

O'REILLY®

Делай как в Google

Разработка
программного
обеспечения



Титус Винтерс, Том Маншрек, Хайрам Райт

Software Engineering at Google

Lessons Learned from Programming Over Time

Titus Winters, Tom Manshreck, and Hyrum Wright

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Делай как в Google

Разработка
программного
обеспечения

Титус Винтерс, Том Маншрек, Хайрам Райт



Санкт-Петербург · Москва · Минск

2021

ББК 32.973.2-018
УДК 004.4
В50

Винтерс Титус, Маншрек Том, Райт Хайрам

В50 Делай как в Google. Разработка программного обеспечения. — СПб.: Питер, 2021. — 544 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1774-1

Современные программисты должны не только эффективно программировать, но и знать надлежащие инженерные практики, позволяющие сделать кодовую базу стабильной и качественной. В чем же разница между программированием и программной инженерией? Как разработчик может управлять живой кодовой базой, которая развивается и реагирует на меняющиеся требования на всем протяжении своего существования? Основываясь на опыте Google, инженеры-программисты Титус Винтерс и Хайрам Райт вместе с Томом Маншреком делают откровенный и пронизательный анализ того, как ведущие мировые практики создают и поддерживают ПО. Речь идет об уникальной инженерной культуре, процессах и инструментах Google, а также о том, как эти аспекты влияют на эффективность разработки. Вы изучите фундаментальные принципы, которые компании разработчиков ПО должны учитывать при проектировании, разработке архитектуры, написании и сопровождении кода.

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

ББК 32.973.2-018
УДК 004.4

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.

ISBN 978-1492082798 англ.

Authorized Russian translation of the English edition
of Software Engineering at Google
ISBN 9781492082798 © 2020 Google, LLC.

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 978-5-4461-1774-1

© Перевод на русский язык ООО Издательство «Питер», 2021
© Издание на русском языке, оформление ООО Издательство «Питер», 2021
© Серия «Бестселлеры O'Reilly», 2021

Краткое содержание

Предисловие	14
Вступление	15
Часть I. Тезисы	21
Глава 1. Что такое программная инженерия?.....	22
Часть II. Культура	43
Глава 2. Успешная работа в команде.....	44
Глава 3. Обмен знаниями.....	59
Глава 4. Инженерия равенства	82
Глава 5. Как стать лидером в команде	93
Глава 6. Масштабируемое лидерство	116
Глава 7. Оценка продуктивности инженеров	132
Часть III. Процессы	147
Глава 8. Правила и руководства по стилю	148
Глава 9. Код-ревью.....	169
Глава 10. Документация	186
Глава 11. Основы тестирования	206
Глава 12. Юнит-тестирование	231
Глава 13. Тестирование с дублерами	256
Глава 14. Крупномасштабное тестирование	279
Глава 15. Устаревание	308
Часть IV. Инструменты	321
Глава 16. Управление версиями и ветвями.....	322
Глава 17. Code Search	346
Глава 18. Системы и философия сборки.....	366
Глава 19. Critique: инструмент обзора кода в Google.....	394
Глава 20. Статический анализ.....	411
Глава 21. Управление зависимостями	424
Глава 22. Крупномасштабные изменения.....	452
Глава 23. Непрерывная интеграция	472
Глава 24. Непрерывная поставка.....	497
Глава 25. Вычисления как услуга	509
Часть V. Заключение	539
Послесловие	540
Об авторах	542
Об обложке	543

Оглавление

Предисловие	14
Вступление	15
Программирование в долгосрочной перспективе	15
Точка зрения Google	16
Что не отражено в книге	17
Заключительные примечания	17
Условные обозначения	17
Благодарности	18
От издательства	20
ЧАСТЬ I. ТЕЗИСЫ	21
Глава 1. Что такое программная инженерия?	22
Время и изменения	24
Масштабирование и эффективность	30
Компромиссы и затраты	36
Программная инженерия и программирование	41
Заключение	42
Итоги	42
ЧАСТЬ II. КУЛЬТУРА	43
Глава 2. Успешная работа в команде	44
Помоги мне скрыть мой код	44
Миф о гениальности	45
Соккрытие вредно	47
Весь секрет в командной работе	50
Заключение	58
Итоги	58

Глава 3. Обмен знаниями	59
Сложности в обучении.....	59
Философия	60
Создание условий: психологическая безопасность	61
Расширение знаний	63
Масштабирование вопросов: вопросы к сообществу	65
Распространяйте знания: вам всегда есть чему научить других	67
Распространение знаний с ростом организации	71
Удобочитаемость: наставничество через обзоры кода	77
Заключение	81
Итоги.....	81
Глава 4. Инженерия равенства	82
Предвзятость — это проблема	82
Понимание необходимости разнообразия	84
Выстраивание мультикультурного потенциала.....	85
Сделать разнообразие действенным	87
Отказ от единых подходов.....	87
Бросьте вызов устоявшимся процессам	89
Ценности и результаты.....	90
Оставайтесь любознательными, двигайтесь вперед.....	91
Заключение.....	91
Итоги.....	92
Глава 5. Как стать лидером в команде	93
Руководители и технические лидеры (и те и другие)	93
Переход от роли разработчика к роли лидера	95
Руководитель	97
Антипаттерны.....	100
Положительные паттерны	104
Неожиданный вопрос	111
Другие советы и рекомендации	111
Люди похожи на растения	113
Заключение.....	115
Итоги.....	115

Глава 6. Масштабируемое лидерство.....	116
Всегда принимайте решение.....	116
Всегда уходите	121
Всегда масштабируйте себя.....	125
Заключение.....	131
Итоги.....	131
Глава 7. Оценка продуктивности инженеров	132
Зачем оценивать продуктивность инженеров?.....	132
Расстановка приоритетов: что измерять?	133
Выбор значимых метрик с использованием целей и сигналов.....	137
Цели.....	138
Сигналы	140
Метрики	140
Использование данных для проверки метрик	141
Принятие мер и оценка результатов.....	145
Заключение.....	145
Итоги.....	145
ЧАСТЬ III. ПРОЦЕССЫ	147
Глава 8. Правила и руководства по стилю.....	148
Зачем нужны правила?.....	149
Создание правил	149
Изменение правил.....	159
Руководства.....	162
Применение правил.....	164
Заключение.....	168
Итоги.....	168
Глава 9. Код-ревью	169
Поток обзора кода.....	170
Как проводятся обзоры кода в Google.....	170
Преимущества обзоров кода.....	172
Передовые практики обзора кода.....	178
Виды обзоров кода	182

Заключение.....	185
Итоги.....	185
Глава 10. Документация.....	186
Что считается документацией?	186
Зачем нужна документация?.....	187
Документация как код	188
Знание своей аудитории	190
Виды документации	192
Обзоры документации	199
Философия документирования	201
Когда привлекать технических писателей?	204
Заключение.....	204
Итоги.....	205
Глава 11. Основы тестирования	206
Почему мы пишем тесты?	207
Проектирование набора тестов	213
Тестирование в масштабе Google	222
История тестирования в Google	224
Ограничения автоматизированного тестирования	228
Заключение.....	229
Итоги.....	230
Глава 12. Юнит-тестирование	231
Важность удобства сопровождения	232
Как предотвратить хрупкие тесты.....	232
Создание ясных тестов	239
Повторное использование тестов и кода: DAMP, не DRY	247
Заключение.....	254
Итоги.....	255
Глава 13. Тестирование с дублерами	256
Влияние тестовых дублеров на разработку ПО	256
Тестовые дублеры в Google	257
Базовые понятия	258

Приемы использования тестовых дублеров	261
Реальные реализации.....	263
Имитации	267
Заглушки.....	271
Тестирование взаимодействий	273
Заключение.....	278
Итоги.....	278
Глава 14. Крупномасштабное тестирование	279
Что такое большие тесты?.....	279
Большие тесты Google	284
Структура большого теста	287
Типы больших тестов	294
Большие тесты и рабочий процесс разработчика.....	302
Заключение.....	307
Итоги.....	307
Глава 15. Устаревание	308
Почему необходимо заботиться об устаревании?.....	309
Почему устаревание вызывает такие сложности?	310
Подходы к прекращению поддержки	313
Управление процессом прекращения поддержки.....	316
Заключение.....	319
Итоги.....	320
ЧАСТЬ IV. ИНСТРУМЕНТЫ.....	321
Глава 16. Управление версиями и ветвями	322
Что такое управление версиями?.....	322
Управление ветвями.....	331
Управление версиями в Google.....	335
Монолитные репозитории.....	340
Будущее управления версиями.....	342
Заключение.....	344
Итоги.....	345

Глава 17. Code Search	346
Пользовательский интерфейс Code Search.....	347
Как гуглеры используют Code Search?	348
Зачем понадобился отдельный веб-инструмент?	350
Влияние масштаба на дизайн	353
Реализация в Google.....	356
Некоторые компромиссы	361
Заключение.....	365
Итоги.....	365
Глава 18. Системы и философия сборки	366
Назначение системы сборки	366
Так ли необходимы системы сборки?	367
Современные системы сборки.....	370
Модули и зависимости.....	386
Заключение.....	392
Итоги.....	393
Глава 19. Critique: инструмент обзора кода в Google	394
Принципы оснащения обзора кода инструментами	394
Процесс обзора кода	395
Этап 1: добавление изменений.....	397
Этап 2: запрос на рецензирование	401
Этапы 3 и 4: исследование и комментирование изменения.....	403
Этап 5: одобрение изменений (оценка изменений).....	406
Этап 6: фиксация изменения	408
Заключение.....	409
Итоги.....	410
Глава 20. Статический анализ	411
Характеристики эффективного статического анализа	411
Ключевые уроки внедрения статического анализа	413
Tricorder: платформа статического анализа в Google	415
Заключение.....	422
Итоги.....	422

Глава 21. Управление зависимостями	424
Почему управлять зависимостями так сложно?.....	425
Импортирование зависимостей.....	428
Теория управления зависимостями.....	433
Ограничения SemVer	438
Управление зависимостями с бесконечными ресурсами	443
Заключение.....	450
Итоги.....	450
Глава 22. Крупномасштабные изменения	452
Что такое крупномасштабное изменение?	452
Кто занимается крупномасштабными изменениями?	454
Препятствия к атомарным изменениям.....	456
Инфраструктура для крупномасштабных изменений.....	461
Процесс крупномасштабных изменений.....	465
Заключение.....	471
Итоги.....	471
Глава 23. Непрерывная интеграция	472
Идеи непрерывной интеграции.....	474
Непрерывная интеграция в Google	487
Заключение.....	496
Итоги.....	496
Глава 24. Непрерывная поставка	497
Идиомы непрерывной поставки в Google	498
Скорость — это командная победа: как разделить процесс развертывания на управляемые этапы	499
Оценка изменений в изоляции: флаги управления функциями	500
Стремление к гибкости: создание серии выпусков.....	501
Качество и ориентация на пользователя: поставляйте только то, что используется.....	503
Сдвиг влево: раннее принятие решений на основе данных.....	504
Изменение культуры команды: дисциплина развертывания.....	506
Заключение.....	507
Итоги.....	507

Глава 25. Вычисления как услуга	509
Приручение вычислительной среды	510
Написание ПО для управляемых вычислений	515
SaaS во времени и масштабе	522
Выбор вычислительной услуги.....	528
Заключение.....	537
Итоги.....	538
ЧАСТЬ V. ЗАКЛЮЧЕНИЕ	539
Послесловие.....	540
Об авторах.....	542
Об обложке	543

Предисловие

Я всегда восхищался тем, как в Google все устроено, и мучил своих друзей-гуглеров (сотрудников Google) расспросами. Как им удается поддерживать такое огромное монолитное хранилище кода? Как десятки тысяч инженеров согласованно работают в тысячах проектов? Как они поддерживают качество систем?

Сотрудничество с «экс-гуглерами» только усилило мое любопытство. Если в одной команде с вами работает бывший инженер Google, вы будете часто слышать: «А вот мы в Google...» Переход из Google в другие компании кажется шокирующим опытом, по крайней мере для инженера. Человек со стороны считает системы и процессы разработки кода в Google одними из лучших в мире, учитывая масштаб компании и то, как часто люди хвалят ее.

В книге «Разработка ПО. Делай как в Google» группа гуглеров (и экс-гуглеров) раскрывает обширный набор практик, инструментов и даже культурных аспектов, лежащих в основе программной инженерии в Google. Но эта книга не ограничивается простым описанием инструментария (о котором можно говорить бесконечно) и дополнительно описывает философию команды Google, которая помогает сотрудникам адаптироваться к разным обстоятельствам. К моему восхищению, несколько глав книги посвящены автоматизированному тестированию, которое продолжает встречать активное сопротивление в отрасли.

Самое замечательное в программной инженерии — это возможность добиться желаемого результата несколькими способами. В каждом проекте инженер должен учитывать множество компромиссов. Что можно заимствовать из открытого исходного кода? Что может спроектировать команда? Что имеет смысл поддерживать для масштаба? У своих друзей-гуглеров я хотел узнать, как устроен гигантский мир Google, богатый талантами и средствами и отвечающий высочайшим требованиям к программному обеспечению (ПО). Их разнообразные ответы познакомили меня с неожиданными аспектами.

В этой книге изложены эти аспекты. Конечно, Google — уникальная компания, и неверно считать ее способ управления разработкой ПО единственно верным. Цель этой книги — помочь вам организовать работу и аргументировать принятие решений, связанных с тестированием, обменом знаниями и совместной деятельностью в команде.

Возможно, вам никогда не придется создавать свой Google и вы не захотите использовать методы этой компании. Но, отказавшись от знакомства с практиками, разработанными в Google, вы лишите себя богатого опыта, накопленного десятками тысяч инженеров, совместно работающих над разработкой софта более двух десятилетий. Это знание слишком ценно, чтобы закрывать на него глаза.

*Камиль Фурнье,
автор книги «От разработчика до руководителя» (М.: МИФ, 2018)*

Вступление

Что именно мы подразумеваем под программной инженерией? Что отличает «программную инженерию» от «программирования» или «computer science»? И как подход Google связан с другими подходами в этой области, описанными во множестве книг в последние пятьдесят лет?

Термины «программирование» и «программная инженерия» используются в отрасли взаимозаменяемо, хотя каждый из них имеет собственное значение. Студенты университетов, как правило, изучают computer science и работают как «программисты».

Но «инженерия» подразумевает применение теоретических знаний для создания чего-то реального и точного. Инженеры-механики, инженеры-строители, авиационные инженеры и специалисты в других инженерных дисциплинах — все они занимаются инженерным делом: используют теоретические знания для создания чего-то реального. Инженеры-программисты также создают «нечто реальное», но это «нечто» неосвязаемо.

В отличие от других инженерных сфер, программная инженерия более открыта для вариантов. Авиационные инженеры должны следовать стандартам, поскольку ошибки в их расчетах могут нанести реальный ущерб. Программирование традиционно не имеет жестких ограничений, но, поскольку ПО интегрируется в нашу жизнь, мы должны определить и использовать более надежные методы его разработки. Надеемся, что эта книга поможет вам увидеть путь к созданию современного ПО.

Программирование в долгосрочной перспективе

На наш взгляд, программная инженерия охватывает не только написание кода, но и все инструменты и процессы создания и дальнейшей поддержки этого кода. Какие методики может внедрить организация, занимающаяся разработкой ПО, чтобы сохранять ценность кода в долгосрочной перспективе? Как инженеры могут сделать кодовую базу более устойчивой, а саму дисциплину программной инженерии более строгой? У нас нет конкретных ответов на эти вопросы, но мы надеемся, что командный опыт Google, накопленный за последние два десятилетия, поможет нам их найти.

Одна из ключевых идей, которой мы делимся в этой книге, заключается в том, что программную инженерию можно рассматривать как «программирование, интегрированное во времени». Какие практики сделают код *устойчивым* — способным реагировать на изменения — в течение его жизненного цикла (от проекта его создания и внедрения до его устаревания)?

В этой книге подчеркиваются три фундаментальных принципа, которые, по нашему мнению, следует учитывать при проектировании, разработке и написании кода.

Время и изменения

Как код должен адаптироваться на протяжении срока действия.

Масштаб и рост

Как организация должна адаптироваться по мере своего развития.

Компромиссы и издержки

Как организация должна принимать решения, основываясь на показателях времени, изменений, масштаба и роста.

В книге мы часто будем возвращаться к этим принципам и показывать, как они влияют на инженерные практики и обеспечивают их устойчивость. (Полное обсуждение этих принципов смотрите в главе 1.)

Точка зрения Google

В компании Google сложился свой уникальный взгляд на рост и развитие устойчивого ПО, основанный на его масштабе и долговечности. Надеемся, что уроки, которые мы извлекли, найдут отражение и в вашей компании, которая тоже развивается и использует все более устойчивые практики.

Темы, обсуждаемые в этой книге, мы разделили на три основных аспекта ландшафта программной инженерии, сложившихся в Google:

- культура;
- процессы;
- инструменты.

Культура Google уникальна, но опыт, который мы получили, изучая ее, широко применим. В главах, посвященных культуре (часть II), мы подчеркнули коллективный характер разработки ПО и необходимость культуры сотрудничества для развития организации и сохранения ее эффективности.

Методики, описанные в части III «Процессы», знакомы большинству инженеров-программистов, но мы отметили, что большой размер кодовой базы Google и долгий срок ее действия обеспечивают надежную основу для выработки оптимальных практик. Мы научились учитывать время и масштаб, а также определили области, где у нас пока нет удовлетворительных ответов.

Наконец, в части IV «Инструменты» мы использовали инвестиции в инфраструктуру инструментов, чтобы получить дополнительные преимущества по мере роста и устаревания кодовой базы. Некоторые из этих инструментов специфичны для Google, но по возможности мы даем ссылки на их альтернативы. Мы полагаем, что основные идеи с успехом можно применять в большинстве инженерных организаций.

Культура, процессы и инструменты, представленные в этой книге, описывают уроки, которые типичный инженер-программист извлекает в процессе работы. Конечно, Google не обладает монополией на полезные советы, и мы делимся своим опытом

не для того, чтобы диктовать вам правила. Эта книга — наша точка зрения, и мы надеемся, что вы найдете ее полезной и сможете применять наш опыт непосредственно или использовать его в качестве отправной точки при выработке своих практик, специфичных для вашей предметной области.

Также эта книга не является учебником. В самой компании Google по-прежнему недостаточно эффективно применяются многие идеи, представленные здесь. Наш опыт является результатом наших неудач: мы все еще допускаем ошибки, внедряем несовершенные решения и должны последовательно улучшать код. Тем не менее огромный размер Google гарантирует существование множества решений для каждой проблемы. Мы надеемся, что в этой книге нам удалось собрать самые лучшие из них.

Что не отражено в книге

Эта книга не охватывает вопросы проектирования ПО (собранная по этой теме информация заслуживает отдельной книги). Мы привели некоторый код, но только для примера, чтобы показать основные принципы, которые не зависят от языка, и в этих главах практически нет рекомендаций по «программированию». Также эта книга не охватывает многих важных вопросов, касающихся разработки ПО, таких как управление проектами, разработка API, усиление безопасности, интернационализация, применение фреймворков пользовательского интерфейса и других проблем, связанных с конкретным языком программирования. Мы решили оставить эти темы в стороне, понимая, что не сможем раскрыть их в достаточной степени, и уделили больше внимания инженерии, а не программированию.

Заключительные примечания

Эта книга является результатом совместных усилий и должна послужить окном в мир программной инженерии крупной организации, создающей свои продукты. Надеемся, что вместе с другими источниками она поможет нашей отрасли внедрять наиболее передовые и устойчивые практики. Но самое главное, мы хотим, чтобы книга вам понравилась и вы взяли на вооружение некоторые из представленных здесь уроков.

Том Маншрек

Условные обозначения

В этой книге приняты следующие обозначения:

Курсив

Используется для обозначения новых терминов.

Моноширинный шрифт

Применяется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена переменных и функций, баз данных, типов данных, переменных окружения, инструкций и ключевых слов.



Так обозначаются советы, предложения и примечания общего характера.

Благодарности

Эта книга является результатом труда множества людей. Все представленное здесь является квинтэссенцией знаний, накопленных нами и другими сотрудниками Google за долгие годы. Мы лишь передаем вам опыт, которым поделились с нами старшие коллеги из Google и других компаний. Перечислить их всех трудно, но мы выражаем им свою благодарность.

Также мы благодарим Мелоди Мекфессель (Melody Meckfessel) за поддержку этого проекта на начальном этапе, а также Дэниела Джаспера (Daniel Jasper) и Дэнни Берлина (Danny Berlin) за помощь в его завершении.

Эта книга была бы невозможна без колоссальных совместных усилий наших кураторов, авторов и редакторов. Авторы и редакторы указаны в каждой главе или врезке, а здесь мы отметим всех, кто внес вклад в каждую главу, участвуя в обсуждении и подготовке к печати.

- **Что такое программная инженерия?** Санджай Гемават (Sanjay Ghemawat), Эндрю Хаятт (Andrew Hyatt).
- **Успешная работа в команде:** Сибли Бэкон (Sibley Bacon), Джошуа Мортон (Joshua Morton).
- **Обмен знаниями:** Дмитрий Глазков, Кайл Лимонс (Kyle Lemons), Джон Риз (John Reese), Дэвид Симондс (David Symonds), Эндрю Тренк (Andrew Trenk), Джеймс Такер (James Tucker), Дэвид Колбрэннер (David Kohlbrenner), Родриго Дамацио Бовендорп (Rodrigo Damazio Bovendorp).
- **Инженерия равенства:** Камау Бобб (Kamau Bobb), Брюс Ли (Bruce Lee).
- **Как стать лидером в команде:** Джон Уили (Jon Wiley), Лоран Ле Брун (Laurent Le Brun).
- **Масштабируемое лидерство:** Брайан О'Салливан (Bryan O'Sullivan), Бхарат Медиратта (Bharat Mediratta), Дэниел Джаспер (Daniel Jasper), Шайндел Шварц (Shaindel Schwartz).
- **Оценка продуктивности инженеров:** Андреа Найт (Andrea Knight), Коллин Грин (Collin Green), Кетлин Садовски (Caitlin Sadowski), Макс-Канат Александер (Max-Kanat Alexander), Илай Янг (Yilei Yang).
- **Правила и руководства по стилю:** Макс-Канат Александер (Max-Kanat Alexander), Титус Винтерс (Titus Winters), Мэтт Аустерн (Matt Austern), Джеймс Деннетт (James Dennett).

- **Код-ревью:** Макс-Канат Александер (Max-Kanat Alexander), Брайан Ледгер (Brian Ledger), Марк Баролак (Mark Barolak).
- **Документация:** Йонас Вагнер (Jonas Wagner), Смит Хинсу (Smit Hinsu), Джеффри Ромер (Geoffrey Romer).
- **Основы тестирования:** Эрик Куфлер (Erik Kufler), Эндрю Тренк (Andrew Trenk), Диллон Блай (Dillon Bly), Джозеф Грейвс (Joseph Graves), Нил Норвитц (Neal Norwitz), Джей Корбетт (Jay Corbett), Марк Стрибек (Mark Striebeck), Брэд Грин (Brad Green), Мишко Хевери (Miško Hevery), Антуан Пикар (Antoine Picard), Сара Сторк (Sarah Storck).
- **Юнит-тестирование:** Эндрю Тренк (Andrew Trenk), Адам Бендер (Adam Bender), Диллон Блай (Dillon Bly), Джозеф Грейвс (Joseph Graves), Титус Винтерс (Titus Winters), Хайрам Райт (Hugum Wright), Оги Факлер (Augie Fackler).
- **Тестирование с дублерами:** Джозеф Грейвс (Joseph Graves), Геннадий Цивил (Gennadiy Civil).
- **Крупномасштабное тестирование:** Адам Бендер (Adam Bender), Эндрю Тренк (Andrew Trenk), Эрик Кюфлер (Erik Kuefler), Мэтью Бомонт-Гай (Matthew Beaumont-Gay).
- **Устаревание:** Грег Миллер (Greg Miller), Энди Шульман (Andy Shulman).
- **Управление версиями и ветвями:** Рейчел Потвин (Rachel Potvin), Виктория Кларк (Victoria Clarke).
- **Code Search:** Дженни Ван (Jenny Wang).
- **Системы и философия сборки:** Хайрам Райт (Hugum Wright), Титус Винтерс (Titus Winters), Адам Бендер (Adam Bender), Джефф Кокс (Jeff Cox), Жак Пиенаар (Jacques Pienaar).
- **Critique: инструменты обзора кода в Google:** Миколай Додела (Mikołaj Dądela), Герман Луз (Hermann Loose), Ева Мэй (Eva May), Элис Кобер-Соцек (Alice Kober-Sotzek), Эдвин Кемпин (Edwin Kempin), Патрик Хизель (Patrick Hiesel), Оле Ремсен (Ole Rehmsen), Ян Мацек (Jan Macek).
- **Статический анализ:** Джеффри ван Гог (Jeffrey van Gogh), Сиера Джаспан (Sierra Jaspan), Эмма Седерберг (Emma Söderberg), Эдвард Афтандилиан (Edward Aftandilian), Коллин Винтер (Collin Winter), Эрик Хо (Eric Haugh).
- **Управление зависимостями:** Расс Кокс (Russ Cox), Николас Данн (Nicholas Dunn).
- **Крупномасштабные изменения:** Мэтью Фаулз Кулукундис (Matthew Fowles Kulukundis), Адам Зарек (Adam Zarek).
- **Непрерывная интеграция:** Джефф Листфилд (Jeff Listfield), Джон Пеникс (John Penix), Каушик Шридаран (Kaushik Sridharan), Санджив Дханда (Sanjeev Dhanda).

- **Непрерывная поставка:** Дэйв Оуэнс (Dave Owens), Шери Шипе (Sheri Shipe), Бобби Джонс (Bobbi Jones), Мэтт Дафтлер (Matt Duftler), Брайан Шутер (Brian Szuter).
- **Вычисления как услуга:** Тим Хокин (Tim Hockin), Коллин Винтер (Collin Winter), Ярек Кузьмиерек (Jarek Kuśmierек).

Также мы хотим поблагодарить Бетси Бейер за то, что поделилась опытом работы над книгой «Site Reliability Engineering. Надежность и безотказность как в Google»¹. Спасибо Кристоферу Гузиковски (Christopher Guzikowski) и Алисии Янг (Alicia Young) из O'Reilly, которые запустили наш проект и довели его до публикации.

Кураторы также хотели сказать отдельное спасибо:

Том Маншрек (Tom Manshreck): «Маме и папе за то, что помогли мне поверить в себя и решали со мной домашние задания за обеденным столом».

Титус Винтерс (Titus Winters): «Папе за мой путь. Маме — за мой голос. Виктории — за мое сердце. Рафу — за мою спину. А также мистеру Снайдеру, Ранве, Зеду (Z), Майку, Заку, Тому и всем Пейнам, Меку (mec), Тоби, Кгд (cgd) и Мелоди за уроки, наставничество и доверие».

Хайрам Райт (Hyrum Wright): «Маме и папе за поддержку. Брайану и завсегдаатаям Бейкерленда за мое первое погружение в мир ПО. Дюэйну (Dewaune) за продолжение этого путешествия. Ханне, Джонатану, Шарлотте, Спенсеру и Бену за их дружбу и интерес. Хизер за то, что все это время была рядом».

От издательства

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

Мы будем рады узнать ваше мнение!

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

¹ Бейер Б., Джоунс К., Петофф Д., Мёрфи Н. Р. Site Reliability Engineering. Надежность и безотказность как в Google. СПб.: Питер, 2021. 592 с.: ил.

ЧАСТЬ I

Тезисы

Что такое программная инженерия?

*Автор: Титус Винтерс
Редактор: Том Манирек*

Ничто не строится на камнях; все построено на песке, но мы должны строить так, как если бы песок был камнем.

Хорхе Луис Борхес

Программирование и программная инженерия имеют три важных отличия: время, масштаб и компромиссы. В отличие от обычных программистов, инженеры-программисты должны больше внимания уделять течению времени и необходимости внесения изменений, больше заботиться о масштабе и эффективности как самого ПО, так и организации, которая его производит. Инженеры-программисты принимают более сложные решения, дороже платят за ошибки и часто опираются на неточные оценки времени и роста.

Мы в Google иногда говорим: «Программная инженерия — это программирование, интегрированное во времени». Программирование, безусловно, является важной частью программной инженерии: в конце концов, именно в процессе программирования создается новый софт. Но если мы разделяем понятия, то нужно разграничить задачи программирования (разработку) и программной инженерии (разработку, изменение, сопровождение). Время — это новое важное измерение в программировании. Куб — это не квадрат, расстояние — это не скорость. Программная инженерия — это не программирование.

Чтобы понять, как время влияет на программу, задумайтесь: «Как долго будет жить¹ код?» Крайние оценки в ответах на этот вопрос могут отличаться в 100 000 раз. Легко представить код, который просуществует несколько минут, или другой код, служащий десятилетиями. Как правило, короткоживущий код не зависит от времени: едва ли нужно адаптировать утилиту, которая проживет час, к новым версиям базовых библиотек, операционной системы (ОС), аппаратного обеспечения или языков

¹ Мы не имеем в виду «продолжительность выполнения», мы имеем в виду «продолжительность поддержки» — как долго код будет продолжать развиваться, использоваться и поддерживаться? Как долго это ПО будет иметь ценность?

программирования. Недолговечные системы фактически являются «мимолетными» задачами программирования и напоминают куб, который сжат вдоль одного измерения до вида квадрата. Но с увеличением продолжительности жизни кода изменения становятся более важными для него. За десяток лет большинство программных зависимостей, явных или неявных, почти наверняка изменятся. Понимание этого явления помогает отличать программную инженерию от программирования.

Это отличие лежит в основе *устойчивости* в мире ПО. Проект будет более *устойчив*, если в течение ожидаемого срока его службы инженер-программист сможет сохранить способность реагировать на любые важные технические или коммерческие изменения. Именно «сможет» реагировать в том случае, если обновление будет иметь ценность¹. Когда вы теряете способность реагировать на изменения в базовых технологиях или развитии продукта, не надейтесь, что такие изменения никогда не превысят критического уровня. В условиях проекта, развивающегося несколько десятилетий, такие надежды обречены на провал².

Взглянуть на программную инженерию можно со стороны оценки масштаба проекта. Сколько людей вовлечено в разработку? Как меняются их роли при разработке и сопровождении проекта с течением времени? Программирование часто является актом индивидуального творчества, но программная инженерия — это командная работа. Одно из первых и самых точных определений программной инженерии звучит так: «Разработка многоверсионных программ для большого числа людей»³. Оно означает, что различие между программной инженерией и программированием определяется количеством пользователей и сроком действия продукта. Командная работа создает новые проблемы, но также открывает такие возможности для создания систем, какие не может предложить один программист.

Организация команды, состав проекта, а также стратегия и тактика его развития — важные компоненты программной инженерии, которые зависят от масштаба организации. Растет ли эффективность производства софта по мере увеличения организации и расширения ее проектов? Растет ли эффективность рабочего процесса по мере развития организации и насколько пропорционально этому растет стоимость стратегий тестирования и управления версиями? Проблемы масштаба, связанные с увеличением числа сотрудников и налаживанием коммуникации между ними, обсуждались с первых дней программной инженерии, начиная с появления

¹ Это вполне точное определение технического долга: он возникает тогда, когда что-то «должно» быть сделано, но еще не сделано, и является разницей между текущим кодом и будущим желаемым кодом.

² Тут вполне уместен вопрос: как заранее узнать, что проект будет долгосрочным?

³ Сейчас трудно установить авторство определения: одни считают, что впервые оно было сформулировано Брайаном Рэнделлом (Brian Randell) или Маргарет Гамильтоном (Margaret Hamilton), другие — что автором является Дейв Парнас (Dave Parnas). Это определение часто приводят как цитату из отчета «Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee», Рим, Италия, 27–31 октября 1969 г., Брюссель, отдел по научным вопросам, НАТО.

мифического человеко-месяца¹. Часто они имеют политический характер, и от их решения во многом зависят устойчивость ПО и ответ на вопрос: «Как дорого обойдется то, что придется делать снова и снова?»

Еще одно отличие программной инженерии от программирования заключается в сложности принятия решений и цене ошибок. Постоянная оценка компромиссов между несколькими путями движения вперед часто основана на несовершенных данных, и иногда цена ошибки очень высока. Работа инженера-программиста или лидера команды инженеров состоит в стремлении к устойчивости организации, продукта и процесса разработки, а также в управлении ростом затрат. Иногда можно откладывать технические изменения или даже принимать плохо масштабируемые политики, которые потом придется пересмотреть. Но, делая такой выбор, инженер-программист должен понимать, к каким затратам эти решения приведут в будущем.

Универсальные решения в программной инженерии встречаются редко, как вы увидите и в этой книге. Учитывая разброс в 100 000 раз между ответами на вопрос «Как долго будет жить ПО?» и в 10 000 раз — между ответами на вопросы «Сколько инженеров работает в компании?» и «Сколько вычислительных ресурсов доступно для проекта?», опыт Google почти наверняка не будет соответствовать вашей ситуации. Поэтому в этой книге мы постарались показать, как в Google искали правильные пути в разработке и сопровождении ПО, рассчитанного на десятилетия, имея десятки тысяч инженеров и вычислительные ресурсы мирового масштаба. Большинство методов, применение которых мы считаем обязательным в таком масштабе, также хорошо подойдут для меньших организаций: считайте эту книгу отчетом одной инженерной экосистемы, который может вам пригодиться. Иногда сверхбольшие масштабы связаны с повышенными расходами, и, возможно, благодаря нашим предупреждениям, когда ваша организация вырастет до таких масштабов, вы сможете найти более удачное решение по издержкам.

Прежде чем перейти к обсуждению особенностей командной работы, культуры, политики и инструментов, давайте подробнее рассмотрим время, масштаб и компромиссы.

Время и изменения

Когда человек учится программировать, срок действия его кодов обычно измеряется часами или днями. Задачи и упражнения по программированию, как правило, пишутся с нуля, почти без рефакторинга и, конечно, без долгосрочного сопровождения. Часто после первого использования эти программы не пересобираются и не применяются на практике. Получая среднее или высшее образование, вы могли участвовать в групповой реализации проекта, который живет месяц или чуть дольше. Такой проект может включать рефакторинг кода, например в ответ на из-

¹ Брукс Ф. Мифический человеко-месяц, или Как создаются программные системы. СПб.: Питер, 2021. — Примеч. пер.

меняющиеся требования, но маловероятно, что он будет зависеть от более широких изменений в его окружении.

Разработчиков короткоживущего кода можно также найти в некоторых отраслях. Мобильные приложения, например, обычно имеют довольно короткий срок действия¹ и часто переписываются заново. На ранних стадиях развития стартапов инженеры могут вполне обоснованно сосредоточиться на ближних целях в ущерб долгосрочным инвестициям, поскольку сама компания, в которой они работают, может просуществовать недолго. Разработчик серийного стартапа вполне может иметь десятилетний опыт разработки и почти или совсем не иметь опыта поддержки ПО, которое должно работать дольше одного-двух лет.

С другой стороны, есть успешные проекты с практически неограниченным сроком службы: трудно предсказать, когда прекратит свое существование Google Search, ядро Linux или Apache HTTP Server. Большинство проектов Google должны существовать неопределенно долго и периодически претерпевать обновления зависимостей, языковых версий и т. д. С течением времени такие долгоживущие проекты *рано или поздно* начинают восприниматься иначе, чем задачи по программированию или развитию стартапа.

На рис. 1.1 показаны два программных проекта на противоположных концах спектра «ожидаемого срока службы». Как обслуживать проект с ожидаемым сроком службы, измеряемым часами? Должен ли программист бросить все и заняться обновлением, если во время работы над сценарием на Python, который будет выполнен всего один раз, вышла новая версия ОС? Конечно, нет: такое обновление не критично. Но если проект Google Search, находящийся на противоположном конце спектра, застрянет на версии ОС 1990-х годов, обслуживание станет проблемой.



Рис. 1.1. Срок жизни и важность обновлений

¹ Как заявляют в компании Appcelerator, «ничто в мире не определено, кроме смерти, налогов и короткого срока службы мобильных приложений» (https://oreil.ly/pnT2_, блог Axway Developer, 6 декабря 2012 года).

Наличие точек на спектре сроков службы, соответствующих низкой и высокой важности обновлений, предполагает, что где-то есть переход. Где-то на линии, соединяющей одноразовую программу и проект, развивающийся десятилетиями, есть этап появления реакции проекта на изменение внешних факторов¹. Любой проект, в котором изначально не планировались обновления, переживает переход болезненно по трем причинам, каждая из которых усугубляет две другие:

- обновления еще не выполнялись в этом проекте: по ним есть только предположения;
- инженеры едва ли имеют опыт проведения обновлений;
- большой объем обновлений: одновременно приходится применять обновления, накопившиеся за несколько лет, вместо постепенного применения небольших обновлений.

Далее, выполнив такое обновление один раз (полностью или частично), вы рискуете переоценить стоимость следующего обновления и решить: «Никогда больше». Компании, которые приходят к такому выводу, заканчивают тем, что просто выкидывают старый код и пишут его заново или решают никогда не обновлять его снова. Вместо того чтобы поддаться естественному желанию избежать болезненной процедуры, часто полезнее инвестировать в этот процесс, чтобы сделать его менее болезненным. Впрочем, выбор зависит от стоимости обновления, его ценности и ожидаемого срока службы проекта.

Суть устойчивости проекта заключается не только в преодолении первого крупного обновления, но и в достижении уверенности, что проект идет в ногу со временем. Устойчивость требует оценки влияния необходимых изменений и управления ими. Мы считаем, что достигли такой устойчивости во многих проектах в Google, в основном путем проб и ошибок.

Итак, чем отличается программирование короткоживущего кода от производства кода с более долгим ожидаемым сроком службы? С течением времени мы стали намного четче осознавать разницу между «работающим по счастливой случайности» и «удобным в сопровождении». Не существует идеального решения вышеназванных проблем. Это прискорбно, потому что длительное сопровождение ПО — это постоянная борьба.

Закон Хайрама

Если вы поддерживаете проект, который используется другими инженерами, то между «работающим по счастливой случайности» и «удобным в сопровождении» есть одно важное отличие, которое мы назвали *законом Хайрама*:

¹ Точка перехода во многом зависит от ваших приоритетов и предпочтений. Судя по нашему опыту, большинство проектов подходит к черте, когда обновление становится необходимостью, где-то через пять лет работы ПО. По более консервативным оценкам этот переход находится где-то между пятью и десятью годами службы.

Если число пользователей API достаточно велико, неважно, что вы обещаете в контракте: любое наблюдаемое поведение системы будет зависеть от чьих-то действий.

По нашему опыту, эта аксиома является доминирующим фактором в любом обсуждении изменения ПО. На концептуальном уровне ее можно сравнить с энтропией: закон Хайрама обязательно должен учитываться при обсуждении изменений и сопровождения¹, так же как при обсуждении вопросов эффективности или термодинамики должна учитываться энтропия. Тот факт, что энтропия никогда не уменьшается, не означает, что мы не должны стремиться к эффективности. То, что закон Хайрама будет действовать применительно к сопровождению ПО, не означает, что мы не должны планировать или пытаться лучше понять это сопровождение. Можно смягчить последствия проблем, даже если мы знаем, что они никогда не исчезнут.

Закон Хайрама основан на практическом понимании, что даже при наличии самых лучших намерений, лучших инженеров и широкого круга методик проверки кода нельзя надеяться на полное соблюдение опубликованных контрактов или передовых практик. Как владелец API вы имеете *некоторую* свободу, четко понимая возможности интерфейса, но на практике сложность изменения также зависит от того, насколько полезным для пользователя является наблюдаемое поведение API. Если пользователи не зависят от него, изменить API будет легко. Но с течением времени и при достаточном количестве пользователей даже самые безобидные изменения *обязательно* что-то нарушат². Анализируя ценность изменений, учитывайте трудности, связанные с поиском, выявлением и устранением нарушений, которые они вызовут.

Пример: упорядоченный хеш

Рассмотрим пример упорядочения итераций по хешу. Если вставить в хеш пять элементов, в каком порядке мы их получим?

```
>>> for i in {"apple", "banana", "carrot", "durian", "eggplant"}: print(i)
...
durian
carrot
apple
eggplant
banana
```

Большинство программистов знают, что хеш-таблицы не упорядочивают данные явно. Но почти никто не знает, что, *возможно*, хеш-таблица, которую они используют, возвращает содержимое в определенном порядке. Этот факт кажется не-примечательным, но за последнюю пару десятилетий опыт использования хэша эволюционировал.

¹ Надо признать, что сам Хайрам собирался назвать этот закон «законом неявных зависимостей», но в Google предпочли более краткое название «закон Хайрама».

² См. комикс «Workflow» (<http://xkcd.com/1172>) на сайте xkcd.

- Атаки *переполнения хеша* (hash flooding)¹ стимулируют недетерминированный характер хранения данных в хеше.
- Потенциальный выигрыш от поиска усовершенствованных алгоритмов хеширования или хеш-контейнеров требует изменения порядка итераций в хеше.
- Согласно закону Хайрама, программисты по возможности пишут программы, зависящие от порядка обхода хеш-таблицы.

Если спросить эксперта: «Можно ли положиться на конкретный порядок обхода элементов в хеш-контейнере?» — он наверняка ответит: «Нет». Это правильный ответ, но слишком упрощенный. Более точный ответ мог бы звучать так: «Если код недолговечный и не предполагает будущих изменений в аппаратном или программном окружении или структуре данных, то это вполне допустимо. Но если известно, что код будет жить долго или нельзя гарантировать, что зависимости никогда не изменятся, то предположение неверно». Более того, даже если ваша собственная реализация не зависит от порядка хранения данных в хеш-контейнере, этот порядок может использоваться другим кодом, неявно создающим такую зависимость. Например, если ваша библиотека сериализует значения перед вызовом удаленной процедуры (RPC, remote procedure call), вызывающая сторона может оказаться в зависимости от порядка следования этих значений.

Это очень простой пример различия между «это работает» и «это правильно». Зависимость недолговечной программы от порядка хранения данных в контейнере не вызывает технических проблем. С другой стороны, для проекта, срок жизни которого преодолевает некоторый порог, такая зависимость представляет значительный риск: по прошествии времени кто-то или что-то может сделать этот порядок ценным. Ценность проявляется по-разному: как эффективность, безопасность или просто пригодность структуры данных для изменения в будущем. Когда ценность очевидна, взвесьте все за и против, выбирая между этой ценностью и проблемами, с которыми могут столкнуться разработчики или клиенты.

Некоторые языки намеренно изменяют порядок хеширования в каждой следующей версии библиотеки или при каждом запуске программы, чтобы предотвратить появление зависимости от этого порядка. Но даже в этом случае закон Хайрама преподносит сюрпризы, поскольку такое упорядочивание происходит с использованием генератора случайных чисел. Устранение случайности может нарушить работу кода пользователей, полагающихся на него. В любой термодинамической системе энтропия увеличивается, и точно так же к любому наблюдаемому поведению применим закон Хайрама.

Между кодами, написанными для «работы сейчас» и для «работы всегда», можно выделить четкие взаимосвязи. Рассматривая код как артефакт с переменным (в ши-

¹ Разновидность атак типа «отказ в обслуживании» (DoS, denial-of-service), при которых злоумышленник, зная внутреннюю организацию хеш-таблицы и особенности хеш-функции, может сформировать данные таким образом, чтобы снизить алгоритмическую производительность операций над таблицей.

роких пределах) временем жизни, можно определить стили программирования: код, зависящий от хрупких и недокументированных особенностей, скорее всего, будет классифицирован как «хакерский» или «хитрый», тогда как код, следующий передовым практикам и учитывающий возможность развития в будущем, вероятно, будет считаться «чистым» и «удобным для сопровождения». Оба стиля имеют свои цели, но выбор одного из них во многом зависит от ожидаемого срока службы кода. Мы привыкли использовать термин *программирование*, если характеристика «хитрый» — это комплимент, и *программная инженерия*, если слово «хитрый» имеет отрицательный оттенок.

Почему бы просто не «отказаться от изменений»?

Все наше обсуждение времени и реакции на изменения связано с неизбежностью изменений. Верно?

Как и все остальное, о чем говорится в книге, принятие решения об обновлении зависит от обстоятельств. Мы готовы подтвердить, что «в большинстве проектов, существующих достаточно долго, рано или поздно возникает необходимость обновления». Если у вас есть проект, написанный на чистом C и не имеющий внешних зависимостей (или имеющих только зависимости, гарантирующие стабильность в течение долгого времени, такие как стандарт POSIX), вы вполне можете избежать рефакторинга или сложного обновления. Разработчики языка C прикладывают значительные усилия, чтобы обеспечить его стабильность.

Но большинство проектов подвержены изменениям в базовой технологии. В основном языки программирования и среды выполнения меняются активнее, чем язык C. Даже библиотеки, реализованные на чистом C, могут меняться для поддержки новых функций и тем самым влиять на пользователей. Проблемы безопасности есть во всех технологиях, от процессоров до сетевых библиотек и прикладного кода. *Каждый* элемент технологии может стать причиной критической ошибки и уязвимости безопасности, о которых вы не узнаете заранее. Если вы не применили исправления для Heartbleed (<http://heartbleed.com>) или не смягчили проблемы с предупреждающим выполнением, такие как Meltdown и Spectre (<https://meltdownattack.com>), потому что полагали (или были уверены), что ничего не изменится, последствия будут серьезными.

Повышение эффективности еще больше осложняет картину. Мы стараемся оснастить вычислительные центры экономичным оборудованием, разумно использующим процессоры. Но старые алгоритмы и структуры данных на новом оборудовании работают хуже: связанный список или двоичное дерево поиска продолжают работать, но растущий разрыв между скоростями работы процессора и памяти влияет на «эффективность» кода. Со временем ценность обновления аппаратного обеспечения может уменьшаться в отсутствие изменений в архитектуре ПО. Обратная совместимость гарантирует работоспособность старых систем, но не гарантирует эффективности старых оптимизаций. Нежелание или неспособность воспользоваться новыми аппаратными возможностями чреваты большими издержками. Подобные

проблемы сложны: оригинальный дизайн может быть логичным и разумным для своего времени, но после эволюции обратных совместимых изменений предпочтительным становится новый вариант (ошибок не было, но время сделало изменения ценными и востребованными).

Подобные проблемы объясняют, почему в долгосрочных проектах необходимо уделять внимание устойчивости независимо от того, влияют ли они непосредственно на нас или только на технологии. Изменения — это не всегда хорошо. Мы не должны меняться только ради перемен, но должны готовиться к изменениям и инвестировать в их удешевление. Как известно каждому системному администратору, одно дело — знать, что есть возможность восстановить данные с ленты, и совсем другое — знать, как это сделать и во что это обойдется. Практика и опыт — это двигатели эффективности и надежности.

Масштабирование и эффективность

Как отмечается в книге Бетси Бейер и др. «Site Reliability Engineering. Надежность и безотказность как в Google» (СПб.: Питер, 2019) (далее Site Reliability Engineering — SRE), производственная система в компании Google относится к числу самых сложных систем, созданных человеком. Формирование такой машины и поддержание ее бесперебойной работы потребовали бесчисленных часов размышлений, обсуждений и проектирования с участием экспертов со всего мира.

Большая часть *этой* книги посвящена сложностям, связанным с масштабированием в организации, которая производит такие машины, и процессам поддержания этой машины в рабочем состоянии в течение долгого времени. Давайте снова вернемся к понятию устойчивости кодовой базы: «База кода организации является *устойчивой*, если вы *можете* без опаски изменять то, что нужно, в течение срока ее службы». В данной формулировке под возможностями также понимаются издержки: если обновление чего-либо сопряжено с чрезмерными затратами, оно, скорее всего, будет отложено. Если со временем затраты растут сверхлинейно, это значит, что выполняемая операция не масштабируется¹ и наступит момент, когда изменения станут неизбежными. Когда проект вырастет вдвое и понадобится выполнить эту операцию снова, окажется ли она вдвое более трудоемкой? Найдутся ли человеческие ресурсы для ее выполнения в следующий раз?

Люди — не единственный ограниченный ресурс, который необходимо наращивать. Само ПО должно хорошо масштабироваться в отношении традиционных ресурсов, таких как вычислительная мощность, память, объем хранилища и пропускная способность. Разработка ПО также должна масштабироваться с точки зрения количества участвующих в ней людей и объема вычислительных ресурсов. Если стоимость вы-

¹ Всякий раз, когда в этой главе мы используем слово «масштабируемость» в неформальном контексте, мы подразумеваем «сублинейное масштабирование в отношении человеческих возможностей».

числений на тестовом кластере растет сверхлинейно и на каждого человека в квартал приходится все больше вычислительных ресурсов, это значит, что положение неустойчиво и скоро придется что-то изменить.

Наконец, самый ценный актив организации, производящей ПО, — кодовая база — тоже нуждается в масштабировании. При сверхлинейном росте системы сборки или системы управления версиями (VCS, version control system) (возможно, в результате увеличения истории изменений) может наступить момент, когда работать с ней станет невозможно. Многим аспектам, таким как «время для полной сборки», «время для получения новой копии репозитория» или «стоимость обновления до новой языковой версии», не уделяется должного внимания из-за того, что они меняются очень медленно. Но они с легкостью могут превратиться в метафорическую сварившуюся лягушку (<https://oreil.ly/clqzN>): медленно накапливающиеся проблемы слишком легко усугубляются и почти никогда не проявляются в виде конкретного момента кризиса. Только имея полное представление об организации в целом и стремясь к масштабированию, вы, возможно, сможете оставаться в курсе этих проблем.

Все, что организация использует для производства и поддержки кода, должно быть масштабируемым с точки зрения издержек и потребления ресурсов. В частности, все операции, которые организация должна выполнять снова и снова, должны быть масштабируемыми относительно человеческих усилий. С этой точки зрения многие распространенные политики не могут считаться хорошо масштабируемыми.

Плохо масштабируемые политики

Даже небольшой опыт разработки позволяет легко определять политики, плохо поддающиеся масштабированию, чаще всего по изменению объема работы, приходящемуся на одного инженера, при расширении компании. Если организация увеличится в 10 раз, увеличится ли в 10 раз нагрузка на одного инженера? Увеличится ли объем работы, которую он должен выполнять, при увеличении кодовой базы? Если на какой-то из этих вопросов будет дан положительный ответ, найдутся ли механизмы для автоматизации или оптимизации работы этого инженера? Если нет, значит, в организации есть явные проблемы с масштабированием.

Рассмотрим традиционный подход к устареванию (подробнее об устаревании в главе 15) в контексте масштабирования. Представьте, что принято решение использовать новый виджет вместо старого. Чтобы мотивировать разработчиков, руководители проекта говорят: «Мы удалим старый виджет 15 августа, не забудьте перейти к использованию нового виджета».

Такой подход хорошо работает в небольших проектах, но быстро терпит неудачу с увеличением глубины и широты графа зависимостей. Команды зависят от постоянно растущего числа виджетов, и любое нарушение в сборке может повлиять на рост компании. При этом вместо переключивания хлопот, связанных с миграцией нового кода, на клиентов команды могут сами внедрить все необходимое, используя преимущества экономии от масштабируемого решения.

В 2012 году мы начали смягчать проблему обновления и поручили командам, отвечающим за инфраструктуру, переводить внутренних пользователей на применение новых версий или выполнять обновление на месте с обеспечением обратной совместимости. Эта стратегия, которую мы назвали «правилом обновления», хорошо масштабируется: зависимые проекты перестали тратить все больше и больше усилий, чтобы просто не отстать. Мы также выяснили, что специальная группа экспертов внедряет обширные изменения намного лучше, чем пользователи: эксперты некоторое время изучают всю глубину изменения, а затем применяют полученные знания в каждой подзадаче. Внедрение изменений пользователем вызывает замедление в работе: он вынужден решать проблему непосредственно у себя, а затем выбрасывать ставшие бесполезными знания. Опыт экспертов масштабируется лучше.

Еще один пример плохо масштабируемой политики — традиционное использование ветвей разработки. Если слияние крупных изменений с главной ветвью дестабилизировало продукт, можно сделать вывод: «Нужно более жесткое управление слиянием. Слияния должны производиться реже!» и для каждой команды (или функциональной возможности) создать отдельные ветви разработки. Когда такая ветвь достигнет конечной точки разработки, она будет протестирована и объединена с главной ветвью, из-за чего другие инженеры, работающие над другими ветвями, будут вынуждены повторно синхронизировать репозитории и проводить тесты. Такой способ управления ветвями можно использовать в небольшой организации, где одновременно разрабатывается 5–10 подобных ветвей. Но по мере роста организации (и количества ветвей) этот подход увеличивает накладные расходы на многократное выполнение одной и той же задачи. Более эффективный подход обсудим в главе 16.

Хорошо масштабируемые политики

Какие политики помогают оптимизировать затраты по мере роста организации? Точнее, какие виды политики можно внедрить, чтобы обеспечить суперлинейный рост ценности при росте организации?

Одна из наших любимых внутренних политик позволяет командам поддержки инфраструктуры безопасно вносить изменения в инфраструктуру. «Если в работе продукта есть сбой или другие проблемы после изменений в инфраструктуре, но их причины не выявлены тестами в системе непрерывной интеграции (CI, continuous integration), значит, эти причины не связаны с изменениями». Другими словами: «Если вам что-то понравилось, добавьте соответствующий тест в систему непрерывной интеграции». Мы называем это «правилом Бейонсе»¹. С точки зрения масштабирования правило Бейонсе подразумевает, что сложные одноразовые специализированные тесты, которые не запускаются системой непрерывной интеграции, не нужно учитывать:

¹ Это отсылка к популярной песне «Single ladies», в которой рефреном звучит фраза: «If you liked it then you shoulda put a ring on it». («Если я тебе нравилась, так надел бы мне на палец колечко».)

инженер инфраструктурной команды может найти другую команду, добавившую код, и спросить, как этот код тестировать. Это удобно, когда в команде сто инженеров, но со временем нам пришлось расширить эту стратегию.

Мы обнаружили высокую эффективность форумов для общения в крупных организациях. Когда инженеры обсуждают вопросы на форумах, знания быстро распространяются, и новые специалисты набираются опыта. Если у вас есть сотня инженеров, пишущих на Java, то один опытный эксперт по Java, готовый ответить на их вопросы, вскоре даст вам сотню инженеров, пишущих лучший код на Java. Знания — это вирусы, а эксперты — их носители. Более подробно коснемся этой темы в главе 3.

Пример: обновление компилятора

Рассмотрим сложную задачу обновления компилятора. Теоретически обновление компилятора не должно вызывать сложностей, учитывая, сколько усилий прилагают разработчики языков, чтобы сохранить обратную совместимость. Но насколько простым является это обновление на практике? Если прежде вам никогда не приходилось выполнять такое обновление, как бы вы оценили его совместимость с вашей базой кода?

По нашему опыту, обновление языка и компилятора — тонкая и сложная задача, даже если ожидается, что новая версия сохранит обратную совместимость. Обновление компилятора почти всегда приводит к незначительным изменениям в его поведении и требует исправления ошибок компиляции, настройки флагов оптимизации или других ранее не выявленных изменений. Как оценить исправность всей кодовой базы, учитывая все потенциальные проблемы обновления компилятора?

Самое сложное обновление компилятора за всю историю Google было проведено в 2006 году. На тот момент мы уже несколько лет имели в штате тысячи инженеров и не обновляли компиляторы около пяти лет. Большинство инженеров не имели опыта смены компилятора, а большая часть кода компилировалась только одной версией компилятора. Обновление стало трудной и утомительной задачей для (в основном) добровольцев, которая в конечном итоге свелась к поиску коротких путей обхода изменений с проблемами адаптации¹. Внедрение изменений происходило болезненно: многие проблемы, согласно закону Хайрама, проникли в кодовую базу и углубили его зависимости от конкретной версии компилятора, сломать которые было очень трудно. Инженеры пошли на риск и применили изменения, четко не представляя последствия и не зная правила Бейонсе и вездесущей системы непрерывной интеграции.

Этот опыт не был чем-то необычным. Инженеры многих компаний могут рассказать похожие истории. Необычность заключается в нашем осознании сложности

¹ В частности, на интерфейсы из стандартной библиотеки C++ нужно было ссылаться с использованием пространства имен `std`, а изменения в оптимизациях для `std::string` отрицательно повлияли на производительность кода, что потребовало искать дополнительные обходные пути.

задачи. Благодаря тому опыту мы стали уделять больше внимания изменениям в технологиях и организации, чтобы потом менее болезненно преодолевать проблемы масштабирования и находить в нем преимущества. Мы стали развивать автоматизацию (чтобы один человек мог сделать больше), согласованность (чтобы ограничить влияние проблем на низком уровне) и обмен опытом (чтобы несколько человек могли сделать больше).

Чем чаще вы вносите изменения в инфраструктуру, тем меньше проблем они вызывают. Мы обнаружили, что в большинстве случаев код, переживший обновление, становится менее хрупким и его легче обновлять в будущем. В экосистеме, где большая часть кода прошла несколько обновлений, он перестает зависеть от нюансов базовой реализации и начинает зависеть от фактических абстракций, гарантированных языком программирования или ОС. Независимо от того, что именно обновляется, первое обновление всегда будет обходиться базе кода значительно дороже, чем последующие, даже с учетом других факторов.

Мы выявили множество факторов, влияющих на гибкость кодовой базы.

Опыт

Мы знаем, как это сделать. Для некоторых языков мы провели сотни обновлений компиляторов на множестве платформ.

Стабильность

Благодаря регулярному обновлению версий нам приходится вносить меньше изменений. Для некоторых языков мы внедряем обновления компиляторов раз в 1–2 недели.

Согласованность

Объем кода, не прошедшего обновление, постоянно уменьшается, опять же благодаря регулярным обновлениям.

Осведомленность

Поскольку обновления происходят достаточно регулярно, мы можем выявить избыточные операции в процессе обновления и попытаться их автоматизировать. Это в значительной степени совпадает со взглядом SRE на рутину¹.

Стратегия

У нас есть процессы и политики, такие как правило Бейонсе, применяя которые мы делаем обновления выполнимыми. Команды поддержки инфраструктуры беспокоятся не о каждом неизвестном случае использования, а только о том, что видно в системе непрерывной интеграции.

Мы осознали не частоту или сложность обновления компилятора, а необходимость обновлений и проводим их с постоянным числом инженеров даже при постоянном

¹ Site Reliability Engineering. Надежность и безотказность, как в Google, глава 5 «Избавляемся от рутины». — *Примеч. пер.*

росте кодовой базы¹. Если бы мы решили, что эта задача слишком дорогостоящая и ее следует избегать в будущем, мы могли бы до сих пор использовать версию компилятора десятилетней давности и из-за упущенных возможностей оптимизации заплатить за вычислительные ресурсы на 25 % больше. А наша центральная инфраструктура могла бы подвергнуться значительным рискам безопасности, поскольку компилятор 2006 года, безусловно, не смягчил бы уязвимости упреждающего выполнения. Стагнация — это вариант, но не самый разумный.

Сдвиг влево

Одна из общих истин, которую мы считаем верной, заключается в том, что выявление проблем на ранних этапах разработки обычно снижает издержки. Рассмотрим временную шкалу разработки некоторой функциональной возможности (рис. 1.2). Процесс разработки движется слева направо — от обсуждения идеи и проектирования к реализации, проверке, тестированию, фиксации и канареечному развертыванию до развертывания в производственной среде. Сдвиг момента выявления проблемы влево на этой временной шкале удешевляет ее устранение.

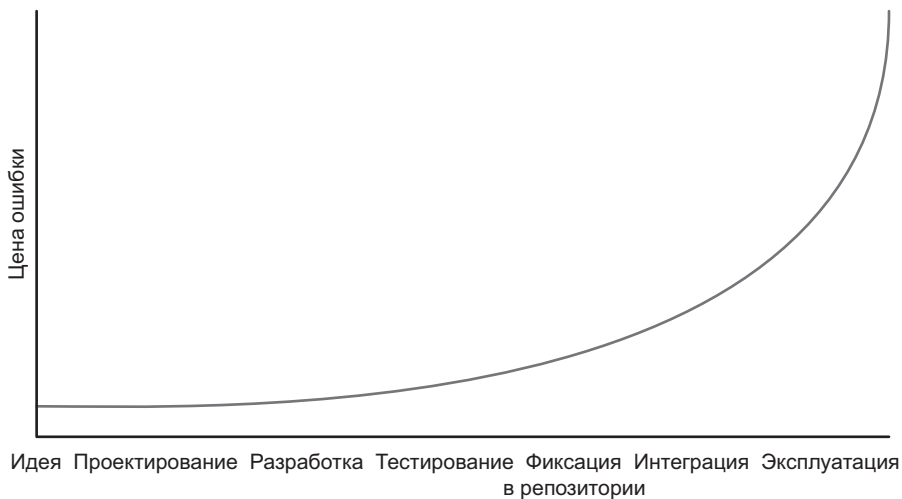


Рис. 1.2. Временная шкала процесса разработки

Это утверждение основано на аргументе, что решение проблем безопасности не должно откладываться до конца процесса разработки, поскольку если она будет об-

¹ По нашему опыту, средний инженер-программист производит относительно постоянное количество строк кода за единицу времени. В фиксированной группе программистов база кода растет линейно — пропорционально количеству человеко-месяцев. Если ваши задачи требуют усилий, объем которых зависит от количества строк кода, переоценить влияние совместной работы будет трудно.

наружена после выпуска продукта в производство, ее исправление обойдется очень дорого. В противном случае для ее устранения могут потребоваться значительные усилия, но конечная цена исправлений будет ниже. Если проблему удастся выявить до того, как разработчик отправит уязвимый код в репозиторий, ее исправление обойдется совсем дешево: разработчик понимает, как работает код, и исправит проблему с меньшими затратами, чем кто-то другой.

В этой книге много раз повторяется один и тот же основной паттерн. Исправление ошибок, обнаруженных статическим анализом и проверкой кода перед его отправкой в репозиторий, обходится намного дешевле устранения проблем, возникающих в ходе эксплуатации кода. Использование инструментов и методов, подтверждающих качество, надежность и безопасность на ранних этапах разработки, является общей целью для наших команд поддержки инфраструктуры. Никакой отдельный процесс или инструмент не может быть идеальным, поэтому мы должны использовать подход эшелонированной защиты, который, надеемся, поможет выявить большинство дефектов в левой части шкалы.

Компромиссы и затраты

Если вы умеете программировать, знаете, как долго будет служить ПО и как его поддерживать с увеличением штата инженеров, производящих и сопровождающих новые функциональные возможности, вам остается только научиться принимать правильные решения. Очевидно, что в программной инженерии, как и везде, хороший выбор ведет к хорошим результатам. Однако на практике эту истину легко упустить из виду. В Google неприемлема фраза: «Потому что я так сказал». В любом нашем обсуждении по любой теме участвуют человек, принимающий решения, и люди, не согласные с этими решениями. Наша цель — согласие, а не единогласие. Это нормально, и я привык слышать фразу: «Я не согласен с вашими метриками/оценками, но я понимаю, как вы могли прийти к такому выводу». В основе этого подхода лежит идея, что всему должна быть причина, а аргументы «просто потому», «потому что я так сказал» или «потому что все так делают» — это признаки плохих решений. Мы всегда должны уметь объяснить и обосновать выбор между затратами двух инженерных решений.

Что мы подразумеваем под затратами? Не только деньги, а сумму издержек, включающую любые или все следующие факторы:

- финансовые затраты (например, деньги);
- затраты ресурсов (например, процессорное время);
- затраты на персонал (например, инженерный труд);
- операционные издержки (например, стоимость принятых мер);
- издержки упущенных возможностей (например, стоимость непринятых мер);
- социальные издержки (например, влияние нашего выбора на общество в целом).

Традиционно легче всего игнорировать социальные издержки. Тем не менее Google и другие крупные технологические компании сегодня могут с уверенностью развешивать продукты с миллиардами пользователей. Масштаб этих продуктов не только открывает возможности, но и усиливает последствия даже небольших проблем в удобстве использования, доступности, справедливости, особенно в отношении маргинализированных групп. ПО проникает во многие аспекты общества и культуры, поэтому с нашей стороны полезно осознавать как достоинства, так и недостатки продукта и его технической стороны при его обсуждении (глава 4).

Затраты также зависят от предубеждений: стремления сохранить текущее положение дел, нежелания что-то потерять и др. Оценивая затраты, мы должны иметь в виду все перечисленные факторы. Здоровье организации — это не только наличие денег в банке, но также осознание ее сотрудниками своей ценности и продуктивности. В творческих и прибыльных областях, таких как программная инженерия, в первую очередь нужно оценивать не финансовые затраты, а затраты на персонал. Увеличение эффективности труда от того, что инженеры довольны, сосредоточены и вовлечены, требует контроля, потому что показатели сосредоточенности и продуктивности настолько изменчивы, что легко представить себе разницу от 10 до 20 %.

Пример: маркеры

Во многих организациях простые маркеры для белой доски считаются ценным товаром. Их распределение жестко контролируются, и они всегда в дефиците. Практически всегда половина маркеров рядом с любой доской уже высохла и непригодна. Вы часто присутствовали на встречах, прерванных из-за отсутствия маркера? Часто теряли ход мыслей, когда маркер заканчивался? Случалось, что все маркеры просто пропадали, например когда их забирала другая команда? А ведь этот товар стоит меньше доллара.

В Google, как правило, есть открытые шкафы, полные канцелярских принадлежностей, включая маркеры. Достаточно просто послать уведомление, чтобы получить десятки маркеров различных цветов. В этом смысле мы пошли на очевидный компромисс: гораздо важнее обеспечить беспрепятственный мозговой штурм, чем разбираться, кому маркер сейчас нужнее.

Мы стремимся открыто и честно взвешивать компромиссы между затратами и выгодой, от канцелярских принадлежностей и льгот для сотрудников до обмена опытом между разработчиками и всеобщей поддержки во всем. Мы часто говорим: «Google — это культура, основанная на данных». На самом деле это упрощение: даже в отсутствие *данных* могут существовать *доказательства, прецеденты и аргументы*. Принятие правильных инженерных решений — это взвешивание всех доступных входных данных и выбор компромиссов. Мы можем основать решение на интуиции или общепринятой практике, но только после применения подходов, основанных на измерениях или оценках истинных затрат.

В конце концов, выбор решения в группе инженеров должен сводиться к одному из двух вариантов:

- мы будем делать так, потому что обязаны (по требованиям законодательства или клиента);
- мы будем делать так, потому что это лучший вариант (как определено кем-то, наделенным правом решения), в чем можно убедиться, основываясь на текущих данных.

Решения не должны обосновываться фразой: «Мы будем делать так, потому что я так сказал»¹.

Основа для принятия решений

Оценивая данные, мы используем два основных сценария:

- Все учитываемые величины измеримы или, по крайней мере, могут быть оценены. Обычно это означает возможность оценки компромиссов между процессорами и сетью, деньгами и оперативной памятью или двумя неделями труда инженеров и экономией N процессоров в вычислительных центрах.
- Некоторые величины незначительны или мы не знаем, как их измерить. Например, иногда «мы не знаем, сколько времени нужно инженерам». Работать с такими величинами иногда сложнее, чем «оценить затраты на разработку плохо спроектированного API» или «оценить влияние выбора продукта на общество».

В первом случае не должно быть недостатка в исходных данных. Любая организация, занимающаяся разработкой ПО, может и должна контролировать текущие затраты на вычислительные ресурсы, трудозатраты инженеров и др. Если вы не хотите публиковать точные суммы финансовых затрат, создайте таблицу для пересчета: какое количество процессоров стоит столько же, сколько стоит данный объем ОЗУ или данная пропускная способность сети.

Имея согласованную таблицу пересчета, каждый инженер сможет провести собственный анализ: «Если я потрачу две недели на преобразование связанного списка в структуру с более высокой производительностью, то использую на 5 Гбайт больше оперативной памяти, но сэкономлю две тысячи процессоров. Стоит ли овчинка выделки?» Ответ на этот вопрос зависит не только от относительной стоимости оперативной памяти и процессоров, но и от затрат на персонал (две недели поддержки инженера-программиста) и стоимости упущенных возможностей (что еще этот инженер мог бы произвести за две недели?).

Выбрать решение во втором случае сложнее. Обсуждая трудноизмеримые величины, мы полагаемся на опыт, лидерство и прецедент и вкладываем средства в исследования, которые помогут количественно оценить то, что трудно поддается количественной

¹ Это не означает, что решения должны приниматься единогласно или широким консенсусом. Кто-то должен взять на себя ответственность за принятие решения. Мы описали сам процесс принятия решений.

оценке (глава 7). Главное, осознавать, что не все измеримо или предсказуемо, и подходить к решениям с осторожностью. Часто трудноизмеримые величины не менее важны, чем измеримые, но сложнее в управлении.

Пример: распределенная сборка

Рассмотрим пример сборки. Согласно ненаучным опросам в Twitter, 60–70 % разработчиков выполняют сборку ПО (даже большого и сложного) на локальном компьютере. Это стало предметом множества шуток, как в комиксе «Compiling» (<https://xkcd.com/303>). Сколько рабочего времени вы теряете в ожидании окончания сборки? Сравните это с затратами на использование чего-то вроде distcc в небольшой группе или с затратами на создание маленькой сборочной фермы для большой группы. Сколько недель/месяцев потребуется, чтобы эти затраты окупились?

Еще в середине 2000-х сборка (проверка и компиляция кода) в Google выполнялась исключительно на локальных компьютерах. Да, были огромные локальные компьютеры (позволяющие собирать Google Maps!), но с ростом кодовой базы время компиляции все росло и росло. Неудивительно, что увеличивались затраты на персонал (из-за потерянного времени), а также на ресурсы (из-за закупки более мощных локальных машин). Затраты на ресурсы были особенно заметны, поскольку большую часть времени высокопроизводительные машины простаивали. Мы посчитали неправильным вкладывать деньги в невостребованные ресурсы.

Так в Google была создана своя система распределенной сборки. Разумеется, разработка этой системы потребовала определенных затрат: инженерам понадобилось время на разработку, изменение привычек и рабочих процессов, освоение новой системы, и, конечно, были задействованы дополнительные вычислительные ресурсы. Но общая экономия того стоила: сборка ускорилась, трудозатраты на разработку окупились, а инвестиции в оборудование были направлены на общую инфраструктуру (подмножество нашего производственного парка), а не на приобретение все более мощных настольных компьютеров. Подробнее о нашем подходе к распределенной сборке в главе 18.

Итак, мы создали новую систему, внедрили ее в производство и ускорили процесс сборки для всех. Можно ли назвать это счастливым концом истории? Не совсем: со временем распределенная сборка стала замедляться, поскольку в граф сборки стали бесконтрольно проникать лишние зависимости. Раньше каждый отдельный инженер страдал от неоптимальной сборки, был заинтересован в ее ускорении и стремился что-то улучшить. Избавив инженеров от проблемы оптимизации процесса сборки, мы создали ситуацию, в которой потребление ресурсов вышло из-под контроля. Это было похоже на парадокс Джевонса¹ (<https://oreil.ly/HL0sl>): потребление ресурса может *увеличиться* в ответ на повышение эффективности его использования.

В целом затраты, связанные с внедрением распределенной системы сборки, намного перевесили расходы, связанные с ее созданием и обслуживанием. Но мы не преду-

¹ https://ru.wikipedia.org/wiki/Парадокс_Джевонса. — Примеч. пер.

смотрели расходов, обусловленных ростом потребления. Забегая вперед, скажу, что мы оказались в ситуации, когда нам пришлось переосмыслить цели и ограничения системы и особенности ее использования, определить оптимальные подходы (небольшое число зависимостей, машинное управление зависимостями) и финансировать создание инструментов для обслуживания новой экосистемы. Даже относительно простой компромисс в виде «мы потратим вот столько на вычислительные ресурсы, чтобы окупить время инженера» имел непредвиденные последствия.

Пример: выбор между временем и масштабированием

Часто темы, связанные со временем и масштабированием, пересекаются и дополняют друг друга. Хорошо масштабируемые стратегии, такие как правило Бейонсе, помогают управлять ситуацией с течением времени. Изменение интерфейса ОС одинаково влияет на все проекты, поэтому мелкие изменения в проектах, вызванные изменениями в ОС, хорошо масштабируются.

Но иногда время и масштабирование вступают в конфликт, и особенно четко это проявляется в базовом вопросе: лучше добавить зависимость или создать (заимствовать) новую ветвь, чтобы удовлетворить локальные потребности?

Этот вопрос может возникнуть на разных уровнях стека, поэтому обычно индивидуальное решение для узкой предметной области бывает эффективнее более общего решения. Ветвление действующего решения и его подгонка под свои нужды упрощает добавление новых возможностей и дает больше уверенности при его оптимизации, будь то микросервис, кеш в памяти, подпрограмма сжатия или что-то еще в программной экосистеме. При этом вы получаете контроль: изменения в базовых зависимостях не будут определены другой командой или сторонним поставщиком, и вы сами решите, как и когда реагировать на возникающую необходимость перемен.

С другой стороны, если вместо повторного использования каждый разработчик будет создавать новые версии всего, что нужно проекту, пострадают масштабируемость и устойчивость. Реагирование на проблему безопасности в базовой библиотеке перестанет быть вопросом обновления отдельной зависимости: оно потребует определить все уязвимые версии этой зависимости и их пользователей.

И снова программная инженерия не дает универсального ответа, что выбрать. Если проект имеет небольшую продолжительность жизни или ветвление имеет ограниченный объем, применение ветвления уместно. Но избегайте ветвления интерфейсов, которые могут жить дольше проекта (структур данных, форматов сериализации, сетевых протоколов). Согласованность имеет большое значение, но она тоже имеет цену, и часто выгоднее создавать свои решения, если делать это осторожно.

Пересмотр решений, совершение ошибок

Одним из недооцененных преимуществ приверженности культуре, основанной на данных, является возможность признавать ошибки. Представьте решение,

основанное на имеющихся данных (вероятно, достоверных) и нескольких предположениях, неявно выведенных из этих данных. По мере поступления новых данных, изменения контекста или исключения предположений может выясниться, что решение было ошибочным или потеряло актуальность. Такая ситуация характерна для организаций, работающих много лет: со временем меняются не только технические зависимости и программные системы, но и данные, используемые для принятия решений.

Мы твердо верим в решения, основанные на данных, но понимаем, что данные со временем меняются. Поэтому время от времени в течение срока службы системы решения должны пересматриваться. Для долгоживущих проектов важно иметь возможность менять направление после принятия первоначального решения, и лица, принимающие решения, должны иметь право на ошибки. Вопреки стереотипам лидеры, способные признавать ошибки, пользуются большим уважением.

Опирайтесь на доказательства, но имейте в виду, что даже то, что невозможно измерить, может иметь ценность. Если вы лидер, придерживайтесь суждения, что нет ничего маловажного. Подробнее о лидерстве в главах 5 и 6.

Программная инженерия и программирование

Узнав, по каким признакам мы различаем программную инженерию и программирование, вы можете спросить, есть ли у нас какие-то внутренние суждения об их ценности. Можно ли сказать, что программирование хуже программной инженерии? Или утверждать, что проект, который, как ожидается, будет развиваться сотнями людей в течение десятилетий, ценнее проекта, который просуществует месяц и будет создан двумя людьми?

Конечно нет. Мы не считаем, что программная инженерия лучше программирования. Это две разные предметные области с разными ограничениями, ценностями и методами. Мы осознаем, что некоторые инструменты хороши в одной области и непригодны в другой. Едва ли имеет смысл внедрять интеграционные тесты (глава 14) и непрерывное развертывание (*continuous deployment*, глава 24) в проект, который просуществует несколько дней. Точно так же наши достижения в области семантического управления версиями (*SemVer*) и управления зависимостями в программной инженерии (глава 21) не применимы к краткосрочным проектам, в которых можно смело использовать все, что доступно.

Научитесь различать схожие термины «программирование» и «программная инженерия». Большая часть их различий заключается в подходах к управлению кодом, влиянии времени на масштабирование и особенностях принятия решений. Программирование — это непосредственный акт создания кода. Программная инженерия — это набор стратегий, методов и инструментов, помогающих сохранить полезность кода в течение всего времени его использования и обеспечить возможность совместной работы в команде.

Заключение

В этой книге обсуждаются политики как для организаций, так и для отдельных программистов, оценка и совершенствование методов, а также инструменты и технологии, используемые в ПО, удобном для сопровождения. Компания Google немало потрудилась, чтобы получить устойчивые кодовую базу и культуру. Мы не считаем свой подход единственно верным, но он наглядно показывает, чего можно добиться. Надеемся, что он послужит полезной основой для размышлений об общей проблеме: как правильно поддерживать код, чтобы он работал столько времени, сколько понадобится.

Итоги

- «Программная инженерия» более широкое понятие, чем «программирование». Программирование — это создание кода. Программная инженерия добавляет к этому понятию обслуживание кода для увеличения срока его использования.
- Продолжительность жизни короткоживущего и долгоживущего кода может отличаться как минимум в 100 000 раз. Было бы неправильно полагать, что одни и те же практики применимы на обоих концах спектра.
- ПО устойчиво, если в течение ожидаемого срока его службы мы сохраняем способность реагировать на изменения в зависимостях, технологиях или требованиях к продукту. Мы можем ничего не менять, но должны быть готовы к изменениям.
- Закон Хайрама: при достаточно большом количестве пользователей API не имеет значения, что вы обещаете в контракте: любое наблюдаемое поведение системы будет зависеть от чьих-то действий.
- Каждая задача, которую организация должна выполнять снова и снова, должна масштабироваться (линейно или еще лучше) в отношении участия человека. Политики — прекрасный инструмент для масштабирования процесса.
- Проблемы, обусловленные неэффективностью процесса, и некоторые другие имеют свойство накапливаться медленно. Они с легкостью могут превратиться в метафорическую сварившуюся лягушку.
- Опыт окупается особенно быстро в сочетании с экономией за счет масштабирования.
- «Потому что я так сказал» — это плохое обоснование решения.
- Принятие решений на основе данных — хорошее правило, но на практике большинство решений основывается на сочетании данных, предположений, прецедентов и аргументов. Лучше всего, когда объективные данные составляют большую часть основы для принятия решений, но не всегда есть возможность опираться *только* на них.
- Принятие решений на основе данных подразумевает необходимость изменения направления, если данные изменились или предположения были опровергнуты. Ошибки и пересмотр планов неизбежны.

ЧАСТЬ II

Культура

Успешная работа в команде

Автор: Брайан Фитцпатрик

Редактор: Риона Макнамара

В этой главе рассмотрены культурные и социальные аспекты программной инженерии в Google, и сначала мы сосредоточимся на изменяемой величине, которую вы можете контролировать, — на вас самих.

Люди по своей природе несовершенны. Мы любим говорить, что люди — это комплекс периодически возникающих ошибок. Но прежде чем искать ошибки в своих коллегах, научитесь распознавать ошибки в себе. Подумайте о своих реакциях, поведении и отношениях и получите более или менее полное представление о том, как стать более эффективным и успешным инженером-программистом, который тратит меньше энергии на решение проблем с людьми и больше — на создание отличного кода.

Главная идея этой главы в утверждении, что разработка ПО — это командная работа. Чтобы добиться успеха в команде инженеров или другом творческом коллективе, нужно организовать свое поведение в соответствии с основными принципами: смирением, уважением и доверием.

Но не будем забегать вперед и начнем с наблюдения за обычным поведением инженеров-программистов.

Помоги мне скрыть мой код

За последние двадцать лет мы с моим коллегой Беном¹ не раз выступали на разных конференциях по программированию. В 2006 году мы в Google запустили услугу хостинга для проектов с открытым исходным кодом (OSS, open source software) (в настоящее время не поддерживается) и получали много вопросов и просьб. Но примерно в середине 2008 года мы начали замечать определенную направленность запросов:

«Не могли бы вы добавить в поддержку Subversion в Google Code возможность скрывать определенные ветки?»

«Не могли бы вы добавить возможность создать проект с открытым исходным кодом, который изначально закрыт, а потом открывается, когда будет готов?»

¹ Бен Коллинз-Сассмэн, один из авторов этой книги.

«Привет, я хочу переписать весь свой код с нуля, не могли бы вы стереть всю историю?»

Заметили, что объединяет эти просьбы?

Ответ: *неуверенность*. Люди боятся, что другие увидят и оценят их работу. С одной стороны, неуверенность — это часть человеческой природы, никто не любит, когда его критикуют, особенно за то, что еще не закончено. Осознание этого помогло нам заметить более общую тенденцию в разработке ПО: неуверенность на самом деле является признаком более серьезной проблемы.

Миф о гениальности

Многие неосознанно творят себе кумира и поклоняются ему. Для инженеров-программистов это могут быть: Линус Торвалдс, Гвидо Ван Россум, Билл Гейтс — все они выдающиеся личности, изменившие мир. Линус в одиночку написал Linux, верно?

На самом деле Линус написал всего лишь начальную версию Unix-подобного ядра для проверки идей и опубликовал его код в списке рассылки. Безусловно, это большое и впечатляющее достижение, но оно было только началом. Современное ядро Linux в сотни раз больше той начальной версии и разрабатывается тысячами умных людей. Настоящее достижение Линуса в том, что он сумел взять на себя руководство этими людьми и скоординировать их работу. Linux — блестящий результат, но не его первоначальной идеи, а коллективного труда сообщества. (И ОС Unix была написана не только Кеном Томпсоном и Деннисом Ритчи, а группой умных людей из Bell Labs.)

Аналогично: разве Гвидо Ван Россум один создал весь Python? Да, он написал первую версию. Но в разработке последующих версий, реализации новых идей и исправлении ошибок принимали участие сотни других людей. Стив Джобс руководил командой, которая создавала Macintosh. И хотя Билл Гейтс известен тем, что написал интерпретатор BASIC для первых домашних компьютеров, его самым большим достижением стало создание успешной компании вокруг MS-DOS. Все они стали лидерами и символами коллективных достижений своих сообществ. Миф о гениальности — это общая тенденция, следуя которой мы, люди, приписываем успех команды одному человеку.

А что насчет Майкла Джордана?

Та же история. Мы идеализировали его, но он не сам выиграл каждый баскетбольный матч. Его истинная гениальность в том, как он взаимодействовал с командой. Тренер Фил Джексон был чрезвычайно умен, и его приемы стали легендарными. Он понимал, что один игрок никогда не выигрывает чемпионат, и поэтому собрал целую «команду мечты» вокруг Майкла. Эта команда работала как исправный механизм — играла столь же впечатляюще, как и сам Майкл.

Итак, почему мы постоянно стремимся найти кумира? Покупаем товары, одобренные знаменитостями, хотим купить платье, как у Мишель Обамы, или обувь, как у Майкла Джордана?

Тяга к известности — вот одна из причин. У людей есть естественный инстинкт находить лидеров и образцы для подражания, преклоняться перед ними и пытаться подражать им. Нам всем нужны герои для вдохновения, и в мире программирования тоже они есть. Феномен «технократической знаменитости» почти перетек в мифологию. Мы все хотим написать что-то, что изменит мир, как Linux, или спроектировать еще один потрясающий язык программирования.

В глубине души многие инженеры хотят, чтобы их считали гениями. Эта фантазия выглядит примерно так:

- у вас возникает потрясающая новая идея;
- вы запираетесь в своем убежище на несколько недель или месяцев, создавая совершенное воплощение этой идеи;
- затем «выпускаете» в мир готовое ПО и шокируете всех своей гениальностью;
- ваши коллеги преклоняются перед вашим умом;
- люди выстраиваются в очередь, чтобы заполучить ваше ПО;
- слава и удача преследуют вас.

А теперь остановимся и посмотрим, что мы имеем в реальности. Вы, скорее всего, не гений.

Не обижайтесь! Конечно, мы верим, что вы очень умный человек. Но понимаете ли вы, насколько редки настоящие гении? Конечно, вы пишете код, и это сложный навык. Но даже если вы гений, этого для успеха недостаточно. Гении, как и все, делают ошибки, а блестящие идеи и редкие навыки программирования не гарантируют, что ваш софт станет хитом. Хуже того, вы можете обнаружить, что способны решать только аналитические проблемы, но не *человеческие*. Быть гением не означает быть «не от мира сего»: любой человек — гений он или нет — с плохими социальными навыками, как правило, является плохим партнером по команде. Для работы в Google (и в большинстве других компаний!) не требуется иметь гениальный интеллект, но точно требуется обладать минимальным уровнем социальных навыков. Взлет или падение вашей карьеры, особенно в такой компании, как Google, во многом зависит от того, насколько хорошо вы умеете сотрудничать с другими людьми.

Оказывается, этот миф о гениальности — еще одно проявление нашей неуверенности. Многие программисты боятся выкладывать на всеобщее обозрение свою работу, которую только начали, потому что тогда коллеги увидят их ошибки и поймут, что автор кода не гений.

Вот что однажды сказал мой друг:

«Я знаю, что КРАЙНЕ не уверен в людях, и опасаясь показывать им что-то, что еще не закончено, как будто они осудят меня или подумают, что я идиот».

Это чрезвычайно распространенное чувство среди программистов, и естественной реакцией на него становится желание спрятаться в своем убежище, чтобы работать, работать, работать, а затем полировать, полировать, полировать и пребывать в уве-

ренности, что никто не увидит глупых ошибок и вы обнаружите шедевр, когда он будет закончен. Спрятать код, пока он не станет идеальным.

Другой распространенный мотив прятать свою работу — это страх, что кто-то другой сможет понять вашу идею и воплотить ее раньше вас. Сохраняя в секрете, вы контролируете идею.

Мы знаем, о чем вы сейчас могли подумать: ну и что? Разве плохо, если люди будут работать так, как хотят?

На самом деле — да, плохо. Мы утверждаем, что так работать — это неправильно, и это *очень важно*. И вот почему.

Соккрытие вредно

Работая все время в одиночку, вы увеличиваете риск ненужных неудач и мешаете росту своего потенциала. Даже при том что разработка ПО — это интеллектуальная работа, требующая глубокой концентрации и уединения, вы не должны забывать о ценности (и необходимости!) совместной работы и анализа.

Подумайте сами: как, работая в одиночку, вы сможете понять, что выбрали верный путь?

Представьте, что вы увлекаетесь проектированием велосипедов и однажды вам в голову пришла блестящая идея совершенно нового механизма переключения передач. Вы заказали детали и закрылись на несколько недель в гараже, пытаетесь создать прототип. Когда ваш сосед — также конструктор-энтузиаст — спросил вас, почему вы избегаете встреч, вы решили не рассказывать ему о своей идее, пока она не воплотится. Проходит еще несколько месяцев; вы не можете заставить прототип правильно работать и не спрашиваете совета у друзей.

И вот в один прекрасный день сосед показывает вам свой велосипед с радикально новым механизмом переключения передач. Оказывается, он конструировал что-то похожее на ваше изобретение, но с помощью друзей из магазина велосипедов. Вы испытываете удар по самолюбию и показываете ему свой результат. Он быстро находит простые недостатки вашего дизайна, которые можно было исправить в первую неделю работы. Из этой истории можно извлечь несколько поучительных уроков.

Раннее выявление

Если вы скрываете свою прекрасную идею от мира и отказываетесь показывать что-либо кому-либо, пока реализация не будет отшлифована, вы фактически пускаетесь в авантюру. На ранней стадии разработки легко допустить фундаментальные ошибки. Вы рискуете заново изобрести колесо¹. А также лишаетесь преимуществ совместной

¹ В прямом смысле, если вы конструируете велосипед.

работы: обратите внимание, насколько быстрее продвигался сосед, сотрудничая с друзьями. Люди пробуют воду, прежде чем нырнуть: убедитесь, что движетесь в правильном направлении и ваш проект не был создан раньше. Шансы допустить ошибку в начале пути высоки. И чем больше отзывов вы получаете на ранних этапах, тем меньше рискуете¹. Вспомните проверенную временем мантру: «Ошибки должны выявляться как можно раньше, как можно быстрее и как можно чаще».

Раскрытие своей идеи на раннем этапе ее реализации не только предотвращает личные ошибки и помогает проверить идею — оно помогает укрепить то, что мы называем фактором автобуса.

Фактор автобуса

Фактор автобуса определяет, сколько участников проекта должно попасть под автобус, чтобы проект провалился.

Насколько рассеяны знания и опыт между участниками проекта? Если вы единственный, кто понимает, как работает прототип, вам должна быть обеспечена особая безопасность — если вас собьет автобус, проект прогорит. Но, взяв кого-нибудь в напарники, вы удвоите фактор автобуса. А если с вами будет работать небольшая команда специалистов, вместе создающих и разрабатывающих прототипы, ситуация улучшится еще больше — проект будет продолжен после ухода одного члена команды. Сотрудники могут жениться, уехать, уйти из компании или взять отпуск по уходу за родственником. *Даже простое наличие* хорошей документации и одного-двух ведущих специалистов в каждой области поможет обеспечить успех проекта в будущем и увеличит фактор автобуса проекта. Надеемся, что большинство инженеров понимают, что лучше быть частью удачного проекта, чем критической частью неудачного.

Помимо фактора автобуса существует проблема общего темпа развития. Легко забыть, что работа в одиночку — это сложная задача, которая решается гораздо медленнее, чем хотелось бы. Как много вы узнаете, работая в одиночку? Как быстро вы движетесь к цели? Google и Stack Overflow являются отличными источниками мнений и информации, но они не могут заменить реальный человеческий опыт. Работа с другими людьми напрямую увеличивает коллективные знания. Когда вы остановились из-за абсурдной проблемы, сколько времени вы потратите, чтобы вытащить себя из ямы? Подумайте, насколько изменится ваш опыт, если пара коллег будет смотреть вам через плечо и сразу указывать на ошибки и варианты их устранения. Именно поэтому в компаниях, занимающихся разработкой ПО, члены одной команды часто сидят вместе (или даже занимаются парным программированием). Программирование — сложная наука. Программная инженерия — еще сложнее. Вам нужна эта вторая пара глаз.

¹ Отметим, что иногда на ранних этапах процесса опасно получать слишком много отзывов, если вы не уверены в выборе направления или цели.

КЕЙС: ИНЖЕНЕРЫ И ОФИСЫ

Двадцать пять лет назад считалось, что единственный способ работать над решением задачи, не отвлекаясь на посторонние дела, — иметь отдельный кабинет с закрытой дверью.

Думаю, большинству инженеров¹ не только не нужно находиться в отдельном кабинете, но и вредно. Современное ПО создается командами, а не одиночками, и постоянный контакт с другими членами команды ценнее соединения с интернетом. В отдельном кабинете вы будете иметь массу времени для обдумывания своих решений, но, двигаясь в неправильном направлении, вы потратите это время впустую.

К сожалению, многие современные технологические компании (включая Google) качнули маятник в другую крайность. Зайдите в их офисы — и найдете сотни инженеров, сидящих вместе в огромных залах. В настоящее время ведутся жаркие споры о целесообразности опен-спейсов и, как следствие, растет неприятие их. Там, где самый малозначительный разговор становится публичным, люди перестают общаться, чтобы не раздражать соседей. Это так же плохо, как отдельные кабинеты!

Мы считаем, что лучшее решение — золотая середина. Сгруппируйте команды по четыре-восемь человек в небольших комнатах (или больших офисах), чтобы упростить случайное общение между ними.

Конечно, инженерам нужна возможность фильтровать шум. И они придумывали способы сообщить, что их не следует прерывать. Например, использовали протокол голосового прерывания — фразу «Прервись, Мэри», где Мэри — имя человека, с которым нужно поговорить. По возможности Мэри поворачивалась лицом и слушала. Если она была занята, то отвечала: «Не сейчас».

Также они ставили на свои мониторы флажки или игрушки, чтобы показать, что прерывать их можно только в случае крайней необходимости. В некоторых командах инженерам выдавались наушники с шумоподавлением. К слову, во многих компаниях сам факт ношения наушников обычно расценивается как сигнал, означающий «не беспокойте меня, если это не очень важно». Многие инженеры, приступая к написанию кода, переходят в режим работы «в наушниках», что может быть полезно, если период изоляции длится недолго и не создает в офисе глухих стен.

Не поймите нас неправильно — мы все еще считаем, что инженеры должны иметь возможность сосредоточиться на написании кода, но мы думаем, что им также необходимо беспрепятственное общение с командой. Если коллега не может просто задать вам вопрос — это проблема, а найти правильный баланс — это искусство.

Темп развития

Еще одна аналогия. Вспомните, как вы используете компилятор. Тратите несколько дней, чтобы написать 10 000 строк кода, а затем нажимаете кнопку «компилировать». Так? Конечно нет. Представьте неприятные последствия такого подхода. Программисты увереннее работают, когда пишут код небольшими фрагментами, используя

¹ Понимаю, что интровертам, вероятно, нужно больше тишины и одиночества и спокойная обстановка; возможность выделить им отдельный кабинет могла бы способствовать более продуктивной их работе.

короткие циклы с обратной связью: написали новую функцию — скомпилировали, добавили тест — скомпилировали, выполнили рефакторинг участка кода — скомпилировали. Это позволяет быстро выявлять и исправлять опечатки и ошибки в коде. Компилятор нужен нам для проверки каждого маленького шага. Некоторые среды программирования могут даже компилировать код сразу после ввода. Современная философия DevOps, направленная на повышение продуктивности, четко говорит о целях: получать отзывы как можно раньше, тестировать как можно раньше и начинать думать о безопасности в продакшене как можно раньше. Все это связано с идеей «сдвига влево» в рабочем процессе разработчика и удешевлении устранения ошибки.

Короткий цикл обратной связи необходим не только на уровне кода, но и на уровне всего проекта. Амбициозные проекты развиваются быстро и должны адаптироваться к меняющимся условиям. Они могут сталкиваться с непредсказуемыми архитектурными ограничениями, политическими препятствиями или техническим несоответствием. Требования могут меняться неожиданно. Как обеспечить быструю обратную связь, чтобы вовремя узнавать, когда следует менять планы или решения? Ответ: работать в команде. Большинству инженеров известна истина: «Чем больше глаз, тем заметнее ошибки». Но ее можно перефразировать иначе: «Чем больше глаз, тем успешнее проект справляется с вызовами». Люди, работающие в затворничестве, рискуют однажды обнаружить, что мир изменился, и несмотря на полноту и правильность первоначальных представлений, проект стал неактуальным.

Проще говоря, не прячьтесь!

Итак, «сокрытие» сводится к следующему: работа в одиночку рискованнее, чем работа в команде. Вас не должно беспокоить, что кто-то украдет вашу идею или подумает, что вы не умны, гораздо хуже, если вы потратите время, двигаясь не туда.

Не становитесь частью печальной статистики.

Весь секрет в командной работе

Итак, давайте оглянемся назад и соберем воедино все рассмотренные идеи.

В программировании ремесленники-одиночки встречаются крайне редко, и никто из них не достигает вершин самостоятельно: их достижения, меняющие мир, почти всегда являются результатом искры вдохновения, сопровождаемой упорной командной работой.

Великая команда блестяще использует своих суперзвезд, и целое всегда больше, чем сумма его частей. Но создать команду из суперзвезд очень сложно.

Сформулируем эту идею проще: *программирование — это командная работа.*

Эта идея прямо противоречит представлению о гениальном программисте, знакомом многим из нас. Не пытайтесь изменить мир или вызвать восхищение у миллионов пользователей, пряча и шлифуя свое секретное изобретение. Работайте с другими

людьми. Поделитесь своим видением. Разделите труд. Учитесь у других. Создайте блестящую команду.

Подумайте, сколько есть примеров широко используемого ПО, написанного одним человеком? (Кто-то вспомнит «LaTeX», но вряд ли этот продукт «широко используется», если только вы не считаете, что число людей, пишущих научные статьи, является статистически значимой частью всех пользователей компьютеров!)

Высокоэффективные команды — это золотая жила и верный ключ к успеху. Приложите все силы, чтобы создать такую команду.

Три столпа социального общения

Итак, если командная работа — лучший путь к созданию блестящего ПО, как создать (или найти) хорошую команду?

Чтобы достичь nirваны совместной работы, сначала запомните «три столпа» социальных навыков. Они не только смазывают колеса взаимопонимания, но и являются основой для здорового взаимодействия и сотрудничества.

Столп 1: смирение

Вы не центр Вселенной (как и ваш код!). Вы не всезнающий и не безгрешный. Вы открыты для самосовершенствования.

Столп 2: уважение

Вы искренне заботитесь о тех, с кем работаете, относитесь к ним по-доброму и цените их способности и достижения.

Столп 3: доверие

Вы верите, что другие тоже компетентны, действуют правильно и вы сможете передать им руль, если понадобится¹.

Если проанализировать первопричины практически любого социального конфликта, в конечном итоге можно проследить его до отсутствия смирения, уважения и (или) доверия. Поначалу это утверждение может показаться неправдоподобным, но убедитесь сами — поразмышляйте над какой-нибудь неприятной или неудобной ситуацией в вашей жизни. Все ли участники конфликта были смиренны? Действительно ли они относились друг к другу с уважением? Имело ли место взаимное доверие?

Почему эти столпы так важны?

Начиная читать эту главу, вы, наверное, не планировали записываться в группу поддержки. Сочувствуем. Социальные проблемы трудно решать: люди беспорядочны, непредсказуемы и часто раздражают своим общением. Вместо того чтобы вкладывать энергию в анализ социальных проблем и искать стратегические решения, гораздо

¹ Это невероятно сложно, если в прошлом вы погорели, передав управление некомпетентным людям.

проще пустить все на самотек. Работать с предсказуемым компилятором намного проще, не так ли? Зачем вообще погружаться в социальную сферу?

Вот цитата из известной лекции Ричарда Хэмминга (http://bit.ly/hamming_paper):

«Взяв за правило рассказывать секретарям анекдоты и проявлять немного дружелюбия, я получал от них очень ценную помощь. Например, однажды по какой-то дурацкой причине все копирующие службы в Мюррей Хилл оказались заняты. Не знаю почему. Надо было что-то делать. Моя секретарша позвонила кому-то в Холмдел, прыгнула в служебный автомобиль, провела час в дороге, скопировала материал и вернулась. Это стало мне наградой за то, что я не ленился приободрять ее, шутил и вообще был дружелюбен. Мне это ничего не стоило, но позже такое мое поведение окупилось. Столкнувшись с необходимостью использовать систему и изучая возможные способы заставить систему делать вашу работу, вы учитесь менять систему под себя».

Вывод прост: не стоит недооценивать социальные игры. Это не обман или манипулирование, а выстраивание отношений, помогающих достичь цели. Отношения всегда живут дольше, чем проекты. При хорошем отношении коллеги будут готовы пройти лишнюю милю, если вам это потребуется.

Смирение, уважение и доверие на практике

Все эти речи о смирении, уважении и доверии звучат как проповедь. Давайте спустимся на грешную землю и подумаем, как применить эти идеи в жизни. Для начала рассмотрим список конкретных моделей поведения и примеры. Многие из этих моделей покажутся очевидными, но вскоре вы заметите, как часто вы (и ваши коллеги) не следуете им — мы заметили это в себе!

Усмирите свое эго

Если вы замечаете, что кому-то не хватает смирения, проще всего попросить этого человека быть скромнее. Никто не горит желанием работать с тем, кто постоянно ведет себя как самый важный человек в комнате. Даже если вы уверены в своем превосходстве, не показывайте этого. Подумайте, так ли необходимо, чтобы последнее слово всегда оставалось за вами? Так ли важно комментировать каждую деталь в предложении или обсуждении? Вы знаете людей, которые так себя ведут?

Скромным быть важно, но это не значит, что можно показывать слабину — в уверенности в себе нет ничего плохого. Просто не стройте из себя всезнайку. Думайте о «коллективном» эго — старайтесь поддерживать командный дух и коллективную гордость. Например, Apache Software Foundation имеет долгую историю создания сообществ вокруг программных проектов. Эти сообщества отличает невероятно сильное самосознание: они отвергают людей, озабоченных саморекламой.

Эго имеет множество проявлений и часто мешает работать. Вот еще одна замечательная история из лекции Хэмминга, которая прекрасно иллюстрирует проявление личного эго (выделение наше):

«Джон Тьюки почти всегда одевался очень небрежно. Когда он приходил в офис, руководители не сразу понимали, что перед ними первоклассный специалист, к мнению которого стоит прислушаться. Джон тратил много времени и сил на преодоление такого неприятия! Я не говорю, что вы должны соответствовать окружению, но обращаю ваше внимание, что “видимость соответствия здорово помогает”. Если вы будете утверждать свое эго, заявляя: “Это будет по-моему”, то будете расплачиваться за это на протяжении всей своей карьеры. И в других сферах жизни это будет приводить к огромному числу ненужных неприятностей. <...> Столкнувшись с необходимостью использовать систему и изучая возможные способы заставить систему делать вашу работу, вы учитесь менять систему под себя. *В противном случае вам придется бороться с ней постоянно и всю свою жизнь вести маленькую необъявленную войну*».

Учитесь критиковать и принимать критику

Несколько лет назад Джо устроился на новую работу программистом. Спустя неделю он начал копать в кодовой базе. Его заинтересовало, какой вклад в код внес каждый член команды. Он стал отправлять им простые обзоры кода по электронной почте, вежливо спрашивая о проектных решениях или указывая места, где можно улучшить логику. Через пару недель его вызвали в кабинет директора. «В чем проблема? — спросил Джо. — Я сделал что-то неправильно?» «У нас было много жалоб на ваше поведение, — сказал директор. — По-видимому, вы были очень резки по отношению к своим товарищам по команде, критикуя их налево и направо. Они расстроены. Вам следует быть вежливее». Джо был совершенно сбит с толку. Конечно, он думал, что коллеги оценят и с радостью примут его обзоры кода. Однако Джо должен был проявить большую осторожность и уважительность к членам команды и использовать более тонкие средства для внедрения код-ревью в культуру — возможно, предварительно обсудить идеи с командой и предложить попробовать новый подход в течение нескольких недель.

В профессиональной среде программной инженерии критика почти никогда не бывает личной — обычно она просто является частью процесса улучшения проекта. Хитрость в том, чтобы убедиться, что вы (и те, кто вас окружает) понимаете разницу между конструктивной критикой творческих результатов и откровенным оскорблением чувств. Последнее — мелочно и недействительно. Первое может (и должно!) быть полезным, если содержит рекомендации по улучшению. И самое главное, конструктивная критика проникнута уважением: человек, выступающий с такой критикой, искренне заботится о других и хочет, чтобы они совершенствовались себя или свой труд. Научитесь уважать своих коллег и вежливо подавать конструктивную критику. Если вы действительно уважаете кого-то, у вас будет мотивация выбирать тактичные и понятные формулировки — навык, приобретаемый с опытом (глава 9).

С другой стороны, нужно учиться принимать критику. Не только скромнее оценивать свои навыки, но и верить, что в глубине души коллеги действуют в ваших интересах (и в интересах проекта!) и на самом деле не считают вас умственно отсталым.

Программирование — это обычный навык. Если коллега указал вам пути улучшения мастерства, разве вы воспримете это как оскорбление и принижение вашего достоинства? Надеемся, что нет. Точно так же ваша самооценка не должна быть связана с кодом, который вы пишете, или любым творческим проектом, над которым вы работаете. Вы — это не ваш код: повторяйте это снова и снова. Вы — это не то, что вы делаете. Поверьте в это сами и заставьте коллег считать так же.

Например, если у вас есть неуверенный в себе коллега, вам точно не следует говорить ему: «Дружище, ты совершенно неправильно понял поток управления в этом методе. Ты должен использовать стандартный шаблон *xyzyzy*, как и все остальные». Так поступать не следует: говорить человеку, что он «не прав» (как если бы мир был черно-белым), требовать изменений и заставлять его чувствовать себя хуже других. Коллега вне всяких сомнений обидится, и его реакция наверняка окажется чрезмерно эмоциональной.

То же самое можно сказать более тактично: «Знаешь, меня смущает поток управления в этой части. Интересно, сможет ли применение шаблона *xyzyzy* сделать его понятнее и проще в поддержке?» Обратите внимание, как в этом случае проявляется смирение, как акцент ставится на себе, а не на собеседнике. Вы утверждаете, что не коллега ошибся, а вы не поняли код. Этой фразой вы предлагаете лишь прояснить ситуацию для вас, что, возможно, способствует достижению цели устойчивого развития проекта в долгосрочной перспективе. Вы также ничего не требуете — даете коллеге возможность мирно отклонить предложение. Обсуждение сосредоточено на коде, а не на чьих-то умственных способностях или навыках.

Получив отрицательный результат, повторите попытку

В деловом мире существует городская легенда о менеджере, который допустил ошибку и потерял 10 миллионов долларов. На следующий день он уныло входит в офис и начинает собирать свои вещи, а когда ему звонят с неизбежной фразой: «Генеральный директор хочет видеть вас в своем кабинете», он тащится в кабинет директора и тихо кладет заявление на стол.

«Что это?» — спрашивает директор.

«Мое заявление об уходе, — говорит менеджер. — Полагаю, вы вызвали меня, чтобы уволить».

«Уволить? Вас? — отвечает директор недоверчиво. — Зачем мне вас увольнять? Я только что заплатил 10 миллионов долларов за ваше обучение!»¹

Конечно, это весьма необычная история, но генеральный директор в ней понимает, что увольнение менеджера не возместит потерю 10 миллионов долларов, а только усугубит ситуацию потерей ценного работника, который, в чем можно не сомневаться, не повторит своей ошибки.

¹ В интернете можно найти десятки вариантов этой легенды, приписываемых разным знаменитым менеджерам.

Один из наших любимых девизов в Google: «Неудача — тоже вариант». Как известно, «не ошибается тот, кто ничего не делает» и «кто не рискует, тот не пьет шампанское». Неудача — это прекрасная возможность учиться и совершенствоваться¹. На самом деле Томас Эдисон часто повторял: «Если я попробую 10 000 способов, которые не сработают, я не потерплю неудачу. Я не отчаюсь, потому что каждая неудачная попытка — это еще один шаг вперед».

В Google X — подразделении, занимающемся прорывными технологиями, такими как беспилотные автомобили и ретрансляторы для доступа в интернет на воздушных шарах, — неудачи намеренно встроены в систему стимулирования. Люди придумывают диковинные идеи, а их коллегам активно рекомендуется как можно быстрее доказать невозможность их воплощения. Люди получают вознаграждение (и даже соревнуются) за опровержение определенного количества идей или одной идеи в течение определенного времени. И только когда никому не удалось опровергнуть идею, она переходит на первую стадию создания прототипа.

Культура подробного анализа причин неудачи

Извлечению уроков из ошибок главным образом способствует документирование неудач с помощью анализа основных причин и описания «результатов вскрытия» проблем (как это называют в Google и многих других компаниях). Будьте особенно внимательны: документ с описанием результатов вскрытия не должен быть бесполезным списком оправданий — он должен содержать объяснение причин проблем и план дальнейших действий и быть доступным всем, чтобы команда действительно следовала предложенным изменениям. Правильное документирование ошибок помогает сохранить понимание того, что произошло, и избежать повторения истории. Не стирайте следы — освещайте их, как взлетно-посадочную полосу, для тех, кто следует за вами!

Хорошее описание результатов вскрытия должно включать:

- краткое описание события;
- историю развития события от его обнаружения до расследования и получения выводов;
- основную причину события;
- оценку влияния события и ущерба;
- перечень действий и мероприятий, предпринятых для немедленного исправления проблемы;
- перечень действий и мероприятий, направленных на предотвращение повторения события;
- уроки и выводы.

¹ Если вы снова и снова продолжаете делать одно и то же и раз за разом терпите неудачу, то это не неудача, а некомпетентность.

Учитесь терпению

Несколько лет назад я работал над инструментом для преобразования репозитория CVS в Subversion (а позже в Git). Из-за капризов CVS я все время наткнулся на странные ошибки. Поскольку мой давний друг и коллега Карл хорошо знал CVS, мы решили поработать вместе и исправить эти ошибки.

Когда мы начали парное программирование, возникла проблема: я предпочитал восходящий способ и погружался в самый низ, быстро выискивая варианты пути наверх, минуя детали. Карл, напротив, отдавал предпочтение нисходящему способу: он стремился получить полную картину о ландшафте и погрузиться в реализацию почти каждого метода в стеке вызовов, прежде чем приступить к устранению ошибки. Это приводило к межличностным конфликтам, разногласиям, а иногда и ожесточенным спорам. Дошло до того, что мы просто не могли программировать вместе — это было слишком сложно для нас обоих.

Тем не менее мы давно питали безграничное доверие и уважение друг к другу. Небольшая доля терпения помогла нам выработать новый метод сотрудничества. Мы садились вместе за компьютер, выявляли ошибку, а затем разделяли ее и атаковали сразу с двух сторон (сверху вниз и снизу вверх), а потом объединяли полученные результаты. Наши терпение и готовность экспериментировать с новыми стилями работы спасли не только проект, но и дружбу.

Будьте открыты для сотрудничества

Чем больше вы открыты для сотрудничества, тем выше ваша способность влиять на других и тем сильнее вы кажетесь. Это заявление выглядит противоречивым. Но практически каждый может вспомнить коллегу-упрямца — кто бы что ему ни говорил, он еще больше заперся в своем упрямстве. Что в итоге происходит с такими членами команды? Люди перестают прислушиваться к их мнению или возражениям, начинают «обходить» таких коллег как препятствие, от которого никуда не деться. Вы наверняка не захотите быть таким человеком, поэтому помните: это нормально, когда кто-то хочет вас переубедить. В первой главе этой книги мы говорили, что инженерия — это компромиссы. Нельзя быть правым во всем и всегда, если у вас нет неизменного окружения и совершенных знаний. Поэтому вы должны менять свою точку зрения, когда вам представляют новые доказательства. Тщательно расставляйте свои приоритеты: чтобы быть услышанным, сначала выслушайте других. Соберите аргументы коллег, *прежде* чем остановить обсуждение и твердо объявить свое решение, — если вы будете постоянно колебаться, меняя свою точку зрения, люди подумают, что вы слабовольный и нерешительный.

Идея уязвимости может показаться странной. Если кто-то признается в своем незнании рассматриваемой темы или путей решения проблемы, на какое доверие он может рассчитывать? Уязвимость — это проявление слабости, которое разрушает доверие, верно?

Неверно! Признание, что вы допустили ошибку или просто не подтвердили свой уровень, может в долгосрочной перспективе повысить ваш статус. На самом деле готовность показать уязвимость — это внешнее проявление смирения, которое

демонстрирует надежность и готовность брать на себя ответственность, а также сообщает, что вы доверяете коллегам. В ответ люди будут уважать вашу честность и силу. Иногда действительно лучше прямо сказать: «Я не знаю».

Профессиональные политики, например, печально известны тем, что никогда не признают свои ошибки или невежество, даже если всем очевидно, что они ошибаются или не знают предмета обсуждения. Такое поведение обусловлено тем, что политики постоянно подвергаются нападкам со стороны своих противников, и именно из-за него большинство не верит словам политиков. Однако вам, пишущим ПО, не нужно постоянно защищаться — ваши товарищи по команде являются вашими единомышленниками, а не конкурентами. У вас с ними одна цель.

Действовать по-гугловски

У нас в Google есть собственная внутренняя версия принципов «смирения, уважения и доверия».

С самого начала существования нашей культуры мы называли действия «по-гугловски» (googley) или «не по-гугловски». Ни в одном словаре вы не найдете это слово, но оно означает «не будь злым», «поступай правильно» или «будь добр к другим». Со временем мы начали использовать термин «по-гугловски» как неформальный признак соответствия нашей культуре у кандидатов на должность инженера или у коллег. Люди часто выражают мнение о других, используя этот термин, например «человек пишет хороший код, но действует не совсем по-гугловски».

В конце концов, мы поняли, что термин «по-гугловски» перегружен смыслом и может стать источником неосознанной предвзятости при найме или оценке работы. Если для каждого сотрудника выражение «по-гугловски» будет означать что-то свое, есть риск свести его к «совсем как я». Очевидно, что это не лучший тест для найма новых сотрудников — мы не хотим нанимать людей, которые «совсем как мы», нам нужны люди из разных слоев общества, с разными мнениями и опытом. Личное желание интервьюера выпить пива с кандидатом (или коллегой) *никогда* не должно рассматриваться как признак, что кто-то является хорошим кандидатом или сотрудником и сможет добиться успеха в Google.

В конечном итоге мы в Google решили эту проблему, явно определив, что под «по-гугловски» подразумевается набор черт характера, которые мы ищем, включающие сильные лидерские способности и приверженность «смирению, уважению и доверию».

Развитие в неопределенности

Умение, имея дело с противоречивыми сообщениями или указаниями, достичь согласия и добиться прогресса в решении проблемы, даже если окружающая среда постоянно меняется.

Бережное отношение к обратной связи

Умение уважительно принимать критику и давать обратную связь и понимание ценности обратной связи для личного (и командного) развития.

Стремление изменить текущее положение вещей

Умение ставить амбициозные цели и достигать их, даже когда коллеги сопротивляются или проявляют инертность.

Пользователь прежде всего

Понимание пользователей продуктов Google, уважительное отношение к их мнению и готовность предпринять действия, отвечающие их интересам.

Заботливое отношение к команде

Сочувственное и уважительное отношение к коллегам, готовность оказывать им активную помощь без лишних вопросов и улучшать сплоченность команды.

Правильные поступки

Соблюдение этических норм во всем и готовность принимать сложные или неудобные решения для защиты целостности команды и продукта.

В настоящее время, определив желательные черты характера и образ поведения сотрудника, мы начали уходить от использования термина «по-гугловски». Всегда лучше четко обозначать свои ожидания!

Заключение

Основой практически любого ПО является сплоченная команда. Миф о гениальности разработчика, в одиночку создающего ПО, все еще сохраняется, но правда в том, что в действительности никто не работает один. Чтобы организация, занимающаяся созданием ПО, выдержала испытание временем, она должна иметь здоровую культуру, основанную на смирении, доверии и уважении и вращающуюся вокруг команды, а не отдельного человека. Кроме того, творческий характер разработки ПО *требует*, чтобы люди рисковали и иногда терпели неудачу, принять которую помогает здоровая командная среда.

Итоги

- Помните о недостатках работы в изоляции.
- Оцените, сколько времени вы и ваша команда тратите на общение и межличностные конфликты. Даже небольшие вложения в понимание личностей и разных стилей работы могут иметь большое значение для повышения эффективности.
- Если вы хотите эффективно работать в команде или крупной организации, помните о своем предпочтительном стиле работы и стилях работы коллег.

Обмен знаниями

Авторы: Нина Чен и Марк Баролак

Редактор: Риона Макнамара

Искать ответы на вопросы, связанные с вашей предметной областью, лучше внутри организации, а не в интернете. Но для этого организации нужны эксперты *и* механизмы распространения знаний, о которых мы поговорим в этой главе. Эти механизмы могут быть очень простыми (задать вопрос и написать ответ) и более структурированными, такими как учебные пособия и классы. Самое главное — в организации должна поддерживаться *культура обучения*, которая требует создания чувства психологической безопасности, позволяющего людям признавать недостаток собственных знаний.

Сложности в обучении

Обмен опытом между организациями — непростая задача, особенно при отсутствии устойчивой культуры обучения. Компания Google уже сталкивалась с рядом проблем, главным образом в период роста.

Отсутствие чувства психологической безопасности

Среда, в которой люди боятся рисковать или совершать ошибки из страха наказания. Часто проявляется как культура страха или стремление избегать прозрачности.

Информационные островки

Фрагментация знаний, возникающая, когда разные подразделения организации не общаются друг с другом или не используют общие ресурсы. В такой среде каждая группа создает свой способ действий¹. Это часто приводит к следующему:

Фрагментация информации

Каждый островок представляет лишь часть общей картины.

Дублирование информации

На каждом островке изобретается свой способ решения одной и той же задачи.

¹ Другими словами, вместо того чтобы разрабатывать один глобальный максимум, в организации создается множество локальных максимумов (<https://oreil.ly/6megY>).

Информационный перекоз

На островах по-разному используется одно и то же решение.

Единая точка отказа (ЕТО)

Узкое место, возникающее, когда важная информация поступает только от одного человека (вспомните фактор автобуса (<https://oreil.ly/IPT-2>) из главы 2).

Она может появиться из самых благих намерений вроде предложения «давай я сделаю это вместо тебя» и давать кратковременное увеличение эффективности («так быстрее») в ущерб долгосрочной перспективе развития (команда не учится выполнять работу). ЕТО приводит к победе принципа «все или ничего».

Принцип «все или ничего»

Команда, в которой проведена грань между теми, кто знает «все», и новичками. Если эксперты всегда и все делают сами, не тратя времени на подготовку новых экспертов путем наставничества или создания инструкций, знания и обязанности продолжают накапливаться у тех, кто уже имеет опыт, а новые члены команды брошены и развиваются медленно.

Попугайство

Повторение без понимания сути. Обычно характеризуется бездумным копированием паттернов или кода без понимания их цели на основе предположения, что это необходимо по неизвестным причинам.

Кладбища с привидениями

Места, обычно в коде, которых люди стараются не касаться, потому что боятся, что что-то может пойти не так. В отличие от вышеупомянутого *попугайства*, для кладбищ с привидениями характерно отсутствие каких-либо действий из-за страха и суеверий.

В оставшейся части главы мы рассмотрим успешные политики преодоления этих сложностей, обнаруженные инженерами Google.

Философия

Программную инженерию можно определить как командную разработку многоверсионного ПО¹, в центре которой находятся люди. Да, код — это важный результат, но лишь малая часть создания продукта. Необходимо отметить, что код не возникает сам по себе из ничего, равно как и опыт. Каждый эксперт когда-то был новичком: успех организации зависит от роста ее сотрудников и инвестиций в них.

Совет эксперта всегда неоценим. Члены команды могут разбираться в разных областях, поэтому выбор товарища по команде, которому можно задать любой вопрос,

¹ Parnas D. L. Software Engineering: Multi-person Development of Multi-version Programs. Heidelberg: Springer-Verlag Berlin, 2011.

очень важен. Но если эксперт уходит в отпуск или переходит в другой коллектив, команда может оказаться в беде. И хотя один человек может оказать помощь многим, эта помощь не масштабируется и ограничивается числом «многих».

Документированные знания лучше масштабируются не только в команде, но и во всей организации. Такие механизмы, как командная электронная база знаний (wiki), позволяют множеству авторов делиться своим опытом с большими группами людей. Но масштабируемость документации сопровождается компромиссами между ее обобщенностью и применимостью к конкретным ситуациям, а также требует дополнительных затрат на обслуживание и поддержание ее актуальности.

Между знаниями отдельных членов команды и задокументированными знаниями находятся коллективные знания. Эксперты часто знают то, что нигде не задокументировано. Если задокументировать эти знания и поддерживать их, они станут доступными не только тем, кто имеет возможность общаться с экспертом, но и всем, кто сможет найти и прочитать документацию.

Получается, что в идеальном мире, где все всегда и сразу документируется, нет нужды консультироваться с человеком, верно? Не совсем. Письменные знания имеют преимущества масштабирования, но целевая личная помощь ничуть не менее ценна. Человек-эксперт может синтезировать свои знания, оценить, какая информация применима к конкретному случаю, определить актуальность документации и знать, где ее найти или к кому обратиться.

Коллективные и письменные знания дополняют друг друга. Даже идеальная команда экспертов с совершенной документацией должна общаться друг с другом, координировать действия с другими командами и адаптировать свои политики. Никакой подход к обмену знаниями не может быть правильным сразу для всех типов обучения, и лучшее сочетание, скорее всего, будет зависеть от особенностей организации. Организационные знания, вероятно, будут меняться по мере роста организации. Тренируйтесь, сосредоточьтесь на обучении и росте и создайте свою группу экспертов. Инженерных знаний много не бывает.

Создание условий: психологическая безопасность

Психологическая безопасность имеет решающее значение для формирования образовательной среды.

Чтобы учиться, вы должны сначала осознать, что есть что-то, чего вы не понимаете. Мы должны приветствовать такую честность (<https://xkcd.com/1053>), а не наказывать ее. (В Google эта работа поставлена довольно хорошо, но иногда инженеры не хотят признавать, что они чего-то не понимают.)

Значительная часть обучения — это способность пробовать что-то и чувствовать себя в безопасности. В здоровой среде люди чувствуют себя комфортно, задают вопросы, ошибаются и приобретают новые знания. Исследование в Google (<https://>

oreil.ly/sRqWg) показало, что психологическая безопасность является самым важным условием создания эффективной команды.

Наставничество

Мы в Google стараемся задать тон общения, когда к компании присоединяется «нуглер» (новый гуглер). Чувство психологической безопасности мы создаем с помощью назначения наставника — человека, который не является членом команды, менеджером или техническим руководителем, в обязанности которого входит отвечать на вопросы и оказывать помощь нуглеру. Наличие официально назначенного наставника для обращения за помощью облегчает новичку работу и избавляет его от страха, что он может отнимать время у коллег.

Наставник — это доброволец, работающий в Google больше года и готовый консультировать новичка по любым вопросам, от использования инфраструктуры до навигации по культуре Google. Он поддерживает нуглера, когда тот не знает, к кому обратиться за советом. Наставник не входит в одну команду с новичком, что позволяет второму чувствовать себя более комфортно при обращении за помощью.

Наставничество формализует и облегчает обучение, но самообучение — это непрерывный процесс. Коллеги всегда учатся друг у друга: как новые сотрудники, так и опытные инженеры, изучающие новые технологии. В здоровой команде коллеги могут не только давать ответы, но и задавать вопросы, не боясь показать, что они чего-то не знают.

Психологическая безопасность в больших группах

Обратиться за помощью к коллеге по команде легче, чем к большой группе в основном незнакомых людей. Но, как мы видели, индивидуальные решения плохо масштабируются. Групповые решения более масштабируемы, но они также более пугающие. Новичкам может быть страшно задать вопрос большой группе, зная, что он может храниться в архиве много лет. Потребность в психологической безопасности усиливается в больших группах. Каждый член группы должен сыграть свою роль в создании и поддержании безопасной среды, гарантирующей, что новички не будут бояться задавать вопросы, а начинающие эксперты будут чувствовать, что способны помочь новичкам, не опасаясь нападок со стороны авторитетных экспертов.

Самый действенный способ создать такую безопасную и гостеприимную среду — заменить состязательность групп их сотрудничеством. В табл. 3.1 приведены некоторые примеры рекомендуемых паттернов (и соответствующих им антипаттернов) групповых взаимодействий.

Эти антипаттерны могут возникать непреднамеренно: кто-то, возможно, пытается помочь, но случайно проявляет снисходительность и неприветливость. Мы считаем, что здесь уместно перечислить социальные правила сообщества Recurse Center (<https://oreil.ly/zGvAN>):

Никакого притворного удивления («Что?! Я не могу поверить! Ты не знаешь, что такое стек?!»)

Притворное удивление разрушает чувство психологической безопасности и заставляет членов группы бояться признаться в нехватке знаний.

Никаких «на самом деле»

Скрупулезные исправления, которые, как правило, связаны с игрой на публику, а не с точностью.

Никаких подсказок из задних рядов

Прерывание текущего обсуждения, чтобы предложить свое мнение, без корректного вступления в разговор.

Никаких «-измов» («Это так просто, что даже моя бабушка справилась бы!»)

Выражения предвзятости (расизм, эйджизм, гомофобия), которые могут заставить людей чувствовать себя ненужными, неуважаемыми или незащищенными.

Таблица 3.1. Паттерны групповых взаимодействий

Рекомендуемые паттерны (сотрудничество)	Антипаттерны (сопостязательность)
Простые вопросы или ошибки правильно воспринимаются и находят ответы и решения	К простым вопросам или ошибкам придираются, и человек, задавший вопрос, наказывается
Объяснения направлены на обучение человека, задавшего вопрос	Объяснения направлены на демонстрацию своих знаний
На вопросы даются полезные ответы с терпением и доброжелательностью	Ответы даются свысока, в язвительной и неконструктивной форме
Взаимодействия принимают форму общего обсуждения для поиска решений	Взаимодействия принимают форму спора между «победителями» и «проигравшими»

Расширение знаний

Обмен знаниями начинается с вас. Важно понимать, что вам всегда есть чему поучиться. Следующие рекомендации позволят вам расширить свои знания.

Задавайте вопросы

Главный урок этой главы: постоянно учитесь и задавайте вопросы.

Мы говорим нуглерам, что на раскочку может уйти до шести месяцев. Этот длительный период необходим для освоения большой и сложной инфраструктуры Google, но он также подтверждает идею о том, что обучение — это непрерывный итеративный процесс. Одна из самых больших ошибок, которую допускают новички, — не просят помощи в сложной ситуации. Возможно, вам нравится преодолевать проблемы в одиночку или вы боитесь, что ваши вопросы «слишком просты». «Я должен по-

пробовать решить проблему сам, прежде чем просить о помощи», — думаете вы. Не попадайтесь в эту ловушку! Часто ваши коллеги являются лучшим источником информации: используйте этот ценный ресурс.

Не заблуждайтесь: день, когда вы вдруг обнаружите, что знаете все, не наступит никогда. Даже у инженеров, годами работающих в Google, возникают ситуации, когда они не знают, что делать, и это нормально! Не бойтесь говорить: «Я не знаю, что это такое, не могли бы вы объяснить?» Относитесь к незнанию как к области возможностей, а не страха¹.

Неважно, кто вы — новичок или ведущий специалист: вы всегда должны находиться в среде, где есть чему поучиться. В отсутствие такой среды неизбежен застой, при котором вам стоит поискать новую среду.

Моделировать такое поведение особенно важно для тех, кто играет руководящую роль: важно не приравнивать «руководство» к «знанию всего». На самом деле чем больше вы знаете, тем больше понимаете, что ничего не знаете (<https://oreil.ly/VWusg>). Открыто задавая вопросы² или показывая пробелы в знаниях, вы подтверждаете, что для других это тоже нормально.

Терпение и доброжелательность отвечающего способствуют созданию атмосферы, в которой нуждающиеся в помощи чувствуют себя в безопасности. Помогите человеку, задающему вам вопрос, преодолеть нерешительность. Инженеры, вероятно, *смогли бы* самостоятельно разобраться в коллективных знаниях, но они пришли не для того, чтобы работать в вакууме. Адресная помощь помогает инженерам работать быстрее, что, в свою очередь, повышает эффективность всей команды.

Вникайте в контекст

Обучение — это не только узнавание чего-то нового, но и развитие понимания решений, лежащих в основе существующего дизайна и реализации. Представьте, что вашей команде поручили развивать унаследованную кодовую базу критически важной части инфраструктуры, существовавшей много лет. Ее авторы давно ушли, а код непонятен. Может возникнуть соблазн переписать все заново и не тратить время на изучение существующего кода. Но вместо того чтобы воскликнуть: «Я ничего не понимаю», и закончить на этом, попробуйте погрузиться глубже: какие вопросы вы должны задать?

Рассмотрим принцип «забора Честерсона» (<https://oreil.ly/Ijv5x>): перед тем как удалить или изменить что-то, сначала разберитесь, зачем это «что-то» здесь находится.

¹ Синдром самозванца (<https://oreil.ly/2FIPq>) не редкость среди успешных учеников, и гуглеры не исключение. Большинство авторов этой книги тоже подвержены синдрому самозванца. Мы понимаем, что страх неудачи пугает людей с синдромом самозванца и может затормозить развитие.

² См. статью «How to ask good questions» (<https://oreil.ly/rr1cR>).

В деле реформирования, в отличие от деформирования, существует один простой и ясный принцип, который можно назвать парадоксом. Представьте, что существуют определенные нормы или законы, например забор, построенный поперек дороги. Молодой реформатор радостно подходит к забору и говорит: «Я не вижу в нем смысла, давайте уберем его». На что более умный реформатор вполне может ответить: «Коль скоро вы не увидите пользы, я не позволю убрать его. Идите и подумайте. А потом скажете мне, что вы видите в нем определенную пользу, и я позволю снести его».

Это не означает, что код не может быть лишен ясности или существующие паттерны проектирования не могут быть неправильными, но часто инженеры склонны приходить к выводу «это плохо!» быстрее, чем нужно, особенно в отношении незнакомого кода, языков или парадигм. Google тоже не застрахован от такого подхода. Ищите и вникайте в контекст, особенно в решениях, которые кажутся необычными. Когда вы поймете суть и цель кода, подумайте, имеет ли смысл ваше изменение. Если да, то смело меняйте код. Но если нет, задокументируйте свои рассуждения для будущих читателей.

Многие руководства по стилю в Google включают дополнительные подробности, чтобы помочь читателям понять причины существующих рекомендаций по стилю, а не просто запомнить список произвольных правил. Понимание причин появления данного руководства позволяет авторам принимать обоснованные решения о том, когда руководство должно перестать применяться и его следует обновить (глава 8).

Масштабирование вопросов: вопросы к сообществу

Индивидуальная помощь бесценная, но она имеет ограниченный масштаб. Кроме того, ученику может быть трудно запомнить все детали. Сделайте своему будущему «я» одолжение: когда узнаете что-то новое в обсуждении тет-а-тет, *запишите это*.

Скорее всего, у следующих новичков появятся те же вопросы, что были у вас. Сделайте им одолжение тоже и *поделитесь тем, что записали*.

Обмен полученными ответами полезен, но также полезно обращаться за помощью к более широкому сообществу. В этом разделе мы рассмотрим различные формы обучения в сообществе. Все представленные подходы — групповые чаты, списки рассылки и системы вопросов и ответов — имеют свои достоинства и недостатки и дополняют друг друга. Они позволяют получить помощь от широкого круга коллег и экспертов, а также открывают доступ к ответам для нынешних и будущих членов сообщества.

Групповые чаты

Сложно получить помощь, если вы не знаете, к кому обратиться, или человек, которому вы хотите задать вопрос, занят. В таких ситуациях на выручку придут групповые чаты. Они хороши тем, что позволяют задать вопрос сразу нескольким

людям и быстро обменяться мнениями с кем угодно. Дополнительный плюс: другие участники чата могут учиться одновременно с вами, просматривая вопросы и ответы, а во многих случаях чаты можно автоматически архивировать и создавать на их основе базы знаний.

В групповых чатах, как правило, есть разделы, посвященные отдельным темам или командам. Тематические чаты обычно открыты, и любой может зайти в них и задать вопрос. Они, как правило, привлекают экспертов и быстро набирают большое число участников, что ускоряет получение ответов. Командные чаты, напротив, значительно меньше и ограничивают участие в них. Они могут не иметь такого же охвата, как тематические чаты, но благодаря меньшему размеру кажутся новичкам безопаснее.

Групповые чаты отлично подходят для быстрого получения ответов, но они недостаточно структурированы, что может затруднить извлечение нужной информации из беседы, в которой вы не принимаете активного участия. Когда вам понадобится поделиться информацией за пределами группы или сделать ее доступной для последующего использования, создайте документ или отправьте сообщение в список рассылки.

Списки рассылки

Для большинства тем в Google есть список рассылки `topic-users@` или `topic-discuss@` в Google Groups, к которому может присоединиться любой сотрудник компании. Вопросы в общедоступной рассылке задаются так же, как в групповом чате: их видят множество людей, знающих и не знающих ответ. В отличие от групповых чатов, общедоступные списки рассылки распространяют информацию среди более широкой аудитории: они упакованы в архивы, лучше структурированы и предусматривают возможность поиска. В Google списки рассылки также индексируются и их легко найти с помощью Moma — поисковой системы во внутренней сети Google.

После получения ответа на вопрос может возникнуть соблазн немедленно продолжить работу. Не торопитесь! Возможно, в будущем кому-то пригодится полученная вами информация (<https://xkcd.com/979>), поэтому опубликуйте ответ в списке рассылки.

Списки рассылки не лишены недостатков. Они хорошо подходят для сложных вопросов, которые требуют описания большого количества деталей, но для быстрого обмена информацией между участниками больше подходят групповые чаты. Тема, посвященная конкретной проблеме, обычно наиболее полезна, когда она активно обсуждается. Архивы электронной почты остаются практически неизменными, и порой бывает трудно определить, соответствует ли ответ в старой ветке обсуждения текущей ситуации. Кроме того, в списке рассылки может быть больше шума, чем, например, в официальной документации, поскольку решение, найденное в списке рассылки, может оказаться неприменимым для вас.

ЭЛЕКТРОННАЯ ПОЧТА В GOOGLE

Google печально известна чрезмерно широким проникновением электронной почты во все аспекты ее деятельности. Инженеры Google получают сотни (если не больше) электронных писем каждый день с различной степенью важности. Нуглеры могут потратить несколько дней на настройку фильтров электронной почты, чтобы справиться с объемом уведомлений, поступающих от групп, на которые они автоматически подписаны; многие из них просто сдаются и не пытаются успевать читать все письма. Некоторые группы по умолчанию вкладывают большие списки рассылки в каждое обсуждение, создавая нежелательный шум.

Мы склоняемся к организации рабочих процессов с использованием электронной почты. Электронная почта не является лучшим средством передачи информации, но к ней мы привыкли. Обдумайте целесообразность ее использования, когда будете выбирать формы коммуникации.

YAQS: система вопросов и ответов

YAQS (yet another question system) — это внутренняя версия Stack Overflow для Google (<https://oreil.ly/iTtbm>). Она позволяет гуглерам ссылаться на существующий или разрабатываемый код, а также обсуждать конфиденциальную информацию.

Так же как Stack Overflow, система YAQS обладает многими преимуществами списков рассылки и добавляет дополнительные достоинства: ответы, отмеченные как полезные, продвигаются вверх в пользовательском интерфейсе, и пользователи могут редактировать вопросы и ответы, чтобы они оставались точными и полезными при изменении кода и фактов. Сегодня в Google одни списки рассылки заменены системой YAQS, а другие превратились в дискуссионные списки более широкой направленности и в меньшей степени ориентированные на решение проблем.

Распространяйте знания: вам всегда есть чему научить других

Обучать могут не только эксперты, и опыт не является двоичной величиной «либо есть, либо нет». Опыт — это многомерный вектор знаний: люди обладают разными знаниями в разных областях. Это одна из причин, почему разнообразие кадров имеет решающее значение для успеха компании: разные люди приносят разные точки зрения и опыт (глава 4). Инженеры Google обучают коллег на собраниях и в учебных классах, через технические доклады, документацию и обзор кода.

Время консультаций

Иногда важно иметь возможность поговорить с человеком, и в таких случаях консультации могут оказаться хорошим решением. Консультация — это регулярно проводимое мероприятие (обычно раз в неделю), при котором один или несколько человек отвечают на вопросы по определенной теме. Она почти никогда не является

предпочтительным вариантом обмена знаниями: часто невозможно ждать следующей консультации, если вопрос срочный, а поиск ответа на него требует много времени и сил. Тем не менее консультации дают людям возможность лично поговорить с экспертом. Это особенно полезно, если проблема достаточно неоднозначна и инженер пока не знает, какие вопросы задавать (например, в самом начале разработки новой службы), или решения не описаны в документации.

Технические обсуждения и учебные классы

В Google выработана надежная культура внутренних и внешних¹ технических обсуждений и учебных классов. Наша команда engEDU (engineering education) специализируется на организации обучения в области computer science для самой широкой аудитории, от инженеров Google до студентов по всему миру. На более низком уровне наша программа g2g (googler to googler) позволяет гуглерам регистрироваться и читать (или посещать) лекции и занятия от коллег в Google². Эта программа пользуется оглушительным успехом, и тысячи участвующих в ней гуглеров преподают самые разные темы, от технических (например, «Понимание векторизации в современных процессорах») до довольно забавных (например, «Танец свинг для начинающих»). Технические обсуждения обычно проводятся лектором, выступающим непосредственно перед аудиторией. Занятия в классе, с другой стороны, могут включать лекцию, но больше сосредоточены на упражнениях и требуют активного участия слушателей. Как результат, занятия под руководством наставника обычно более трудоемкие и дорогостоящие в организации и поддержке, чем технические обсуждения, и предназначены для более важных или сложных тем. Тем не менее после создания класса его относительно легко масштабировать, потому что сразу несколько преподавателей могут читать курс, опираясь на одни и те же материалы. Мы обнаружили, что классы предпочтительнее при следующих условиях:

- Тема достаточно сложна, из-за чего часто является источником недоразумений. Создание классов — кропотливая работа, которая должна быть обоснована.
- Тема относительно стабильная. Обновление материалов класса — трудная задача, поэтому, если предмет быстро развивается, используйте другие формы передачи знаний.
- По данной теме наставники готовы отвечать на вопросы и оказывать персональную помощь. Если учащиеся легко могут самостоятельно освоить тему, эффективнее использовать инструменты самообучения, такие как документация или аудио- и видеозаписи. Ряд вводных курсов в Google имеет версии для самостоятельного изучения.

¹ Вот пара примеров: <https://talksat.withgoogle.com> и <https://www.youtube.com/GoogleTechTalks>.

² Программа g2g подробно описана в книге Ласло Бока «Работа рулит! Почему большинство людей в мире хотят работать именно в Google» (М.: МИФ, 2015), которая включает описания различных аспектов программы, а также способы оценки воздействия и рекомендации, на что следует обратить внимание при создании аналогичных программ. — *Примеч. пер.*

- Спрос на регулярное проведение класса достаточно высок. В противном случае потенциальные учащиеся могут получить необходимую информацию другими способами. В Google классы особенно актуальны для небольших, географически удаленных офисов.

Документация

Документация — это письменное знание, основная цель которого — помочь читателям чему-то научиться. Не все письменные знания являются документацией. Например, ответ на вопрос можно найти в списке рассылки, но основная цель списка в том, чтобы найти ответы, и только во вторую очередь — задокументировать обсуждение для других.

В этом разделе мы сосредоточимся на возможностях внести вклад в создание формальной документации, от исправления опечаток до более существенных усилий, таких как документирование коллективных знаний.



Подробнее о документации в главе 10.

Обновление документации

Когда вы узнаете что-то новое, уделите внимание улучшению существующей документации и учебных материалов. Освоив новый процесс или систему, вы забудете, с какими сложностями сталкивались раньше и чего вам не хватало во вводной документации. Если найдете ошибку или упущение в документации, исправьте ее! Оставьте место стоянки чище, чем оно было до вашего прихода¹, и пробуйте обновлять документацию самостоятельно, даже если она принадлежит другому подразделению.

В Google инженеры чувствуют себя вправе обновлять документацию, кому бы она ни принадлежала, даже если исправление касается опечатки. Уровень поддержки сообщества заметно повысился с внедрением g3doc², значительно упростившим гуглерам поиск владельца документации для передачи ему своих предложений. Кроме того, такой подход помогает оставить след в истории изменений.

Создание документации

Наращивая мастерство, пишите новую документацию и обновляйте существующую. Например, если вы настроили новый процесс разработки, запишите, какие шаги выполнили. Так вы облегчите путь тем, кто следует за вами. Но еще лучше помочь другим самостоятельно находить ваш документ. Недоступность или неизвестность

¹ См. «The Boy Scout Rule» (<https://oreil.ly/2u1Ce>) и книгу Кевина Хиннея (Kevin Henney) «97 Things Every Programmer Should Know» (Boston: O'Reilly, 2010).

² Название g3doc означает «документация google3». google3 — это название текущего расположения монолитного репозитория исходного кода в Google.

местоположения документации равноценны ее отсутствию. В g3doc документация находится рядом с исходным кодом, а не в (недоступном) документе или на неизвестной веб-странице.

Наконец, убедитесь в наличии механизма обратной связи. Если у читателей не будет простого способа сообщить, что документация устарела или неточна, они, скорее всего, ничего и никому не расскажут и следующий новичок столкнется с той же проблемой. Люди охотнее вносят изменения, если чувствуют, что кто-то действительно заметит и рассмотрит их предложения. В Google есть возможность отправить сообщение об ошибке непосредственно в самом документе.

Кроме того, гуглеры могут легко оставлять комментарии на страницах g3doc. Другие гуглеры имеют возможность читать и отвечать на эти комментарии, и поскольку оставление комментария автоматически вызывает отправку сообщения об ошибке владельцу документации, читателю не нужно выяснять, к кому обратиться.

Продвижение документации

Иногда бывает трудно стимулировать инженеров на документирование своих действий. Работа над документацией требует времени и сил, ее результаты неочевидны, и ими в большинстве случаев пользуются другие. Подобная асимметрия, при которой многие извлекают выгоду из затрат времени немногих, полезна для организации в целом, но она требует хороших стимулов. Мы обсудим некоторые из них в разделе «Стимулы и признание» ниже.

Тем не менее автор документа часто тоже может получить непосредственную выгоду от написания документации. Предположим, что члены команды всегда обращаются к вам за помощью в устранении определенных видов сбоев в работе. Документирование используемых вами процедур потребует затрат времени, но выполнив эту работу один раз, вы сэкономите время в будущем, передав членам команды ссылку на документацию и отвлекаясь на практическую помощь только при необходимости.

Написание документации также помогает команде и организации расширяться. Во-первых, информация в документации становится каноничной: члены группы могут ссылаться на общий документ и даже обновлять его самостоятельно. Во-вторых, она может распространиться за пределы команды, если некоторые части документации не являются уникальными для команды и будут полезны для других команд, решающих подобные задачи.

Код

На метауровне код — это знание, поэтому процесс написания кода можно считать формой описания знаний. Обмен знаниями может не являться прямым намерением при разработке кода, но является побочным эффектом, который можно усилить читабельностью и ясностью кода.

Документирование кода является еще одним из способов обмена знаниями. Четкая документация приносит пользу не только пользователям библиотеки, но и тем, кто

в будущем будет ее сопровождать. Комментарии в коде тоже передают знания: они пишутся специально для будущих читателей (включая вас в будущем!). С точки зрения компромиссов комментарии в коде имеют те же недостатки, что и документация в целом: они должны активно поддерживаться, чтобы не устареть и не начать противоречить коду.

Обзоры кода (глава 9) часто дают возможность обучения и авторам, и рецензентам. Например, предложение рецензента может познакомить автора с новым паттерном тестирования или рецензент может узнать о новой библиотеке, увидев, как автор использует ее в своем коде. Google стандартизирует наставничество посредством обзоров, определяя *процесс повышения удобочитаемости* (подробнее в конце этой главы).

Распространение знаний с ростом организации

С ростом организации все труднее обеспечивать надлежащий обмен опытом между подразделениями. Если культура важна на каждой стадии роста, то создание справочных источников информации актуальнее для более зрелых организаций.

Развитие культуры обмена знаниями

Организационная культура — это хрупкое человеческое творение, полезность которого многие компании осознают с опозданием. Но мы в Google считаем, что внимание к культуре и окружающей среде¹ позволяет добиться большего, чем сосредоточение на результатах этой среды, таких как код.

Крупные организационные изменения — сложная задача, и на эту тему написано бесчисленное количество книг. Мы не претендуем на знание всех ответов в этой области, но можем поделиться конкретными шагами, которые предпринимались в Google для создания культуры, способствующей обучению.

Прочитайте книгу «Работа рулит!», чтобы больше узнать о культуре Google.

Уважение

Отталкивающее поведение нескольких человек может сделать неприветливыми целую команду или сообщество (https://oreil.ly/R_Y7N). В недружелюбной среде новички учатся задавать вопросы в другом месте, а перспективные специалисты опускают руки, не имея возможности расти. В худшем случае в группе остаются только наиболее токсичные ее члены. Выбраться из этого состояния может быть очень трудно.

Обмен знаниями может и должен осуществляться с доброжелательностью и уважением. В технологических организациях терпимость или, что еще хуже, почтение к «блестящему ничтожеству» одновременно и распространены, и вредны, но быть

¹ Бок Л. Работа рулит! Почему большинство людей в мире хотят работать именно в Google. М.: МИФ, 2015.

экспертом и быть добрым — это не взаимоисключающие характеристики. В разделе «Лидерство» корпоративного документа Google четко говорится об этом:

«Хотя некоторая степень технического лидерства приветствуется на более высоких уровнях, тем не менее лидерство не всегда направлено на решение технических проблем. Лидеры помогают окружающим людям расти, улучшают психологический климат команды, создают культуру командной работы и сотрудничества, смягчают напряженность в команде, доказывают личным примером важность культуры и ценностей Google и делают Google более динамичным и интересным местом для работы. Хамы — плохие лидеры».

Урс Хельцле (Urs Hölzle, старший вице-президент по технической инфраструктуре) и Бен Трейнор Слосс (Ben Treynor Sloss, вице-президент, основатель Google SRE) написали регулярно цитируемый внутренний документ «No Jerks» («Не хами») о том, почему гуглеры должны заботиться об уважительном поведении на работе и как этого достичь.

Стимулы и признание

Хорошую культуру необходимо активно развивать, и для продвижения культуры обмена знаниями необходимо иметь систему поощрений и вознаграждений. Для многих организаций характерна распространенная ошибка, когда на словах говорят о наборе ценностей, а на деле активно поощряют поведение, не связанное с этими ценностями. Люди реагируют на стимулы, а не на дежурные фразы, поэтому при создании системы поощрений и вознаграждений важно, чтобы слова не расходились с делом.

Google использует различные механизмы признания заслуг, от общекорпоративных стандартов, таких как анализ эффективности и критерии продвижения, до взаимных поощрений между гуглерами.

Штатное расписание, которое мы используем для измерения вознаграждений, таких как премии и карьерный рост, побуждает инженеров делиться знаниями. На более высоких уровнях поощряется авторитет сотрудника, который должен увеличиваться с карьерным ростом. На высших уровнях примеры лидерства включают следующее:

- Растите будущих лидеров, служите наставниками для младших сотрудников, помогайте им развиваться как в техническом плане, так и в корпоративном.
- Поддерживайте и развивайте сообщества разработчиков ПО в Google, выполняя обзоры кода и дизайна, организуя инженерное образование и разработку, а также давая экспертные рекомендации.



Подробнее о лидерстве в главах 5 и 6.

Штатное расписание — это способ управления культурой сверху. Но в Google культура также управляется снизу, например через программу вознаграждения коллег — это

денежное вознаграждение и формальное признание, которыми любой гуглер может поощрить другого гуглера за его работу¹. Например, когда Рави предлагает дать премию коллеге Джулии за то, что она является главным автором списка рассылки — регулярно отвечая на вопросы, чем приносит пользу многим сотрудникам, — он публично признает ее вклад в обмен знаниями и ее авторитет за пределами команды. Поскольку вознаграждения со стороны коллег ориентированы на сотрудников, они интересуют всех.

Кроме вознаграждений от коллег распространено также публичное признание вклада (как правило, меньшего по отдаче или усилиям, чем те, которые заслуживают вознаграждения от коллег), повышающее авторитет гуглера среди коллег.

Когда гуглер дает другому гуглеру вознаграждение или похвалу, он может отправить копию наградного письма группам или отдельным лицам, способствуя увеличению известности своего коллеги. Кроме того, руководители, получая такие наградные письма, обычно рассылают их членам своей команды, чтобы те могли отметить достижения друг друга.

Система, в которой люди могут выражать официальное признание своим коллегам, является мощным инструментом поощрения, стимулирующим сотрудников продолжать творить и удивлять коллег. Важно не вознаграждение, а признание.

Создание канонических источников информации

Каноническими называют централизованные, общекорпоративные источники информации, которые позволяют стандартизировать и распространять экспертные знания. Они лучше всего подходят для хранения информации, которая имеет отношение ко всем инженерам в организации и препятствуют образованию информационных островков. Например, руководство по организации рабочего процесса разработчика должно быть каноническим, тогда как руководство по запуску локального экземпляра Frobber в основном используют только инженеры, работающие над Frobber.

Создание канонических источников информации требует больше инвестиций, чем поддержка локальной информации, такой как командная документация, но также дает более широкие преимущества. Наличие централизованного источника справочных материалов в организации упрощает поиск информации, а также избавляет от проблем, связанных с ее фрагментацией, которые могут возникнуть, когда несколько групп, сталкивающихся с похожими проблемами, создают свои, часто противоречивые, руководства.

Поскольку каноническая информация хорошо видна и обеспечивает общее на уровне организации понимание данных, важно, чтобы ее содержимое активно поддерживалось и проверялось экспертами в предметной области. Чем сложнее тема, тем важнее наличие явных владельцев канонического источника сведений о ней.

¹ К числу вознаграждений от коллег относятся денежные премии и сертификаты, а также наградная запись гуглера во внутреннем инструменте gThanks.

Читатели могут заметить, что какая-то каноническая информация устарела, но обычно у них нет достаточного опыта для ее изменения, даже если инструменты позволяют это сделать.

Создание и поддержка централизованных канонических источников информации — дорогостоящий и трудоемкий процесс, и не все материалы можно совместно использовать на уровне организации. При оценке усилий, которые придется приложить для формирования этого ресурса, учитывайте особенности аудиторией. Кому нужна информация? Вам? Команде? Всем участникам проекта? Всем инженерам?

Руководства разработчиков

В Google имеется широкий набор официальных руководств для инженеров, включая руководства по стилю (<http://google.github.io/styleguide>), официальные рекомендации по разработке ПО¹, руководств по проверке кода² и тестированию³, а также «Советы недели» (TotW, tips of the week)⁴.

Объем информации чрезвычайно велик, и бессмысленно ожидать, что инженеры прочтут ее всю от начала до конца, а тем более усвоят одномоментно. Поэтому эксперт, уже знакомый с некоторым руководством, может отправить ссылку на него коллеге-инженеру для ознакомления. Так эксперт экономит время, которое мог потратить на объяснение практики, принятой в компании, а учащийся узнает о существовании канонического источника информации, к которому можно обратиться в любое время. Такой процесс масштабирует знания, поскольку позволяет экспертам определять и удовлетворять конкретные информационные потребности, используя общие ресурсы.

Ссылки go/

Ссылки `go/` (которые иногда называют ссылками `goto/`) — это сокращения внутренних URL в Google⁵. Большинство внутренних ссылок в Google имеют, по крайней мере, одну внутреннюю ссылку `go/`. Например, «`go/spanner`» предоставляет информацию о Spanner, а «`go/python`» ссылается на руководство Google для разработчиков на Python. Сама информация может храниться в любом репозитории (`g3doc`, Google Drive, Google Sites и т. д.), но наличие ссылки `go/` обеспечивает запоминающийся и интуитивно определяемый способ доступа к данным. Такое решение дает ряд преимуществ:

- Ссылки `go/` настолько короткие, что ими легко обмениваться в разговоре («Обязательно проверь `go/frobber!`»). Это гораздо проще, чем искать ссылку и отправлять ее в сообщении. Удобство обмена ссылками повышает скорость распространения знаний.

¹ Например, книги о разработке ПО в Google.

² См. главу 9.

³ См. главу 11.

⁴ Доступны для нескольких языков. Извне доступен сборник для C++ (<https://abseil.io/tips>).

⁵ Ссылки `go/` не имеют отношения к языку Go.

- Ссылки `go/` имеют постоянный характер и будут работать, даже если изменится базовый URL. Когда владелец перемещает контент в другой репозиторий (например, из Google doc в `g3doc`), он может просто обновить целевой URL ссылки `go/`. Сама ссылка `go/` остается неизменной.

Ссылки `go/` настолько укоренились в культуре Google, что возник благоприятный замкнутый круг: ищущий информацию о Frobber сначала проверяет ссылку `go/frobber`, и если она не указывает на «Руководство разработчика Frobber» (вопреки ожиданиям), то гуглер сам настраивает ссылку. В результате гуглеры часто угадывают правильную ссылку `go/` с первой попытки.

Лабораторные работы

Лабораторные работы (codelab) Google размещены в учебных пособиях, которые объясняют инженерам новые идеи или процессы путем объединения теории, примеров и упражнений¹. Каноническая коллекция лабораторных работ для технологий, широко используемых в Google, доступна на `go/codelab`. Перед публикацией лабораторные работы проходят несколько этапов формального анализа и тестирования и являются интересным промежуточным звеном между статической документацией и учебными классами, однако обладают не только лучшими, но и худшими их чертами. Практический характер лабораторных работ делает их более привлекательными, чем традиционная документация, — инженеры могут получить к ним доступ по запросу и выполнить самостоятельно, но они требуют дорогостоящей поддержки и не учитывают индивидуальных потребностей учащегося.

Статический анализ

Инструменты статического анализа — это мощный способ обмена знаниями, которые можно проверить в программе. Каждый язык программирования имеет свои особые инструменты статического анализа, но все они объясняют авторам и рецензентам кода, как можно улучшить код с помощью стиля и передовых практик. Некоторые инструменты идут дальше и предлагают автоматически применить рекомендации.



Подробнее об инструментах статического анализа и их использовании в Google в главе 20.

Подготовка инструментов статического анализа требует вложения сил и средств, но зато они легко масштабируются. Когда в инструмент добавляется проверка использования передовых практик, каждый инженер, использующий этот инструмент, узнает об их существовании. У инженеров появляется дополнительное свободное время для изучения чего-то другого. Инструменты статического анализа расширяют знания инженеров и позволяют организациям внедрять передовой опыт последовательно.

¹ Извне лабораторные работы доступны по адресу: <https://codelabs.developers.google.com>. (Лабораторные работы на русском языке, хотя и не все, можно найти по адресу: <https://codelabs.developers.google.com/lang-ru/> — Примеч. пер.)

Важно оставаться в курсе событий

Есть информация, которая имеет решающее значение для работы, например знание особенностей типичного рабочего процесса разработки. Другая информация, такая как обновления популярных инструментов оценки производительности, — менее критична, но все же важна. Таким образом, требования к среде обмена информацией зависят от важности распространяемых сведений. Например, пользователь ожидает, что официальная документация будет обновляться, но обычно не ждет того же от информационных бюллетеней, благодаря чему владелец может уменьшить затраты на их поддержку и обслуживание.

Новостные рассылки

В Google широко используются новостные рассылки для всех инженеров: EngNews (технические новости), Ownd (новости, касающиеся конфиденциальности и безопасности) и Google Greatest Hits (отчеты о наиболее интересных ситуациях за квартал). Это хороший способ передачи информации, которая представляет интерес для инженеров, но не является критически важной. Изучив эту сторону проблемы информирования, мы обнаружили, что новости вызывают больший интерес, когда рассылаются реже и содержат исключительно полезные сведения. В противном случае они воспринимаются как спам.

Большинство рассылок в Google распространяется по электронной почте, однако создатели некоторых из них проявили креатив. «Тестирование в туалете» (советы по тестированию) и «Обучение в уборной» (советы по повышению производительности) — это одностраничные информационные бюллетени, размещаемые в туалетных комнатах. Этот уникальный способ распространения новостей выделяется среди других источников информации, а все выпуски бюллетеней сохраняются в электронном архиве.



Историю о том, как появился бюллетень «Тестирование в туалете», вы найдете в главе 11.

Сообщества

Гуглеры любят создавать сообщества, объединяющие сотрудников разных подразделений, для обмена знаниями по различным темам. Эти открытые каналы связи позволяют учиться у коллег, находящихся за пределами вашего круга общения, и предотвращать образование информационных островков и дублирование сведений. Особой популярностью пользуются Google Groups: в Google есть тысячи внутренних групп, официальных, полуофициальных и неофициальных. Некоторые из них обсуждают вопросы устранения неполадок, другие (например, Code Health) больше ориентированы на дискуссии и оказание помощи. Внутренняя служба Google+ также популярна среди гуглеров как неформальный источник информации, где специали-

сты описывают интересные технические неполадки или подробности о проектах, над которыми работают.

Удобочитаемость: наставничество через обзоры кода

Термин «удобочитаемость» имеет в Google более широкое толкование, чем просто удобочитаемость кода. Это стандартизированный общекорпоративный процесс наставничества, главной целью которого является распространение передовых практик программирования. Удобочитаемость охватывает широкий спектр знаний, включая языковые идиомы, структуру кода, дизайн API, использование распространенных библиотек, документацию и тестирование.

Внедрение поддержки удобочитаемости начиналось с усилий одного человека. В самом начале существования Google Крейг Сильверстейн (идентификационный номер сотрудника 3) лично садился рядом с каждым новым сотрудником и проводил построчную «проверку удобочитаемости» его первого кода, отправляемого в репозиторий. Это была очень придирчивая проверка, которая охватывала все — от способов улучшения кода до следования соглашениям об использовании пробелов и отступов. Благодаря таким проверкам база кодов в Google приобрела единообразный вид, но, что особенно важно, в ходе обзора новые сотрудники учились передовым практикам, узнавали, что из общей инфраструктуры им доступно, и получали наглядное представление о том, как следует писать код в Google.

Со временем поток людей, нанимаемых в Google, вырос настолько, что один человек уже не мог уделить внимание всем новичкам. Многие инженеры нашли прошлый опыт настолько ценным, что добровольно приняли участие в программе. Сегодня около 20 % инженеров Google участвуют в процессе поддержки удобочитаемости как рецензенты или авторы кода.

В чем заключается процесс поддержки удобочитаемости?

Код-ревью обязателен в Google. Каждый список изменений (CL, changelist)¹ требует одобрения удобочитаемости владельцем *сертификата удобочитаемости*. Сертифицированные авторы неявно подтверждают удобочитаемость своих собственных CL, а CL других авторов одобряют один или несколько квалифицированных рецензентов. Это требование появилось после того, как компания Google выросла до такой степени, что стало невозможно обеспечить получение каждым инженером обзоров кода, которые учили бы передовым практикам с желаемой степенью строгости.



Подробнее о том, как в Google осуществляются обзоры кода и что в этом контексте означает «одобрение», в главе 9.

¹ Список изменений (changelist) — это список файлов в VCS, в которые вносились изменения. Список изменений является синонимом *набора изменений* (changeset, <https://oreil.ly/9jkgp>).

Обладание сертификатом удобочитаемости в Google обычно называется «владением удобочитаемостью» для некоторого языка. Инженеры, владеющие удобочитаемостью, демонстрируют умение писать понятный, идиоматичный и легкосопровождаемый код, который иллюстрирует лучшие практики Google и лучший стиль оформления кода для данного языка. Для этого централизованная группа *рецензентов удобочитаемости* проводит CL через процесс оценки удобочитаемости, после которого сообщает, насколько представленные изменения демонстрируют уровень мастерства автора. По мере того как авторы усваивают рекомендации по удобочитаемости, они получают все меньше и меньше замечаний, пока наконец сами не получают сертификат удобочитаемости. Обладателю сертификата доверяют продолжать применять свои знания в своем коде и выступать в роли рецензента кода другого инженера.

Примерно 1–2 % инженеров Google являются рецензентами удобочитаемости. Все рецензенты — это добровольцы, и любой, кто получит сертификат удобочитаемости, может выдвигать себя на роль рецензента удобочитаемости. Рецензенты придерживаются самых высоких стандартов, потому что от них ожидается не только глубокое знание языка, но и способность обучать. Они должны воспринимать поддержку удобочитаемости прежде всего как процесс наставничества и сотрудничества, а не контроля или состязания. Рецензентам и авторам CL рекомендуется дискутировать в процессе рецензирования. Рецензенты добавляют ссылки в свои комментарии, чтобы авторы могли узнать мотивы, по которым те или иные рекомендации вошли в руководство по стилю («забор Честерсона»). Если обоснование какой-то рекомендации неясно, авторы должны просить разъяснений («задавать вопросы»).

Поддержка удобочитаемости — это процесс, управляемый человеком, цель которого — масштабировать знания стандартизированным и в то же время персонализированным способом. Дополняя комплекс письменных и коллективных знаний, удобочитаемость сочетает в себе преимущества документации, доступ к которой можно получить с помощью ссылок, и общения с наставниками, которые знают, на какие фрагменты руководства лучше сослаться. Канонические и языковые рекомендации полностью документированы (и это хорошо!), но объем информации настолько велик¹, что усвоить его трудно, особенно новичкам.

Зачем нужен этот процесс?

Код читают множество людей, особенно в масштабах Google и нашего (очень большого) монолитного репозитория². Любой инженер может заглянуть в код других

¹ По состоянию на 2019 год одно только руководство по стилю для C++ насчитывало 40 страниц. Дополнительный документ, описывающий совокупность передовых практик, во много раз длиннее.

² Узнать, почему в Google используется монолитный репозиторий, можно здесь: <https://cacm.acm.org/magazines/2016/7/204032-why-google-stores-billions-of-lines-of-code-in-a-single-repository/fulltext>. Также имейте в виду, что не весь код в Google хранится в монолитном репозитории: процесс контроля удобочитаемости, о котором мы сказали, применяется только к монолитному репозиторию, потому что это понятие согласованности внутри репозитория.

команд и извлечь нужные уроки, а мощные инструменты, такие как Kythe (<https://kythe.io>), позволяют легко находить ссылки по всей кодовой базе (глава 17). Важной особенностью документирования передовых практик (глава 8) является обеспечение согласованных стандартов для всего кода в Google. Удобочитаемость является механизмом как соблюдения, так и распространения стандартов.

Одно из основных преимуществ программы поддержки удобочитаемости заключается в том, что она дает инженерам больше, чем просто коллективные знания, накопленные их командой. Чтобы получить сертификат удобочитаемости, инженеры должны отправлять CL централизованной команде рецензентов, которые проверяют код всей компании. Централизация проверки дает существенные выгоды: программа ограничена линейным масштабированием с ростом организации, но позволяет упростить соблюдение согласованности, препятствует появлению островков и (часто непреднамеренному) отклонению от установленных норм.

Значение согласованности всей кодовой базы невозможно переоценить: даже если десятки тысяч инженеров будут писать код десятилетиями, согласованность гарантирует, что код на данном языке будет выглядеть одинаково во всем корпусе. Это позволяет читателям сосредоточиться на том, что делает код, а не на том, как он выглядит. Авторам масштабных изменений (глава 22) легче вносить изменения по всему монолитному репозиторию для тысяч команд. Люди могут переходить из команды в команду и быть уверенными, что способ использования языка в новой команде не сильно отличается от способа, применявшегося в предыдущей команде.

Эти преимущества имеют свою цену: поддержка удобочитаемости — это сложный процесс по сравнению с другими средствами распространения знаний, такими как документация и учебные курсы, потому что она обязательна и обеспечивается инструментами Google (глава 19). Эти затраты нетривиальны и выражаются:

- в дополнительных сложностях для команд, в которых нет рецензентов удобочитаемости, потому что им приходится искать рецензентов вне своей команды, чтобы получить одобрение удобочитаемости для CL;
- в дополнительных затратах времени на проверку кода авторов, которые сами не являются рецензентами;
- в линейном масштабировании специфической работы узкого круга рецензентов.

Перевешивает ли выгода эти затраты? Нужно учитывать фактор времени: выгоды и затраты имеют разный временной охват. Программа сознательно идет на краткосрочные затраты, вызванные задержками на проверку кода, чтобы получить долгосрочные выгоды в виде высококачественного кода, согласованного по всему репозиторию, и накопление опыта инженерами. Долгосрочные преимущества обусловлены ожиданием, что написанный код будет служить годы, если не десятилетия¹.

¹ По этой причине код, если известно, что он будет существовать недолго, освобождается от требований удобочитаемости. Примерами могут служить каталог `experimental/` (явно предназначенный для экспериментального кода, который не может использоваться в про-

В большинстве инженерных процессов всегда есть возможность что-то улучшить. Некоторые затраты можно уменьшить с помощью инструментов. Многие проблемы удобочитаемости могут обнаруживаться статически и автоматически комментироваться с помощью инструментов статического анализа. Благодаря постоянному развитию таких инструментов рецензенты удобочитаемости могут фокусироваться на аспектах более высокого порядка: делать код понятным сторонним читателям, зная, что, например, пробелы в конце строки будут обнаружены автоматизированными средствами.

Но одного стремления к лучшему недостаточно. Удобочитаемость вызывает споры: некоторые инженеры считают, что это ненужное бюрократическое препятствие и неэффективное использование рабочего времени инженера. Действительно ли выгоды от программы поддержки удобочитаемости перевешивают затраты на нее? За ответом мы обратились к нашей команде по исследованию продуктивности (EPR, *engineering productivity research*).

Команда EPR провела углубленные исследования положительных и отрицательных сторон программы поддержки удобочитаемости, в том числе выяснив, мешает ли этот процесс людям, научились ли они чему-нибудь или изменили ли свое поведение в ходе участия в программе. Исследования показали, что процесс поддержки удобочитаемости оказывает положительное влияние на скорость разработки. На проверку CL авторов, овладевших навыками удобочитаемости, обычно уходит меньше времени¹. Удовлетворенность инженеров качеством своего кода, по их личным отзывам (в отсутствие более объективных показателей качества), выше среди тех, кто участвовал в программе удобочитаемости. Значительное большинство инженеров, прошедших программу, сообщают об удовлетворенности процессом и считают его стоящим. Они утверждают, что многому научились у рецензентов и изменили свое поведение, стараясь избегать проблем с удобочитаемостью при разработке и рецензировании кода.



Подробное описание этого исследования и внутреннего исследования продуктивности инженеров в Google в главе 7.

В Google очень сильная культура рецензирования кода, и удобочитаемость является естественным ее продолжением. Поддержка удобочитаемости выросла из страсти одного инженера в формальную программу с участием экспертов, передающих знания всем инженерам Google. Эта программа развивалась и изменялась с ростом Google и будет продолжать развиваться по мере изменения потребностей Google.

дуктах) и программа Area 120 (<https://area120.google.com>) — творческая мастерская для экспериментальных продуктов Google.

¹ На этот показатель влияет множество факторов, включая срок работы в Google и тот факт, что для проверки списков изменений авторов, не владеющих удобочитаемостью, обычно требуется проводить дополнительные раунды рецензирования.

Заключение

Знания в некотором смысле являются наиболее важным (хотя и нематериальным) капиталом организации, занимающейся разработкой ПО, и обмен этими знаниями имеет решающее значение для устойчивости организации и ее готовности к изменениям. Культура, способствующая открытому и честному обмену знаниями, обеспечивает эффективное распространение информации и позволяет организации расширяться. В большинстве случаев инвестиции в упрощение обмена знаниями приносят многократные дивиденды в течение жизни компании.

Итоги

- Чувство *психологической безопасности* является основой для создания среды обмена знаниями.
- Начните с малого: задавайте вопросы и записывайте ответы.
- Дайте людям возможность получать помощь, облегчив доступ к экспертам и справочным руководствам.
- Систематически поощряйте и вознаграждайте тех, кто уделяет время распространению знаний и опыта в своей команде или в организации.
- Универсального рецепта передачи информации не существует: расширение возможностей обмена знаниями требует применения комплекса стратегий, а идеальное для вашей организации сочетание средств для обмена знаниями почти наверняка будет меняться со временем.

Инженерия равенства

*Автор: Демма Родригес
Редактор: Риона Макнамара*

В предыдущих главах мы исследовали различия между программированием — процессом разработки кода, решающего текущие проблемы, и программной инженерией — подходом, предполагающим более широкое применение инструментов, стратегий и процессов для решения текущих и будущих возможных проблем в коде, который может служить десятилетия. В этой главе мы обсудим уникальные обязанности инженера при разработке программных продуктов для широкого круга пользователей и оценим, как организация, используя разнообразие подходов, может проектировать системы «для всех».

Мы все еще довольно слабо представляем, какое влияние программная инженерия оказывает на различные группы людей и общества, и все еще учимся понимать, как создавать продукты, которые расширяют возможности и уважают всех пользователей. У нас было много громких неудач в отношении наиболее уязвимых пользователей, и поэтому мы решили написать эту главу: путь к более качественным продуктам начинается с оценки неудач и стимулирования роста.

Этой главой мы также хотим подчеркнуть растущий дисбаланс сил между теми, кто принимает решения, и теми, на кого эти решения влияют (в частности, маргинализованные сообщества). Важно поразмыслить над тем, что мы узнали, и поделиться идеями с инженерами-программистами следующего поколения, чтобы они работали лучше, чем мы сегодня.

Тот факт, что вы взяли в руки эту книгу, скорее всего, означает ваше стремление стать выдающимся инженером. Вы хотите решать проблемы. Стремитесь создавать продукты, обеспечивающие положительные результаты для самого широкого круга людей, включая тех, кому трудно угодить. Представьте, как будут использоваться инструменты, созданные вами, чтобы изменить траекторию развития человечества.

Предвзятость — это проблема

Даже самые талантливые инженеры рискуют подвести пользователей, не учитывая их национальные, этнические, расовые, гендерные, возрастные и социально-экономические особенности, а также их способности и убеждения. Они могут сделать это

намеренно: у всех людей есть предубеждения, и в последние несколько десятилетий социологи не раз признавали, что большинство людей проявляет бессознательную предвзятость, навязывая и пропагандируя стереотипы. Бессознательная предвзятость коварна, и часто ее легче просто исключить, чем смягчить. Даже желая поступать правильно, мы можем не осознавать своих предубеждений. На организационном уровне важно понимать, что предубеждения существуют, и стараться их исключить из разработки продуктов и работы с пользователями.

Из-за предвзятости компании Google иногда не удавалось справедливо представить пользователей в своих продуктах и уделить достаточное внимание недопредставленным группам. Многие пользователи связывают это с тем, что основной контингент наших инженеров составляют мужчины, белые или азиаты, которые, конечно, не представляют все сообщества, использующие наши продукты. Отсутствие представителей всевозможных групп пользователей среди наших сотрудников¹ означает, что мы не обладаем необходимым разнообразием для понимания, какое влияние может оказывать использование наших продуктов.

КЕЙС: В GOOGLE НЕДОСТАТОЧНО ПОЛНО ПРЕДСТАВЛЕНЫ РАЗНЫЕ РАСОВЫЕ ГРУППЫ

В 2015 году инженер-программист Джеки Алсине (Jacky Alciné) заметил², что алгоритмы распознавания изображений в Google Photos классифицируют его темнокожих друзей как «горилл». Google не спешила реагировать на эти ошибки.

Чем вызван такой феерический провал? Тому есть несколько причин:

- Алгоритмы распознавания изображений зависят от наличия «правильного» (часто имеется в виду «полного») набора данных. Набор фотографий, на которых обучался алгоритм распознавания изображений в Google, был явно неполным — данные не представляли всего населения планеты.
- В самой компании Google (и технической индустрии в целом) негроидная раса недостаточно широко представлена³, и это повлияло на субъективные решения при разработке алгоритмов и сборе данных. Бессознательная предвзятость самой организации привела к тому, что был выпущен недостаточно готовый продукт.
- Мы не включили в потенциальную целевую аудиторию Google Photos недопредставленные группы. Тесты Google не заметили ошибки, которые затем увидели пользователи и справедливо расценили как проявление неуважения.

Даже в конце 2018 года мы в Google все еще не могли адекватно решить эту проблему⁴.

¹ Ежегодный отчет о разнообразии в Google за 2019 год (<https://diversity.google/annual-report>).

² @jackyalcine. 2015. «Google Photos, Y'all Fucked up. My Friend's Not a Gorilla». Twitter, June 29, 2015. <https://twitter.com/jackyalcine/status/615329515909156865>.

³ Во многих отчетах за 2018–2019 годы отмечалось отсутствие разнообразия в высокотехнологических сферах. В числе наиболее известных можно назвать: National Center for Women & Information Technology (<https://oreil.ly/P9ocC>) и Diversity in Tech (<https://oreil.ly/Y1pUW>).

⁴ Simonite T. When It Comes to Gorillas, Google Photos Remains Blind. *Wired*, January 11, 2018.

Наш продукт, упомянутый в этом примере, был неверно спроектирован и реализован, он не учитывал должным образом все расовые группы и в результате подвел наших пользователей и вызвал плохие отзывы о Google в прессе. Другие наши продукты тоже страдают от подобных провалов: функция автозаполнения может вернуть оскорбительные или расистские результаты. Рекламная система Google может использоваться для показа расистских или оскорбительных объявлений. YouTube может не выявить ненавистнических высказываний, которые технически запрещены на этой платформе.

Во всех этих случаях технологии не виноваты. Например, функция автозаполнения не предназначена для каких-то конкретных групп пользователей или их дискриминации. Но ее конструкция оказалась недостаточно гибкой, чтобы исключить дискриминационные высказывания, которые считаются языком ненависти. Вред для самой компании Google также очевиден: снижение доверия пользователей и сокращение сотрудничества с компанией. Например, соискатели из числа темнокожих, латиноамериканцев и евреев могут потерять веру в Google как платформу или даже в возможность существования всеохватывающей среды, что подрывает цель Google — достижение сбалансированного представления разных групп при найме.

Как такое могло случиться? В конце концов, Google стремится нанимать специалистов с безупречным образованием или профессиональным опытом — выдающихся программистов, которые пишут лучший код и проверяют свою работу. Девиз Google: «Создавай для всех», — но правда в том, что нам предстоит пройти еще долгий путь, прежде чем мы сможем подтвердить его. Одно из решений этих проблем — помочь самой организации, занимающейся программной инженерией, иметь в своих рядах представителей всех групп людей, для которых мы создаем продукты.

Понимание необходимости разнообразия

Мы в Google считаем, что выдающийся инженер должен быть сосредоточен на привлечении различных точек зрения в разработку и внедрение продуктов. Это также означает, что гуглеры, отвечающие за собеседование и прием на работу других инженеров, должны содействовать расширению представления в компании различных групп. Например, проводя собеседование, важно выяснить, как решения о приеме на работу влияют на баланс представленных групп. Чтобы действительно создавать продукты для всех, мы должны поощрять более широкий спектр образовательной подготовки у наших инженеров.

Прежде всего важно разрушить представление о том, что кандидат с ученой степенью и/или опытом работы в области computer science по определению обладает всеми навыками, необходимыми выдающемуся инженеру. Также важно избавиться от идеи, что только люди с образованием в области computer science могут проектировать и создавать продукты. В настоящее время большинство программистов имеет такое образование (<https://oreil.ly/2Bu0H>): они с успехом пишут код, создают теории, изменяющие мир, и применяют методологии для решения проблем. Однако, как

показывают вышеупомянутые примеры, *этого недостаточно для всеохватывающей и равноправной инженерии.*

Инженеры должны учитывать в работе все особенности экосистемы, на которую они стремятся влиять. Как минимум понимать демографические характеристики пользователей, уделять внимание людям, которые отличаются от них самих, особенно тем, которые могут использовать выпускаемые продукты для причинения вреда другим. Сложнее всего учитывать особенности людей, лишенных права голоса в среде, где они получают доступ к технологиям. Для этого инженеры должны понимать интересы представителей всех существующих и потенциальных групп пользователей. В отсутствие разнообразного представления групп в командах каждый инженер должен учиться создавать продукты, пригодные для всех пользователей.

Выстраивание мультикультурного потенциала

Отличительной чертой выдающегося инженера является *проницательность* — способность понять, как продукты могут приносить пользу и вред различным группам людей, и умение выявлять и отвергать функции или продукты, которые приводят к неблагоприятным результатам. Это высокая и труднодостижимая цель, поскольку становление высокопроизводительного инженера требует индивидуального подхода. Тем не менее, чтобы добиться успеха, мы должны принять во внимание не только наши сообщества, но и миллиарды новых пользователей, которые появятся в будущем, а также нынешних пользователей, которые могут быть лишены права голоса или оставаться неучтенными в наших продуктах.

Со временем вы можете создать инструменты, которыми ежедневно пользуются миллиарды людей, влияющие на представления о ценности человеческой жизни, контролирующие человеческую деятельность и собирающие и сохраняющие конфиденциальные данные, например фотографии детей и близких. Как инженер вы можете обладать большей властью, чем думаете, и прямо влиять на общество. Важно, чтобы вы понимали ответственность и не использовали свою власть во вред кому-либо. Для начала осознайте свою предвзятость, вызванную многими социальными и образовательными факторами. После этого вы сможете замечать упускаемые из виду варианты использования, из-за которых ваши продукты могут быть полезны или вредны.

Технологии продолжают развиваться с возрастающей скоростью, создавая почву для новых сценариев использования искусственного интеллекта (ИИ) и машинного обучения. Чтобы оставаться конкурентоспособными, мы стремимся к увеличению охвата и эффективности в создании высококвалифицированных инженерно-технических кадров. Однако обратите внимание, что в настоящее время одни имеют возможность создавать будущие технологии, а другие — нет. Нам необходимо понять, будут ли способствовать создаваемые нами программные системы процветанию общества и обеспечению равного доступа к технологиям.

Исторически сложилось, что компании стремятся к быстрому достижению стратегической цели, связанной с получением прибыли и доминированием на рынке,

отвергая исследования, которые этот процесс замедляют. Они ценят производительность и превосходство, но зачастую не могут действовать беспристрастно. Внимание к недопредставленным группам пользователей дает нам возможность двигаться навстречу справедливости. Чтобы сохранить конкурентоспособность в технологическом секторе, нам необходимо научиться создавать глобальное равенство.

Сегодня нас волнуют разработки технологий для сканирования, захвата и идентификации изображений людей, идущих по улице. Мы беспокоимся о конфиденциальности и о том, как правительства могут использовать эту информацию сейчас и в будущем. Однако большинство наших специалистов не имеют необходимых знаний о недопредставленных группах, чтобы понять влияние расовых различий на процедуру распознавания лиц и оценить корректность результатов, полученных от ИИ.

В настоящее время ПО для распознавания лиц по-прежнему ставит в невыгодное положение этнические меньшинства или людей с другим цветом кожи. Наше исследование недостаточно полно и не включает весь спектр различных оттенков кожи. Нельзя ожидать надежных результатов, если обучающие данные и создатели ПО представляют лишь малую часть пользователей. Мы должны быть готовы отложить разработку и попытаться получить более полные и точные данные, чтобы создать всеобъемлющий продукт.

Однако наука о данных сама по себе сложна. Даже при наличии достаточно полного представления некой группы в команде разработчиков обучающий набор все еще может быть предвзятым и давать неверные результаты. Исследование, завершенное в 2016 году, показало, что в базе данных по распознаванию лиц в правоохранительных органах находятся более 117 миллионов взрослых американцев¹. Из-за непропорционального внимания полиции к темнокожим при использовании такой базы данных для распознавания лиц могут иметь место расовые ошибки. Несмотря на то что ПО постоянно развивается и совершенствуется, независимое тестирование фактически отсутствует. Для исправления этой вопиющей оплошности нужно иметь честность, чтобы приостановиться и сформировать входные данные так, чтобы они были как можно менее предвзятыми. В настоящее время Google предлагает идею статистического обучения ИИ, чтобы гарантировать непредвзятость наборов данных.

Таким образом, смещение фокуса в отраслевом опыте в сторону комплексного мультикультурного расового и гендерного образования — это не только *ваша* ответственность, но и *ответственность вашего работодателя*. Технологические компании должны следить за тем, чтобы профессиональное развитие их сотрудников было комплексным и многопрофильным. Требование не в том, чтобы один человек взял на себя обязательство изучать другие культуры или демографию в одиночку. Каждый из нас как специалист или руководитель команды должен вносить свой

¹ *Gaines S., Williams S. The Perpetual Lineup: Unregulated Police Face Recognition in America. Center on Privacy & Technology at Georgetown Law, October 18, 2016.*

вклад в непрерывное профессиональное совершенствование, которое развивает не только навыки разработки ПО и лидерские качества, но также способность понимать разнообразный опыт всего человечества.

Сделать разнообразие действенным

Всеобщие равенство и справедливость достижимы, если мы готовы признать, что все несем ответственность за постоянную дискриминацию в технологическом секторе и провалы в системе. Откладывание или абстрагирование личной ответственности неэффективно, а в некоторых случаях даже губительно. Также безответственно полностью приписывать динамику в компании или команде более широким социальным проблемам, способствующим неравенству. Любимая фраза как сторонников разнообразия, так и его недоброжелателей звучит примерно так: «Мы усердно работаем над исправлением (вставьте сюда «проблему системной дискриминации»), но учесть все очень трудно. Как победить (вставьте сюда «многовековую») исторически сложившуюся дискриминацию?» Это направление уводит нас в сторону философской или академической беседы и не связано с целенаправленными усилиями по улучшению условий или результатов труда. Для создания мультикультурного потенциала требуется более полное понимание того, как неравенство в обществе влияет на рабочие места, особенно в технологическом секторе.

Если вы инженер-руководитель, стремящийся принимать на работу больше людей из недопредставленных групп, то должны учитывать историческое влияние дискриминации в мире. При этом важно выйти за рамки академической беседы и сосредоточиться на количественных и практических шагах для обеспечения равенства и справедливости. Например, занимаясь наймом инженеров-программистов, вы несете ответственность за сбалансированность кадров. Есть ли женщины или другие недопредставленные группы в списке кандидатов? Какие возможности для роста вы предоставили после найма кого-то и является ли распределение возможностей справедливым? У каждого технического лидера или управленца в сфере программной инженерии есть средства увеличения равенства в командах. Важно понимать, что даже при наличии серьезных системных проблем мы все являемся частью этой системы. Это наша проблема, и мы должны ее исправить.

Отказ от единых подходов

Мы не можем использовать какую-то одну методологию для устранения неравенства в технологическом секторе. Наши проблемы сложны и многообразны. Поэтому мы должны отказаться от единого подхода к представлению различных групп на рабочих местах, даже если он продвигается людьми, вызывающими восхищение или имеющими институциональную власть.

Одна из наиболее важных особенностей высокотехнологичной индустрии заключается в том, что недостаток представления определенных групп среди работников

можно устранить, только исправив процедуру найма. Да, это фундаментальный шаг, и мы не сделаем его сразу. В наших силах признать системное неравенство и сосредоточить внимание, например, на более репрезентативном найме и образовательном неравенстве по расовым, гендерным, социально-экономическим признакам и иммиграционному статусу.

В высокотехнологичной индустрии многие из недопредставленных групп ежедневно лишаются возможностей и продвижения по службе. Самая высокая текучесть кадров в Google происходит среди темнокожих сотрудников (<https://oreil.ly/JFbTR>) и мешает достижению целей более полного их представления. Если мы хотим добиться изменений в этой области, необходимо оценить, создаем ли мы экосистему, в которой могут процветать все начинающие инженеры и другие специалисты в области технологий.

Полное понимание предметной области имеет решающее значение для определения путей исправления ситуации. Это относится ко всему — от миграции критически важных данных до найма новых сотрудников. Например, если вы, как руководитель, хотите нанять больше женщин, недостаточно сосредоточиться на выстраивании процесса найма. Нужно также уделить внимание другим аспектам экосистемы найма и продвижения по службе и подумать, способны ли кадровики выявлять сильных кандидатов, как женщин, так и мужчин. Если вы управляете разнородной командой инженеров, сосредоточьтесь на психологической безопасности и вкладывайте средства в расширение мультикультурного потенциала в команде, чтобы новые ее члены чувствовали себя востребованными.

Современная общепринятая методология заключается в том, чтобы сначала создать сценарий для большинства вариантов использования, оставив улучшения на потом. Но этот подход несовершенен: он дает преимущества пользователям, которые уже имеют преимущества в доступе к технологиям, и увеличивает неравенство. Откладывать рассмотрение всех групп пользователей до момента завершения проектирования означает понизить планку качества работы инженера. Вместо этого мы с самого начала создаем всеохватывающий дизайн и повышаем стандарты разработки, чтобы сделать инструменты интересными и доступными для *всех*, особенно для людей, которые борются за доступ к технологиям.

Ориентирование на пользователя, который меньше всего похож на вас, не просто полезно — это лучшая практика. Существуют прагматичные и непосредственные шаги, которые все специалисты независимо от предметной области должны учитывать при разработке продуктов, не сторонясь неприятных или непонятных им пользователей. Все начинается со всестороннего исследования пользовательского опыта. Это исследование должно проводиться с пользователями, которых отличают язык, место жительства, культура, принадлежность к социально-экономическому классу, возраст и способности. Сначала сосредоточьтесь на самом сложном сценарии использования.

Бросьте вызов устоявшимся процессам

Под созданием более справедливых систем подразумевается не только разработка всеобъемлющих спецификаций продуктов, но и вызов устоявшимся процессам, приводящим к неверным результатам.

Рассмотрим недавний случай и оценим его с точки зрения справедливости. Несколько команд инженеров в Google работали над разработкой глобальной системы вакансий. Система поддерживала не только наем сотрудников извне, но и перемещения внутри компании. Инженеры и руководители проекта проделали большую работу, выслушав запросы тех, кого они считали своей основной группой пользователей: рекрутеров. Рекрутеры хотели сэкономить время менеджеров по персоналу и соискателей и передали команде разработчиков примеры использования, ориентированные на охват и эффективность. В частности, они попросили инженеров добавить функцию выделения рейтингов эффективности (особенно низких) сразу после того, как внутренний соискатель выразит заинтересованность в работе.

На первый взгляд ускорение процесса оценки и экономия времени соискателей — достойная цель. Так в чем же несправедливость? Были подняты следующие вопросы:

- Можно ли использовать оценки эффективности в прошлом для прогнозирования эффективности в будущем?
- Являются ли оценки эффективности свободными от индивидуальной предубежденности?
- Стандартизованы ли оценки эффективности в разных организациях?

Ответ «нет» на любой из этих вопросов означает, что представление рейтингов эффективности может приводить к несправедливым и, следовательно, неверным результатам.

Когда выдающийся инженер усомнился в том, что оценки эффективности в прошлом способны предсказать эффективность в будущем, команда рецензентов решила провести тщательный анализ. В итоге было установлено, что кандидаты, получившие низкий рейтинг эффективности, почти наверняка улучшат этот рейтинг, если найдут новую команду. Фактически они с той же вероятностью могли бы достичь удовлетворительного или высокого рейтинга эффективности, как и кандидаты, которые никогда не имели низкого рейтинга. Проще говоря, эти рейтинги сообщают только, *как человек выполнял свою работу на момент его оценивания*, и не должны использоваться для оценки пригодности внутреннего кандидата для работы в другой команде. (Однако их можно использовать для оценки, насколько сотрудник подходит для текущей команды и где он может продвигаться вперед.)

На этот анализ ушло значительное время, но в результате был реализован более справедливый процесс внутренней миграции.

Ценности и результаты

Google имеет большой опыт инвестирования в сферу найма персонала. Как показывает предыдущий пример, мы постоянно оцениваем процессы, чтобы повысить их справедливость и охват. В более широком смысле наши основные ценности основаны на уважении и непоколебимой приверженности разнообразию работников. Тем не менее год за годом мы упускали свой шанс принимать на работу представителей разных групп, являющихся отражением наших пользователей по всему миру. Борьба за улучшение справедливости результатов продолжается, несмотря на проводимую политику и программы, направленные на поддержку инициатив по увеличению разнообразия и совершенствованию процедур найма и продвижения по службе. Причина неудачи не в ценностях, намерениях или инвестициях компании, а скорее в *реализации* политик.

Старые привычки трудно сломать. Пользователи, которых можно привлечь для разработки и от которых мы привыкли получать отзывы, могут не представлять всех пользователей, с которыми мы хотим наладить контакт. Это имеет место в отношении всех видов продуктов, от носимых устройств, которые не подходят женщинам, до ПО для видеоконференций, которое не всегда подходит людям с темным цветом кожи.

Так какой же вывод отсюда следует?

1. **Внимательно посмотрите в зеркало.** Наш девиз: «Создавай для всех». Как это сделать в отсутствие репрезентативной рабочей силы или модели взаимодействий, централизующей отзывы сообщества? Это невозможно. Правда в том, что иногда мы не могли публично защитить наших наиболее уязвимых пользователей от расистского, антисемитского и гомофобного контента.
2. **Не нужно создавать для всех подряд. Создавайте со всеми.** Мы пока не создаем для всех. Но мы работаем не в вакууме, и, конечно, нельзя сказать, что наши технологии не репрезентативны для общества в целом. Мы уже не можем что-то создать и об этом забыть. Итак, как создавать для всех? Мы стараемся вовлекать в процесс создания наших пользователей — представителей разных групп — и ставим наиболее уязвимые сообщества в центр нашей работы. Мы не должны учитывать их мнения по остаточному принципу.
3. **При проектировании учитывайте пользователей, которые будут испытывать наибольшие сложности при использовании продукта.** Создание для тех, кто может столкнуться с дополнительными сложностями, сделает продукт лучше для всех. Не гонитесь за быстрой прибылью.
4. **Не стройте предположений о справедливости: оценивайте ее во всех системах.** Поймите, что лица, принимающие решения, также подвержены предвзятости и могут быть недостаточно осведомлены о причинах неравенства. У вас может не хватить опыта, чтобы определить или оценить масштаб проблемы неравенства.

Удовлетворение нужд одной группы пользователей может означать ущемление прав другой группы, и часто эти компромиссы трудно обнаружить и невозможно повернуть вспять. Сотрудничайте с экспертами в области разнообразия, справедливости и интеграции.

5. **Перемены возможны.** Проблемы, с которыми мы сталкиваемся сегодня в области высоких технологий, от дезинформации до преследования в интернете, поистине огромны. Мы не можем решить их, используя неудачные подходы из прошлого или только имеющиеся навыки. Нам нужно измениться.

Оставайтесь любознательными, двигайтесь вперед

Путь к справедливости долог и сложен. Однако мы можем и должны перейти от простого создания инструментов и услуг к более глубокому пониманию, как наши продукты влияют на человечество. Бросьте вызов своему образованию, влияйте на свои команды и руководителей, проводите более всестороннее исследование пользователей — все это позволит добиться прогресса. Перемены сопряжены с неудобствами, а путь к высокой эффективности может быть болезненным, но они возможны благодаря сотрудничеству и творчеству.

Наконец, как будущие выдающиеся инженеры мы должны в первую очередь сосредоточиться на пользователях, наиболее подверженных влиянию предвзятости и дискриминации. Мы можем вместе работать над ускорением прогресса, сосредоточившись на постоянном совершенствовании и анализе наших неудач. Становление инженера — сложный и непрерывный процесс, цель которого состоит в том, чтобы инициировать перемены, продвигающие человечество вперед без ущемления чьих-либо прав. Как будущие выдающиеся инженеры мы верим, что сможем предотвратить неудачи.

Заключение

Разработка ПО и формирование организации, занимающейся разработкой ПО, — это командная работа. По мере увеличения масштаба организации необходимо адекватно формировать базу пользователей, которая в современном взаимосвязанном компьютерном мире включает людей, географически удаленных друг от друга. Необходимо приложить больше усилий, чтобы команды, разрабатывающие ПО, и создаваемые ими продукты в полной мере отражали ценности такого разнообразного и всеохватывающего множества пользователей. И если софтверная организация хочет развиваться и расти, она не может игнорировать недопредставленные группы. Инженеры из этих групп могут не только дополнять саму организацию, но и предоставлять уникальные перспективы для разработки и внедрения ПО, действительно полезного для всего мира.

Итоги

- Предвзятость — это проблема.
- Разнообразие кадров необходимо для правильного формирования всеохватывающей базы пользователей.
- Всеохватность имеет решающее значение не только для совершенствования процесса найма и расширения представительства недопредставленных групп, но и для создания действительно благоприятной среды для всех людей.
- Скорость разработки продукта должна регулироваться с учетом его полезности для всех пользователей. Лучше приостановить работу, чем выпустить продукт, который может причинить вред некоторым пользователям.

Как стать лидером в команде

Автор: Брайан Фицпатрик
Редактор: Риона Макнамара

Итак, мы познакомились с культурой и составом команд, пишущих ПО, и в этой главе мы обсудим человека, который в конечном счете несет ответственность за выполнение работы.

Никакая команда не сможет работать слаженно без лидера, особенно в Google, где инженерия — это почти исключительно командная работа. Мы в Google различаем две лидерские роли: *руководитель*, управляющий людьми, и *технический лидер*, направляющий технические усилия. Обязанности в этих ролях весьма различны, но обе они требуют определенных навыков.

Корабль без капитана — это такой плавучий зал ожидания: если кто-то не возьмет руль и не запустит двигатель, корабль будет бесцельно плыть по течению. Разработка ПО отчасти похожа на этот корабль: если никто не управляет ею, инженеры потратят драгоценное время на ожидание или, что еще хуже, будут писать код, который никому не нужен. Хотя эта глава посвящена управлению людьми и техническому лидерству, ее стоит прочитать всем разработчикам, чтобы лучше понять руководителей.

Руководители и технические лидеры (и те и другие)

В любой команде инженеров обычно есть лидер. Иногда в команду приходит опытный руководитель, а иногда простой разработчик продвигается на роль лидера (обычно из небольшой команды).

В зарождающихся командах обе роли может играть один и тот же человек — *технический руководитель* (TLM, tech lead manager). В больших командах обязанности руководителя может выполнять опытный менеджер по персоналу, если старший инженер с большим опытом работы берет на себя функции технического лидера. Несмотря на то что руководитель и технический лидер играют важные роли в развитии команды, навыки, необходимые для достижения успеха в каждой роли, совершенно разные.

Руководитель

Многие компании привлекают для управления командами разработчиков обученных руководителей, которые могут почти ничего не знать о разработке ПО. Однако

с самого начала в Google было решено, что руководители команд, занимающихся программной инженерией, должны иметь опыт инженерной работы. Это предполагает прием на работу опытных руководителей, которые раньше были инженерами-программистами, или обучение инженеров-программистов руководству (подробнее об этом ниже).

На самом высоком уровне руководитель отвечает за эффективность, производительность и удовлетворенность каждого члена команды, включая технического лидера, и при этом следит за тем, чтобы продукт удовлетворял потребности бизнеса. Поскольку потребности бизнеса и потребности отдельных членов команды не всегда совпадают, руководитель часто находится в затруднительном положении.

Технический лидер

Технический лидер (TL, tech lead) команды — это человек, который отчитывается перед руководителем команды, отвечает (сюрприз!) за технические аспекты продукта, включая выбор технических решений, архитектуру, приоритеты, скорость работы и общее управление проектом (в крупных командах ему могут помогать руководители программ). Технический лидер обычно работает рука об руку с руководителем, чтобы укомплектовать команду кадрами и распределить задачи между инженерами в соответствии с их навыками и знаниями. Большинство технических лидеров одновременно являются разработчиками, что часто ставит их перед выбором: что-то быстро сделать самому или делегировать работу члену команды, который справится с ней медленнее. Последнее чаще всего является правильным решением, поскольку помогает увеличить уровень способностей и возможности команды.

Технический руководитель

В небольшой и только зарождающейся команде, где лидер должен обладать обширными техническими навыками, руководство часто осуществляется техническим руководителем — одним человеком, способным руководить людьми и управлять техническими потребностями команды. Иногда технический руководитель — это самый старший член команды, который до недавнего времени был простым разработчиком.

В Google принято, чтобы крупные, хорошо зарекомендовавшие себя команды имели пару лидеров — технического лидера и руководителя, работающих вместе как партнеры, поскольку трудно совмещать обе роли без риска психологического выгорания.

Работа технического руководителя сложна и часто требует умения балансировать между выполнением работы разработчика, делегированием этой работы и управлением персоналом. Поэтому техническим руководителям часто требуются высокая степень наставничества и помощь со стороны более опытных технических руководителей. (Рекомендуем начинающему техническому руководителю помимо изучения курсов, которые предлагает Google по этой теме, найти более опытного наставника, который сможет регулярно консультировать его по мере вхождения в новую роль.)

КЕЙС: ВЛИЯНИЕ БЕЗ ПОЛНОМОЧИЙ

Конечно, вы можете заставить подчиненных выполнять работу, но совсем другое дело управлять теми, кто находится за пределами компании или даже занимается созданием других продуктов. Способность «влиять без полномочий» — одно из самых сильных лидерских качеств, которое вы можете развить.

Например, в течение многих лет Джефф Дин, старший инженер-разработчик и, пожалуй, самый известный гуглер *внутри* Google, возглавлял очень небольшую команду инженеров Google, но его умение влиять на технические решения и направления развития помогало достигать целей как внутри инженерной организации, так и за ее пределами (благодаря его статьям и публичным выступлениям).

Другой пример: в свое время я создал команду «The data liberation front», насчитывающую человек пять инженеров, которая смогла организовать экспорт данных более чем в 50 продуктов Google через созданный нами Google Takeout. Наши стремления не были подкреплены официальными приказами на исполнительном уровне Google. Так как же нам удалось сподвигнуть сотни инженеров внести свой вклад в нашу идею? Стратегическая потребность в идее, демонстрация ее связи с целями и приоритетами компании и взаимодействие с небольшой группой инженеров, занимающихся разработкой инструмента, помогли другим группам легко и быстро интегрироваться с Takeout.

Переход от роли разработчика к роли лидера

Назначенный официально или нет, кто-то должен сесть за руль. Если ваш продукт начал развитие и вы имеете достаточно мотивации и не хотите ждать, тогда этим «кем-то» можете стать вы. Вас могут втянуть в разрешение конфликтов, принятие решений и координацию действий другие члены команды. Это происходит постоянно и часто совершенно случайно. Возможно, вы не собирались становиться «лидером», но это произошло — вы заболели «менеджеритом».

Даже если вы поклялись никогда не руководить, в какой-то момент в своей карьере вы можете оказаться на руководящей должности, особенно если добьетесь успеха в роли разработчика. В остальной части этой главы мы поможем вам понять, что делать в такой ситуации.

Мы не собираемся убеждать вас стать руководителем, а просто хотим показать, что лучшими считаются лидеры, которые служат команде, используя принципы смирения, уважения и доверия. Понимание тонкостей лидерства является важным умением, помогающим влиять на направление работы. Если вы хотите управлять кораблем проекта, а не просто плыть на нем в роли пассажира, вам нужно знать, как ориентироваться в бурных водах, иначе вы сами (и проект) сядете на мель.

Единственное, чего нужно бояться, ...ЭМ-М, всего

Помимо общего чувства тревоги, которое испытывают многие, когда слышат слово «руководитель», существует еще масса причин, почему большинство людей не хотят становиться руководителями. Самая веская причина, которую можно услышать

в мире разработки ПО, — необходимость тратить гораздо меньше времени на создание кода. Это верно и для технических лидеров, и для руководителей, и я расскажу об этом далее в этой главе, но сначала мы рассмотрим еще несколько причин, по которым большинство из нас не горят желанием становиться руководителями.

Те, кто большую часть карьеры занимается созданием кода, обычно заканчивают день чем-то, что можно продемонстрировать (код, документ с описанием дизайна или целый список исправленных ошибок), и говорят: «Вот что я сделал сегодня». Но в конце напряженного дня, посвященного «управлению», нередко возникает мысль: «Сегодня я ни черта не сделал». Однако это равносильно тому, что, потратив годы на подсчет яблок, которые вы собирали каждый день, и перейдя на работу по выращиванию бананов, вы в конце каждого дня будете говорить себе: «Я не собрал ни одного яблока», забывая про цветущие банановые пальмы рядом с собой. Оценить количественно управленческую работу сложнее, чем подсчитать созданные виджеты, однако даже простая удовлетворенность и продуктивность команды являются ничуть не менее важной оценкой работы лидера. Просто не попадайтесь в ловушку с подсчетом яблок, занимаясь выращиванием бананов¹.

Еще одна важная причина уклонения от руководящей работы, часто не высказываемая прямо, корнями уходит в знаменитый «принцип Питера», который гласит: «Каждый сотрудник имеет тенденцию подняться до уровня своей некомпетентности». Чтобы избежать этого, в Google человеку предлагается выполнять более сложную работу (то есть «превзойти самого себя») в течение ограниченного времени, и только потом принять решение о переходе на новый уровень. Многие из нас сталкивались с руководителями, неспособными выполнять свою работу или просто плохо управлявшими людьми², и мы знаем тех, кому довелось работать только с плохими руководителями. Если всю свою карьеру человек работал только с плохими руководителями, *откуда* у него может возникнуть желание самому стать руководителем? Разве может возникнуть стремление получить роль, которую вы не можете выполнять?

Есть, однако, веские причины, чтобы стать техническим лидером или руководителем. Во-первых, это возможность масштабирования своих знаний и умений. Даже если вы прекрасно пишете код, его объем ограничен. А теперь представьте, сколько кода может написать команда великих инженеров под вашим руководством! Во-вторых, вы можете открыть в себе новые таланты — многие, оказавшись втянутыми в управление проектом, обнаруживают, что прекрасно справляются с выбором решений, оказанием помощи и обеспечением нужд команды или компании. Кто-то должен вести за собой, так почему не вы?

¹ Еще одно отличие, к которому нужно привыкнуть: плоды труда руководителя обычно зреют намного дольше.

² Не стоит принуждать людей становиться руководителями: если инженер способен писать отличный код и вообще не хочет управлять людьми или руководить командой, заставив его работать в роли руководителя или технического лидера, вы потеряете прекрасного инженера и получите плохого руководителя. Принуждение — не только плохая, но и по-настоящему вредная идея.

Лидерство как служение

Иногда кажется, что руководителей поражает что-то вроде болезни, когда они забывают обо всех ужасах, которые с ними творили их руководители, и внезапно начинают делать то же самое со своими подчиненными. Вот далеко не полный список симптомов этого заболевания: микроуправление, игнорирование неэффективных сотрудников и прием на работу слабых специалистов. Без своевременного лечения эта болезнь может убить всю команду. Лучший совет, который я получил, когда впервые занял руководящий пост в Google, дал мне Стив Винтер, занимавший в то время должность технического директора. Он сказал: «Прежде всего, сопротивляйся желанию управлять». Одно из главных побуждений новоиспеченного руководителя — активно «управлять» своими сотрудниками, потому что именно это должен делать руководитель, верно? Часто это устремление приводит к катастрофическим последствиям.

Избавиться от острых форм «менеджерита» поможет мантра: «Лидерство — это служение». Она прямо говорит, что самое ценное, что можно сделать в роли лидера, — служить своей команде, подобно тому как дворецкий или мажордом служит своему хозяину и заботится о его здоровье и благополучии. Как лидер-слуга вы должны стремиться создать атмосферу смирения, уважения и доверия. Это может выражаться в устранении бюрократических препятствий, с которыми не справляются члены команды, помощи в достижении согласия или даже покупке еды для тех, кто допоздна работает в офисе. Лидер-слуга засыпает ямы, чтобы облегчить путь своей команде, консультирует при необходимости, но при этом не боится испачкать руки. Единственное, чем должен управлять лидер-слуга, — это техническое обеспечение и социальное здоровье команды. Как бы ни было заманчиво сосредоточиться исключительно на техническом обеспечении, социальное здоровье команды столь же важно (но управлять им намного сложнее).

Руководитель

Итак, что же должен делать руководитель в современной софтверной компании? До эры повсеместного распространения компьютеров слова «управление» и «труд» описывали почти противоположные понятия. Руководитель располагал всей полнотой власти, чтобы направлять коллективный труд на достижение своих целей. Но современные софтверные компании работают иначе.

«Начальник» — почти бранное слово

Прежде чем говорить об основных обязанностях руководителя в Google, обратимся к истории. Современные представления о начальнике как об узурпаторе сформировались при военной иерархии и были подхвачены в ходе промышленной революции более ста лет назад! В то время повсюду появлялись фабрики, для обслуживания машин которых требовались рабочие (обычно неквалифицированные) и начальники — для управления. Спрос на работу был высок, и у начальников не было доста-

точно мотивации для хорошего отношения к подчиненным или улучшения условий их труда. Такие негуманные трудовые отношения успешно существовали много лет, пока работники выполняли относительно простые задачи.

Начальники часто обращались с сотрудниками как погонщики с мулами: если не удавалось приманить морковкой — били палкой. Метод кнута и пряника пережил переход от фабрик¹ к современным офисам, где в середине XX века среди работников, выполнявших одну и ту же работу много лет, процветал стереотипный образ невозмутимого начальника-погонщика.

В некоторых отраслях эта традиция продолжается и поныне, даже там, где требуются творческое мышление и умение решать проблемы, несмотря на многочисленные исследования, подтверждающие неэффективность и вредность кнута и пряника в творческих областях. В прошлом работники сборочного конвейера могли обучиться за несколько дней и быть быстро заменены по желанию начальника. Но современным разработчикам ПО могут потребоваться месяцы, чтобы освоиться в новой команде, поэтому они нуждаются в заботе, времени и пространстве, чтобы думать и творить.

Современный руководитель

Большинство людей все еще используют слово «начальник», несмотря на то что его первоначальное значение устарело. Само название часто побуждает новых руководителей *управлять* своими подчиненными. Руководители могут вести себя подобно родителям², на что сотрудники вполне естественно реагируют как дети. А теперь сформулируем понятие «руководить» в контексте смирения, уважения и доверия: если руководитель выказывает свое доверие сотруднику, тот испытывает позитивное давление и старается это доверие оправдать. Это так просто. Хороший руководитель прокладывает путь команде, заботясь о ее безопасности и благополучии, и в то же время следит, чтобы ее потребности были удовлетворены. Если хотите вынести какой-то общий урок из этой главы, тогда запомните следующее:

Рядовые начальники заботятся о том, как что-то сделать, а великие руководители — о том, что можно сделать (и доверяют своей команде выяснение деталей, как это сделать).

Несколько лет назад к моей команде присоединился новый инженер Джерри. Простой руководитель Джерри (в другой компании) был непреклонен, заставляя подчиненных сидеть за столами с 9:00 до 17:00 и считая, что если работник куда-то отлучался, значит, он работал не в полную силу (что, конечно, совершенно нелепо). В свой первый рабочий день Джерри пришел ко мне в 16:40 и пробормотал извинения

¹ Подробнее об оптимизации передвижения работников на фабрике читайте в статьях о научной организации труда или тейлоризме, где особо подчеркивается важность влияния на моральный дух работников.

² Если у вас есть дети, то вы наверняка вспомните, когда в первый раз сказали своему ребенку что-то, что заставило вас остановиться и воскликнуть (возможно, даже вслух): «Черт возьми, я веду себя как мои родители!»

за то, что вынужден уйти на 15 минут раньше, так как у него запланирована встреча, которую он не смог перенести. Я улыбнулся и сказал: «Пока вы выполняете свою работу, мне все равно, когда вы покинете офис». На несколько секунд Джерри замер в недоумении, затем кивнул и пошел дальше. Я относился к Джерри как ко взрослому — он всегда выполнял свою работу, и мне никогда не приходилось беспокоиться о том, сидит ли он за столом, потому что ему не нужна нянька. Если ваши сотрудники настолько не заинтересованы в работе, что их должен подгонять начальник-нянька, *это* и есть ваша настоящая проблема.

НЕУДАЧА — ТОЖЕ ВАРИАНТ

Еще один способ стимулировать команду — дать людям почувствовать себя защищенными, чтобы они начали рисковать. Если вам удастся создать в команде атмосферу психологической безопасности, ее члены смогут быть самими собой и не опасаться негативной реакции от вас или коллег. Риск — захватывающая штука, но большинство людей боятся рисковать, и во многих компаниях принято избегать риска любой ценой, выбирать консервативный подход к работе и стремиться к небольшим успехам, даже когда больший риск может дать многократно больший результат. В Google говорят, что, пытаясь достичь невозможного, вы, вероятно, потерпите неудачу, но, скорее всего, такая неудача даст вам больше, чем выполнение работы, которая точно вам под силу. Чтобы создать культуру, в которой допустим риск, команда должна знать, что неудачи — это нормально.

Итак, давайте примем раз и навсегда: терпеть неудачу — это нормально. На самом деле мы предпочитаем думать о неудаче как о быстром способе научиться чему-то (при условии, что одна и та же неудача не повторяется), а не обвинять в ней кого-то. Быстро потерпеть неудачу — это хорошо, потому что на карту поставлено не так много. Неудача после длительной работы тоже может преподать ценный урок, но она воспринимается болезненно, потому что при ней потери больше (обычно времени инженера). Таким образом, неудача, влияющая на клиентов, является самой нежелательной, но она же преподносит наиболее ценный урок. Как упоминалось выше, каждый раз, когда в Google происходит серьезная неудача, мы выполняем вскрытие, документируем события, приведшие к неудаче, и разрабатываем последовательность шагов, которые предотвратят подобное в будущем. Эта процедура — не поиск виноватых и не бюрократическая проверка: главная ее цель — сосредоточиться на сути проблемы и решить ее раз и навсегда. Это очень сложно, но эффективно (и надежно).

Личные успехи и неудачи немного отличаются. Одно дело хвалить за личные успехи, но стремление возложить личную вину в случае неудачи — отличный способ рассорить команду и отбить в людях охоту рисковать. Потерпеть неудачу — допустимо, но неудачи должны быть общими для всей команды, и вся команда должна учиться на ошибках. Если человек добивается личного успеха, хвалите его перед командой. Если терпит неудачу — изложите конструктивную критику в частном порядке¹. В любом случае воспользуйтесь неудачей как возможностью проявить смирение, уважение и доверие, чтобы помочь команде учиться на ошибках.

¹ Публичная критика не только неэффективна (она заставляет людей защищаться), но и редко необходима, а чаще является просто излишне жестокой. Можете быть уверены, что остальные члены команды уже знают, что человек потерпел неудачу, поэтому нет необходимости заострять на этом внимание.

Антипаттерны

Прежде чем перейти к списку «паттернов проектирования» для технических лидеров и технических руководителей, рассмотрим коллекцию паттернов, которым вы не должны следовать, если хотите стать успешным руководителем. Мы замечали проявления этих деструктивных паттернов не только в работе руководителей, с которыми нам довелось сотрудничать, но и в своих действиях.

Антипаттерн: прием на работу слабых специалистов

Руководитель, чувствующий себя неуверенно в своей роли (по любой причине), может испытывать соблазн обезопасить себя от сомнений окружающих в его способностях, нанимая людей, которыми можно помыкать, — менее умных, недостаточно амбициозных или просто еще более неуверенных. Даже если такой подход укрепит позицию лидера или лица, принимающего решения, команда не сможет сама идти вперед и подчиненных придется вести, как собак на поводке. Создав команду из слабых специалистов, вы не сможете взять отпуск, ведь стоит вам выйти из кабинета, и производительность резко падает. Но это же невысокая цена за то, чтобы чувствовать себя в безопасности на работе, верно?

Вместо этого нужно стремиться нанимать людей, которые умнее вас и могут вас заменить. Это может быть трудно, потому что такие подчиненные будут регулярно оспаривать ваше лидерство (и сообщать вам, что вы ошибаетесь). Однако эти же люди будут постоянно удивлять вас и добиваться успеха. Они мотивируют себя сами, а некоторые даже пытаются управлять командой, но воспринимайте это не как угрозу вашей власти, а как возможность возглавить дополнительную команду, исследовать новое или даже взять отпуск, не беспокоясь о необходимости контролировать команду каждый день. Это также отличный шанс учиться и развиваться — намного легче расширить свой опыт, когда вас окружают люди, опытнее вас.

Антипаттерн: игнорирование неэффективных сотрудников

Когда в Google я впервые раздавал премии команде, то сказал своему руководителю: «Мне нравится быть начальником!» На что тот ответил: «Иногда ты бываешь зубной феей, а иногда — стоматологом».

Дергать зубы не доставляет никакого удовольствия. Мы видели, как руководители команд делают все, чтобы получить сильную команду, но эта команда терпит неудачу (и в конечном итоге разваливается) из-за одного-двух неэффективных игроков. Мы понимаем, что человеческий фактор — самый сложный компонент в разработке ПО, а самая трудная часть работы с людьми — это общение с человеком, который не отвечает ожиданиям. Иногда люди не добиваются успеха, потому что работают недостаточно долго или усердно, но, возможно, человек просто не способен выполнять свою работу, даже если очень старается.

У команды SRE в Google есть девиз: «Надежда — это не стратегия». И нигде надежда так не используется в качестве политики, как в работе с неэффективным исполнителем. Большинство лидеров команд стискивают зубы, отводят глаза и просто *надеются*, что слабый игрок волшебным образом исправится или просто уйдет. Однако такие надежды оправдываются очень редко.

Пока лидер надеется, а слабый сотрудник не совершенствуется (или не уходит), высококлассные специалисты в команде тратят драгоценное время, чтобы увлечь неэффективного коллегу, и в результате моральный дух команды иссякает. Можете быть уверены, что команда знает о существовании неэффективного коллеги, даже если вы его игнорируете, потому что несет часть его груза.

Игнорирование неэффективных сотрудников не только мешает присоединению к команде новых высококлассных специалистов, но и подталкивает к уходу уже имеющихся эффективных исполнителей. В результате в команде остаются только неэффективные работники, потому что они единственные, кто не ушел по собственному желанию. Наконец, оставляя их в команде, вы оказываете им плохую услугу: тот, кто плохо работает в вашей команде, может оказаться намного эффективнее где-то еще.

Чем быстрее вы выявите неэффективного работника, тем быстрее сможете помочь ему начать расти или уйти. Если он проявил неэффективность сразу, его нужно просто поощрить или подтолкнуть к переходу на более высокий уровень. Если же слишком долго ждать, что неэффективный сотрудник исправится, его отношения с командой могут испортиться до такой степени, что вы уже не сможете их наладить.

Как лучше подтягивать отстающих членов команды? Отличная аналогия — представить, что вы помогаете человеку после тяжелой болезни снова научиться ходить, затем бегать трусцой, а потом бежать наравне с остальными членами команды. Это почти всегда требует временного микроменеджмента и всегда — большого смирения, уважения и доверия (уважения особенно). Установите конкретные сроки (скажем, два месяца), в которые сотрудник постепенно должен достичь четко определенных измеримых целей. Дайте ему возможность пройти череду маленьких успехов. Встречайтесь с ним каждую неделю, чтобы проверить, как он продвигается, и оценить успех или неудачу. Если сотрудник не успевает, это сразу станет очевидно для вас обоих и он либо признает, что ему лучше уйти, либо включит решимость и «улучшит свою игру», чтобы оправдать ваши ожидания. В любом случае, работая напрямую с неэффективным сотрудником, вы поспособствуете важным и необходимым изменениям.

Антипаттерн: игнорирование личных проблем

Руководитель работает с командой по двум основным направлениям: социальному и техническому. Обычно руководители в Google сильнее проявляют себя в техническом направлении, и поскольку большинство из них получают повышение после технической должности (то есть раньше основной их задачей было решение технических проблем), они могут игнорировать человеческий фактор. Это так заманчиво — сконцентрировать всю свою энергию на технической стороне работы команды, ведь именно этому учили в вузе.

Пример: у Джейка родился первенец. Джейк и Кэти долго работали вместе — и удаленно, и в одном офисе, поэтому несколько первых недель после рождения ребенка Джейк работал из дома. Такой режим работы отлично подошел для пары, и Кэти была всецело согласна с этим, потому что у них с Джейком уже был опыт удаленной работы. Они прекрасно справлялись, пока их руководитель Пабло (работавший в другом офисе) не узнал об этом и не выразил недовольство. Джейк попытался объяснить Пабло, что он работает так же эффективно, как в офисе, и ему с женой будет намного удобнее работать из дома еще в течение нескольких недель. Пабло ответил: «Джейк, у многих есть дети. Ты обязательно должен ходить в офис». Разумеется, Джейк (по природе мягкий человек) был взбешен и потерял уважение к Пабло.

Пабло мог бы разрешить ситуацию другим способом, например выразить понимание желания Джейка проводить больше времени дома с женой, и если его эффективность осталась прежней и команда не пострадала, просто позволить ему продолжать работать удаленно еще какое-то время. Он мог договориться с Джейком, что тот будет посещать офис один или два дня в неделю. Независимо от конечного результата беседы немного сочувствия помогло бы Джейку остаться довольным в этой ситуации.

Антипаттерн: быть другом для всех

Первый опыт руководства большинство людей получают, становясь руководителями или техническими лидерами команды, членами которой они были ранее. Многие лидеры не хотят терять друзей в своих командах, поэтому стараются поддерживать дружеские отношения с подчиненными. Это может привести к печальным последствиям. Не путайте дружбу с мягким лидерством: ваши подчиненные, понимая, что их карьера зависит от вас, могут чувствовать себя вынужденными отвечать взаимностью на жесты дружбы.

Помните, что вы можете руководить и поддерживать согласие, не будучи близким другом для членов команды (или жестким, бескомпромиссным начальником). Точно так же можно быть жестким лидером, не отказываясь от дружбы. По нашему опыту, совместный обед с командой эффективно способствует поддержанию социальной связи с ней — он не вызывает чувства дискомфорта у членов команды, а у вас появляется возможность неформального общения с коллегами.

Иногда бывает сложно войти в руководящую роль, например в отношении хорошего друга и коллеги. Если друг, которым требуется руководить, не отличается самоконтролем и трудолюбием, это может вызвать стресс у всех. Мы советуем не попадать в такую ситуацию, но если избежать этого невозможно, уделите особое внимание отношениям с таким человеком.

Антипаттерн: компромисс в требованиях к новым сотрудникам

Стив Джобс сказал: «Отличники нанимают других отличников, но хорошисты нанимают троечников». Невероятно легко стать жертвой этой поговорки, особенно если вы хотите набрать людей быстро. За пределами Google широко распространена

практика, когда для найма пяти инженеров команда разбирает кучу заявок, проводит опрос 40–50 кандидатов и отбирает пять лучших из них независимо от того, соответствуют ли они уровню требований.

Это один из самых быстрых способов создать посредственную команду.

Затраты на поиск подходящего человека, будь то оплата услуг рекрутеров, рекламы или мониторинга резюме, ничто по сравнению с затратами на работу с сотрудником, которого, по логике, вы вообще не должны были нанимать. Неудачный выбор кандидата проявится в снижении эффективности команды, росте напряжения, усилении контроля и в итоге приведет к бумажной работе и стрессу в процессе увольнения (поскольку отставлять неподходящего работника еще дороже). Если вы руководите командой, у вас нет права голоса при найме новых сотрудников и вы недовольны кандидатами, то боритесь изо всех сил за высококлассных инженеров. Но если, несмотря на все ваши усилия, команда продолжает пополняться неподходящими инженерами, возможно, пришло время искать другую работу. Без материала, из которого можно слепить отличную команду, вы обречены.

Антипаттерн: отношение к членам команды как к детям

Лучший способ показать членам команды, что вы им не доверяете, — относиться к ним как к детям. Люди склонны вести себя так, как вы к ним относитесь, поэтому, если вы относитесь к ним как к детям или заключенным, не удивляйтесь, что они будут вести себя соответственно. Такое отношение проявляется, когда вы стараетесь управлять всеми мельчайшими аспектами работы или просто неуважительно высказываетесь о способностях сотрудников и не даете им возможности нести ответственность за свой вклад. Если вам постоянно приходится давать подчиненным руководящие указания, потому что вы им не доверяете, значит, вы потерпели провал (или исполнили свою мечту создать команду, за которой надо присматривать). Если вы нанимаете людей, достойных доверия, и демонстрируете им свое доверие, они обычно хорошо справляются со своей работой (если исходить из основной предпосылки, упоминавшейся выше, что вы наняли хороших людей).

Доверие простирается на все уровни общения, вплоть до обыденных вещей, таких как офисные и компьютерные принадлежности. Например, Google предоставляет сотрудникам шкафы с различными ручками, блокнотами и прочими инструментами для творчества, которые можно брать по мере необходимости. IT-отдел поддерживает многочисленные площадки «Tech Stop» — зоны самообслуживания, похожие на мини-магазины электроники. В них можно найти множество компьютерных аксессуаров и компонентов (блоки питания, кабели, мыши, USB-накопители и т. д.), которые любой может просто взять и уйти. Поскольку сотрудникам Google оказывается полное доверие, они чувствуют себя ответственными за взятые вещи. Многие корпорации с ужасом реагируют на это, восклицая, что люди наверняка «крадут» эти вещи, из-за чего Google теряет деньги. Такое, конечно, возможно, но как насчет затрат на работников, которые ведут себя как дети или тратят драгоценное время,

оформляя официальные заказы на дешевые канцелярские товары? Конечно, это обходится дороже, чем несколько ручек и USB-кабелей.

Положительные паттерны

Теперь, познакомившись с антипаттернами, обратимся к позитивным паттернам успешного лидерства и управления, о которых мы узнали из опыта работы в Google, наблюдая за другими успешными лидерами, и в первую очередь за нашими наставниками. Здесь мы перечислим не только паттерны, которые помогли нам добиться больших успехов, но и те, которые использовали уважаемые нами лидеры.

Забудьте про эго

Мы уже говорили о необходимости «забыть про свое эго» несколько глав назад, когда впервые рассматривали смирение, уважение и доверие, однако для лидера команды это особенно важно. Этот паттерн часто неправильно понимается как разрешение другим вытирать о себя ноги, но это совсем другое. Конечно, грань между смирением и позволением другим вас использовать очень тонкая, но смирение — это не то же самое, что отсутствие уверенности. Вы можете быть уверенным в себе и в своем мнении, не будучи при этом эгоистом. С личным эго трудно справиться в любой команде, особенно с эго ее лидера. Поэтому вы должны работать над воспитанием сильного коллективного эго команды и ее самосознания.

Под «забыть про свое эго» также подразумевается доверие: доверяйте команде. Уважайте способности и прошлые достижения членов команды, даже если коллеги недавно присоединились к вам.

Если вы не стремитесь управлять всем и вся в своей команде, то можете быть уверены, что люди, занимающиеся практической реализацией, знают тонкости своей работы лучше, чем вы. Вы все еще можете оставаться единственным, кто ведет команду к согласию и помогает определить цели, но детали достижения ваших целей лучше оставить людям, которые создают продукт. Это даст им не только чувство причастности, но и ответственности за успех (или неудачу) продукта. Если у вас хорошая команда и вы позволяете ей самой устанавливать планку качества и скорости работы, то вы достигнете большего, чем если бы стояли над подчиненными с кнутом и пряником.

Впервые вступая в руководящую должность, большинство людей чувствуют огромную ответственность за правильность решений и считают, что должны знать ответы на все вопросы. Можем заверить, что у вас не все и не всегда будет получаться правильно и вы не будете знать ответы на все вопросы, а стремясь к этому, быстро потеряете уважение команды. Очень многое зависит от вашего чувства безопасности. Вспомните время, когда вы были простым разработчиком. Вы могли почувствовать запах неуверенности за милю. Имейте в виду, что, когда кто-то ставит под сомнение ваше решение или заявление, часто этот человек просто пытается лучше вас понять. Поощряя вопросы коллег, у вас будет гораздо больше шансов получить конструктив-

ную критику, которая сделает вас лучшим лидером лучшей команды. Найти людей, способных дать хорошую конструктивную критику, невероятно сложно, еще сложнее получить такую критику от подчиненных. Подумайте о цели, которую вы пытаетесь достичь с командой, и открыто принимайте отзывы и критику: избегайте соблазна оградить свою территорию.

Наконец, последний элемент формулы «забыть про свое эго»: многие инженеры предпочли бы поджариться на сковородке, чем признать свою ошибку. Это не значит, что повсюду в разговоре нужно вставлять слово «извините» — желание признать ошибку должно быть искренним. Вы будете совершать ошибки, и независимо от того, будете вы их признавать или нет, ваша команда узнает о них. Извинения есть не просят. Люди с огромным уважением относятся к лидерам, которые извиняются, допустив ошибку, и вопреки распространенному мнению извинения не сделают вас уязвимым. Фактически, извиняясь перед людьми, вы заслуживаете их уважение, потому что извинения говорят людям, что вы трезво оцениваете ситуацию и — возвращаясь к смирению, уважению и доверию — смиренны.

Будьте мастером дзен

Как инженер вы, вероятно, отлично развили в себе скептика и циника, но эти качества могут стать помехой, когда придет ваше время возглавить команду. Это не означает, что вы должны быть наивно оптимистичны, просто поумерьте свой скепсис, когда будете сообщать команде, что знаете тонкости и препятствия, связанные с работой. Умеренность в проявлении реакций и спокойствие важны, потому что вы ведете за собой людей и команда (осознанно или нет) будет искать в вашем поведении подсказки, как действовать и реагировать на происходящее вокруг.

Вот простой способ визуализировать этот эффект: представьте организационную схему компании в виде цепочки шестеренок, внизу которой находятся отдельные разработчики в виде маленьких шестеренок с несколькими зубьями, а на каждом последующем уровне управления есть более крупные шестеренки, представляющие руководителей и генерального директора — самую большую шестеренку с сотнями зубьев. Когда шестеренка-руководитель (с несколькими десятками зубьев) совершает один оборот, шестеренка-разработчик совершает два или три оборота. А генеральный директор одним небольшим движением может заставить несчастного сотрудника в конце цепочки, шестью-семью уровнями ниже, вращаться с бешеной скоростью! Чем выше вы поднимаетесь в иерархии управления, тем быстрее можете заставить вращаться шестеренки под вами, независимо от того, собираетесь вы это делать или нет.

Ту же ситуацию можно представить иначе по принципу, что лидер всегда на виду: не только когда проводит собрание или выступает с речью, но даже когда просто сидит за столом и отвечает на письма. Ваши коллеги наблюдают за вами, стараясь найти невербальные подсказки во время бесед и изучая реакции тела. Что они прочитают? Уверенность или страх? Как лидер вы должны вдохновлять, но вдохновение — это

работа 24/7. Ваше видимое отношение ко всему — независимо от того, насколько оно тривиально, — неосознанно замечается и влияет на команду.

Один из первых руководителей в Google, Билл Кафран, вице-президент по инженерным разработкам, по-настоящему освоил способность сохранять спокойствие в любой обстановке. Что бы ни взорвалось, что бы ни случилось, насколько бы ни был велик огненный шторм, Билл никогда не паниковал. Обычно он скрещивал руки на груди, подпирал подбородок ладонью и задавал вопросы о проблеме запаниковавшему инженеру. Это успокаивало инженера и помогало сосредоточиться на решении проблемы, а не бегать, как курица с отрубленной головой. Мы часто шутили, что если кто-то войдет к Биллу и скажет, что 19 офисов компании подверглись нападению космических пришельцев, тот ответит: «Есть идеи, почему не 20?»

Это подводит нас к еще одной хитрости в искусстве управления: задавать вопросы. Когда к вам обращается член команды за советом, у вас может возникнуть неподдельное желание наконец что-то исправить. Это же именно то, что вы делали годы и годы, прежде чем перейти на руководящую должность. В таком случае вы можете переключиться в режим решения, но это самое последнее, что вы должны сделать. Человек, спрашивающий совета, обычно не хочет, чтобы *вы* решили его проблему, но хочет, чтобы вы помогли ему найти решение, и самый простой способ сделать это — задать этому человеку вопросы. Вы не должны играть роль шара предсказаний (невыносимого и бесполезного). Но вы можете проявить смирение, уважение и доверие и попытаться помочь человеку решить проблему самостоятельно, уточняя и исследуя проблему вместе с ним. Обычно это наталкивает сотрудника на ответ¹, и это будет ответ, найденный им самим, что возвращает его к авторству и ответственности, о которых мы говорили выше в этой главе. Независимо от того, знаете вы ответ или нет, этот прием почти всегда оставляет у сотрудника впечатление, что вы помогли ему. Хитро, правда? Сократ гордился бы вами.

Будьте катализатором

В химии катализатором называют вещество, ускоряющее химическую реакцию, но само не участвующее в ней. Примером катализатора (такого как ферменты) может служить вещество, способствующее сближению реагентов: вместо случайного броуновского движения в растворе молекулы реагентов сближаются под воздействием катализатора и взаимодействуют друг с другом. Это та роль, которую вам часто придется играть в качестве лидера, и сыграть ее можно множеством способов.

Чаще всего лидеру приходится добиваться согласия в команде. Для этого вы можете пытаться управлять процессом от начала до конца, а можете просто слегка подтолкнуть его в нужном направлении, чтобы ускорить сближение. Работа по формированию согласия в команде — это лидерский навык, который часто исполь-

¹ См. также статью в Википедии «Rubber duck debugging» (<https://oreil.ly/BKkvk>). (Перевод статьи на русский язык можно найти по адресу: https://ru.wikipedia.org/wiki/Метод_утенка. — Примеч. пер.)

зуются неофициальными лидерами. Диктовать направление менее эффективно, чем стимулировать согласие¹. Если команда хочет двигаться быстро, иногда она добровольно уступает власть и руководство одному или нескольким лидерам. Диктатура, выбранная добровольно, это форма согласия.

Устраняйте препятствия на пути

Иногда команда приходит к согласию относительно того, что делать, но наталкивается на препятствие и останавливается. Это препятствие может быть техническим или организационным, и умение перепрыгнуть через него, чтобы помочь команде продолжить движение, — это обычный прием лидерства. Если препятствие непреодолимо для членов команды, то их понимание, что вы готовы (и способны) его устранить, очень ценно.

Однажды команда потратила несколько недель, пытаясь преодолеть препятствие, возникшее в юридическом отделе Google. Когда команда наконец исчерпала все свои возможности и подошла к своему руководителю с проблемой, тот решил ее менее чем за два часа, просто потому что знал, с каким человеком нужно связаться для обсуждения вопроса. В другой раз команде потребовались дополнительные серверные ресурсы, которые трудно получить. К счастью, руководитель команды имел дружеские связи с другими командами в компании и сумел предоставить нужные ресурсы команде в тот же день. А когда у одного из инженеров возникли проблемы с таинственным фрагментом кода на Java, руководитель команды, не будучи экспертом в Java, смогла связать этого инженера с другим инженером, который точно знал, в чем проблема. Вам не нужно знать все ответы, чтобы помочь устранить препятствия, обычно достаточно знать людей, которые могут помочь, и часто такое знание более ценно, чем знание правильного ответа.

Будьте учителем и наставником

Одна из самых больших сложностей в работе технического лидера — наблюдать, как джун тратит три часа на что-то, что вы могли решить за 20 минут. Поначалу обучение людей и предоставление им возможности учиться самостоятельно может быть невероятно трудным делом, но это жизненно важный компонент эффективного лидерства. Особенно важно дать время новым сотрудникам, которые кроме технологий и кодовой базы изучают культуру команды и ответственность, которую они могут взять на себя. Хороший наставник должен находить компромиссы между временем обучения подопечного и его вкладом в общее дело, одновременно прикладывая все усилия для масштабирования команды по мере ее роста.

Большинство не стремится стать наставниками (как и руководителями) — обычно наставника для нового члена команды ищет лидер. Чтобы стать наставником,

¹ Попытка достичь 100%-ного согласия также может быть вредной. Вы должны быть способны принять решение продолжить работу, даже если не все поддерживают ваше мнение или остается некоторая неопределенность.

не требуется формального подтверждения или подготовки. Для этого нужны три составляющие: опыт работы с процессами и системами в команде, способность объяснять и умение оценивать, какая именно помощь нужна подопечному. Самое важное — дать подопечному достаточно информации, но если вы будете все время повторять объяснения, подопечный, скорее всего, отключится от вас, вместо того чтобы вежливо сказать, что он узнал, что хотел.

Ставьте четкие цели

Это один из тех паттернов, которые, несмотря на всю очевидность, игнорируются огромным количеством лидеров. Если вы хотите, чтобы все члены команды быстро двигались в одном направлении, убедитесь, что они понимают выбранное направление и соглашаются с ним. Представьте, что ваш продукт — это большой воз (а не куча трубок). Каждый член команды держит в руках веревку, привязанную к возу, и, работая над продуктом, они тянут этот воз, каждый в своем направлении. Если вы хотите как можно быстрее переместить воз (или продукт) на север, все члены команды должны тянуть его на север. Чтобы сформулировать четкие цели, установите четкие приоритеты и помогите команде решить, на какие компромиссы она должна пойти, когда придет время.

Самый простой способ поставить четкую цель и заставить команду тянуть продукт в одном направлении — кратко описать цели и задачи команды. После этого вы сможете сделать шаг назад и дать команде больше автономии, периодически проверяя, что сотрудники сохраняют правильное направление. Это не только поможет вам освободить время для решения других лидерских задач, но и значительно повысит эффективность команды. Инженеры могут добиться (и добиваются) успеха без четких целей, но тогда они впустую тратят много энергии. Это будет расстраивать вас, замедлять прогресс команды и заставлять вас тратить больше собственной энергии для корректировки курса.

Будьте честными

Нет, мы не предполагаем, что вы лжете команде, но этот аспект заслуживает упоминания, потому что вы неизбежно окажетесь в положении, когда не сможете сказать что-то подчиненным или, что еще хуже, будете вынуждены сказать им что-то неприятное. Один наш знакомый руководитель говорит новым членам команды: «Я не буду вам лгать, но буду сообщать, когда что-то не смогу вам сказать или просто не знаю ответа».

Если член команды обращается к вам с вопросом по теме, которую вы не можете с ним обсудить, скажите ему, что вы знаете ответ, но не имеете права говорить его. Если же вы не знаете ответа на вопрос сотрудника (такое бывает чаще), прямо скажите, что не знаете ответа. Это еще одна из тех истин, которые кажутся ослепительно очевидными, но, оказавшись в роли руководителя, многие почему-то считают, что незнание ответа доказывает их слабость или оторванность от реаль-

ности. На самом деле единственное, что доказывает правдивый ответ, — руководитель тоже человек.

Давать четкий ответ... ну, это сложно. В первый раз сообщить сотруднику, что он допустил ошибку или не оправдал ожиданий, может быть очень трудно. В большинстве пособий для руководителей, чтобы смягчить удар, рекомендуется использовать «бутерброд с комплиментами», который выглядит примерно так:

«Вы хороший сотрудник и один из наших лучших инженеров. Однако ваш код слишком запутан и его почти никто не сможет понять. Но у вас огромный потенциал и чертовски крутая коллекция футболок».

Конечно, это смягчит удар, но после такого хождения вокруг да около большинство людей выйдут со встречи, думая только: «Ха! У меня классные футболки!» Мы настоятельно рекомендуем не использовать бутерброд с комплиментами, но не потому, что, на наш взгляд, вы должны быть жестким или грубым, а потому, что из-за него люди не слышат критического сообщения. Можно проявить уважение, добрые намерения и сочувствие и высказывать конструктивную критику без бутерброда с комплиментами. На самом деле доброта и сочувствие имеют решающее значение, если вы хотите, чтобы человек услышал критику, а не начал защищаться.

Несколько лет назад наш коллега принял в команду нового члена — Тима, забрав его у другого руководителя, настаивавшего на том, что с Тимом невозможно работать, потому что тот не реагирует на критику и продолжает работать как хочет. Наш коллега поприсутствовал на нескольких встречах того руководителя с Тимом, чтобы понаблюдать за их общением, и заметил, что руководитель часто использовал бутерброд с комплиментами. Новый руководитель четко объяснил Тиму, как тот должен изменить свой подход к работе, чтобы эффективно влиться в команду:

«Мы знаем, что вы не обращаете внимания на то, что ваш способ взаимодействия с командой отталкивает и сердит коллег. Если вы хотите быть эффективным, постарайтесь усовершенствовать свои навыки общения. Мы готовы вам в этом помочь».

Он не высказывал Тиму никаких комплиментов и не приукрашивал проблему, но самое главное, он не сердился, а просто изложил факты так, как он их видел, основываясь на общении Тима с предыдущей командой. И вот, спустя несколько недель (и после нескольких «повторных» встреч) эффективность Тима значительно улучшилась. Тиму просто нужны были более четкие объяснения.

Когда вы прямо высказываете замечания или критику, это гарантирует, что ваше сообщение будет услышано. Но если вы заставите подопечного защищаться, он будет думать не о том, что нужно изменить, а о том, какие аргументы подобрать, чтобы доказать вашу неправоту. Наш коллега Бен когда-то руководил инженером, назовем его Дином, который имел свое мнение по всем вопросам и любил спорить с командой. Он пылко критиковал все, от целей и задач команды до размещения виджета на веб-странице, и отвергал любые возражения. После нескольких месяцев такого поведения Дина Бен решил сообщить ему, что тот слишком агрессивен.

Если бы Бен просто сказал: «Дин, не будь таким придурком», — то Дин полностью проигнорировал бы это сообщение. Бен много думал о том, как заставить Дина понять, что его действия негативно отражаются на команде, и придумал следующую метафору:

Принятие решения похоже на поезд, проезжающий через город. Выпрыгивая перед поездом, чтобы остановить его, вы замедляете его и раздражаете машиниста. Каждые 15 минут прибывает новый поезд, и если вы будете выпрыгивать перед каждым из них, то не только потратите кучу времени на остановку поездов, но в конечном итоге один из машинистов разозлится настолько, что просто переедет вас. Нет ничего плохого, если вы будете пытаться остановить какие-то поезда, но тщательно выбирайте поезда, которым действительно не повредит остановка.

Эта метафора не только помогла разрядить ситуацию, но и позволила Бену и Дину обсудить влияние остановок на команду, не переходя на личности.

Следите за удовлетворенностью

Еще один из способов сделать членов команды более продуктивными (и уменьшить вероятность их ухода) в долгосрочной перспективе — потратить некоторое время на оценку их удовлетворенности. Лучшие лидеры, с которыми мы работали, были немножко психологами и время от времени оценивали моральное состояние членов команды, следя за тем, чтобы они получали признание за свою работу и оставались довольными ею. Один известный нам технический руководитель составлял таблицу всех грязных, неблагодарных задач, которые необходимо выполнить, и равномерно распределял эти задачи между членами команды. Другой технический руководитель считал часы работы своей команды и устраивал выходные с совместными веселыми выездами, чтобы избежать выгорания и утомления. Третий проводил встречи один на один с членами команды и обсуждал технические проблемы, стараясь растопить лед в отношениях, а затем узнавал, есть ли у сотрудников все необходимое для выполнения работы. После того как беседа приобретала более или менее непринужденный характер, он ненавязчиво расспрашивал инженера, насколько ему нравится его работа, и рассказывал, что его ждет дальше.

Надежный способ оценить удовлетворенность команды¹ — спрашивать в конце личной встречи: «Что вам нужно?» Этот простой вопрос позволяет подвести итоги и убедиться, что у каждого члена команды есть все, что ему нужно для продуктивной работы и удовлетворенности ею, но если задавать его при каждой личной встрече, члены команды запомнят эту особенность и начнут приходить к вам со списками своих потребностей.

¹ Каждый год в Google проводится опрос под названием «Googlegeist», в ходе которого оценивается уровень удовлетворенности сотрудников по многим параметрам. Это хорошая обратная связь, но мы не назвали бы этот способ «простым».

Неожиданный вопрос

Вскоре после того как я начал работать в Google, у меня состоялась первая встреча с генеральным директором Эриком Шмидтом. В конце Эрик спросил: «Вам что-нибудь нужно?» Я подготовил миллион защитных ответов на самые сложные вопросы, но к этому вопросу оказался не готов. Я сидел совершенно ошарашенный. Но в следующий раз подготовился к такому вопросу!

Для лидера может быть полезно обратить внимание на удовлетворенность членов команды вне офиса. Наш коллега Мекка начинает личные беседы с подчиненными, предлагая им оценить свою удовлетворенность по шкале от 1 до 10, и его сотрудники часто используют эту шкалу для обсуждения удовлетворенности в офисе и вне его. Не нужно думать, что у людей нет жизни вне работы — нереалистичные ожидания относительно количества времени, которое люди способны потратить на работу, могут заставить их потерять уважение к вам или, что еще хуже, выгореть. Мы не предлагаем вмешиваться в жизнь членов команды, но если вы будете внимательны к их личным обстоятельствам, то сможете лучше понимать, почему их эффективность меняется в разные периоды. Небольшие послабления сотруднику, который в настоящее время испытывает сложности в личной жизни, могут побудить его уделять работе больше времени потом, когда перед командой будут поставлены сжатые сроки.

В значительной степени удовлетворенность членов команды определяется их отношением к своей карьере. Если вы спросите сотрудников, как им видится их карьера через пять лет, многие из них просто пожмут плечами. Большинство людей не любят распространяться о своих планах, но обычно у каждого человека есть цель, которую он хотел бы достичь в следующие пять лет: продвигнуться по службе, узнать что-то новое, запустить какой-то важный проект или работать с умными людьми. Если вы хотите стать эффективным лидером, подумайте, как можно помочь осуществить чьи-то мечты, и пусть сотрудники видят ваше участие в их карьере. Самое важное — выявить их неявные цели, чтобы потом, когда доведется представлять рекомендации сотрудникам, у вас был реальный набор показателей для оценки их возможностей.

Однако удовлетворенность определяется не только карьерой, но и возможностью для каждого члена команды стать лучше, получить признание за свою работу и попутно немного повеселиться.

Другие советы и рекомендации

Вот еще несколько советов и рекомендаций, которые мы в Google даем лидерам:

Делегируйте работу другим, но не бойтесь запачкать свои руки

При вступлении в роль лидера самое сложное — достичь баланса. В первое время вы будете склонны выполнять всю работу сами, а после долгого пребывания на руководящей должности выработаете привычку ничего не делать самостоятельно. Сначала вам наверняка придется приложить немало усилий, чтобы заставить себя

передать работу другим инженерам, даже если для ее выполнения им потребуется гораздо больше времени, чем вам. Этим вы не просто сохраните свое душевное равновесие, но и дадите остальной команде возможность учиться. Чтобы быстро завоевать уважение команды и ускорить работу, испачкайте свои руки, например выполнив грязную работу, которую никто не хочет делать. Ваше резюме и список достижений может быть длиной в милю, но ничто так ярко не продемонстрирует команде ваши умения и преданность (и скромность), как выполнение действительно тяжелой работы.

Найдите себе замену

Если вы не хотите выполнять одну и ту же работу до конца своей карьеры, найдите себе замену. Этот поиск начинается, как упоминалось выше, с найма: нанимайте людей «умнее себя». Когда в команде появятся люди, способные выполнять вашу работу, дайте им возможность взять на себя больше обязанностей или попробовать возглавить команду. Поступая так, вы скоро увидите, кто из них может и хочет руководить. Но не забывайте, что некоторые сотрудники предпочитают быть просто хорошими специалистами, и это нормально. Мы всегда удивлялись компаниям, которые ставят своих лучших инженеров, вопреки их желаниям, на руководящие должности, тем самым теряя хороших специалистов и получая посредственных руководителей.

Знайте, когда гнать волну

Часто и неизбежно у вас будут возникать трудные ситуации, когда внутренний голос будет кричать вам, что не надо ничего делать. Вы можете быть недовольны инженером: его навыками, склонностью выпрыгивать перед каждым поездом или отсутствием мотивации при работе 30 часов в неделю. «Просто подожду немного, и все наладится», — скажете вы себе. «Все пройдет само собой», — добавите вы. Не попадайтесь в эту ловушку — это ситуации, в которых вы должны гнать самые большие волны и делать это без промедления. Такие проблемы редко решаются сами собой, и чем дольше вы будете ждать их исчезновения, тем негативнее они будут влиять на команду и тем чаще вы будете думать о них по ночам. Выжидая, вы лишь откладываете неизбежное и наносите работе серьезный ущерб. Так что действуйте, причем быстро.

Защищайте свою команду от хаоса

Вступая в роль лидера, первое, что вы обнаружите, — мир за пределами команды наполнен хаосом и неопределенностью (или даже безумием), которых вы не замечали, когда были разработчиком. Когда я впервые занял руководящий пост в 1990-х годах (до того как снова стал рядовым разработчиком), я был озадачен ужасающей неопределенностью и организационным хаосом в компании. Я спросил другого руководителя, в чем причина этой внезапной нестабильности в такой спокойной компании, на что он посмеялся над моей наивностью и ответил, что хаос был всегда, просто мой предыдущий руководитель ограждал от него меня и всю нашу команду.

Прикрывайте свою команду

Важно своевременно информировать подчиненных о происходящем в компании «над ними», но не менее важно защищать их от множества необоснованных требований и неопределенности, которые могут навязываться команде извне. Делитесь с сотрудниками как можно большим количеством информации, но не отвлекайте их организационным сумасшествием, которое вряд ли когда-либо на них повлияет.

Сообщайте команде, когда все идет хорошо

Многие начинающие руководители настолько увлекаются искоренением недостатков у членов команды, что недостаточно часто дают им положительные отзывы. Вы должны не только сообщать кому-то, что он облажался, но также отмечать его успехи и обязательно сообщать ему (и его коллегам), что он делает выдающуюся работу.

Наконец, вот еще один совет, который лучшие лидеры знают и часто используют, когда в команде есть инженеры, желающие попробовать что-то новое:

Просто скажите «да», если потом это будет легко отменить

Если у вас есть член команды, желающий взять день или два, чтобы попробовать новый инструмент или библиотеку¹, которые могли бы ускорить работу над вашим продуктом (и сроки не сильно поджимают), просто скажите: «Конечно, попробуй». С другой стороны, если сотрудник хочет, например, запустить новый продукт, который придется поддерживать в течение десяти лет, тогда хорошенько подумайте о целесообразности такой работы. По-настоящему хорошие лидеры понимают, когда что-то можно отменить, но того, что можно отменить, намного больше, чем вы думаете (это касается как технических, так и нетехнических решений).

Люди похожи на растения

Моя жена была младшей из шести детей в семье, и ее мать столкнулась с трудной задачей вырастить шестерых совершенно разных детей, каждому из которых нужны разные вещи. Я спросил свою тещу, как ей это удалось (заметили, что я сделал?), и она ответила, что дети похожи на растения: одни похожи на кактусы, которым нужно немного воды и много солнечного света; другие, как африканские фиалки, нуждаются в рассеянном освещении и влажной почве; а третьи напоминают помидоры и превосходно растут, если дать им немного удобрений. Если у вас шестеро детей и вы даете каждому одинаковое количество воды, света и удобрений, все они получают одинаковое обращение, но высока вероятность, что никто из них не получит того, что им действительно нужно.

Члены вашей команды тоже похожи на растения: кому-то нужно больше света, кому-то — больше воды (а кому-то — больше... удобрений). Ваша задача как лидера

¹ Подробнее о технических изменениях, которые можно «отменить», в главе 22.

определить, кому что нужно, а затем дать им это — то есть мотивировать и направлять в разных пропорциях.

Чтобы все члены команды получили то, что им нужно, поощряйте тех, кто следует колее, и более настойчиво управляйте теми, кто отвлекается или не знает, что делать. Конечно, будут и такие, кто «плывет по течению» и нуждается в мотивации и управлении. Правильно сочетая методы мотивации и управления, вы сможете сделать команду удовлетворенной и продуктивной. Но не давайте слишком много — если людям не нужна мотивация или управление, а вы пытаетесь дать им это, то вызовете раздражение.

Давать указания довольно просто — достаточно иметь базовое понимание того, что нужно сделать, и простые навыки организации и координации, чтобы разбить проблему на управляемые задачи. Владая этими инструментами, вы сможете направить инженера, нуждающегося в помощи. Мотивация, однако, немного сложнее и заслуживает некоторого объяснения.

Внутренняя и внешняя мотивация

Есть два типа мотивации: *внешняя*, которая исходит извне (например, денежная премия), и *внутренняя*, которая исходит изнутри. В книге «Драйв. Что на самом деле нас мотивирует» (М.: Альпина Паблишер, 2019) Дэн Пинк объясняет, что сделать людей самыми счастливыми и продуктивными нельзя одной только внешней мотивацией (например, засыпать их деньгами) — для этого нужно развивать еще и внутреннюю мотивацию, дав людям три вещи: самостоятельность, мастерство и цель¹.

Самостоятельность появляется у человека, когда он может действовать без чьего-то назойливого управления². Самостоятельным сотрудникам (а Google стремится нанимать в основном самостоятельных инженеров) можно задать общее направление развития продукта и предоставить возможность самим решать, как его делать. Это развивает мотивацию не только потому, что сотрудники теснее связаны с продуктом (и, вероятно, лучше вас знают, как его создавать), но также и потому, что дает им гораздо более сильное чувство владения продуктом. Чем больше их вклад в развитие продукта, тем выше их интерес к его успеху.

Под мастерством в данном случае понимается возможность человека улучшить имеющиеся навыки и овладеть новыми. Наличие благодатной почвы для повышения мастерства не только помогает мотивировать людей, но со временем делает их лучше, от чего выигрывает вся команда³. Навыки подобны лезвию ножа: можно потратить десятки тысяч долларов, чтобы привлечь в команду людей с самыми

¹ Предполагается, что людям платят достаточно хорошо, чтобы они не испытывали стресса из-за низкого дохода.

² Предполагается, что в команде есть люди, которым не требуется микроменеджмент.

³ Конечно, это также означает, что они становятся более ценными и востребованными сотрудниками, которые могут найти другое место работы, если утратят удовлетворенность от работы с вами. См. выше паттерн «Следите за удовлетворенностью».

отточенными навыками, но если использовать один нож годами, не затачивая, вы получите тупой и неэффективный нож, а в некоторых случаях даже бесполезный. Инженерам в Google предоставляются широкие возможности для освоения нового и совершенствования своего мастерства.

Конечно, никакие самостоятельность и мастерство не смогут мотивировать человека, работающего безо всякой цели, поэтому вы должны обозначить цель. Многие люди работают над продуктами, имеющими большое значение для компании, клиентов и мира в целом, но остаются в стороне от положительного результата своей деятельности. Даже если значение продукта небольшое, вы можете мотивировать им команду. Когда сотрудники увидят цель, вы заметите рост их мотивации и производительности¹. Одна наша знакомая руководительница следит за отзывами о своем продукте (довольно незначительном), которые приходят по электронной почте в Google. Обнаружив сообщение от клиента, рассказывающего, как продукт помог ему лично или его бизнесу, она немедленно рассылает этот отзыв членам команды. Это не только мотивирует членов команды, но и часто вдохновляет их подумать об улучшении продукта.

Заключение

Руководство командой сильно отличается от работы инженера-программиста. Поэтому хорошие инженеры-программисты не всегда становятся хорошими руководителями, и это нормально — эффективные организации позволяют продвигаться по карьерной лестнице и рядовым разработчикам, и руководителям. В Google давно поняли, что для руководителей важно иметь опыт в программной инженерии, но гораздо важнее иметь социальные навыки. Хорошие руководители помогают командам работать эффективно, сосредоточивая их усилия на правильных целях и изолируя их от внешних проблем, следуя трем столпам: смирению, доверию и уважению.

Итоги

- Не будьте «начальником» в традиционном смысле: сосредоточьтесь на лидерстве, влиянии и служении команде.
- Делегируйте все, что сможете; не старайтесь сделать все сами.
- Обратите особое внимание на сосредоточенность, направление и скорость работы команды.

¹ *Grant A. M.* The Significance of Task Significance: Job Performance Effects, Relational Mechanisms, and Boundary Conditions. *Journal of Applied Psychology*, 93, No. 1 (2018), http://bit.ly/task_significance.

Масштабируемое лидерство

Автор: Бен Коллинз-Сассмэн

Редактор: Риона Макнамара

В главе 5 мы поговорили о том, как перейти от роли «рядового разработчика» к роли лидера команды. В этой главе мы рассмотрим, как наиболее эффективно продолжить путь лидера и перейти от руководства одной командой к руководству группой смежных команд.

Все лучшие практики руководства сохраняют свою актуальность по мере движения лидера вверх по карьерной лестнице. Вы останетесь «лидером-слугой», несмотря на увеличение числа подчиненных. Однако проблемы, которые вы будете решать, станут более широкими и абстрактными и вынудят вас подняться на «более высокий уровень» — меньше разбираться в технических деталях, чтобы видеть «ширину», а не «глубину» проблем. Вас будет огорчать не востребованность своих инженерных знаний, а ваша эффективность начнет напрямую зависеть от *общей* технической интуиции и способности побудить инженеров двигаться в правильном направлении.

Этот процесс часто деморализует до тех пор, пока вы не заметите, что оказываете гораздо большее влияние как лидер, а не как разработчик. Это сладкое осознание имеет привкус горечи.

Итак, учитывая, что мы понимаем основы лидерства, подумайте, что нужно, чтобы превратить себя в *действительно хорошего* лидера? Об этом мы поговорим здесь и используем то, что мы называем «три всегда в лидерстве»: всегда принимайте решение, всегда уходите, всегда будьте масштабируемым.

Всегда принимайте решение

Управлять командой из команд означает принимать больше решений на более высоких уровнях. Такая работа больше связана со стратегией высокого уровня, чем с решением какой-то конкретной инженерной задачи. Большинство решений на этом уровне касаются поиска правильного набора компромиссов.

Притча о самолете

Линдсей Джонс (<http://lindsayjones.com>) — наш друг, театральным звукорежиссер и композитор. Он часто летает по США от одного производства к другому и полон

безумных (и правдивых) историй о воздушных путешествиях. Вот одна из наших любимых историй:

Шесть часов утра, мы сели в самолет и готовы взлетать. Вдруг командир воздушного судна включает громкую связь и сообщает, что кто-то переполнил топливный бак на 10 000 галлонов. Я много летал на самолетах, но не знал, что такое возможно. Я имею в виду, что если я переполню бак в машине хотя бы на галлон, то залью бензином свои ботинки.

Как бы то ни было, командир говорит, что есть два варианта: можно подождать, пока подъедет заправщик и откачает лишнее топливо из самолета, что займет более часа, или двадцать человек сойдут с самолета, чтобы выровнять вес.

Но никто даже не сдвинулся с места.

В салоне был парень, сидевший через проход от меня в первом классе, невероятно злой. Он напомнил мне Фрэнка Бернса из М*А*S*Н¹. Он был жутко возмущен и, брызгая слюной, требовал назвать виновного. Это было потрясающее зрелище, как будто он герой фильма-буффонады.

В какой-то момент он схватил кошелек и достал огромную пачку денег! И такой: «Я не могу опоздать на эту встречу! Я дам 40 долларов любому, кто сойдет с самолета прямо сейчас!»

Конечно же, нашлись люди, решившиеся этим воспользоваться. Он выдал по 40 долларов 20 пассажирам (а это 800 долларов наличными!), и они сошли с самолета.

Теперь все в порядке и самолет начинает выруливать на взлетно-посадочную полосу. Но тут снова включается командир и сообщает, что компьютер самолета перестал работать и никто не знает почему. Теперь самолет нужно отбуксировать обратно к выходу на посадку.

Фрэнка Бернса чуть не хватил удар. Seriously! Я думал, у него случится инсульт. Он сыпал проклятиями. Все остальные просто переглядывались между собой.

Мы вернулись к воротам, и этот парень потребовал отправить его другим рейсом. Ему предложили место на самолете, вылетающем в 9:30. Он спросил: «Разве нет рейса до 9:30?»

А служащий аэропорта у выхода на посадку ответил: «Ну, на 8:00 есть еще один рейс, но он полон. Они сейчас закрывают двери».

А он такой: «Полон?! Что значит полон? В этом самолете нет ни одного свободного места?!»

А служащий аэропорта говорит: «Нет, сэр, на этом самолете были места, пока вдруг из ниоткуда не появились 20 пассажиров и не заняли их. Это были самые

¹ Американский телесериал, созданный Ларри Гелбертом, в российском прокате вышел под названием «Чертова служба в госпитале МЭШ». — *Примеч. пер.*

счастливые пассажиры, которых я когда-либо видел, они смеялись всю дорогу до трапа».

Рейс на 9:30 был, пожалуй, самым тихим.

Эта история, как вы понимаете, о компромиссах. Большая часть этой книги посвящена различным техническим компромиссам в инженерных системах, но оказывается, что компромиссы применимы и к поведению людей. Как лидер вы обязаны принимать решения о том, что должны делать ваши команды каждую неделю. Иногда компромиссы очевидны («если мы будем работать над этим проектом, то задержим другой»); иногда они имеют непредсказуемые последствия, как в той истории про самолет.

На самом высоком уровне ваша работа как лидера состоит в том, чтобы направлять людей к решению сложных и неоднозначных проблем. Под неоднозначностью мы подразумеваем, что проблема не имеет очевидного решения и даже может оказаться неразрешимой. В любом случае проблема должна быть изучена, исследована и (желательно) проконтролирована. Если сравнить написание кода с вырубкой деревьев, то ваша задача как лидера состоит в том, чтобы «увидеть лес за деревьями», найти правильный путь через этот лес и направить инженеров к нужным деревьям. На этом пути вы должны сделать три важных шага: снять *шоры*, выявить *компромиссы* и, наконец, принять *решение* и повторить процесс.

Снятие шор

Приступая к решению проблемы, часто можно обнаружить, что есть группа людей, борющаяся с ней уже много лет. Эти люди так долго погружались в проблему, что неосознанно надели на глаза «шоры» и больше не видят леса, продолжая делать массу предположений о проблеме (или ее решении). «Мы всегда так поступали», — говорят они, потеряв способность критически оценивать ситуацию. Иногда можно обнаружить странные механизмы преодоления или рационализации, которыми эти люди оправдывают существующее положение. Именно здесь у вас, имеющего свежий взгляд, есть большое преимущество. Вы можете увидеть эти шоры, задать вопросы, а затем рассмотреть новые политики. (Конечно, незнание задачи не является обязательным условием для хорошего руководства.)

Выявление компромиссов

Неоднозначные проблемы по определению *не* имеют универсальных волшебных решений. И нет решения, универсального для всех ситуаций. Есть только *решение, лучшее на данный момент*, и оно почти наверняка предполагает компромисс в том или ином направлении. Ваша задача — выявить компромиссы, объяснить их, а затем помочь сотрудникам найти лучший баланс выгод и затрат.

КЕЙС: РЕШЕНИЕ ПРОБЛЕМЫ «ЗАДЕРЖЕК» В WEB SEARCH

При управлении командой из команд возникает естественное стремление заняться развитием не одного продукта, а целого «класса» продуктов или, может быть, решением более широкой проблемы, характерной для нескольких продуктов. Рассмотрим, например, наш самый старый продукт — Web Search.

Десять лет тысячи инженеров в Google работали над общей проблемой улучшения результатов поиска. Но побочным эффектом этих усилий стало постепенное замедление скорости работы продукта. Когда-то результаты поиска в Google отображались на странице в виде десятка синих ссылок на веб-сайты. И тысячи крошечных изменений добавили к ним изображения, видеоролики, блоки с фактами из Википедии и даже элементы интерактивного интерфейса. Это увеличило нагрузку на серверы: по сети передавалось больше байтов, а клиенту (обычно по телефону) отображалась более сложная разметка HTML. Несмотря на то что за эти же десять лет быстродействие сети и компьютеров заметно увеличилось, скорость работы страницы поиска только уменьшалась из-за задержки ее отображения. Кому-то может показаться, что в этом нет ничего особенного, но задержка в работе продукта напрямую влияет (в совокупности с другими проблемами) на вовлеченность пользователей и частоту использования продукта. Даже увеличение времени отображения всего на 10 мс имеет значение. Задержка нарастает медленно. Это не вина конкретной команды инженеров, скорее это длительное коллективное отравление среды. В какой-то момент общая задержка в Web Search выросла до такой степени, что ее эффект стал сводить на нет внесенные улучшения и отрицательно сказался на вовлеченности пользователей.

Многие лидеры долго боролись с этой проблемой, но не смогли решить ее на систематическом уровне. Шоры, которые все носили, предполагали, что единственный способ устранить задержку — раз в два-три года объявлять «желтый код»¹ и бросать все силы на оптимизацию кода и ускорение работы продукта. Эта стратегия позволяла добиться улучшения на какое-то время, после чего задержка снова начинала расти и через месяц или два возвращалась к прежнему уровню.

Что изменилось с тех пор? В какой-то момент мы сделали шаг назад, выявили недостатки в работе и произвели полную переоценку компромиссов. Мы поняли, что погоня за «качеством» имеет не одну, а *две* разные издержки. Первая издержка — для пользователя: более высокое качество обычно означает больший объем отправляемых данных и соответственно большую задержку. Вторая издержка — для Google: более высокое качество означает больше работы, чтобы сгенерировать данные, и больше затрат процессорного времени на наших серверах — того, что мы называем «пропускной способностью». Руководство очень внимательно относилось к компромиссу между качеством и пропускной способностью, но оно никогда не рассматривало задержку как фактор, влияющий на принятие решения. Как гласит старая шутка, «Выбери два из хорошо, быстро и дешево». Самый простой способ наглядно представить компромисс — нарисовать треугольник с вершинами хорошо (качество), быстро (задержка) и дешево (пропускная способность).

¹ «Желтый код» — в Google этот термин обозначает «экстренный брейншторм с целью решения критической проблемы». Предполагается, что вовлеченные команды приостановят всю работу и сосредоточатся на решении проблемы, пока чрезвычайное положение не будет отменено.

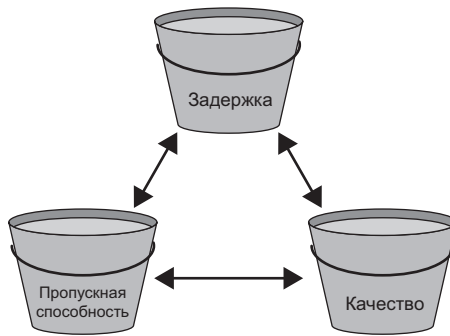


Рис. 6.1. Компромиссы в Web Search: выбери два!

Именно это и произошло. Легко улучшить любую из этих характеристик в ущерб, по крайней мере, одной из двух других. Например, можно улучшить качество, добавив больше данных на страницу с результатами поиска, но это уменьшит пропускную способность и увеличит задержку. Также можно пойти на прямой компромисс между задержкой и пропускной способностью, увеличив трафик на обслуживающий кластер. Если вы передадите кластеру больше запросов, то получите увеличенную пропускную способность, то есть более полное использование процессоров и более быструю окупаемость средств, вложенных в аппаратное обеспечение. Но более высокая нагрузка увеличит частоту конфликтов за обладание ресурсами на компьютере, что усугубит среднюю задержку обработки запроса. Намеренное уменьшение трафика (ослабление нагрузки) приведет к уменьшению пропускной способности, но при этом каждый отдельный запрос будет обрабатываться быстрее.

Главное здесь — это понимание. Понимание всех компромиссов позволило нам начать экспериментировать с новыми способами балансировки. Теперь мы можем рассматривать задержку не как неизбежный и случайный побочный эффект, а как типичный результат, стоящий в одном ряду с другими нашими результатами. Мы выработали новые политики. Например, наши специалисты по обработке данных смогли точно измерить, насколько задержка повлияла на вовлеченность пользователей. Это помогло им определить метрику, позволяющую сопоставить улучшения в качестве с вовлеченностью пользователей в краткосрочной перспективе, и ущерб, вызванный задержками, с вовлеченностью пользователей в долгосрочной перспективе. Теперь, если небольшое изменение улучшает качество, но увеличивает задержку, мы можем количественно оценить, стоит ли внедрять это изменение или нет. Мы каждый месяц определяем, насколько сбалансированы наши изменения в качестве, задержке и пропускной способности.

Принятие решения и повторение процесса

После того как вы выявите компромиссы и разберетесь в них, у вас появится опора для принятия лучшего решения в текущем месяце. В следующем месяце вам придется пересмотреть компромиссы и баланс выгод и затрат. Мы говорим: «Всегда принимай решение», имея в виду, что процесс принятия решений итеративный.

Если не воспринимать процесс как непрерывное балансирование, команды рискуют попасть в ловушку поиска идеального решения и прийти к тому, что некоторые называют «параличом анализа». Сделайте итерации привычным делом для команд. Для этого можете понизить ставки и успокоить нервы фразой: «Попробуем это решение и посмотрим, что получится. В следующем месяце мы сможем отменить изменения или принять другое решение». Такой подход сделает сотрудников гибкими и готовыми извлекать уроки из своего выбора.

Всегда уходи

На первый взгляд совет «всегда уходи» звучит ужасно. Почему хороший лидер должен уходить? На самом деле это известная цитата Бхарата Медиратты, бывшего технического директора Google. Он имел в виду, что ваша задача — не только решить неоднозначную проблему, но и заставить других решить ее *самостоятельно*, без вашего участия. Это освободит вас для перехода к новой проблеме (или в новую организацию) и позволит оставить след успеха на вашем пути.

Антипаттерном здесь, конечно, будет ваше стремление стать единой точкой отказа. Вспомните термин «фактор автобуса»: *количество людей, которым нужно попасть под автобус, чтобы проект провалился*.

Разумеется, «автобус» — это всего лишь метафора. Люди заболевают, уходят в другие команды или компании. Проверьте себя: вспомните сложную проблему, в решении которой ваша команда добилась хороших результатов. А теперь представьте, что вы исчезли. Сможет ли команда продолжить двигаться вперед? Это движение будет успешным? Вот еще более простой тест: подумайте о последнем вашем отпуске, который длился как минимум неделю. Вы продолжали проверять рабочую почту? (Большинство лидеров так и делают.) Спросите себя: зачем? Рухнет ли что-то, если вы перестанете обращать на это внимание? Если да, тогда вы, скорее всего, стали единой точкой отказа и должны это исправить.

Ваша цель: создать команду, способную работать самостоятельно

Возвращаясь к цитате Бхарата, быть успешным лидером означает построить организацию, которая сможет самостоятельно решать сложные проблемы. Она должна иметь сильных лидеров, надежные инженерные процессы и позитивную самоподдерживаемую культуру, которая функционирует в течение долгого времени. Да, это трудно, но руководство командой из команд действительно больше связано с организацией людей, а не с техническим мастерством. И есть три основных шага для построения такой самодостаточной группы: распределение задач, делегирование подзадач и итерация по мере необходимости.

Распределение задач

Сложные задачи обычно состоят из множества более мелких сложных задач. Если вы возглавляете команду из команд, вам потребуется распределить небольшие задачи между командами. Но учтите, что меньшие задачи могут меняться со временем и жесткие ограничения по их выполнению могут помешать заметить эти изменения и приспособиться к ним. Если есть возможность, пересматривайте организационную структуру, попробуйте перестраивать ее, чтобы получить больше свободы: меняйте размеры подгрупп, дайте сотрудникам возможность переходить из группы в группу, перераспределяйте задачи между группами. Назначайте не «слишком жесткие» и не «слишком расплывчатые» границы выполнения: с одной стороны, подгруппы должны четко понимать цель задачи и средства ее достижения, но с другой стороны, исполнители должны иметь свободу выбора направления и пробовать что-то новое в ответ на меняющуюся обстановку.

Пример: распределение «задачи с задержкой» в Google Search

Подходя к проблеме задержки в Google Search, мы поняли, что эту задачу можно разделить как минимум на два основных направления: устранение *симптомов* задержки и устранение ее *причин*. Было очевидно, что нужно создать несколько проектов и укомплектовать их людьми, чтобы оптимизировать кодовую базу для повышения скорости, но *одного* ускорения было недостаточно. Тысячи инженеров продолжали наращивать сложность и «качество» результатов поиска, быстро сводя на нет оптимизации скорости, поэтому нам также нужно было направить людей на предотвращение задержки. Мы выявили пробелы в метриках анализа задержек, обучении разработчиков и документации. Назначив разные команды для одновременной работы над причинами и симптомами задержек, мы смогли обеспечить надежный систематический контроль над задержками. (Обратите внимание, что эти команды занимались *задачами*, а не конкретными решениями!)

Делегирование подзадач лидерам

Тема «делегирования» в книгах по менеджменту — это клише, но для ее упоминания есть причина: научиться делегированию *действительно трудно*. Оно идет вразрез с нашими представлениями об эффективности по пословице: «Хочешь сделать хорошо — сделай это сам».

Но если вы согласны с тем, что ваша цель — создать самостоятельную организацию, основным способом ее достижения является делегирование. Вы должны воспитать плеяду самодостаточных лидеров, и делегирование — это самый эффективный способ их обучения. Вы поручаете им задание, позволяете потерпеть неудачу, а затем даете возможность пробовать снова и снова. В Кремниевой долине широко известны мантры о «быстрых неудачах и повторных попытках». Эта философия относится не только к инженерному проектированию, но и вообще к обучению.

Портфель лидера постоянно пополняется важными задачами, которые необходимо решать. Большинство из этих задач довольно просты для вас. Представьте, что вы

усердно разбираете свою почту, отвечая на вопросы, а затем решаете прерваться на 20 минут, чтобы устранить старую надоевшую проблему. Но прежде чем сделать это, остановитесь и задайте себе вопрос: *действительно ли я единственный, кто сможет выполнить эту работу?*

Возможно, вы справитесь с задачей намного *эффективнее*, чем кто-либо, но тогда ваши лидеры ничему не научатся. Если задача не горит, тогда сделайте усилие над собой и поручите работу кому-нибудь другому — скорее всего, тому, кто точно справится с ней, но медленнее, чем вы. Помогайте, если понадобится. Ваши лидеры должны научиться «повышать уровень» и выполнять работу самостоятельно, чтобы вы не попадали в критические ситуации.

Отсюда следует, что вы должны помнить о своей цели как лидера лидеров. Приходя утром на работу, задавайте себе важный вопрос: *что я могу сделать, чего никто другой в моей команде не сможет?*

На этот вопрос есть много хороших ответов. Например, вы можете защитить команды от организационной политики, поддержать их и создать благоприятный психологический климат, продвигая культуру смирения, доверия и уважения. Также важно уделить внимание «управлению вверх», то есть следить за тем, чтобы вышестоящее руководство понимало, чем занимается ваша группа, и оставаться в курсе событий в компании. Но часто самый верный ответ на этот вопрос: «Я вижу лес за деревьями». Другими словами, вы можете *определять высокоуровневые политики*. Ваша стратегия должна охватывать не только общее техническое направление, но и организационное. Вы определяете план решения неоднозначной проблемы и способ его претворения в жизнь вашей организацией в течение долгого времени. Вы постоянно составляете карту леса, а вырубку деревьев поручите другим.

Корректировка и повторение

Давайте предположим, что вы достигли момента, когда создали самодостаточную организацию. Вы больше не единая точка отказа. Поздравляем! Чем заняться дальше?

Прежде чем ответить, обратите внимание, что вы действительно освободили себя — теперь у вас есть свобода «всегда уходить». Вы можете заняться решением новой или смежной проблемы или перейти в другой отдел и в другую предметную область, освободив место для перспективных лидеров, которых вы обучили. Это отличные способы избежать выгорания.

Простой ответ на вопрос «что дальше?»: *направлять* созданную организацию и поддерживать ее. Но без явной на то необходимости вы должны действовать тонко. В книге «Debugging Teams»¹ есть притча о том, как выполнять осмысленные корректировки:

Есть история о Мастере-механике, который давно вышел на пенсию. В компании, где он раньше работал, возникла проблема, которую никто не мог решить. Со-

¹ Fitzpatrick B. W., Collins-Sussman B. Debugging Teams: Better Productivity through Collaboration. Boston: O'Reilly, 2016.

трудники вызвали Мастера, чтобы узнать, сможет ли он помочь найти причину проблемы. Мастер осмотрел сломавшуюся машину, послушал ее, вытащил кусочек мела и нарисовал маленький крестик сбоку на ее корпусе. Затем сказал технику, что в этом месте не закреплен провод. Техник вскрыл машину, затянул проволоку и тем самым решил проблему. Когда Мастер предъявил счет на 10 000 долларов, разгневанный генеральный директор потребовал объяснить, почему Мастер требует столь высокую плату просто за то, что нарисовал мелом крестик! Тогда Мастер предъявил другой счет, в котором указал, что 1 доллар стоит мел, которым был нарисован крестик, и 9999 долларов — знание, где его поставить.

Это весьма поучительная история: единственная, тщательно продуманная корректировка может принести весомый результат. Это ценный прием при управлении людьми. Мы представляем, что наша команда летит на большом дирижабле, медленно и уверенно двигаясь в определенном направлении. Вместо того чтобы заниматься микроменеджментом и пытаться постоянно корректировать курс, мы большую часть недели внимательно наблюдаем и слушаем, а в конце делаем маленькую отметку мелом в нужном месте — то есть выполняем небольшое, но важное «действие», чтобы скорректировать курс.

Вот что такое хороший менеджмент: 95 % наблюдения и 5 % важных корректировок в нужном месте. Слушайте своих лидеров и подчиненных. Разговаривайте со своими клиентами и помните, что часто (особенно если ваша команда строит инженерную инфраструктуру) ваши «клиенты» — это не конечные пользователи, а ваши коллеги. Удовлетворенность клиентов требует такого же внимательного отношения, как и удовлетворенность подчиненных. Узнавайте, что работает, а что — нет и сохраняет ли дирижабль правильное направление? Ваше управление должно быть итеративным, но вдумчивым и с минимальными корректировками курса. Если вы окунаетесь в микроменеджмент, то рискуете стать единой точкой отказа! «Всегда уходи» — это призыв к *макроменеджменту*.

Позаботьтесь о самоидентификации команды

Распространенная ошибка — считать, что команда должна отвечать за конкретный продукт, а не за общую задачу. Продукт — это *решение* задачи. Срок службы решений может быть коротким, и продукты могут быть вытеснены более удачными решениями. Однако *задача* — при правильном выборе — может быть вечной. Самоидентификация команды по конкретному решению («Мы — команда, управляющая репозиториями Git») может со временем порождать всевозможные страхи ее членов. Если компания захочет использовать новую VCS, скорее всего, команда, идентифицирующая себя по решению, будет «упорствовать», защищать свое решение, противостоять переменам и даже начнет вредить организации. Она будет цепляться за свои шоры, потому что решение стало ее самоидентификацией. Если, напротив, команда будет связывать себя с *задачей* («Мы — команда, обеспечивающая управление версиями для компании»), она сможет свободно экспериментировать с различными решениями, появляющимися время от времени.

Всегда масштабируйте себя

Во многих книгах о менеджменте говорится о «масштабировании» в контексте «максимизации вашего влияния» — стратегиях развития команды и вашего лидерства. Мы не собираемся обсуждать этот вопрос здесь, потому что уже не раз затрагивали его. Должно быть очевидно, что создание самостоятельной организации с сильными лидерами само по себе является отличным рецептом роста и успеха.

Вместо этого мы обсудим масштабирование команды с *оборонительной* и личной точек зрения, а не с наступательной. *Самый ценный ресурс лидера — это ограниченный запас времени, внимания и энергии.* Если вы агрессивно будете наращивать ответственность и власть своих команд, не научившись защищать свое здравомыслие, то масштабирование вашего лидерства обречено на провал. Поэтому мы поговорим о том, как эффективно масштабировать себя.

Цикл успеха

Команда решает сложную задачу по стандартной схеме. Вот как она выглядит:

Анализ

Вы выявляете проблему и начинаете ее решать. Вы определяете шоры, выявляете все компромиссы и выстраиваете мнение о том, как ими управлять.

Сражение

Вы начинаете работать независимо от того, есть ли у команды все необходимое. Вы готовитесь к неудачам и повторным попыткам. В этот период вашу работу можно сравнить с ношением воды в решете. Вы подталкиваете своих лидеров и экспертов к выработке мнений, а затем внимательно выслушиваете их и разрабатываете общую стратегию, пусть и не совсем верную¹.

Движение вперед

Команда начнет приходить к пониманию задачи. Вы принимаете все более разумные решения, и начнется реальное движение вперед. Моральный дух коллектива укрепляется. Вы пробуете компромиссы, и организация начинает самостоятельно решать проблему. Отличная работа!

Награда

Внезапно ваш руководитель поздравляет вас с успехом. Вы обнаруживаете, что ваша награда — не похлопывание по плечу, а *совершенно новая задача*, которую нужно решить. Все верно: награда за успех — новая работа... и новая ответственность! Часто новая задача похожа на предыдущую, но не менее сложна.

¹ На этом этапе легко может возникнуть синдром самозванца. Один из способов борьбы с ощущением, что вы занимаетесь не своим делом, — просто притворяться, что кто-то из экспертов точно знает, что делать, просто он в отпуске, а вы временно его заменяете. Это отличный способ снять с себя лишний груз ответственности и дать себе разрешение на неудачу и обучение на ошибках.

И вы попадаете в трудное положение. Вам дали новую задачу, но (как правило) не дали больше людей. Теперь вам нужно каким-то образом заняться решением *двух* задач, а это значит, что, скорее всего, первая задача должна быть решена *вдвое меньше* количеством людей и за вдвое меньшее время. Вторая половина ваших людей вам понадобится, чтобы заняться новой работой! Последний шаг решения задачи мы называем *стадией сжатия*: вы берете все, что делали, и сжимаете до вдвое меньших размеров.

В действительности цикл успеха — это скорее спираль (рис. 6.2). В течение многих месяцев и лет ваша организация масштабируется, решая задачи, а затем выясняя, как их сжать, чтобы параллельно вступить в схватку с новыми задачами. Если повезет, вам разрешат нанять еще людей. Однако чаще расширение штата сотрудников не будет поспевать за масштабированием. Ларри Пейдж, один из основателей Google, скорее всего, назвал бы эту спираль «невероятно захватывающей».



Рис. 6.2. Спираль успеха

Спираль успеха — сложная головоломка, с которой трудно справиться, и все же она является главной парадигмой масштабирования команды из команд. Сжатие задачи заключается не только в максимизации эффективности команды, но и в умении распределять время и внимание в соответствии с новой, более широкой областью ответственности.

Важное против срочного

Вспомните время, когда вы были простым беззаботным сотрудником. Если раньше вы были программистом, ваша жизнь была более спокойной. У вас был список дел, и каждый день вы методично работали по этому списку, писали код и устраняли проблемы. Расстановка приоритетов, планирование и выполнение работы были простыми и понятными.

Однако, став лидером, вы замечаете, что ваш режим работы стал менее предсказуемым и больше похож на пожаротушение. То есть ваша работа стала менее *проактивной* и более *реактивной*. Чем выше занимаемый пост, тем больше форс-мажоров возникает в работе. Вы — предложение «finally» в длинном списке блоков кода! Все ваши средства связи — электронная почта, чаты, встречи — напоминают атаки «отказ в обслуживании» на ваши время и внимание. На самом деле, если не проявить осторожность, вы в конечном итоге будете тратить все свое время на работу в реактивном режиме. Люди будут бросать вам мячи, а вы — отчаянно прыгать из стороны в сторону, стараясь поймать их все.

Эта проблема обсуждалась во многих книгах. Автор книг по менеджменту Стивен Кови известен своей идеей необходимости отличать важное от срочного. Фактически эту идею популяризировал президент США Дуайт Эйзенхауэр в своем высказывании в 1954 году:

«У меня есть два вида задач — срочные и важные. Срочные не важны, а важные не срочны».

Напряжение в работе представляет одну из самых больших опасностей для вашей эффективности как лидера. Если вы позволите себе окунуться в реактивный режим (что происходит почти автоматически), вы потратите каждый момент своей жизни на неотложные задачи, которые почти все не играют важной роли в общей картине. Помните, что ваша работа как лидера состоит в том, чтобы делать то, что *можете сделать только вы*, например прокладывать путь через лес. Выстроить такую метастратегию невероятно важно, но почти никогда не срочно. Всегда легче ответить на следующее срочное письмо.

Итак, как заставить себя работать преимущественно над важным, а не срочным? Вот несколько ключевых приемов:

Делегируйте

Многие неотложные дела можно передать другим лидерам. Вас может терзать чувство вины за делегирование тривиальной задачи или беспокойство о ее более медленном решении. Но делегируя, вы тренируете подчиненных и освобождаете свое время для работы над чем-то важным, что можете сделать только вы.

Планируйте время

Регулярно отводите пару часов или больше для спокойной работы *только* над важным, но не срочным, например над выработкой командной политики, опре-

делением направлений карьерного роста ваших лидеров или планированием сотрудничества с другими командами.

Подберите хорошую систему учета

Существуют десятки систем учета, помогающих расставить приоритеты в работе. Некоторые опираются на использование ПО (например, специальные инструменты ведения списка дел), другие — на использование ручки и бумаги (как метод «Bullet Journal», <http://www.bulletjournal.com>), а третьи не зависят от реализации. Относительно последней категории большую популярность среди инженеров завоевала книга Дэвида Аллена «Как привести дела в порядок» (М.: МИФ, 2016). Она описывает абстрактный алгоритм работы с задачами и поддержания «почтового ящика пустым». Попробуйте разные системы и определите, какая лучше подходит для вас. Постарайтесь найти что-то более эффективное, чем стикеры на экране компьютера.

Учитесь ронять мячи

Есть еще один ключевой метод управления своим временем, и на первый взгляд он выглядит чересчур радикальным. Для многих этот метод противоречит многолетним инженерным инстинктам. Как инженер вы обращаете внимание на детали: составляете списки дел, проверяете, что в них не попало, проявляете точность и всегда заканчиваете начатое. Вам приятно закрывать ошибки в баг-трекере или сокращать количество необработанных входящих сообщений до нуля. Но ваше время и внимание как лидера лидеров находятся под постоянной атакой. Независимо от ваших стараний вы, так или иначе, начнете ронять мячи на пол — их просто слишком много. Это ошеломляет, и вы, наверное, все время чувствуете вину за это.

А теперь отступим на шаг назад и посмотрим на ситуацию беспристрастно. Если падение каких-то мячей неизбежно, не лучше ли уронить некоторые из них *преднамеренно*, а не *случайно*? По крайней мере, тогда у вас появится ощущение контроля.

Вот отличный способ сделать это.

Мари Кондо, консультант по организационным вопросам, в книге «Магическая уборка. Японское искусство наведения порядка дома и в жизни» (М.: Эксмо, 2017) рассказала о философии эффективного наведения порядка в доме, которую с успехом можно применить к любому абстрактному беспорядку.

Подумайте о своем физическом имуществе как о трех кучах. Около 20 % ваших вещей просто бесполезны (вы попользовались ими один раз и больше никогда не трогали), их очень легко выбросить. Около 60 % вещей представляют определенный интерес, они различаются для вас по важности, и вы используете их время от времени. И только оставшиеся 20 % ваших вещей действительно важны для вас: вы используете их *постоянно*, они имеют большое эмоциональное значение или, как говорит мисс Кондо, вызывают глубокую «радость», когда вы просто берете их в руки. Главная мысль ее книги состоит в том, что большинство людей неправильно-

но подходят к организации своей жизни: они тратят время, чтобы выбросить 20 % мусора, но оставшиеся 80 % вещей все еще создают ощущение загромождения. Она утверждает, что для наведения *истинного* порядка нужно определить верхние, а не нижние 20 %. Если вы сможете определить только важные вещи, вы должны выбросить остальные 80 %. Звучит экстремально, но довольно эффективно. Этот метод позволяет радикально избавиться от хлама.

Эту философию с успехом можно применить к входящей почте или списку дел — мячам, брошенным вам. Разделите мячи на три кучи: нижние 20 % мячей, вероятно, не являются ни срочными, ни важными, и их можно смело удалить или проигнорировать. Средние 60 % задач, которые могут иметь ту или иную степень срочности или важности, — это смешанная куча. На вершине остаются 20 % мячей, имеющих неоспоримую важность.

А теперь, работая над своими задачами, не пытайтесь взяться за верхние 80 %, иначе вы окажетесь загруженными срочными, но не важными задачами. Вместо этого внимательно определите мячи, которые попадают в верхние 20 %, — критические задачи, которые *можете сделать только вы*, — и сосредоточьтесь только на них. Позвольте себе уронить остальные 80 %.

Поначалу это может показаться ужасным, но если вы намеренно уроните так много мячей, то обнаружите два удивительных обстоятельства. Во-первых, даже если вы никому не будете делегировать средние 60 % задач, ваши подчиненные заметят это и подберут их сами. Во-вторых, если что-то в этой средней куче действительно важно, оно все равно вернется к вам, рано или поздно перейдя в верхние 20 %. Вам просто нужно *поверить*, что задачи, не попавшие в верхние 20 %, будут решены другими или вырастут до уровня действительной важности. Между тем, сконцентрировавшись только на критически важных задачах, вы сможете распределить свое время и внимание так, чтобы верно реагировать на постоянно растущие обязанности вашей группы.

Защитите свою энергию

Мы поговорили о защите вашего времени и внимания, но в уравнении есть еще один член — ваша личная энергия. Постоянные хлопоты о масштабировании утомляют. Как в такой обстановке оставаться заряженным и оптимистичным?

Отчасти ответ заключается в том, что со временем ваша общая выносливость возрастает. В начале карьеры работа в офисе по восемь часов в день может показаться шоком: вспомните, насколько уставшими вы возвращались домой. Но, так же как при подготовке к марафону, ваш мозг и тело со временем накапливают запасы выносливости.

Другая важная часть ответа: лидеры постепенно учатся более разумно *управлять* своей энергией и постоянно уделять этому внимание, приобретают умение оценивать свою энергию в любой момент и делать осознанный выбор, когда и как «подзарядить» себя. Вот несколько интересных примеров разумного управления энергией:

Возьмите настоящий отпуск

Выходные — это не отпуск. Требуется как минимум три дня, чтобы «забыть» о работе, и по меньшей мере неделя, чтобы почувствовать себя обновленным. Но продолжая во время отпуска проверять рабочую почту или чаты, вы *перекрываете* подпитку. Поток беспокойства прорывается в ваш разум, и все преимущества психологического дистанцирования улетучиваются. Отпуск заряжает, только если вы действительно полностью отключаетесь от работы¹. А это возможно, только если вы создали организацию, которая может работать без вас.

Просто отключитесь

Решив отключиться от дел, оставьте свой рабочий ноутбук в офисе. Если на вашем личном телефоне есть рабочие контакты, удалите их. Например, если ваша компания использует G Suite (Gmail, Google Calendar и т. д.), установите эти приложения в «рабочем профиле» на личном телефоне. У вас появятся два приложения Gmail: одно для личной электронной почты, другое — для рабочих. На телефоне с Android достаточно нажать всего одну кнопку, чтобы полностью отключить рабочий профиль. Все значки рабочих приложений станут серыми, как если бы они были удалены, и вы не сможете «случайно» проверить рабочие сообщения, пока вновь не включите рабочий профиль.

Сделайте выходные по-настоящему выходными

Выходные не так эффективны, как отпуск, но они тоже могут немного подзарядить вас. Но имейте в виду, что подзарядка возможна только в том случае, если вы полностью отключитесь от общения по рабочим вопросам. Попробуйте на самом деле выйти из системы в пятницу вечером, провести выходные, занимаясь любимым делом, и войти снова в понедельник утром, когда вернетесь в офис.

Делайте перерывы в течение дня

В естественной среде человеческий мозг работает 90-минутными циклами². Воспользуйтесь возможностью встать и прогуляться по офису или выйти на 10 минут на улицу. Такие маленькие перерывы, конечно, позволяют подзарядиться лишь чуть-чуть, но они могут существенно снизить уровень стресса и улучшить ваше самочувствие в течение следующих двух часов работы.

Дайте себе право на день психологической разгрузки

Иногда без всякой причины работа не задается. Вы могли хорошо выспаться, хорошо поесть, позаниматься спортом, и все равно у вас плохое настроение. Если вы лидер, то это ужасно. Ваше плохое настроение накаляет атмосферу, а это может привести к неверным решениям (электронные письма, которые вы не должны были отправлять, слишком суровые суждения и т. д.). Оказавшись в такой ситу-

¹ Вам нужно заранее приготовить себя к тому, что во время отпуска ваша работа просто не будет выполнена. Упорный (или энергичный) труд непосредственно перед отпуском и после него смягчает эту проблему.

² Узнать больше о циклах покоя-активности мозга можно в Википедии: https://en.wikipedia.org/wiki/Basic_rest-activity_cycle.

ации, просто развернитесь и идите домой, объявив разгрузочный день. В такой день лучше ничего не делать, чем активно наносить урон.

В конце концов, умение управлять энергией так же важно, как и умение управлять временем. Если вы освоите эти умения, то сможете перейти к следующему, более широкому циклу масштабирования ответственности и создания самодостаточной команды.

Заключение

Успешные лидеры берут на себя больше ответственности по мере карьерного роста (и это естественно). Если они не придумают эффективные методы, помогающие быстро принимать правильные решения, делегировать задачи другим при необходимости и управлять своей повышенной ответственностью, они в итоге могут почувствовать себя подавленными. Быть эффективным лидером не означает принимать совершенные решения, делать все самостоятельно или работать вдвое больше. Эффективный лидер должен всегда принимать решения, всегда уходить и всегда быть масштабируемым.

Итоги

- Всегда принимайте решения: неоднозначные проблемы не решаются как по волшебству — все они связаны с поиском правильных компромиссов в данный момент и повторными попытками.
- Всегда уходите: ваша задача как лидера состоит в том, чтобы построить организацию, которая автоматически решает целый класс неоднозначных проблем без вашего присутствия.
- Всегда масштабируйте себя: со временем успех порождает больше ответственности, и вы должны активно управлять *масштабированием* своей работы, чтобы защитить свои ограниченные ресурсы личного времени, внимания и энергии.

Оценка продуктивности инженеров

Автор: Сиера Джаспен

Редактор: Риона Макнамара

Google — компания, управляемая данными. Большинство наших продуктов и проектных решений прочно опираются на данные. Решения, принятые на основе данных и соответствующих метрик, носят объективный, а не субъективный характер. Однако сбор и анализ данных о человеческой натуре имеют свои проблемы. Мы в Google обнаружили, что в сфере разработки ПО наличие команды специалистов, сосредоточенных на продуктивности труда инженеров, очень ценно и важно, потому что компания может масштабировать и использовать идеи такой команды.

Зачем оценивать продуктивность инженеров?

Представьте, что вы владеете процветающим бизнесом (например, развиваете систему поиска в интернете) и хотите расширить его (выйти на рынок корпоративных, облачных или мобильных приложений). Для этого необходимо увеличить размер инженерной организации. Однако расходы на связь растут в квадратичной зависимости от роста размеров организации¹. Вы можете нанять больше людей, но накладные расходы на связь не будут расти линейно с расширением персонала. Линейно масштабировать бизнес с увеличением размеров вашей инженерной организации не получится.

Есть другой способ решить проблему масштабирования: *сделать каждого сотрудника более продуктивным*. Это поможет расширить сферу бизнеса без увеличения накладных расходов на связь.

Компании Google пришлось быстро осваивать новые сферы бизнеса и соответственно учить инженеров работать более продуктивно. Для этого нам нужно было понять, что делает сотрудников продуктивными, найти причины неэффективности инженерных процессов и исправить выявленные проблемы. Эти меры можно повторять при необходимости в процессе непрерывного улучшения и тем самым масштабировать инженерную организацию с учетом все возрастающих потребностей.

¹ Брукс Ф. Мифический человеко-месяц, или Как создаются программные системы. СПб.: Питер, 2021. — *Примеч. пер.*

Однако этот цикл мер *тоже* требует человеческих ресурсов. Было бы нецелесообразно повышать продуктивность десяти инженеров в год за счет труда пятидесяти инженеров в течение года, выявляющих и устраняющих препятствия, мешающие росту продуктивности. *Наша цель — не только повысить продуктивность разработки ПО, но и сделать это эффективно.*

Мы в Google достигли этой цели, создав команду исследователей, занимающихся вопросами повышения продуктивности инженеров. В нее вошли люди, работающие над исследованиями в области программной инженерии, инженеры-программисты общего профиля, а также социологи из разных областей, включая когнитивную психологию и поведенческую экономику. Добавление специалистов социальных наук позволило нам изучать человеческий фактор разработки ПО, включая личные мотивы, структуру стимулов и политики для управления сложными задачами. Мы стремимся оценить и увеличить продуктивность инженерного труда, используя подходы на основе данных.

В этой главе мы рассмотрим, как наша исследовательская группа достигает поставленной цели. Все начинается с расстановки приоритетов. В разработке ПО есть много параметров, которые *можно* измерить, но что *нужно* измерить? Мы рассмотрим, как после выбора проекта исследовательская группа определяет значимые метрики, идентифицирующие проблемные этапы процесса, и как показатели для оценки увеличения продуктивности используются в Google.

В этой главе мы исследуем конкретный пример, представленный языковыми командами C++ и Java: поддержку удобочитаемости. (Подробнее об удобочитаемости в главе 3.) Этот процесс был введен в первые годы существования Google, еще до того, как стали привычными средства автоматического форматирования (глава 8) и статического анализа кода (глава 9), блокирующие его отправку. Сам процесс обходится довольно дорого, потому что требует, чтобы сотни инженеров проверяли удобочитаемость кода, написанного другими инженерами. Некоторые сотрудники считали его архаичным проявлением «дедовщины», не имеющим смысла, и это была их любимая тема в обеденный перерыв. Перед языковыми командами был поставлен вопрос: стоит ли тратить время на повышение удобочитаемости?

Расстановка приоритетов: что измерять?

Для оценки продуктивности инженеров нужно выбрать метрики. Само измерение — дорогостоящий процесс, включающий производство измерений, анализ результатов и распространение полученных данных в компании. Такие мероприятия могут замедлять работу инженерной организации в целом. Даже если замедления не происходит, отслеживание прогресса может влиять на поведение инженеров и отвлекать их от реальных проблем. Измерения и оценка должны производиться с умом — не становиться гаданием на кофейной гуще и не отнимать время и ресурсы на измерение ненужного.

Мы в Google придумали ряд вопросов, чтобы помочь командам определить, стоит ли вообще проводить измерения. Сначала мы просим людей описать в форме конкретного вопроса, что они хотели бы измерить. Как показывает практика, чем конкретнее человек формулирует вопрос, тем выше вероятность, что он получит пользу от ответа. Когда к нам обратилась группа по удобочитаемости, ее вопрос был прост: окупятся ли для компании затраты на работу инженера по повышению удобочитаемости?

Затем мы просим их рассмотреть следующие аспекты их вопроса:

Какой результат вы ожидаете получить и почему?

Не будем притворяться, что мы относимся к теме нейтрально. У нас есть свои представления о вопросе. Признав это с самого начала, мы попытаемся устранить предубеждения и не допустить объяснения результатов постфактум.

Команда по удобочитаемости отметила, что причина возникновения вопроса о целесообразности дополнительной проверки кода в неуверенности. Одно время люди были уверены, что затраты на контроль удобочитаемости окупятся, но после появления автоматизированных инструментов форматирования и статического анализа эта уверенность стала неполной. Развилось убеждение, что этот процесс потерял актуальность. Он еще может давать инженерам преимущества (и результаты опроса показали, что люди стремятся их получить), но неясно, стоят ли эти преимущества времени, затраченного авторами или рецензентами кода.

Если данные подтвердят ожидаемый результат, какие действия будут предприняты?

Бессмысленно что-то измерять, если по результатам измерений не будет принято никаких действий. Обратите внимание, что действием может быть «поддержание текущего положения дел», если запланировано некоторое изменение, которое произойдет независимо от результатов измерений.

Команда удобочитаемости дала простой ответ: если подтвердится, что выгоды от контроля удобочитаемости окупают затраты, команда сошлется на исследование и данные в сборнике часто задаваемых вопросов об удобочитаемости и объявит об этом, чтобы подтвердить ожидания.

Если получится отрицательный результат, будут ли приняты соответствующие меры?

Часто отрицательный результат не так влияет на принятие решение, как другие более важные факторы. В такой ситуации, возможно, стоит отложить измерения. Этот вопрос останавливает большинство проектов, к которым приступает наша исследовательская группа: люди, принимающие решения, заинтересованы в знании результатов измерений, но по каким-то причинам не хотят менять курс.

В случае с удобочитаемостью, однако, командой было твердо обещано, что если по результатам анализа затраты перевесят выгоду или выгоды окажутся незначительными, тогда команда остановит контроль удобочитаемости. Поскольку разные языки программирования имеют разные уровни зрелости инструментов форматирования и статического анализа, каждый язык будет оцениваться отдельно.

Кто будет принимать решение по полученному результату и в какой срок?

Сотрудник, запрашивающий проведение измерений, должен быть уполномочен принимать решения (или действовать от имени ответственного лица). Цель измерения — помочь принять бизнес-решение, и важно знать, кто сделал запрос и данные в какой форме его убедят. Конечно, хорошее исследование должно включать разные подходы (от структурированных опросов до статистического анализа журналов), но время предоставления данных может быть ограничено. Поэтому измерение полезно ориентировать на лица, принимающие решения. Склонны ли они сопереживать историям из опросов?¹ Доверяют ли они результатам опроса или данным из журналов? Могут ли они разобраться в особенностях статистического анализа? Если лицо, принимающее решения, не верит в форму результата в принципе, то нет смысла проводить измерение.

В случае с удобочитаемостью нам были четко названы те, кто будет принимать решения. Две команды, Java и C++, напрямую сделали нам запрос на измерения, а другие языковые группы стали наблюдать, что будет². Лица, принимающие решения, доверяли опыту своих инженеров в понимании удовлетворенности и важности обучения, но хотели видеть «беспристрастные числа», полученные из журналов и характеризующие скорость и качество кода. Мы провели качественный и количественный анализ. Эта работа не была срочной, но ее завершение было приурочено к внутренней конференции, на которой можно было бы объявить о предстоящих изменениях. В результате мы получили несколько месяцев.

Задавая эти вопросы, мы часто обнаруживаем, что проводить измерения просто нет смысла... и это нормально! Есть много веских причин не измерять влияние инструмента или процесса на продуктивность. Вот несколько примеров:

Нет возможности изменить процесс/инструменты прямо сейчас

Этому могут мешать временные или финансовые ограничения. Представьте, что по результатам измерения переключение на более быстрый инструмент сборки сэкономит несколько часов в неделю. Но вы не станете его проводить, поскольку для перехода потребуется приостановить разработку перед важным

¹ В настоящее время в отрасли с пренебрежением относятся к «анекдотным» (anecdota) — историям из личного опыта, и все стремятся «руководствоваться данными». Однако истории существуют, потому что имеют большую выразительную силу. Они, в отличие от чисел, могут дать контекст и глубокое объяснение, вызывающее резонанс. Наши исследователи не принимают решений, опираясь на истории, но мы используем и поощряем такие методы, как структурированные опросы и тематические исследования, для глубокого понимания явлений и описания контекста количественных данных.

² Java и C++ имеют наибольшую инструментальную поддержку. Для обоих языков имеются зрелые средства форматирования и статического анализа, улавливающие распространенные ошибки. Также оба финансируются в основном изнутри. Несмотря на то что другие языковые команды, такие как Python, тоже интересовались результатами, очевидно, что удаление контроля удобочитаемости для Python не даст никаких преимуществ, если мы не сможем показать такие преимущества для Java или C++.

этапом финансирования. Инженерные компромиссы не оцениваются в вакууме — более широкий контекст может полностью оправдывать задержку реакции на результат.

Любые результаты вскоре станут недействительными из-за других факторов

В качестве примера можно привести измерение характеристик процесса разработки ПО в организации непосредственно перед запланированной реорганизацией или измерение суммы технического долга по устаревшей системе.

Лицо, принимающее решение в этих областях, имеет твердые убеждения, и вы вряд ли сможете представить достаточное количество доказательств нужного вида, чтобы изменить их.

Во многом это вопрос знания аудитории. Даже в Google мы встречаем людей с непоколебимыми убеждениями, основанными на опыте. Нам встречались и те, кто вообще не доверяет данным опросов или поддается влиянию убедительных повествований, основанных на небольшом количестве интервью, и, конечно же, те, кто доверяет только результатам анализа журналов. Во всех случаях мы стараемся представить взгляд на истину с разных точек зрения, используя смешанные методы, но если заинтересованная сторона настаивает на использовании метода измерений, который не подходит для данной проблемы, мы откажемся в этом участвовать.

Результаты будут использоваться, только чтобы потешить свое честолюбие

Это, пожалуй, самая распространенная причина, по которой мы отвечаем отказом. Люди принимают решения по нескольким причинам, и совершенствование процесса разработки ПО — только одна из них. Например, команда инструментов выпуска новых версий в Google однажды попросила оценить запланированное изменение в рабочем процессе выпуска версий. Было очевидно, что изменение как минимум не ухудшит текущего состояния, но авторы запроса хотели знать, будет ли это улучшение значительным или нет. Мы спросили команду: если улучшение окажется небольшим, потратят ли они ресурсы на запланированное изменение, даже если затраты не окупят себя? Ответ был «да»! Как оказалось, эти изменения не только повышали продуктивность, но также имели побочный эффект: снижали нагрузку на команду разработчиков инструментов выпуска.

Единственные доступные метрики не позволяют достаточно точно оценить проблему и могут меняться под влиянием других факторов

В некоторых случаях необходимые метрики (см. следующий раздел) просто недоступны, и может возникнуть соблазн провести анализ с использованием других, менее точных метрик (например, подсчитать число строк кода). Однако любые результаты, полученные с помощью таких метрик, не будут поддаваться интерпретации. Если метрика подтверждает ранее существовавшее убеждение заинтересованных сторон, они могут приступить к реализации своего плана без учета того, что метрика не является точной мерой. Если метрика не подтверждает

убеждение, ее неточность будет использована против нее и заинтересованная сторона все так же может приступить к реализации своего плана.

Под успешным измерением характеристик процесса подразумевается не доказательство правильности или неправильности гипотез, а *предоставление заинтересованным сторонам данных, необходимых для принятия решения*. Если заинтересованная сторона не будет использовать полученные данные, измерение можно считать неудачным. Мы должны измерять характеристики процесса, только когда предполагается, что на основе его результатов будет принято конкретное решение. Команда поддержки удобочитаемости четко объявила: если метрики покажут, что процесс приносит пользу, они опубликуют результаты, в противном случае процесс измерений будет свернут. Обратите внимание, что команда поддержки удобочитаемости имела право принять такое решение.

Выбор значимых метрик с использованием целей и сигналов

Приняв решение об измерении характеристик процесса, мы определяем, какие метрики для него использовать. Очевидно, что число строк кода — плохая метрика¹, но чем тогда измерить продуктивность инженеров?

Для создания метрик мы в Google используем модель Цели/Сигналы/Метрики (GSM, Goals/Signals/Metrics).

- *Цель* — это желаемый конечный результат. Она на высоком уровне определяет, что вы хотите понять, и не должна содержать ссылок на конкретные способы измерения.
- *Сигнал* — признак достижения цели. Сигналы — это то, что мы *хотим* измерить, но они могут оказаться неизмеримыми.
- *Метрика* — это отражение сигнала, которое можно измерить. Возможно, метрика не идеально совпадает с сигналом, но мы считаем, что она достаточно близка к нему.

Модель GSM полезна с точки зрения определения метрик. Во-первых, задавая сначала цели, затем сигналы и, наконец, метрики, мы предотвращаем *эффект уличного фонаря*. Этот термин происходит от фразы «искать ключи под уличным фонарем»: освещенные места не всегда совпадают с местом, куда упали ключи. То же верно в отношении метрик: легкодоступные и легкоизмеримые метрики не всегда соответствуют нашим потребностям.

¹ «Следующий шаг — измерение продуктивности программиста количеством строк кода в месяц. И этот метод оценки дорого обходится, потому что поощряет писать раздутый неэффективный код. Но сегодня меня мало интересует, насколько глупой выглядит эта единица измерения труда даже с чисто коммерческой точки зрения. Я хочу подчеркнуть, что если мы считаем количество строк кода, то должны рассматривать их не как доход, а как затраты. Общепринятая практика дошла до такого маразма, что указывает нам записывать число не в ту колонку». Эдсгер Дейкстра (Edsger Dijkstra). «On the cruelty of really teaching computing science», EWD Manuscript 1036 (<https://oreil.ly/ABAX1>).

Во-вторых, модель GSM помогает предотвратить искажение и смещение метрик, вынуждая нас выбирать метрики *до* фактического измерения результата (использовать принципиальный подход). Рассмотрим случай, когда метрики были выбраны без использования принципиального подхода и результаты не совпали с ожиданиями заинтересованных сторон. В такой ситуации заинтересованные стороны могут предложить для измерения другие метрики, и мы не сможем утверждать, что новые метрики не подойдут! Модель GSM заставляет нас выбирать метрики, способные измерять цели. Именно такие метрики заинтересованные стороны считают самыми подходящими.

Наконец, модель GSM может показать нам области охвата измерений. Запуская процесс GSM, мы перечисляем все цели и для каждой определяем сигналы. Как мы увидим в примерах, не все сигналы измеримы, и это нормально! С помощью GSM мы определяем, что именно не поддается измерению, и принимаем решение, как действовать дальше.

При выборе метрик важна поддержка *контроля* — возможность проследить каждую метрику обратно до сигнала, который она отражает, и цели, которую она пытается измерить. Это дает нам уверенность, что мы знаем, какие метрики измеряем и зачем.

Цели

Цель должна быть обозначена в терминах, не относящихся к метрикам. Сами по себе цели не поддаются измерению, но хороший набор целей — это то, с чем все могут согласиться, прежде чем переходить к сигналам, а затем к метрикам.

Определение правильного набора целей может показаться простой задачей: команда наверняка знает цели своей работы! Однако наша исследовательская группа обнаружила, что люди часто забывают включить в этот набор все возможные *компромиссы продуктивности*, которые могут привести к ошибочным оценкам.

Представьте, что команда была настолько сосредоточена на том, чтобы сделать проверку удобочитаемости быстрой, что забыла о качестве кода. Команда включила в измерения оценки, учитывающие время, необходимое для прохождения процесса рецензирования, и степень удовлетворенности инженеров этим процессом. Один из членов команды даже предложил:

«Рецензирование кода можно сделать очень быстрым: надо полностью удалить этот этап».

Конечно, это пример крайности, однако команды постоянно игнорируют компромиссы при измерении: они настолько сосредоточены на увеличении скорости, что забывают измерить качество (или наоборот). Для борьбы с этой забывчивостью наша исследовательская группа делит продуктивность на пять основных аспектов, которые находятся в противоречии друг с другом. Мы призываем команды сверять с ними свои цели, чтобы ничего не упустить. Запомнить все пять аспектов помогает аббревиатура QUANTS:

Quality of the code (*качество кода*)

Как определяется качество кода? Достаточно ли хорошо тесты справляются с предотвращением регрессий? Насколько хорошо архитектура способствует снижению рисков и изменений?

Attention from engineers (*внимание инженеров*)

Как часто инженеры полностью включаются в работу? Как часто им приходится отвлекаться на уведомления? Помогает ли инструмент легко переключать внимание инженера?

Intellectual complexity (*интеллектуальная сложность*)

Какая когнитивная нагрузка требуется для выполнения задания? Какова внутренняя сложность решаемой задачи? Приходится ли инженерам справляться с избыточной сложностью?

Tempo and velocity (*темп и скорость*)

Насколько быстро инженеры справляются со своими задачами? Как быстро они могут выпускать новые версии? Сколько задач они решают за определенный период?

Satisfaction (*удовлетворенность*)

Насколько инженеры удовлетворены своими инструментами? Насколько хорошо инструмент соответствует потребностям инженеров? Насколько инженеры удовлетворены своей работой и конечным продуктом? Чувствуют ли они выгорание?

Для оценки процесса поддержки удобочитаемости, наша исследовательская группа совместно с командой удобочитаемости определила несколько целей продуктивности процесса:

Качество кода

Участвуя в процессе повышения удобочитаемости инженеры пишут более качественный и более согласованный код и вносят свой вклад в культуру единообразия кода.

Внимание инженеров

Повышение внимания инженера в процессе контроля удобочитаемости не является целью. Это нормально! Для принятия компромиссного решения необязательно учитывать все пять аспектов продуктивности.

Интеллектуальная сложность

Участвуя в процессе контроля удобочитаемости, инженеры изучают кодовую базу и лучшие практики программирования в Google, а также получают советы наставников.

Темп и скорость

Благодаря повышению удобочитаемости инженеры справляются с задачами быстрее и эффективнее.

Удовлетворенность

Инженеры видят пользу от контроля удобочитаемости и положительно оценивают свое участие в нем.

Сигналы

Сигналы помогают нам узнать, достигли ли мы своих целей. Не все сигналы измеримы, и между сигналами и целями нет прямой связи. У каждой цели должен быть хотя бы один сигнал, но их может быть и больше. Также некоторые цели могут иметь общий сигнал. В табл. 7.1 перечислены некоторые примеры сигналов, использовавшиеся для оценки процесса контроля удобочитаемости.

Таблица 7.1. Сигналы и цели

Цели	Сигналы
Инженеры пишут более качественный код	Инженеры, получающие отзывы о своем коде, оценивают свой код как более качественный
Инженеры изучают кодовую базу и лучшие практики программирования в Google	Инженеры отмечают, что процесс проверки удобочитаемости дает им новые знания
Инженеры получают советы наставников	Инженеры отмечают положительное влияние общения с опытными инженерами, выступающими в роли рецензентов
Инженеры справляются с задачами быстрее и эффективнее	Инженеры считают, что их продуктивность выше, чем у коллег. Изменения, внесенные инженерами, прошедшими проверку, воспринимаются быстрее и проще, чем изменения, написанные другими инженерами
Инженеры видят пользу от процесса проверки удобочитаемости и положительно оценивают свое участие в нем	Инженеры считают процесс контроля удобочитаемости полезным

Метрики

Метрики определяют, как будут измеряться сигналы, и сами по себе являются измеримым отражением сигналов, поэтому не считаются идеальными оценками. По этой причине сигнал может иметь несколько метрик, обеспечивающих его более точное представление.

Например, чтобы оценить, насколько быстрее читается код инженера, участвующего в контроле удобочитаемости, можно использовать комбинацию из данных опросов и журналов. Ни одна из этих метрик не передаст истинной картины. (Человеку свойственно ошибаться, а метрики из журналов могут неточно измерять время, затраченное инженером на просмотр фрагмента кода, или искажаться неизвестными факторами, действовавшими в тот момент, такими как размер или сложность кода.) Но если эти метрики показывают разные результаты, это прямо говорит о том, что одна из оценок, возможно, неверна и мы должны продолжить исследования. Если

же их результаты одинаковы, мы испытываем уверенность в том, что достигли какой-то истины.

Кроме того, некоторые сигналы могут не иметь соответствующих им метрик на определенном этапе. Рассмотрим пример. Академическая литература предлагает множество вариантов оценок качества кода, но ни один из них не является по-настоящему полноценным. Занявшись исследованием процесса контроля удобочитаемости, мы могли использовать метрику, плохо отражающую сигнал, и затем принять решение на ее основе или просто признать, что на данном этапе ее нельзя измерить. В итоге мы решили не учитывать эту метрику как количественную оценку, хотя и просили инженеров самим оценить качество их кода.

Следование модели GSM — это отличный способ прояснить цели оценивания некоего процесса и порядок измерения. Мы в Google используем качественные данные для проверки наших метрик, чтобы убедиться, что они достаточно полно отражают предполагаемый сигнал и охватывают процесс полностью.

Использование данных для проверки метрик

Однажды мы создали метрику для оценки средней задержки работы инженера из-за ожидания завершения сборки. Цель состояла в том, чтобы зафиксировать «типичное» время ожидания. Мы провели *выборочное исследование*. Согласно методике исследований, инженеры должны были отвлекаться от выполнения интересующей их задачи, чтобы ответить на несколько вопросов. Как только инженер запускал сборку, мы автоматически отправляли ему небольшой список вопросов о том, какие задержки на сборку он ожидает, опираясь на личный опыт. Однако в нескольких случаях инженеры ответили, что еще не начали сборку! Оказалось, что инструменты запускали сборку автоматически и инженеры не тратили время на ожидание результатов и это не «засчитывалось» в «типичную» задержку. Затем мы скорректировали метрику, чтобы исключить подобные сборки¹.

Количественные метрики полезны, потому что позволяют оценить масштаб. Вы можете измерить опыт инженеров по всей компании за длительный период и быть уверенными в результатах. Однако они не предоставляют никакой контекстной или описательной информации. Количественные показатели не объясняют, почему инженер использовал устаревший инструмент для выполнения своей задачи, выбрал необычный рабочий процесс или отказался от стандартного процесса. Только качественные исследования могут дать такую информацию и позволяют получить представление о возможных шагах по улучшению процесса.

Теперь рассмотрим сигналы, представленные в табл. 7.2. Какие метрики можно было бы использовать для измерения каждого из них? Некоторые из этих сигналов можно

¹ Наш опыт работы в Google показывает, что когда количественные и качественные показатели расходятся, это часто объясняется тем, что количественные показатели не отражают ожидаемый результат.

измерить методом анализа журналов. Другие можно измерить только с помощью опроса. Третьи вообще могут оказаться неизмеримыми — например, как измерить фактическое качество кода?

В итоге, исследуя влияние процесса контроля удобочитаемости на продуктивность, мы получили комбинацию метрик из трех источников. Первый источник — результаты опроса конкретно о процессе. Вопросы задавались людям, завершившим этот процесс. Их ответы позволили получить прямую обратную связь о процессе. Как мы предполагали, это должно было избавить нас от предвзятости в отзывах¹, хотя и вводило некоторую предвзятость, обусловленную временем² и ограниченностью выборки³. Второй — результаты крупномасштабного квартального опроса, который не был специально посвящен удобочитаемости; его целью было получение метрик, которые должны влиять на процесс контроля удобочитаемости. И третий — детализированные метрики из журналов инструментов разработчика, помогающие определить, сколько времени требовалось инженерам для выполнения конкретных задач⁴. Полный список метрик с соответствующими сигналами и целями представлен в табл. 7.2.

Таблица 7.2. Цели, сигналы и метрики

QUANTS	Цель	Сигнал	Метрика
Качество кода	Инженеры пишут более качественный код	Инженеры, получающие отзывы о своем коде, оценивают свой код как более качественный	Ежеквартальный опрос: доля инженеров, сообщивших, что удовлетворены качеством своего кода
		Процесс контроля удобочитаемости оказывает положительное влияние на качество кода	Опрос о процессе: доля инженеров, сообщивших, что проверки удобочитаемости не влияют или отрицательно влияют на качество кода
			Опрос о процессе: доля инженеров, сообщивших, что участие в процессе контроля удобочитаемости улучшило качество кода их команды

¹ Под предвзятостью здесь понимается предвзятость памяти. Люди чаще вспоминают особенно интересные или разочаровывающие события.

² Предвзятость, обусловленная временем событий, — это еще одна форма предвзятости памяти, когда люди лучше помнят свой последний опыт. В данном случае они, поскольку только что успешно завершили процесс, могут испытывать особенно яркие положительные эмоции.

³ Поскольку в опросе участвовали только те, кто завершил процесс, мы не учитывали мнение тех, кто этот процесс еще не завершил.

⁴ Есть соблазн использовать такие метрики для оценки отдельных инженеров или даже для определения высоких и низких показателей эффективности. Однако это было бы контрпродуктивно. Если для оценки эффективности использовать метрики продуктивности, инженеры быстро поймут это и такие метрики перестанут быть полезными для измерения и повышения продуктивности во всей организации. Единственный способ заставить эти метрики работать — оценивать ими не индивидуумов, а группы.

QUANTS	Цель	Сигнал	Метрика
	Инженеры пишут более согласованный код	Инженеры получают постоянную обратную связь и рекомендации от рецензентов	Опрос о процессе: доля инженеров, сообщивших о несоответствии между комментариями рецензентов и критериями удобочитаемости
	Инженеры вносят свой вклад в культуру единообразия кода	Инженеры регулярно получают комментарии, касающиеся проблем в оформлении и/или удобочитаемости кода	Опрос о процессе: доля инженеров, сообщивших, что регулярно получают комментарии, касающиеся проблем в оформлении и/или удобочитаемости кода
Внимание инженеров	Не рассмотрено	Не рассмотрено	Не рассмотрено
Интеллектуальная сложность	Инженеры изучают кодовую базу и лучшие практики программирования в Google	Инженеры отмечают, что процесс контроля удобочитаемости дает им новые знания	Опрос о процессе: доля инженеров, сообщивших, что они узнали что-то новое не менее чем в четырех актуальных темах
			Опрос о процессе: доля инженеров, сообщивших, что обучение или приобретение опыта является сильной стороной процесса контроля удобочитаемости
	Инженеры получают советы наставников	Инженеры отмечают положительное влияние общения с опытными инженерами, выступающими в роли рецензентов	Опрос о процессе: доля инженеров, сообщивших, что работа с рецензентами является сильной стороной процесса контроля удобочитаемости
Темп / скорость	Инженеры работают более продуктивно	Инженеры чувствуют себя продуктивнее коллег	Ежеквартальный опрос: доля инженеров, сообщивших, что они достигли высокой продуктивности
		Инженеры отмечают, что завершение процесса контроля удобочитаемости положительно повлияло на скорость их работы	Опрос о процессе: доля инженеров, сообщивших, что отсутствие контроля удобочитаемости снижает скорость работы команды
		Изменения, внесенные инженерами, прошедшими проверку, воспринимаются быстрее и проще, чем изменения, написанные другими инженерами	Данные из журналов: среднее время обзора изменений, внесенных инженерами, участвовавшими и не участвовавшими в процессе контроля удобочитаемости
		Изменения, внесенные инженерами, прошедшими проверку, легче преодолевают процесс рецензирования, чем изменения, написанные другими инженерами	Данные из журналов: среднее время рецензирования изменений, внесенных инженерами, участвовавшими и не участвовавшими в процессе контроля удобочитаемости

QUANTS	Цель	Сигнал	Метрика
		Изменения, внесенные инженерами, прошедшими проверку, быстрее проходят процесс рецензирования, чем изменения, написанные другими инженерами	Данные из журналов: среднее время утверждения изменений, внесенных инженерами, участвовавшими и не участвовавшими в процессе контроля удобочитаемости
		Проверка удобочитаемости не оказывает отрицательного влияния на скорость работы	Опрос о процессе: доля инженеров, сообщивших, что проверка удобочитаемости отрицательно влияет на скорость их работы Опрос о процессе: доля инженеров, сообщивших, что рецензенты быстро давали ответы Опрос о процессе: доля инженеров, сообщивших, что своевременность проверок была сильной стороной процесса контроля удобочитаемости
Удовлетворенность	Инженеры видят пользу от процесса контроля удобочитаемости и положительно оценивают свое участие в нем	Инженеры считают проверку удобочитаемости в целом полезным опытом Инженеры оценивают процесс контроля удобочитаемости как стоящий Инженеры не считают процесс контроля удобочитаемости разочарывающим	Опрос о процессе: доля инженеров, сообщивших, что они получили в целом положительный опыт от участия в процессе контроля удобочитаемости Опрос о процессе: доля инженеров, сообщивших о целесообразности процесса контроля удобочитаемости Опрос о процессе: доля инженеров, сообщивших, что качество проверки удобочитаемости является сильной стороной процесса Опрос о процессе: доля инженеров, сообщивших, что тщательность проверки является сильной стороной процесса Опрос о процессе: доля инженеров, сообщивших, что процесс контроля удобочитаемости является малопонятным, непрозрачным, медленным или разочарывающим Ежеквартальный опрос: доля инженеров, сообщивших, что они удовлетворены скоростью своей работы

Принятие мер и оценка результатов

Вспомним, что нашей целью в этой главе была выработка мер, способствующих увеличению продуктивности. Проводя исследования, наша команда всегда определяет список рекомендаций о направлениях дальнейшего совершенствования. Мы можем предложить добавить в инструменты новые возможности, уменьшить задержки в инструментах, улучшить документацию, избавиться от устаревших процессов или даже изменить структуру стимулов для инженеров. В идеале наши рекомендации основаны на «использовании инструментов»: нет смысла говорить инженерам, что они должны изменить процесс или мнение, если их инструменты не позволяют этого сделать. Вместо этого мы всегда предполагаем, что инженеры оценят соответствующие компромиссы, если у них будут все необходимые данные и подходящие инструменты.

Наше исследование показало, что в целом проверка удобочитаемости — стоящий процесс: инженеры, прошедшие его, были удовлетворены и чувствовали, что извлекли пользу. Журналы показали, что владение навыками удобочитаемости помогло инженерам научиться быстрее проверять свой код даже при меньшем участии рецензентов. Также исследование выявило направления совершенствования процесса: инженеры определили его болевые точки. Языковые группы воспользовались этими рекомендациями, усовершенствовали инструментарий и сделали процесс более быстрым, прозрачным и комфортным для инженеров.

Заключение

Мы в Google обнаружили, что наличие команды специалистов по продуктивности дает широкие преимущества в сфере программной инженерии: каждой команде не требуется определять свой курс в стремлении увеличить продуктивность, если централизованная команда может намного полнее сосредоточиться на решении таких проблем. Как известно, человеческий фактор, как и компромиссы, трудно измерить, и для экспертов важно понимать анализируемые данные. Команда специалистов по продуктивности должна основывать свои рекомендации на данных и стремиться устранить предвзятость.

Итоги

- Перед измерением продуктивности узнайте, будет ли использоваться полученный результат для принятия решений независимо от того, является ли он положительным или отрицательным. Если не предполагается делать никаких выводов по результатам, вероятно, нет смысла что-то измерять.
- Выберите метрики, используя модель GSM. Хорошая метрика — это разумное отражение сигнала. Ее можно проследить обратно до ваших первоначальных целей.

- Выберите метрики, охватывающие все продуктивности (QUANTS). Этим вы гарантируете, что не улучшите один аспект (например, скорость разработки) за счет другого (например, качества кода).
- Качественные метрики — тоже метрики! Подумайте о создании механизма опроса для выявления убеждений инженеров. Качественные метрики также должны соответствовать количественным метрикам. Если они противоречат друг другу, скорее всего, количественные метрики неверны.
- Старайтесь вырабатывать рекомендации, влияющие на рабочий процесс разработчика и структуру стимулов. Несмотря на то что иногда приходится рекомендовать дополнительное обучение или улучшение документации, изменения более вероятно будут применены, если они повлияют на повседневные привычки разработчика.

ЧАСТЬ III

Процессы

Правила и руководства по стилю

Автор: Шайндел Шварц

Редактор: Том Манирек

В большинстве организаций, занимающихся программной инженерией, есть правила, применяющиеся к кодовой базе. Они определяют, как в компании принято хранить файлы с исходным кодом, форматировать код, присваивать имена, использовать паттерны, исключения и потоки выполнения. Большинство инженеров-программистов действуют в рамках набора политик, управляющих их работой. В Google для управления базой кода мы поддерживаем набор руководств по стилю, которые определяют наши правила.

Правила — это не предложения или рекомендации, а обязательные для выполнения законы. При необходимости их можно игнорировать только в особых утвержденных случаях. Руководства, в отличие от правил, содержат рекомендации и описание переносимых практик, следование которым желательно, но не обязательно.

Мы собираем правила и в наших канонических руководствах по стилю программирования определяем допустимое и недопустимое поведение. Термин «стиль» в данном случае можно понять не совсем правильно — как набор вариантов форматирования. В действительности наши руководства по стилю — это нечто большее: они включают полный набор соглашений, управляющих нашим кодом. Это не значит, что эти руководства носят строго обязательный характер. Правила в них могут предлагать, например, использовать имена «настолько описательные, насколько это возможно, в пределах разумного» (<https://oreil.ly/xDNAn>). Скорее они служат основным источником правил, которым следуют наши инженеры.

Мы поддерживаем отдельные руководства по стилю для каждого языка программирования¹. Все они преследуют одну цель — обеспечить устойчивость кода, но между ними есть множество различий по охвату, длине и содержанию. Языки программирования имеют разные сильные стороны, особенности, приоритеты и причины использования в Google. Поэтому руководства для разных языков отличаются. Например, руководства для Dart, R и Shell лаконичны и сфокусированы на нескольких общих принципах, таких как именование и форматирование. Руководства для C++, Python

¹ Многие из наших руководств по стилю имеют внешние версии, которые вы можете найти по адресу <https://google.github.io/styleguide>. В этой главе мы приведем примеры из этих руководств.

и Java включают гораздо больше деталей, углубляются в особенности языка и представлены более подробными документами. В одних руководствах больше внимания уделяется типичному использованию языка вне Google — например, руководство для Go очень короткое и добавляет лишь несколько правил к правилам и методам, изложенным во внешних общепризнанных соглашениях (<https://oreil.ly/RHrvP>). В других содержатся правила, принципиально отличающиеся от внешних норм, — например, наши правила для C++ запрещают применять исключения — возможность языка, широко используемую за пределами Google.

Большая разница даже между нашими руководствами не позволяет точно описать, что должно включать руководство по стилю. В разных организациях могут быть разные требования к устойчивости, для поддержки которой создаются правила. В этой главе мы обсудим принципы и процессы, определившие подходы к разработке правил и руководств в Google, и рассмотрим примеры, в основном из наших руководств по стилю для C++, Python и Java.

Зачем нужны правила?

Итак, зачем нужны правила? Цель правил — поощрить «хорошее» поведение и препятствовать «плохому». Интерпретация понятий «хорошо» и «плохо» во многом зависит от организации и ее целей и потребностей. В одних организациях «хорошими» могут считаться модели, предполагающие использование минимального объема памяти или ориентированные на потенциальные оптимизации во время выполнения. В других «хорошим» поведением считается применение новейших возможностей языка. Третьи организации усиленно заботятся о согласованности и считают отклонения от паттернов «плохим» подходом. Сначала мы должны понять, что ценится в организации, а затем использовать правила и рекомендации, чтобы поощрять желательное поведение и препятствовать нежелательному.

По мере роста организации установленные правила и руководства формируют общий словарь разработки, который помогает инженерам сосредоточиться на том, что должен делать их код, а не на том, как заставить его это делать. Поддерживая словарь, инженеры действуют «хорошо», даже не замечая этого. То есть правила помогают компании развиваться в выбранных направлениях.

Создание правил

При определении набора правил ключевой вопрос заключается не в том, «какие правила мы должны иметь?», а в том — «какую цель мы преследуем?» Концентрация внимания на цели помогает определить, какие правила ее поддерживают. В Google, где руководство по стилю является законом, определяющим нормы кодирования, мы спрашиваем не «что входит в руководство по стилю?», а «почему что-то входит в руководство по стилю?» Что получает организация от набора правил, регулирующих написание кода?

Руководящие принципы

Рассмотрим конкретную ситуацию: в Google работает более 30 000 инженеров с разными навыками и опытом. Ежедневно в репозиторий с базой кода, насчитывающей более двух миллиардов строк, отправляется около 60 000 изменений, которые, вероятно, будут продолжать оставаться в репозитории десятилетиями. Наши продукты уникальны, но мы сталкиваемся с типичной организационной проблемой — как сохранить устойчивость инженерной среды при росте компании с течением времени.

В этом контексте цель наших правил — помочь нам справиться со сложной средой разработки, обеспечить управляемость кодовой базы и поддержать продуктивность инженеров. Здесь мы идем на компромисс: большой свод правил, помогающий достичь цели, ограничивает свободу выбора. Мы теряем некоторую гибкость и даже можем обидеть кого-то, но мы выбираем выигрыш в согласованности и уменьшении числа конфликтов, обеспечиваемый официальным стандартом.

Мы выработали ряд основополагающих принципов, которыми руководствуемся при разработке правил:

- необременительность;
- оптимизация для читателя кода;
- поддержка единообразия кода;
- предотвращение использования необычных или способствующих ошибкам конструкций;
- при необходимости — оценка практической целесообразности применения правила.

Необременительность

Руководство по стилю не должно включать слишком много правил. Чтобы все инженеры в организации изучили и начали выполнять новое правило, требуется время. Если правил окажется слишком много¹, инженерам будет трудно их запомнить, а новичкам — изучить. Кроме того, чем больше свод правил, тем сложнее и дороже его поддерживать.

Мы сознательно решили не фиксировать очевидные правила. Руководство по стилю в Google не носит юридический характер: отсутствие в нем явного запрета на действие не означает законность этого действия. Например, руководство для C++ не содержит правил, запрещающих использование `goto`. Программисты на C++ уже

¹ Особую важность здесь приобретают инструменты. Понятие «слишком много» определяет не количество правил в руководстве, а количество сведений, которое должен запомнить инженер. Например, в старые темные времена, предшествовавшие появлению clang, инженеру требовалось помнить массу правил форматирования. Эти правила никуда не делись, но благодаря появлению современных инструментов соблюдать их стало намного проще. Мы достигли точки, когда кто-то может добавить любое количество правил, и никто не обратит на это внимания, потому что инструмент применит правило автоматически.

склонны избегать этого оператора, поэтому добавление правила, запрещающего его, приведет к ненужным накладным расходам. Один или два инженера могут по ошибке использовать `goto`, но это не повод увеличивать нагрузку для всех созданием нового правила.

Оптимизация для читателя кода

Еще один принцип, которому следуют наши правила, — оптимизация кода для читателя, а не для автора. С течением времени код будет читаться другими людьми гораздо чаще, чем автором. Мы предпочитаем, чтобы код был более утомительным для ввода и менее трудным для чтения. В нашем руководстве по стилю для Python мы отмечаем, что условные выражения короче операторов `if` и удобны для ввода, но читателю кода бывает трудно разобраться в них, поэтому мы ограничиваем их использование (<https://oreil.ly/ftyvG>). Это компромисс: мы понимаем, что инженерам сложнее снова и снова вводить длинные описательные имена переменных и типов, но готовы заплатить эту цену за удобство всех будущих читателей.

В соответствии с этими приоритетами мы также требуем, чтобы инженеры оставляли явное доказательство предполагаемого поведения кода. Читатель должен четко понимать, что делает код. Например, наши руководства для Java, JavaScript и C++ требуют использовать специальную аннотацию или ключевое слово всякий раз, когда метод переопределяет соответствующий метод суперкласса. Без явного подтверждения читателя, вероятно, могут понять намерение автора, но это займет у них больше времени и усилий.

Предупреждение о неожиданном поведении еще более важно. В C++ бывает сложно понять, что владеет указателем, просто прочитав фрагмент кода. Если указатель передается в функцию, поведение которой неизвестно, мы не можем уверенно предсказать ожидаемый результат. Вызывающий код по-прежнему владеет указателем? Ответственность за указатель взяла на себя функция? Можно ли продолжать использовать указатель после возврата из функции или он стал недействительным? Чтобы ответить на эти вопросы, наше руководство для C++ требует использовать `std::unique_ptr` (<https://oreil.ly/h0lFE>), когда предполагается передача права владения указателем, или конструкцию `unique_ptr`, когда в каждый конкретный момент существует только одна копия указателя. Когда функция принимает аргумент типа `unique_ptr` и намеревается стать владельцем указателя, вызывающий код должен явно использовать семантику перемещения:

```
// Функция, которая принимает Foo*.
// По заголовку этой функции невозможно сказать, вступает ли она во владение
// указателем
void TakeFoo(Foo* arg);

// Вызов функции не позволяет читателю понять, как изменится право владения
// указателем после возврата из нее
Foo* my_foo(NewFoo());
TakeFoo(my_foo);
```

Сравните этот фрагмент со следующим:

```
// Функция принимает std::unique_ptr<Foo>.
void TakeFoo(std::unique_ptr<Foo> arg);

// Любой вызов этой функции явно показывает, что ей передается право
// владения указателем и unique_ptr нельзя использовать после возврата из нее
std::unique_ptr<Foo> my_foo(FooFactory());
TakeFoo(std::move(my_foo));
```

Данное правило в руководстве по стилю гарантируют, что все инструкции вызова будут содержать четкие доказательства передачи права собственности, когда это применимо, и читателям кода будет не нужно понимать поведение каждого вызова. Мы предоставляем в API достаточно информации, чтобы дать возможность рассуждать о его взаимодействиях. Такое четкое документирование поведения в точках вызова гарантирует, что фрагменты кода остаются читаемыми и понятными. Мы стремимся локализовать рассуждения, касающиеся событий в точке вызова, и избавить читателей от необходимости искать и читать другой код, включая реализацию функции.

Также мы добавили в руководство по стилю множество правил комментирования. Документирующие комментарии (блочные комментарии в начале файла, а также перед определениями классов и функций) должны описывать структуру или назначение следующего за ними кода. Комментарии внутри реализации (разбросанные по всему коду) должны описывать или выделять неочевидные варианты, объяснять сложные фрагменты и подчеркивать важные части кода. У нас есть правила, охватывающие оба типа комментариев, которые требуют от инженеров давать пояснения для читателей кода.

Поддержка единообразия кода

Наше понимание единообразия в кодовой базе похоже на философию, которую мы применяем в офисах. Команды с большим числом инженеров часто занимают несколько офисов, каждый из которых имеет свою уникальную индивидуальность с учетом местного колорита, но во всех них неизменно присутствует все, что нужно для работы. Пропуск гуглера распознается всеми считывателями, любые устройства в Google подключаются к WiFi, аппаратура для видеоконференций в любом конференц-зале имеет одинаковый интерфейс. Гуглеру не нужно тратить время на изучение особенностей использования устройств, и перемещаясь между офисами, он может быстро приступить к работе.

Такого же единообразия мы стремимся добиться в исходном коде — любой инженер должен быстро вникать в незнакомый код компании. Локальный проект может иметь свою индивидуальность, но его инструменты, методы и библиотеки должны быть узнаваемы и понятны.

Преимущества единообразия

Несмотря на то что запрет на настройку устройств чтения пропусков или интерфейса аппаратуры для видеоконференций ограничивает творческую свободу, с его помощью

мы получаем преимущества единообразия. То же верно и для кода: правила, требующие обеспечивать единообразие, позволяют большему числу инженеров выполнять больше работы с меньшими усилиями¹:

- Когда база кода единообразна по стилю, инженеры, пишущие и читающие код, могут сосредоточиться на смысле кода, а не на его оформлении. Также единообразие позволяет формировать фрагменты для экспертной оценки². Когда мы решаем задачи, используя одинаковые интерфейсы, и единообразно форматируем код, эксперты быстро понимают, что этот код делает. Единообразие также упрощает деление кода на модули и помогает избежать дублирования. По этим причинам мы уделяем большое внимание единообразию в именовании, использовании паттернов, форматировании и структуре. В руководстве по стилю есть множество правил, добавленных исключительно для гарантии единообразия, например определяющих количество пробелов в отступах или ограничивающих длину строки³. Главной ценностью таких правил является их наличие, а не содержание.
- Единообразие облегчает масштабирование. Инструменты — это ключ к масштабированию организации, а единообразный код облегчает создание инструментов, способных понимать, редактировать и генерировать код. Представьте инструмент, который обновляет исходные файлы, добавляя отсутствующие инструкции импорта или удаляя инструкции включения неиспользуемых заголовков. Если в разных проектах к спискам импорта применены разные политики сортировки, для некоторых проектов такой инструмент может оказаться бессильным. Когда все команды используют одни и те же правила, открывается возможность инвестировать в создание инструментов, работающих повсюду, и в автоматизацию многих задач по обслуживанию.
- Единообразие также помогает масштабированию человеческого ресурса в компании. По мере роста организации увеличивается число инженеров. Поддержание единообразия в коде облегчает их переход между проектами, сокращает время на их включение в работу и дает организации возможность гибко приспосабливаться по мере изменения потребностей в численности персонала. В растущей организации также будут появляться люди, взаимодействующие с кодом не как разработчики, например инженеры по надежности, инженеры по библиотекам и «уборщики» кода. Благодаря единообразию эти люди могут работать сразу над несколькими ранее не знакомыми проектами.

¹ Спасибо Х. Райту за фактическое сравнение, сделанное при посещении примерно 15 разных офисов Google.

² «Формирование фрагментов» — это когнитивный процесс группировки элементов информации в значимые «фрагменты», а не их индивидуальный учет. Напоминает рассмотрение общей картины на шахматной доске вместо оценки позиций отдельных фигур.

³ См. разделы: «4.2. Block indentation: +2 spaces» (<https://oreil.ly/jaf6n>), «Spaces vs. Tabs» (<https://oreil.ly/1AMEq>), «4.4. Column limit:100» (<https://oreil.ly/WhufW>) и «Line Length» (<https://oreil.ly/sLctK>).

- Единообразия также обеспечивает устойчивость. Со временем инженеры покидают проекты, собственники меняются, а проекты объединяются или разделяются. Единообразия кодовой базы гарантирует низкую стоимость переходов и дает нам практически неограниченную текучесть как кода, так и инженеров, работающих с ним, упрощая долгосрочную поддержку.
-

В МАСШТАБЕ

Несколько лет назад наше руководство по стилю для C++ строго запрещало нарушения совместимости со старым кодом: «В какой-то момент могут появиться весьма веские аргументы для изменения правил стиля, но мы постараемся сохранить все как есть, чтобы обеспечить единообразия».

Пока база кода была небольшой и в ней почти не было старых пыльных углов, в этом запрете был смысл.

Но по мере разрастания и старения кодовой базы сохранение совместимости перестало быть приоритетной задачей. Такое изменение было (по крайней мере, для арбитров руководства по стилю для C++) осознанным: мы явно заявили, что база кода на C++ никогда больше не будет полностью единообразной.

Было бы слишком обременительно обновлять правила по мере развития современных практик и требовать применения этих правил ко всему, что когда-либо было написано. Наши инструменты и процессы крупномасштабного изменения позволяют нам обновлять почти весь код и приводить его в соответствие почти к каждому новому паттерну или синтаксису, поэтому большая часть старого кода соответствует самому последнему одобренному стилю (глава 22). Однако такой подход не идеален: когда база кода становится такой большой, как наша, невозможно гарантировать, что каждый фрагмент старого кода будет соответствовать современным практикам. Требование идеального единообразия достигло той точки, когда оно стало обходиться слишком дорого.

Установка стандарта. Мы склонны концентрироваться на внутреннем единообразии. Иногда локальные соглашения возникают раньше глобальных, и порой нецелесообразно подгонять внутренние правила под внешние. Мы выступаем за иерархию единообразия: нормы, принятые в данном файле, предшествуют нормам данной команды, предшествующим нормам более крупного проекта, которые, в свою очередь, предшествуют нормам всей кодовой базы. Руководства по стилю содержат правила, которые явно относятся к локальным соглашениям¹ и считают локальное единообразия важнее научно-технического.

Однако не всегда достаточно создать и соблюдать набор внутренних соглашений. Иногда необходимо принимать во внимание стандарты, принятые внешним сообществом.

Если внутренние соглашения уже существуют, желательно привести их в соответствие с внешними правилами. Для небольших, самодостаточных и недолговечных проектов это, вероятно, не будет иметь смысла — внутреннее единообразия важнее

¹ Примером может служить использование констант (<https://oreil.ly/p6RLR>).

происходящего за границами небольшого проекта. Но как только время и возможность масштабирования приобретут для проекта большее значение, возрастет вероятность, что его код будет пересекаться с внешними проектами или выйдет за пределы компании. Тогда соблюдение общепринятых стандартов с лихвой окупится в долгосрочной перспективе.

КОЛИЧЕСТВО ПРОБЕЛОВ

Руководство по стилю для Python в Google изначально обязывало использовать в коде на Python отступы с двумя пробелами. Стандартное руководство по стилю для Python, используемое внешним сообществом, предлагает оформлять отступы четырьмя пробелами. Большая часть кода на Python, написанного нами раньше, была направлена на непосредственную поддержку наших проектов на C++, а не на создание действующих приложений на Python. Поэтому мы решили использовать отступы с двумя пробелами, в соответствии с правилами оформления кода на C++. Шло время, и мы увидели, что это решение не оправдало ожиданий. Инженеры, использующие Python, гораздо чаще читают и пишут код на Python, а не на C++, и им нужно прикладывать дополнительные усилия, чтобы что-то найти или задействовать фрагменты внешнего кода. Мы также сталкивались с большими трудностями каждый раз, когда пытались экспортировать части нашего кода в открытый исходный код, тратя время на согласование различий между нашим внутренним кодом и внешними проектами, к которым мы хотели присоединиться.

Когда пришло время создать для Starlark (<https://oreil.ly/o7aY9>) — языка на основе Python, разработанного в Google для описания процессов сборки — руководство по стилю, мы прописали использование отступов с четырьмя пробелами, чтобы обеспечить единообразие с внешними правилами¹.

Предотвращение использования необычных или способствующих ошибкам конструкций

Наши руководства по стилю ограничивают использование некоторых самых необычных или сложных конструкций, содержащих малозаметные ловушки. Применение этих конструкций без глубокого понимания всей их сложности легко может приводить к ошибкам. Даже если конструкции понятны действующим инженерам, будущие участники проекта и специалисты, занимающиеся сопровождением кода, могут не иметь такого же понимания.

Так, правило в руководстве по стилю для Python запрещает использование функций рефлексии (<https://oreil.ly/ooqIr>), поскольку такие функции, как `hasattr()` и `getattr()`, дают пользователю возможность обращаться к атрибутам объектов, используя строковые имена:

```
if hasattr(my_object, 'foo'):  
    some_var = getattr(my_object, 'foo')
```

¹ Стиль форматирования для файлов BUILD, реализованных на Starlark, используется инструментом сборки `buildifier` (<https://oreil.ly/iGMoM>).

Казалось бы, в этом примере все в порядке. Но взгляните на следующий фрагмент:

`some_file.py`:

```
A_CONSTANT = [  
    'foo',  
    'bar',  
    'baz',  
]
```

`other_file.py`:

```
values = []  
for field in some_file.A_CONSTANT:  
    values.append(getattr(my_object, field))
```

При беглом просмотре не сразу видно, что здесь происходит обращение к полям `foo`, `bar` и `baz`. В коде нет четких этому подтверждений. Вам придется приложить определенные усилия, чтобы выяснить, какие строки используются для доступа к атрибутам вашего объекта. А что, если вместо `A_CONSTANT` код будет читать поля из ответа, полученного от механизма RPC, или из хранилища данных? Такой запутанный код может проделать незаметную брешь в системе безопасности из-за неправильной проверки сообщения. Кроме того, такой код сложно тестировать и проверять.

Динамическая природа Python допускает такое поведение и разрешает использовать `hasattr()` и `getattr()` в очень ограниченных случаях.

Такие возможности языка помогают решать задачи эксперту, знакомому с ними, но часто они слишком сложны для понимания и довольно редко используются инженерами. Нам нужно, чтобы с базой кода работали не только эксперты, но и начинающие программисты. Наши инженеры по надежности находят подозрительные места в коде, даже написанном на языке, которым они владеют недостаточно свободно. Мы придаем большое значение интерпретации кода.

Практическая целесообразность уступок

По словам Ральфа Уолдо Эмерсона, «глупая последовательность — суеверие недалеких умов» (<https://oreil.ly/bRFg2>). В нашем стремлении к единообразию и простоте кодовой базы мы не стараемся слепо игнорировать все остальное. Мы знаем, что некоторые правила в наших руководствах по стилю будут сталкиваться с ситуациями, требующими исключений, и это нормально. При необходимости мы отклоняемся от правил в угоду эффективности и практической целесообразности.

Производительность имеет значение. Иногда есть смысл принять меры по оптимизации производительности даже в ущерб единообразию и удобочитаемости. Например, наше руководство для C++ запрещает использование исключений, но разрешает применение `noexcept` (<https://oreil.ly/EAgN->) — спецификатора языка, связанного с исключениями, который может включать оптимизацию компилятора.

Совместимость тоже имеет значение. Код, предназначенный для взаимодействия с чем-то, разработанным за пределами Google, нужно адаптировать для этой цели.

Например, наше руководство для C++ содержит исключение из общего правила по оформлению имен с использованием ВерюблужьегоРегистра, которое разрешает использование змеиного_регистра, как в стандартной библиотеке, для именования сущностей, имитирующих особенности стандартной библиотеки¹. Руководство для C++ также допускает несоблюдение некоторых правил при программировании для Windows (<https://oreil.ly/xCrwV>), где совместимость с платформой требует использования множественного наследования, которое явно запрещено для всего остального кода на C++ в Google. В наших руководствах по стилю для Java и JavaScript прямо указано, что сгенерированный код, который часто взаимодействует с компонентами, не принадлежащими проекту, или зависит от них, не обязан подчиняться правилам руководств². Единообразия является жизненно важным, а адаптация имеет ключевое значение.

Руководство по стилю

Итак, какие правила входят в руководство по стилю языка? Все они делятся примерно на три категории:

- правила, предотвращающие опасности;
- правила, отражающие передовые практики;
- правила, обеспечивающие единообразие.

Предотвращение опасностей

Прежде всего наши руководства по стилю включают правила об особенностях языка, которые должны или не должны использоваться по техническим причинам. У нас есть правила использования статических членов, переменных, лямбда-выражений, исключений, потоков выполнения, прав доступа и наследования классов. В своих правилах мы определяем, какие особенности языка (например, стандартные типы) можно использовать, а какие следует применять с осторожностью, чтобы избежать появления малозаметных ошибок. В описании каждого правила мы стремимся объяснить плюсы и минусы принятого нами решения. Большинство из этих решений основаны на необходимости обеспечить надежность кода с течением времени, а также поощрить поддерживаемые приемы использования языка.

Отражение передовых практик

Наши руководства по стилю также включают правила применения некоторых передовых практик программирования. Эти правила помогают поддерживать работоспособ-

¹ См. раздел «Exceptions to Naming Rules» (<https://oreil.ly/AiTjH>). Например, наши библиотеки Abseil с открытым исходным кодом используют змеиный_регистр для именования типов, предназначенных для использования вместо стандартных типов. См. определения типов в <https://github.com/abseil/abseil-cpp/blob/master/absl/utility/utility.h>. Это C++11-реализации типов из стандарта C++14, для именования которых использован змеиный_регистр вместо более предпочтительного в Google ВерблужьегоРегистра.

² См. раздел «Generated code: mostly exempt» (<https://oreil.ly/rGmA2>).

ность кодовой базы и простоту ее сопровождения. Например, мы указываем, где и как авторы должны включать комментарии¹. Наши правила, касающиеся комментариев, охватывают общие соглашения о комментировании и определяют конкретные случаи обязательного включения комментариев в код — случаи, когда намерение автора не очевидно, такие как провалы через операторы `case` в инструкции `switch`, пустые блоки перехвата исключений и метапрограммирование шаблонов. У нас также есть правила, определяющие структуру исходных файлов и описывающие организацию их содержимого. У нас есть правила именования пакетов, классов, функций и переменных. Цель всех этих правил — подталкивать инженеров к использованию практик, помогающих писать более здоровый и устойчивый код.

Некоторые из передовых практик, предписываемых нашими руководствами по стилю, призваны сделать исходный код более удобочитаемым, например через правила форматирования. Наши руководства по стилю указывают, когда и как использовать отступы и пустые строки, определяют длину строк и выравнивание скобок. Требования к форматированию для некоторых языков мы реализовали в инструментах автоматического форматирования: `gofmt` для Go и `dartfmt` для Dart. Указывая подробный список требований к форматированию или называя инструмент, который необходимо использовать, мы преследуем одну и ту же цель: применить ко всему коду единообразный набор правил форматирования для улучшения его читабельности.

КЕЙС: ВНЕДРЕНИЕ `std::unique_ptr`

Когда в C++11 появился `std::unique_ptr` — тип «умного» указателя, выражающий исключительное владение динамически размещаемым объектом и удаляющий объект, когда указатель `unique_ptr` выходит из области видимости, — наше руководство по стилю изначально запретило его использовать. Поведение `unique_ptr` и связанная с ним семантика перемещения были не знакомы большинству инженеров. Предотвращение попадания `std::unique_ptr` в кодовую базу казалось нам более безопасным выбором. Мы обновили наши инструменты, чтобы находить ссылки на запрещенный тип, и сохранили в существующем руководстве рекомендацию использовать другие типы «умных» указателей.

Время шло. Инженеры постепенно осваивали семантику перемещения, и мы все больше убеждались, что использование `std::unique_ptr` прямо соответствует целям нашего руководства по стилю. Информация о владении объектом, которую сообщает `std::unique_ptr` в точке вызова функции, значительно упрощает чтение кода. Дополнительная сложность, связанная с этим новым типом и сопровождающей его семантикой перемещения, все еще вызывала серьезную обеспокоенность, но возможность значительно улучшить общее состояние кодовой базы в долгосрочной перспективе решила компромисс в пользу применения `std::unique_ptr`.

¹ Основные правила комментирования для нескольких языков см. в <https://google.github.io/styleguide/cppguide.html#Comments>, <http://google.github.io/styleguide/pyguide#38-comments-and-docstrings> и <https://google.github.io/styleguide/javaguide.html#s7-javadoc>.

Наши руководства по стилю также включают ограничения на использование новых и малопонятных особенностей языка, чтобы установить защитные ограждения вокруг потенциальных ловушек, пока сотрудники не пройдут обучение. Сразу после появления новых особенностей языка мы не всегда уверены в том, что правильно понимаем, как их использовать. По мере их изучения инженеры запрашивают у владельцев руководств по стилю разрешение на их использование. Наблюдая за поступающими запросами, мы получаем представление об особенностях использования новых возможностей и набираем достаточно примеров, чтобы обобщить хорошие практики, отделить их от плохих и, наконец, внести изменения в правила.

Обеспечение единообразия

Наши руководства по стилю также содержат правила, охватывающие множество мелких аспектов. Цель этих правил — просто принять и задокументировать решения. Многие правила в этой категории не оказывают существенного влияния на код. Такие правила, как соглашения об именовании, оформлении отступов и порядке инструкций импорта, обычно не дают четкого и измеримого преимущества одной формы перед другой, что может быть причиной продолжения обсуждения разных вариантов в сообществе¹. Приняв решение, мы останавливаем бесконечный цикл обсуждений и продолжаем двигаться дальше. Наши инженеры больше не выбирают, что лучше — два пробела или четыре. В этой категории правил важен не конкретный выбор, а сам факт выбора.

И все остальное

При этом в наших руководствах по стилю многое отсутствует. Мы стараемся сосредоточиться на том, что оказывает наибольшее влияние на здоровье кодовой базы. Существуют намного лучшие практики, не указанные в этих документах, в том числе фундаментальные советы: не умничайте, избегайте ветвления кодовой базы, не изобретайте велосипед и т. д. Такие документы, как наши руководства по стилю, не могут помочь новичку полностью овладеть мастерством программирования — есть некоторые вещи, которые мы подразумеваем и делаем это намеренно.

Изменение правил

Наши руководства по стилю не статичны. С течением времени ситуация и факторы, повлиявшие на установление правила, могут измениться. Причиной обновления правила может стать выход новой версии языка. Если правило заставляет инженеров прикладывать усилия, чтобы обойти его, а инструменты, используемые для соблюдения правила, становятся чрезмерно сложными и обременительными для поддержки, значит, правило потеряло актуальность. Определение момента, когда правило следует пересмотреть, является важной частью процесса сохранения актуальности и релевантности набора правил.

¹ Такие дискуссии лишь отвлекают внимание (<http://aquamarine.bikeshed.com>) и иллюстрируют закон тривиальности Паркинсона (<https://oreil.ly/L-K8F>).

Решения, лежащие в основе правил в наших руководствах по стилю, подкреплены обоснованиями. Добавляя правила, мы тратим время на обсуждение и анализ его плюсов, минусов и потенциальных последствий, пытаюсь убедиться, что данное изменение подходит для Google, и включаем эти соображения в большинство статей руководств по стилю.

Документирование обоснования того или иного решения дает нам преимущество: позволяет понять, когда что-то должно измениться. С течением времени и изменением условий хорошее решение, принятое ранее, может оказаться не лучшим. Имея перечень четко обозначенных факторов, мы можем определить, когда изменения, связанные с одним или несколькими из этих факторов, требуют переоценки правила.

КЕЙС: ОФОРМЛЕНИЕ ИМЕН С ИСПОЛЬЗОВАНИЕМ ВЕРБЛЮЖЬЕГО РЕГИСТРА

Когда мы в Google выпустили начальный вариант руководства по стилю для Python, мы решили использовать для имен методов ВерблюжийРегистр вместо змеиного_регистра. В руководстве по стилю для Python (PEP 8, <https://oreil.ly/Z9AA7>) и в большей части сообщества Python использовался змеиный_регистр, однако в ту пору в Google язык Python использовался в основном разработчиками на C++ для создания сценариев, действующих поверх кодовой базы на C++. Многие из определяемых ими типов на Python были обертками для соответствующих типов на C++, а поскольку соглашения об именовании для C++, принятые в Google, отдают предпочтение ВерблюжьемуРегистру, считалось важным сохранить единообразие между языками.

Позднее мы подошли к созданию и поддержке независимых приложений на Python силами инженеров, работающих на Python, а не инженеров C++, пишущих короткие сценарии. Наше руководство по стилю создавало некоторые неудобства и проблемы с удобочитаемостью для инженеров Python, требуя придерживаться единого стандарта для нашего внутреннего кода, из-за чего инженеры были вынуждены постоянно корректировать код под другой стандарт, когда им приходилось использовать внешний код. Руководство также затрудняло обучение новых сотрудников, уже имеющих опыт программирования на Python, и их адаптацию к нашим нормам оформления кодовой базы.

С развитием наших проектов на Python наш код все чаще взаимодействовал с внешними проектами на Python. В некоторых проектах мы использовали сторонние библиотеки, что привело к смещению в кодовой базе нашего ВерблюжьегоРегистра с внешним змеиным_регистром. Когда мы начали открывать исходный код наших проектов на Python, необходимость их поддержки во внешнем мире, где наши соглашения считались неконформистскими, добавило сложностей для нас и вызвало настороженность сообщества, которому наш стиль показался непривычным и странным.

Учитывая эти аргументы и после обсуждения отрицательных (потеря единообразия с другим кодом в Google, переобучение гуглеров, использующих наш стиль на Python) и положительных (достижение единообразия с внешним кодом на Python, позволяющим беспрепятственно использовать сторонние библиотеки) сторон, арбитры руководства по стилю для Python решили изменить правило. В результате руководство по стилю для Python в Google было обновлено и разрешило использовать змеиный_регистр в именах, но с оговорками: этот стиль должен единообразно применяться во всем файле, разрешение не распространяется на существующий код и команды сами могут решать, какой стиль лучше для конкретного проекта.

Процесс обновления правил

Понимая, что правила должны периодически изменяться, а также учитывая наше стремление к увеличению срока службы кода и масштабированию, мы определили процесс обновления наших правил. Процесс изменения руководства по стилю основан на решениях. Мы оформляем предложения по обновлению руководства, идентифицируя существующую проблему и представляя предлагаемое изменение как способ ее устранения. «Проблемы» в этом процессе не являются гипотетическими примерами того, что может пойти не так, — они подтверждены паттернами, найденными в существующем коде Google. Рассматривая продемонстрированную проблему и имея подробное обоснование существующего решения по стилю, мы можем пересмотреть его, исследовав оправданность другого решения.

Сообщество инженеров, пишущих код в соответствии с руководством по стилю, часто раньше других замечает необходимость изменить то или иное правило. Но большинство изменений в руководствах Google начинаются с обсуждения в сообществе. Любой инженер может задать вопрос или предложить изменение в списках рассылки по конкретным языкам, посвященным обсуждениям руководств.

Предложения по изменению руководства могут быть полностью сформированными и включать конкретные формулировки или начинаться с вопросов о применимости существующего правила. Поступающие идеи обсуждаются сообществом и получают отзывы от других пользователей языка. Некоторые предложения отклоняются сообществом: признаются ненужными, слишком двусмысленными или бесполезными. Другие получают положительные отзывы, оцениваются как заслуживающие внимания и, возможно, обрастают уточнениями. Предложения, прошедшие обсуждение в сообществе, принимаются для выработки окончательного решения об изменении правила.

Арбитры стилей

В Google окончательные решения в отношении руководства по стилю принимаются арбитрами по стилю. Для каждого языка программирования существует группа опытных экспертов по языку, которые являются владельцами руководства по стилю и назначаются на эту роль лицами, принимающими решения. Арбитрами по стилю могут быть ведущие инженеры из команд, занимающихся разработкой библиотек для данного языка, и другие гуглеры, обладающие соответствующим опытом.

Решение о любом изменении в руководстве по стилю принимается только после обсуждения технических компромиссов. Арбитры принимают решение в контексте согласованных целей, для которых оптимизируется руководство по стилю. При принятии решения учитываются не личные предпочтения, а только компромиссы. В настоящее время группа арбитров по стилю C++ состоит из четырех членов. Это может показаться странным: четное число членов комитета может помешать принятию решения, если голоса разделятся поровну. Однако из-за особенностей подхода к принятию решений, в котором никогда не используется аргумент «потому что я так

думаю», решения принимаются на основе консенсуса, а не голосования. В результате группа из четырех человек прекрасно справляется со своими обязанностями.

Исключения

Да, наши правила — закон, и поэтому некоторые правила требуют исключений. Правила обычно рассчитаны на общий случай, а для конкретных ситуаций может быть полезно определить исключение. Когда возникает нетипичная ситуация, проводятся консультации с арбитрами стиля, чтобы определить, действительно ли есть веские основания для изменения определенного правила.

Внести исключение нелегко. Если для C++ предлагается ввести макрос, руководство по стилю требует, чтобы его имя включало префикс с названием проекта. Из-за того что макросы в C++ обрабатываются как члены глобального пространства имен, для предотвращения конфликтов все макросы, экспортируемые из заголовочных файлов, должны иметь глобально уникальные имена. Правило в руководстве по стилю, касающееся именования макросов, допускает исключения, предоставленные арбитром, для некоторых служебных макросов, которые действительно являются глобальными. Но когда причина запроса на исключение сводится к личным предпочтениям и обусловлена чрезмерной длиной имени макроса или желанием добиться единообразия в рамках одного проекта, такой запрос отклоняется. Целостность кодовой базы важнее единообразия проекта.

Исключения допускаются в тех случаях, когда выясняется, что полезнее разрешить нарушить правило, чем препятствовать этому. Руководство по стилю для C++ запрещает неявные преобразования типов, включая конструкторы с одним аргументом. Однако для типов, предназначенных для прозрачного обертывания других типов, где базовые данные сохраняют точное представление, вполне разумно разрешить неявное преобразование. В таких случаях добавляется исключение из правила неявного преобразования. Наличие четкого обоснования для исключений может указывать на то, что данное правило необходимо уточнить или изменить. Однако для этого конкретного правила поступает достаточно много запросов на исключения, которые только кажутся подходящими для создания исключения, но в действительности таковыми не являются — либо потому, что конкретный рассматриваемый тип не является прозрачным типом-оберткой, либо потому, что тип является оберткой, но в таком исключении нет необходимости.

Руководства

В дополнение к правилам мы курируем руководства по программированию в различных формах, начиная от углубленного обсуждения сложных тем и заканчивая краткими советами по передовым практикам, которые мы поддерживаем.

Руководства представляют опыт, накопленный нашими инженерами, документируют передовые практики и извлеченные уроки. Как правило, руководства фокусируются на часто встречающихся ошибках или на новых возможностях, которые малоизвестны

и поэтому могут стать источниками путаницы. Если выполнять правила «необходимо», то руководства — «следует».

Одним из примеров сборника руководств, который мы развиваем, является набор учебников для некоторых языков, наиболее широко используемых у нас. Если наши руководства по стилю носят предписывающий характер и определяют, какие особенности языка разрешены, а какие запрещены, учебники для начинающих носят более описательный характер и объясняют особенности, одобренные в руководствах по стилю. Они освещают почти все, что должен знать инженер, плохо знакомый с особенностями использования языка в Google, но не углубляются в каждую деталь темы, давая только объяснения и рекомендации. Если инженер хочет узнать, как применять ту или иную особенность, учебники для начинающих послужат ему хорошим ориентиром.

Несколько лет назад мы начали публиковать серию советов по C++, в которых предлагался комплекс общих рекомендаций по использованию этого языка в Google. В этих советах мы рассмотрели сложные аспекты — время жизни объекта, семантику копирования и перемещения, поиск, зависящий от аргумента. Также мы осветили новшества — возможности C++11 в кодовой базе, типы C++17 (такие как `string_view`, `optional` и `variant`) — и затронули другие аспекты, о которых стоит упомянуть; например, что вместо использования директивы `using` нужно обращать внимание на предупреждения, чтобы не пропустить неявные преобразования в тип `bool`. Советы основаны на реальных проблемах, возникавших при решении реальных задач, которые не рассматриваются в каноничных руководствах по стилю. Основанные на практическом опыте, а не абстрактных идеях, советы широко применимы и считаются своего рода «каноном повседневной работы». Советы имеют узкую направленность и краткую форму: каждый из них читается не более нескольких минут. Серия «Совет недели» пользуется большим успехом внутри компании и содержит ссылки на обзоры кода и технические обсуждения¹.

Инженеры-программисты приходят в новый проект или кодовую базу со знанием языка программирования, но они не знают, как этот язык используется в Google. Чтобы восполнить этот пробел, мы поддерживаем серию курсов «<язык> @Google 101» для каждого из используемых языков программирования. Рассчитанные на полный рабочий день курсы посвящены обсуждению особенностей использования языка в нашей кодовой базе. Они охватывают наиболее часто используемые библиотеки и идиомы, внутренние настройки и применение пользовательских инструментов. Эти курсы сделают из инженера C++ хорошего инженера Google C++.

В дополнение к обучающим курсам мы готовим справочные материалы, чтобы быстро включить в работу нового сотрудника, совершенно незнакомого с нашей средой. Эти справочники различаются по форме и охватывают языки, которые мы используем. Вот некоторые полезные справочники, которые мы поддерживаем для внутреннего использования:

¹ Некоторые наиболее популярные советы можно найти по адресу: <https://abseil.io/tips>.

- Рекомендации по конкретным языкам для областей, как правило, вызывающих сложности (например, конкурентность и хеширование).
- Подробное описание новых особенностей, появившихся в обновлении языка, и рекомендации по их использованию в кодовой базе.
- Списки ключевых абстракций и структур данных, предлагаемых нашими библиотеками. Этот справочник помогает нам не изобретать заново структуры, которые уже существуют, и дает ответ на вопрос: «Как это называется в наших библиотеках?»

Применение правил

Природа правил такова, что они приобретают ценность, если выполняются неукоснительно. Обеспечить следование правилам можно педагогическими методами или техническими — с помощью инструментов. У нас в Google есть разные официальные учебные курсы, охватывающие многие из лучших практик, применения которых требуют наши правила. Мы также заботимся об актуальности документации. Ключевой частью нашего подхода к изучению правил являются обзоры кода. Процесс контроля удобочитаемости, в ходе которого инженеры, не знакомые со средой Google, обучаются через обзоры кода, в значительной степени направлен на освоение руководств по стилю (подробнее о контроле удобочитаемости в главе 3).

Некоторый уровень обучения всегда необходим — инженеры должны, в конце концов, выучить правила, чтобы писать адекватный код, но контроль за соблюдением правил мы предпочитаем автоматизировать с помощью инструментов.

Автоматизация контроля за соблюдением правил гарантирует, что правила не будут отброшены или забыты с течением времени или по мере расширения организации. Приходят новые люди, не знающие всех правил, а правила меняются со временем, и даже при налаженном общении люди не могут помнить текущее состояние всех дел. Пока наши инструменты синхронизируются с изменением наших правил, мы уверены, что правила применяются всеми инженерами во всех наших проектах.

Еще одним преимуществом автоматизации контроля за соблюдением правил является минимизация различий в интерпретации и применении правил. Когда мы пишем сценарий или используем инструмент для проверки, то проверяем все входные данные на соответствие правилу и не оставляем инженерам возможности интерпретировать правило по-своему. Инженеры, как и все люди, имеют свои точки зрения и предубеждения, которые могут менять взгляд людей на вещи. Передача инженерам права контроля за соблюдением правил может привести к непоследовательному толкованию и применению правил с потенциально ошибочными представлениями об ответственности. Чем больше ответственности мы возлагаем на инструменты проверки, тем меньше возможностей для разных толкований правила мы оставляем.

Автоматизация проверки также обеспечивает масштабируемость применения правила. По мере развития организации отдельная команда экспертов может создавать

инструменты для использования в остальной части компании. Даже если размер компании удвоится, это не потребует удвоить усилия по обеспечению соблюдения всех правил во всей организации — они останутся примерно на прежнем уровне. Даже с учетом преимуществ, которые мы получаем от внедрения инструментов, автоматизация применения всех правил может оказаться невозможной. Некоторые технические правила требуют человеческого суждения. Например, в руководстве по стилю для C++: «Избегайте сложного метапрограммирования шаблонов»; «Используйте `auto`, чтобы избежать употребления имен типов, которые загромождают код, очевидны или не важны, во всех случаях, когда тип не помогает читателю понять код», «Композиция часто более уместна, чем наследование». В руководстве по стилю для Java: «Нет единого правильного рецепта (упорядочения членов и инициализаторов класса); разные классы могут упорядочивать свое содержимое по-разному»; «Очень редко бывает правильным ничего не делать в ответ на перехваченное исключение»; «Крайне редко требуется переопределять `Object.finalize`». Для оценки соответствия всем этим правилам требуется человеческое суждение, а инструменты этого не могут (пока!).

Другие правила являются скорее социальными, чем техническими, и решать социальные проблемы с помощью технических средств часто неразумно. Для многих правил, попадающих в эту категорию, детали определены менее четко и инструменты, анализирующие их, получатся слишком сложными и дорогостоящими. Контроль за соблюдением этих правил часто лучше оставить людям. Например, когда речь заходит об объеме изменений в коде (то есть о количестве измененных файлов и строк), мы рекомендуем инженерам отдавать предпочтение небольшим изменениям. Небольшие изменения легче проверить, поэтому обзоры, как правило, выполняются экспертами быстрее и тщательнее. Также небольшие изменения менее чреваты ошибками, потому что позволяют легко рассуждать об их потенциальном влиянии и последствиях. Но что значит «небольшое изменение»? Изменения, которые выливаются в одинаковые обновления единственной строки в сотнях файлов, на самом деле могут легко поддаваться проверке. А изменение длиной в 20 строк может содержать сложную логику с побочными эффектами, которые трудно оценить. Мы понимаем, что размеры изменений можно оценивать по-разному и некоторые из этих оценок могут быть весьма субъективными, особенно если учитывать сложность изменения. Именно поэтому у нас нет никаких инструментов для автоматического отклонения изменений, размер которых превышает некоторый порог, выраженный количеством строк. Рецензенты могут (и часто делают это) отвергнуть изменение, если посчитают, что оно слишком велико. Для этого и подобных ему правил контроль за соблюдением целиком возложен на инженеров, которые пишут и проверяют код. Однако в отношении технических правил всегда, когда это возможно, мы выступаем за автоматизацию.

Проверка ошибок

Контроль соблюдения многих правил, регламентирующих использование языка, можно реализовать с помощью инструментов статического анализа. Неофициальное

исследование руководства по стилю для C++, проведенное некоторыми нашими разработчиками библиотек на C++ в середине 2018 года, показало, что вполне можно автоматизировать проверку примерно 90 % его правил. Инструменты проверки ошибок получают набор правил или паттернов и проверяют, насколько полный указанный образец им соответствует. Автоматическая проверка снимает с автора кода бремя запоминания всех обязательных правил — инженеру остается только поиск предупреждений о нарушениях (многие из которых сопровождаются предлагаемыми исправлениями), обнаруженных анализатором кода, тесно интегрированным в процесс разработки. Когда мы начали использовать инструменты выявления фактов использования устаревших функций, основанных на тегах в исходном коде, которые выводили предупреждения с предлагаемыми вариантами исправления, проблема использования устаревших API исчезла почти за одну ночь. Снижение усилий на соблюдение правил увеличивает вероятность, что инженеры с радостью доведут проверку до конца.

Для автоматизации процесса контроля за соблюдением правил (глава 20) мы используем такие инструменты, как `clang-tidy` (<https://oreil.ly/dDHtI>) для C++ и `Error Prone` (<https://oreil.ly/d7wkw>) для Java.

Наши инструменты были специально разработаны и адаптированы для поддержки определяемых нами правил. Правила должны соблюдать все без исключения, поэтому инструментами проверки пользуются все наши инженеры. В некоторых инструментах поддержки передовых практик, если соглашения допускают определенную гибкость, предусмотрены механизмы исключений, позволяющие проектам приспосабливать эти инструменты к своим потребностям.

КЕЙС: GOFMT

Самир Ажмани (Sameer Ajmani)

Компания Google выпустила язык программирования Go с открытым исходным кодом 10 ноября 2009 года. С тех пор Go стал языком для разработки служб, инструментов, облачной инфраструктуры и ПО с открытым исходным кодом¹.

С первого дня мы понимали, что нам нужен стандартный формат для кода на Go, причем после выпуска исходного кода будет почти невозможно этот формат изменить. Поэтому в первоначальную версию Go был включен стандартный инструмент форматирования `gofmt`.

Мотивация

Обзоры кода — одна из передовых практик в программной инженерии, и обсуждение вопросов, связанных с форматированием, требует слишком много времени. Конечно, стандартный формат понравится не всем, но он достаточно хорошо сэкономил время инженера².

¹ В декабре 2018 года Go стал языком № 4 в GitHub по количеству запросов на вытягивание (<https://oreil.ly/tAqDI>).

² Доклад Роберта Гриземера (Robert Griesemer) «The Cultural Evolution of `gofmt`» (<https://oreil.ly/GTJRd>), сделанный в 2015 году, содержит подробную информацию о мотивации, дизайне

Стандартизовав формат, мы заложили основу для инструментов, которые смогут автоматически обновлять код Go. Причем код, отредактированный инструментом, будет неотличим от кода, созданного человеком¹.

Например, несколько месяцев, предшествовавших выходу Go 1.0 в 2012 году, команда Go использовала инструмент `gofix` для автоматического обновления предварительной версии кода языка и библиотек до стабильной версии 1.0. Благодаря `gofmt` в производимые инструментом `gofix` изменения вошло только самое важное, касавшееся использования языка и API. Это облегчило программистам обзор изменений и дало возможность учиться на изменениях, сделанных инструментом.

Влияние

Программисты на Go ожидают, что *весь* код на Go будет отформатирован с помощью `gofmt`. У `gofmt` нет настроек, и его поведение редко меняется. Все основные редакторы и интегрированные среды разработки (IDE, integrated development environment) используют `gofmt` или имитируют его поведение, поэтому почти весь существующий код на Go отформатирован идентично. Сначала пользователи Go жаловались на обязательный стандарт; теперь они часто называют `gofmt` одной из причин, по которым им нравится Go. Этот формат позволяет быстро вникать в любой код на Go.

Тысячи проектов с открытым исходным кодом используют и пишут код на Go². Поскольку все редакторы и IDE обеспечивают единообразное форматирование, инструменты на Go легко переносятся между платформами и интегрируются в новые среды разработки и рабочие процессы с помощью командной строки.

Настройка

В 2012 году мы решили автоматически отформатировать все файлы `BUILD` в Google, используя новый стандартный инструмент форматирования `buildifier`. Файлы `BUILD` содержат правила сборки ПО в Blaze — системе сборки в Google. Стандартизация формата `BUILD` должна была позволить нам создавать инструменты, автоматически редактирующие файлы `BUILD`, не нарушая их формата, как это делают инструменты Go с файлами на Go.

Одному инженеру потребовалось шесть недель, чтобы переформатировать 200 000 файлов `BUILD`, принятых различными командами, в течение которых каждую неделю добавлялось больше тысячи новых файлов `BUILD`. Зарождающаяся инфраструктура Google для внесения масштабных изменений значительно ускорила этот процесс (глава 22).

Форматирование кода

Мы в Google широко используем автоматические средства проверки стиля и форматирования, чтобы обеспечить единообразие оформления нашего кода. Проблема

и влиянии `gofmt` на Go и другие языки.

¹ Расс Кокс (Russ Cox) объяснил в 2009 году (https://oreil.ly/IqX_J), что цель `gofmt` — автоматизация изменений: «Итак, у нас есть все сложные части инструмента манипулирования программами, которые просто ждут, чтобы их использовали. Согласие принять “стиль `gofmt`” запустит этот инструмент в ограниченном объеме кода».

² Пакеты `AST` (<https://godoc.org/go/ast>) и `format` (<https://godoc.org/go/format>) на Go импортируются тысячами проектов.

длины строк перестала существовать¹. Инженеры просто запускают программы проверки стиля и продолжают двигаться вперед. Единый способ форматирования устраняет циклы рецензирования, в ходе которых инженер тратит время на поиск, маркировку и исправление незначительных ошибок в оформлении.

Управляя самой большой в истории базой кода, мы получили возможность сравнить результаты форматирования, выполненного людьми и автоматизированными инструментами. В среднем роботы справляются с большими объемами кода лучше людей. Но в некоторых ситуациях знание предметной области имеет значение, например человек лучше справляется с форматированием матрицы, чем универсальный инструмент. В остальных случаях форматирование кода с помощью автоматических инструментов редко выполняется неправильно.

Также мы автоматизировали проверку форматирования: перед отправкой кода в репозиторий служба проверяет, изменился ли код после его обработки инструментом форматирования. Если изменился, то отправка отклоняется и сопровождается инструкциями, как запустить средство форматирования для исправления кода. Такой предварительной проверке подвергается большая часть кода в Google. Для проверки кода на C++ мы используем `clang-format` (<https://oreil.ly/AbP3F>); для кода на Python — внутреннюю обертку вокруг `yapf` (<https://github.com/google/yapf>); для кода на Go — `gofmt` (<https://golang.org/cmd/gofmt>); для кода на Dart — `dartfmt` и для файлов BUILD — `buildifier` (<https://oreil.ly/-wyGx>).

Заключение

В любой инженерной организации, особенно в такой большой, как Google, правила помогают управлять сложностью процессов и создавать устойчивую кодовую базу. Общий набор правил направляет инженерные процессы на расширение и рост и обеспечивает долгосрочную поддержку кодовой базы и самой организации.

Итоги

- Правила и руководства должны быть нацелены на поддержание устойчивости и масштабирование.
- Корректировки правил должны быть обоснованы данными.
- Правил не должно быть слишком много.
- Единообразие — ключ к успеху.
- Автоматизация контроля за соблюдением правил должна применяться везде, где возможно.

¹ Если учесть, что для обсуждения требуется не менее двух инженеров, умножьте это число на количество раз, когда такое обсуждение может случиться в команде, насчитывающем более 30 000 инженеров, и вы убедитесь, насколько дорогостоящим может стать решение проблемы «количества символов в строке».

Код-ревью

Авторы: Том Манирек и Кейтлин Садовски

Редактор: Лиза Кэри

Код-ревью, или обзор кода, — это процесс проверки внесенных в код изменений сотрудником, который не является их автором, часто до передачи кода в репозиторий. Определение выглядит простым, но реализации процесса обзора кода сильно отличаются в индустрии ПО. В одних организациях вносимые изменения просматривает отдельная группа «цензоров» всей кодовой базы. В других обзоры кода делегируются небольшим командам, что позволяет разным командам требовать разные уровни проверки кода. В Google практически каждое изменение проверяется перед его отправкой в репозиторий и каждый инженер отвечает за инициирование обзоров и проверку изменений.

Обзоры кода обычно представляют собой комбинацию процесса и инструмента, поддерживающего этот процесс. Мы в Google используем для проверки кода специальный инструмент Critique¹, обсуждение которого достойно отдельной главы в этой книге. А сейчас мы представим идеи, связанные с самим процессом обзора кода в Google, которые можно применить к любому специальному инструменту.



Дополнительная информация о Critique в главе 19.

Некоторые из преимуществ обзора кода, такие как выявление ошибок до того, как они окажутся в кодовой базе, хорошо известны² и даже очевидны (если не измерены точно). Но у него есть и скрытые преимущества: процесс обзора кода в Google настолько вездесущий и обширный, что мы смогли отследить другие его аспекты, в том числе психологические, играющие роль в экономии времени и масштабировании.

¹ Также для проверки кода в Git мы используем Gerrit (<https://www.gerritcodereview.com>), но в первую очередь для проектов с открытым исходным кодом. То есть Critique — основной инструмент типичного инженера-программиста в Google.

² Макконнелл С. Совершенный код. Мастер-класс. М.: Русская редакция и Microsoft Press, 2017. — Примеч. пер.

Поток обзора кода

Обзоры кода могут выполняться на разных этапах разработки ПО. В Google обзоры кода выполняются до того, как изменения попадут в кодовую базу, — этот этап мы называем *предварительной проверкой*. Основная цель обзоров кода — получить от инженера, который не является автором данного кода, отметку LGTM («looks good to me» — «хорошо для меня»). Мы используем эту отметку как необходимый «элемент» разрешения на внесение изменения (остальные элементы обсудим ниже).

Типичный обзор кода в Google включает следующие этапы:

1. Пользователь-*автор* вносит изменения в кодовую базу в своем рабочем пространстве. Затем он создает снимок изменений — файл исправлений (patch-файл) и их описание — и выгружает его в инструмент обзора кода. Для выявления изменений этот инструмент создает *diff-файл*, содержащий отличия нового кода от старой кодовой базы.
2. Автор может использовать diff-файл также для комментирования автоматического обзора. Убедившись, что diff-файл точно и полно отражает изменения, автор отправляет его одному или нескольким рецензентам по электронной почте. Рецензенты получают уведомление с просьбой просмотреть и прокомментировать снимок.
3. *Рецензенты* открывают diff-файл в инструменте обзора кода и добавляют к нему комментарии. Некоторые комментарии явно требуют исправить выявленные ошибки. Другие носят чисто информационный характер.
4. Автор вносит необходимые исправления, выгружает новый снимок, полученный на основе отзывов, а затем отправляет его рецензентам. Шаги 3 и 4 могут повторяться несколько раз.
5. Если рецензенты удовлетворены состоянием изменений, они дают им отметку LGTM. По умолчанию требуется только одна отметка LGTM, но соглашение может потребовать, чтобы с изменениями согласились все рецензенты.
6. Автор фиксирует изменения, отмеченные LGTM, в кодовой базе при условии, что он *учел все комментарии* и изменения *одобрены*. Процедура одобрения мы рассмотрим в следующем разделе.

Далее в этой главе мы обсудим этот процесс более подробно.

Как проводятся обзоры кода в Google

Мы уже рассказали, как примерно выглядит типичный процесс обзора кода, но, как всегда, дьявол кроется в деталях. В этом разделе мы подробно опишем, как методы обзора кода в Google позволяют масштабировать это процесс.

Процесс обзора кода в Google имеет три вида «одобрения» любого изменения.

- Проверка и оценка правильности измененного кода и его соответствия заявленным целям других инженеров — часто членам команды автора изменения. Выражается в отметках LGTM.

- Одобрение одного из владельцев кода, подтверждающее, что код подходит для конкретной части кодовой базы (и может быть сохранен в определенный каталог). Это одобрение может быть неявным, если владельцем кода является автор изменения. База кода в Google — это древовидная структура с иерархией владельцев каталогов (глава 16). Владельцы действуют как «цензоры» своих каталогов. Изменение предлагает любой инженер, отметку LGTM ставит любой другой инженер, а добавление изменения в часть кодовой базы *одобряет* владелец соответствующего каталога. Таким владельцем может быть технический руководитель или инженер-эксперт в конкретной области кодовой базы. Как правило, каждая команда сама закрепляет области кода за их владельцами.
- Одобрение от обладателя сертификата удобочитаемости для данного языка¹, подтверждающее, что код соответствует стилю языка и передовым практикам. Это одобрение тоже может быть неявным, если автор изменения обладает сертификатом удобочитаемости, или явным, если оно получено от другого инженера, уполномоченного проверять удобочитаемость на соответствующем языке программирования.

КОД — ЭТО ОТВЕТСТВЕННОСТЬ

Важно помнить (и понимать), что код сам по себе является ответственностью, поскольку в будущем он обслуживается не его автором. Код похож на топливо — груз, без которого самолет не полетит (<https://oreil.ly/TmoWX>).

Добавлять новые возможности стоит, только если они действительно нужны. Повторяющийся код — это не только пустая трата сил, но и лишние затраты. Изменить один паттерн в кодовой базе намного легче, чем множество повторяющихся фрагментов. К написанию совершенно нового кода у нас относятся настолько неодобрительно, что появилась поговорка: «Если ты пишешь код с нуля, значит, делаешь это неправильно!»

Это особенно верно для библиотечного или служебного кода. Если вы пишете утилиту, скорее всего, учитывая размер кодовой базы Google, кто-то еще уже написал нечто подобное. Поэтому инструменты, описанные в главе 17, имеют большое значение как для поиска повторяющегося кода, так и для предотвращения дублирования. Чтобы выявить дубликаты как можно раньше, нужно показать проект новой утилиты ответственным командам до того, как к ней будет написан код.

Конечно, появляются новые проекты, внедряются новые технологии, добавляются новые компоненты и т. д. Но обзор кода не является поводом для переосмысления или обсуждения предыдущих проектных решений. На выработку проектных решений часто требуется много времени для обмена предложениями, обсуждения дизайна (в обзорах API или на встречах) и, возможно, разработки прототипов. В той же мере, в какой обзор совершенно нового кода не должен происходить внезапно, сам процесс рецензирования не должен рассматриваться как возможность пересмотреть предыдущие решения.

¹ В Google под «удобочитаемостью» подразумевается не только простота понимания, но также использование набора стилей и передовых практик, помогающих другим инженерам сопровождать код. Подробнее о контроле удобочитаемости в главе 3.

Хотя такая форма контроля выглядит обременительной — и, по общему мнению, такой и является, — в большинстве случаев все три роли принимает на себя один человек, что значительно ускоряет процесс обзора кода. Важно отметить, что две последние роли может взять на себя сам автор кода, и ему останется только получить LGTM от другого инженера.

Эти требования делают процесс обзора кода достаточно гибким. Технический руководитель, который является владельцем проекта и обладает сертификатом удобочитаемости для этого кода, может отправить изменение в кодовую базу на основании LGTM от другого инженера. Инженер без таких полномочий может отправить такое же изменение в ту же кодовую базу при условии, что он получит одобрение от владельца с сертификатом удобочитаемости для данного языка. Три вышеупомянутых одобрения могут объединяться в любой комбинации. Автор может даже запросить более одного LGTM, явно пометив изменение как получившее LGTM от всех рецензентов.

Чаще всего обзоры выполняются в два этапа: получение LGTM от инженера-коллеги, а затем получение одобрения у соответствующего владельца(-ев) кода и/или рецензента(-ов) удобочитаемости. Это позволяет одобряющим инженерам сосредоточиться на разных аспектах и сэкономить время, необходимое для обзора. Первый рецензент может сосредоточиться на правильности кода, а владелец кодовой базы — на необходимости изменения, без учета деталей в каждой строке. Другими словами, владелец часто ищет нечто иное, чем рецензент. Наконец, тех, кто хочет принять код в свой проект/каталог, волнуют вопросы: «Насколько сложно поддерживать этот код?», «Увеличит ли он мой технический долг?», «Достаточно ли опыта в нашей команде для его поддержки?»

Если все обзоры всех трех типов могут выполняться одним человеком, почему бы просто не поручить таким сотрудникам проводить все обзоры кода? Ответ прост: такой подход не масштабируется. Разделение ролей добавляет гибкости процессу обзора кода. Если вы работаете с коллегой над новой функцией в библиотеке, то можете попросить кого-то из вашей команды проверить правильность и удобочитаемость кода. После нескольких раундов (возможно, в течение нескольких дней) ваш код удовлетворит рецензента и вы получите LGTM. После этого вам останется только получить одобрение владельца библиотеки (часто имеющего сертификат удобочитаемости), чтобы утвердить изменение.

Преимущества обзоров кода

Сами по себе обзоры кода не вызывают споров. Многие (возможно, даже большинство) компании и проекты с открытым исходным кодом имеют некоторую форму обзоров кода и считают этот процесс не менее важным, чем проверка работоспособности при добавлении нового кода в репозиторий. Инженеры-программисты понимают некоторые из наиболее очевидных преимуществ обзоров кода, даже если считают, что эта практика применима не во всех случаях. Но в Google этот процесс, как правило, более тщательный и распространенный, чем в большинстве других компаний.

ВЛАДЕНИЕ

Хайрам Райт

В небольшой команде выделенный доступ к коду в отдельном репозитории обычно распространяется на всех членов команды. В конце концов, они друг друга хорошо знают и предметная область достаточно узкая, чтобы каждый мог быть экспертом, а небольшой объем кода ограничивает влияние потенциальных ошибок.

По мере увеличения численности команды такой подход перестает масштабироваться. В результате происходит беспорядочное разделение репозитория: неясно, кто и какими знаниями обладает и какие обязанности выполняет в разных частях репозитория. Мы в Google называем эту совокупность знаний и обязанностей *владением*, а ее носителей — *владельцами*. Эта идея отличается от владения коллекцией исходного кода и скорее подразумевает разделение базы кода в интересах компании. (Если бы нам снова пришлось вводить культуру владения кодом, мы наверняка выбрали бы термин «распоряжение».)

Специальные файлы с названием OWNERS содержат списки пользователей, которые несут ответственность за каталог и его элементы. Эти файлы могут также содержать ссылки на другие файлы OWNERS или внешние списки управления доступом, но в конечном итоге образуют список лиц. Каждый подкаталог может содержать свой файл OWNERS, при этом файлы в дереве каталогов связаны аддитивным отношением: файл обычно включает всех пользователей, перечисленных в файлах OWNERS, расположенных выше в дереве каталогов. В файлах OWNERS может быть столько записей, сколько пожелает команда, но мы рекомендуем сохранять списки относительно небольшими и точными, чтобы гарантировать ясность разделения зон ответственности.

Владение кодом в Google дает сотруднику право на одобрение изменений в этом коде. Владелец должен сам понимать код или знать, кому задавать вопросы по коду. В разных командах используются разные способы передачи права владения новым членам команды, но мы рекомендуем не использовать это право в качестве обряда посвящения и призываем уходящих членов команд незамедлительно уступать права владения.

Распределенная организация владения позволяет использовать методы, представленные в этой книге. Например, инженеры, перечисленные в корневом файле OWNERS, могут выступать в роли лиц, утверждающих крупномасштабные изменения (глава 22) без привлечения членов локальных команд. Точно так же файлы OWNERS действуют подобно документации, позволяя людям и инструментам легко находить ответственных за данный фрагмент кода: достаточно просто пройти вверх по дереву каталогов. У нас нет центрального органа, который регистрирует новые права владения в новых проектах: для присвоения прав владения достаточно создать новый файл OWNERS.

Это простой, но весьма действенный механизм владения, и за последние два десятилетия он значительно расширился. С его помощью Google гарантирует эффективную работу десятков тысяч инженеров с миллиардами строк кода в одном репозитории.

Культура Google, как и многих софтверных компаний, основана на предоставлении инженерам широких возможностей. Многие считают, что строгие процессы плохо работают в динамично развивающихся компаниях, которым приходится быстро реагировать на появление новых технологий, и бюрократические правила отрицательно влияют на творческую работу. Однако в Google обзоры кода — это обязательное мероприятие, в котором должны участвовать все инженеры-програм-

мисты и практически¹ все изменения в кодовой базе. Это требование влечет затраты и влияет на скорость разработки, замедляя внедрение нового кода и его передачу в производство. (Инженеры-программисты часто жалуются на строгость обзоров кода.) Тогда зачем нам этот процесс? Почему мы считаем, что он приносит выгоду в долгосрочной перспективе?

Хорошо продуманный процесс обзора кода обеспечивает следующие преимущества:

- проверку правильности кода;
- гарантию понятности изменения для других инженеров;
- обеспечение согласованности всей кодовой базы;
- воспитание командной ответственности;
- возможность обмена знаниями;
- хронологическую запись принятия решений об изменениях.

С течением времени многие из этих преимуществ приобретают особую важность для авторов изменений, рецензентов и организации в целом. Рассмотрим указанные преимущества подробнее.

Правильность кода

Очевидным преимуществом процесса обзора кода является возможность проверить «правильность» изменения. Рецензент, который не является автором изменения, проверяет, было ли изменение должным образом протестировано, правильно ли оно спроектировано и насколько эффективно оно функционирует. В основном проверка правильности кода ориентирована на выявление потенциальных ошибок в базе кода, которые может вызвать конкретное изменение.

Многие отмечают, что основной эффект от обзоров кода заключается в предотвращении появления ошибок в ПО. Исследование, проведенное в IBM, показало, что ошибки, обнаруженные на более ранних этапах разработки, можно исправить быстрее, чем ошибки, обнаруженные на более поздних этапах². Время, потраченное на обзор кода, экономит время на тестирование, отладку и устранение регрессий при условии, что сам процесс обзора кода необременителен. Трудоемкие или неправильно масштабированные процессы обзора кода менее надежны³. Мы дадим некоторые рекомендации по упрощению обзора кода в этой главе.

¹ Некоторые изменения в документации и конфигурациях могут добавляться без обзора кода, но обычно мы настаиваем на прохождении этой процедуры.

² «Advances in Software Inspection», *IEEE Transactions on Software Engineering*, SE-12(7): 744–751, July 1986. Конечно, это исследование проводилось до появления надежных инструментов и автоматизированных процессов тестирования в сфере разработки ПО, но его результаты остаются актуальными и в наше время.

³ *Rigby P. C., Bird C.* 2013. Convergent software peer review practices. ESEC/FSE 2013: *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, August 2013: 202–212. <https://dl.acm.org/doi/10.1145/2491411.2491444>.

Чтобы оценка правильности была более объективной, предпочтение обычно отдается авторскому подходу к дизайну или реализации предложенного изменения. Рецензент должен предлагать альтернативы изменения, опираясь не на личное мнение, а на упрощение понимания кода (предлагать менее сложные варианты) или работы с ним (предлагать более эффективные варианты). В целом же инженерам рекомендуется одобрять изменения, улучшающие кодовую базу, даже если они не «идеальны». Это существенно ускоряет обзор кода.

По мере развития инструментов все больше проверок правильности будет выполняться автоматически, с применением таких методов, как статический анализ и автоматическое тестирование (подробнее о задачах, которые не могут решить инструменты, в главе 20). Несмотря на свои ограничения, инструменты ослабили зависимость проверки правильности кода от человека.

Проверка на наличие дефектов на начальном этапе процесса обзора кода является неотъемлемой частью общей политики «сдвига влево», согласно которой мы стремимся выявлять ошибки как можно раньше, чтобы на их устранение ушло меньше усилий и ресурсов. Обзор кода не является ни панацеей, ни единственной проверкой правильности — это элемент эшелонированной обороны ПО от проблем в коде. Поэтому обзор кода необязательно должен быть «идеальным» по результативности.

Как ни удивительно, но проверка правильности — не основное преимущество, которое Google получает от процесса обзора кода. Проверка правильности гарантирует работоспособность изменения, но гораздо важнее, чтобы изменение оставалось понятным и осмысленным с течением времени и по мере увеличения кодовой базы. Рассмотрим проверку понятности изменения.

Понятность кода

Обзор кода обычно является первым взглядом на изменение со стороны. Рецензент делает то, что не может сделать даже самый лучший инженер, — дает непредвзятый отзыв. *Обзор кода часто является первой проверкой понятности изменения для более широкой аудитории.* Код читается гораздо чаще, чем пишется, и его восприятие и понимание чрезвычайно важны.

Часто полезно найти рецензента, который имеет свой взгляд на изменение, отличный от мнения автора, и, возможно, будет в своей работе поддерживать или использовать предлагаемые изменения. Проверка кода основана как на уважении к мнению его автора, так и на принципе «клиент всегда прав». Какое бы число вопросов автор ни получил сейчас, оно многократно умножится с течением времени. Это не означает, что автор должен изменить свой подход или логику в ответ на критику, но означает, что, возможно, ему нужно объяснить свою позицию более четко.

Проверки правильности и понятности кода являются основными критериями для получения LGTM и последующего одобрения. Когда инженер ставит отметку LGTM, он утверждает, что код действует как требуется и он понятен. В Google изменения в коде, который постоянно поддерживается, требуют дополнительного одобрения.

Согласованность всей базы кода

В конечном счете код, который вы пишете, будет поддерживаться кем-то еще. Многим из тех, кто будет сопровождать ваш код, придется прочитать его и понять, что вы хотели выразить. Другим (в том числе автоматизированным инструментам) может понадобиться выполнить рефакторинг вашего кода, когда вы уже перейдете в другой проект. Поэтому код должен соответствовать стандартам согласованности и не быть чрезмерно сложным. В процессе обзора рецензенты могут оценить, насколько код соответствует стандартам кодовой базы. Таким образом, обзор способствует поддержке *работоспособности кода*.

Именно для простоты сопровождения оценка LGTM (указывающая на правильность и простоту кода для понимания) отделена от оценки удобочитаемости. Одобрения, касающиеся удобочитаемости, могут давать только лица, успешно прошедшие процесс обучения удобочитаемости кода на определенном языке программирования. Например, одобрить код на Java может только инженер, получивший сертификат удобочитаемости на Java.

Проверяющему удобочитаемость поручается исследовать код и убедиться, что он следует утвержденным передовым практикам для этого конкретного языка программирования, согласуется с базой кода на этом языке в репозитории Google и не слишком сложен. Согласованный и простой код легче понять и проанализировать с применением инструментов, когда придет время для рефакторинга, что делает его более устойчивым. Если определенный паттерн в кодовой базе всегда реализуется каким-то одним способом, это упростит создание инструмента для его рефакторинга.

Кроме того, код может быть написан одним человеком, но будет прочитан десятками, сотнями или даже тысячами других людей. Согласованность всей базы кода упрощает его понимание всеми инженерами и даже влияет на процесс обзора кода. Иногда она конфликтует с функциональностью: рецензент, оценивающий удобочитаемость, может выступить за то, чтобы изменение стало менее функциональным, но более простым для понимания.

Благодаря согласованности кодовой базы инженерам проще вникать в код других проектов. Для обзора кода инженерам может потребоваться помощь другой команды. Возможность обратиться к экспертам для проверки согласованности кода позволяет инженерам сосредоточиться на правильности и простоте понимания кода.

Командная ответственность

Обзор кода также дает важные культурные преимущества: он укрепляет инженеров-программистов в мнении, что фактически код является не «их» собственностью, а частью коллективного труда. Такие психологические преимущества могут быть едва заметными, но от этого не менее важными. Если убрать процесс обзора кода, большинство инженеров неизбежно начнут склоняться к индивидуальному стилю

и подходу в разработке ПО. Процесс обзора кода заставляет автора не только позволить другим вносить свой вклад в его творение, но и идти на компромиссы ради общего блага.

Человеку свойственно гордиться своим ремеслом и не испытывать особого желания открывать свой код для критики, а также настороженно относиться к критическим отзывам о готовом коде. Процесс обзора кода предусматривает механизм смягчения эмоционального напряжения его автора. Обзор кода не только ставит под сомнение предположения автора, но и делает это предписанным нейтральным образом, чтобы исключить «переход на личности». В конце концов, процесс *требуется* критического обзора (не зря мы назвали наш инструмент для обзора кода «Critique»), поэтому никто не можете обвинить рецензента в том, что он выполняет свою работу и критикует. Таким образом, процесс обзора кода выступает в роли «злого полицейского», тогда как рецензент воспринимается как «добрый полицейский».

Конечно, не всем и даже не большинству инженеров нужны такие механизмы смягчения. Но создание защиты от такой критики в процессе обзора кода часто дает большинству инженеров гораздо более мягкое представление об ожиданиях команды. Перспектива обзора кода пугает многих инженеров, присоединившихся к Google или перешедших в новую команду. Нетрудно понять, что любая форма критики негативно отражается на эффективности работы человека. Но со временем почти все инженеры привыкают, что в процессе обзора их кода возникнут вопросы, и начинают понимать ценность советов и вопросов, предложенных в ходе этого процесса (хотя, по общему признанию, для привыкания требуется время).

Еще одним психологическим преимуществом обзора кода является признание его пригодности. Даже самые способные инженеры могут страдать от синдрома самозванца и быть слишком самокритичными. Процесс обзора кода действует как подтверждение и признание их способностей. Часто этот процесс включает обмен идеями и знаниями (подробнее об этом в следующем разделе), что приносит пользу как рецензенту, так и рецензируемому. По мере расширения своих знаний инженерам иногда бывает трудно получить подтверждение своего совершенствования. Процесс обзора кода дает им такую возможность.

Процедура инициирования обзора кода также заставляет авторов уделять своим изменениям больше внимания. Многие инженеры-программисты не являются перфекционистами — большинство из них признают, что код, «выполняющий свою работу», лучше, чем совершенный код, разработка которого занимает слишком много времени. В отсутствие процесса обзора кода многие авторы кода начнут срезать углы, чтобы исправить дефекты позже. «Да, у меня нет некоторых юнит-тестов, но я потом их добавлю». Обзор кода заставляет инженера решать большинство проблем до отправки изменений. Необходимость собрать все компоненты, составляющие изменение, для обзора кода заставляет инженера убедиться, что код в полном порядке. Короткий период для размышлений перед отправкой изменений — это идеальное время, чтобы еще раз прочитать изменения и убедиться, что ничего не упущено.

Обмен знаниями

Одним из наиболее важных, но недооцененных преимуществ обзора кода является обмен знаниями. Большинство авторов выбирают рецензентов, которые являются экспертами или, по крайней мере, опытными специалистами в предметной области. Процесс обзора позволяет рецензентам передавать авторам знания, давать советы, рассказывать про новые методы или делиться рекомендациями. (Рецензенты могут помечать некоторые свои комментарии как «FYI» — «на заметку», не требуя каких-либо действий, а просто сообщая автору некоторые сведения.) По мере накопления опыта авторы часто становятся владельцами и затем получают возможность выступать в роли рецензентов кода других инженеров.

Кроме обратной связи и подтверждения работоспособности кода процесс обзора кода также включает обсуждение реализации изменения, выбранной автором. Не только авторы, но и рецензенты могут узнавать о новых методах и паттернах в процессе обзора. В Google рецензенты могут напрямую делиться возможными вариантами изменений с автором в самом инструменте обзора кода.

Инженер может читать не все электронные письма, адресованные ему, но, как правило, он отвечает на каждый комментарий, полученный в процессе обзора кода. Этот обмен знаниями может выходить за рамки часовых поясов и проектов благодаря средствам быстрого распространения информации среди инженеров Google по всей базе кода. Обзор кода обеспечивает своевременную и эффективную передачу знаний. (Многие инженеры в Google «знакомятся» с другими инженерами при прохождении обзоров кода!)

Время, затрачиваемое на обзоры кода, позволяет инженерам накопить довольно значительные знания. Конечно, основная задача инженеров в Google по-прежнему заключается в программировании, но большую часть своего времени они все еще тратят на обзоры кода. Процесс обзора кода является одним из основных способов взаимодействия инженеров-программистов. Часто в ходе обзора кода демонстрируются новые паттерны, например с помощью рефакторинга в крупномасштабных изменениях.

Более того, поскольку каждое изменение становится частью кодовой базы, обзоры кода действуют как хронологические записи. Любой инженер может заглянуть в кодовую базу Google и определить, когда был принят какой-то паттерн, а также извлечь искомым обзор кода. Часто такие «археологические изыскания» помогают вникнуть в код не только автору и рецензентам, но и гораздо более широкому кругу инженеров.

Передовые практики обзора кода

Признаем, что обзоры кода могут привести к возникновению трений и задержек. Обычно эти проблемы связаны не с самим обзором, а с выбранной реализацией изменений. Обеспечение бесперебойности и масштабирования процесса обзора кода в Google требует применения передовых практик. Большинство из этих практик указывают, что обзоры кода должны протекать быстро и гладко.

Будьте вежливы и профессиональны

Как отмечалось в главе 2 «Культура» этой книги, Google активно развивает культуру доверия и уважения. Эта культура избавляет нас от предвзятого взгляда на обзор кода. Например, инженеру-программисту достаточно получить одобрение LGTM только от одного другого инженера, чтобы удовлетворить требование к простоте понимания кода. Многие инженеры комментируют и оценивают изменения, понимая, что эти изменения можно внести в кодовую базу без дополнительных проверок. Тем не менее обзоры кода могут вызвать беспокойство и стресс даже у самых способных инженеров. Поэтому крайне важно, чтобы все отзывы и критические замечания не выходили за рамки профессиональной сферы.

В общем случае рецензенты уважают выбранные авторами подходы и предлагают альтернативы, только если авторский подход несовершенен. Если автор сумеет продемонстрировать, что несколько подходов одинаково верны, рецензент должен согласиться с предпочтением автора. Даже если в выбранном подходе обнаружатся дефекты, обзор станет платформой для обучения (для обеих сторон!). Все комментарии должны оставаться в строго профессиональных рамках. Рецензенты не должны спешить с выводами относительно особого подхода автора: лучше сначала спросить, почему автор сделал именно такой выбор, прежде чем предполагать, что его подход неверен.

Рецензенты должны предоставлять отзывы максимально оперативно. Мы в Google ожидаем, что отзывы поступят в течение 24 (рабочих) часов. Если рецензент не укладывается в это время, ему рекомендуется ответить, что он, по крайней мере, увидел изменение и постарается подготовить рецензию как можно скорее. Рецензенты должны избегать рецензирования кода по частям. Ничто так не раздражает автора, как получение отзыва от рецензента, его изучение, а затем получение других отзывов.

Мы ожидаем профессионализма не только со стороны рецензента, но и со стороны автора. Помните, что вы — это не ваш код и предложенное вами изменение — не «ваше», а коллективное. В любом случае, как только вы отправите фрагмент кода в репозиторий, он перестанет быть вашим. Будьте готовы объяснить, почему поступили именно так, а не иначе. Помните, что понятность кода и простота его поддержки в будущем являются одной из зон ответственности автора.

Важно рассматривать каждый комментарий рецензента как элемент TODO («что сделать»). Если вы не согласны с комментарием рецензента, дайте ему знать и опишите причину, но не отмечайте комментарий как решенный, пока каждая сторона не получит возможность предложить альтернативу. Один из распространенных способов сохранить корректность дебатов, когда автор не согласен с рецензентом, — предложить альтернативу и попросить другую сторону рассмотреть ее. Помните, что обзор кода — это возможность обучения не только для автора, но и для рецензента. Такой подход часто помогает снизить вероятность разногласий.

Точно так же, если вы являетесь владельцем кода и отвечаете за обзор кода в вашей кодовой базе, будьте готовы оценить изменения, предложенные внешним автором.

Если изменение направлено на улучшение кодовой базы, вы должны уважительно отнестись к автору, потому что любое такое изменение указывает на то, что в коде еще есть что-то, что можно и нужно улучшить.

Пишите небольшие изменения

Пожалуй, наиболее важной практикой, обеспечивающей гибкость процесса обзора кода, является внесение в код изменений небольшого объема. В идеале обзор кода должен фокусироваться на одной небольшой проблеме. Процесс обзора кода в Google не предназначен для изменений, оформленных в виде полностью сформированных проектов, и рецензенты вправе отклонить такие изменения как слишком большие для одного обзора. Кроме того, небольшие изменения позволяют инженерам не тратить время на ожидание результатов обзора. Небольшие изменения также имеют свои преимущества в процессе разработки ПО. Определить источник ошибки внутри изменения гораздо проще, если оно достаточно мало.

Однако важно понимать, что иногда трудно согласовать процесс обзора кода, опирающийся на небольшие изменения, с внедрением новых масштабных особенностей. Набор небольших последовательных изменений проще проверить по отдельности, но труднее оценить в рамках более крупной схемы. Некоторые инженеры в Google, по их признанию, не являются поклонниками практики внесения небольших изменений. Конечно, существуют методы для управления такими изменениями (разработка в интеграционных ветвях, управление изменениями с использованием базы различий, отличной от HEAD), но эти методы влекут существенные накладные расходы. Рассматривайте предложение вносить изменения небольшими порциями только как оптимизацию и допустите в своем коде возможность редкого появления больших изменений.

«Небольшие» изменения обычно ограничены примерно 200 строками кода. Небольшое изменение должно быть простым для рецензента, чтобы автору не пришлось откладывать другие изменения в ожидании результатов обзора. Большинство изменений в Google рассматриваются в течение дня¹. (По крайней мере, первичный отзыв на изменение дается в течение дня.) Около 35 % изменений в Google затрагивают единственный файл². Чем проще изменение для рецензента, тем быстрее оно попадет в кодовую базу и начнет приносить пользу автору. Ожидание подробного обзора в течение недели или около того может повлиять на последующие изменения. Небольшой первичный обзор предотвращает затраты, которые мог повлечь выбор неправильного подхода.

Поскольку изменения передаются обычно небольшими порциями, почти все обзоры кода в Google выполняются одним человеком. В противном случае, если бы приходилось привлекать целую команду, чтобы оценить все изменения в общей кодовой

¹ *Sadowski C., Söderberg E., Church L., Sipko M., Bacchelli A.* Modern code review: a case study at Google (<https://oreil.ly/m7FnJ>).

² Там же.

базе, сам процесс оказался бы не масштабируемым. Ограничив размер изменений для обзора, мы оптимизировали этот процесс. Нередки ситуации, когда изменение комментируется несколькими людьми. Часто запрос на обзор направляется одному из членов команды, а его копии — соответствующим командам, но основным рецензентом по-прежнему является тот, чья отметка LGTM ожидается, и только одна отметка LGTM необходима для изменения. Любые другие комментарии хотя и важны, но необязательны.

Ограничение размера изменений позволяет рецензентам быстро посмотреть, выполнил ли основной рецензент свою часть обзора, и сосредоточиться исключительно на особенностях, которые привносит изменение в кодовую базу, а также на сохранности работоспособности кода с течением времени.

Пишите хорошие описания к изменениям

Описание должно сообщать тип изменения с краткой сводкой в первой же строке, которая действует подобно строке с темой в электронных письмах — ее видят инженеры Google в сводке истории в инструменте Code Search (глава 17).

В описании следует подробно рассказать, что *и почему* изменяется. Описание «Исправление ошибок» не поможет ни рецензенту, ни будущему «археологу». Если изменение содержит несколько более мелких связанных изменений, перечислите их в виде списка (желательно короткого). Описание является хронологической записью для изменения, и такие инструменты, как Code Search, позволят вам найти, кто и какую строку написал, в любом конкретном изменении в кодовой базе. Поиск и исследование исходного изменения часто помогают при исправлении ошибок.

Описания — не единственное место, куда можно добавить документацию к изменению. При разработке общедоступного API вы едва ли захотите включить в общее описание детали реализации, но непременно сделайте это в фактической реализации, добавив соответствующие комментарии. Если рецензент не поймет вашу задумку, даже правильную, это будет означать, что вашему коду недостает структуры и/или более ясных комментариев. Если в процессе обзора кода будет принято новое решение, обновите описание изменения или добавьте соответствующие комментарии в реализацию. Обзор кода — это не только набор действий, выполняемых сейчас, но и описание ваших действий для будущих читателей.

Привлекайте к обзору минимальное число рецензентов

Большинство обзоров кода в Google выполняются всего одним рецензентом¹. Поскольку проверка корректности, пригодности для владельцев и удобочитаемости кода может производиться одним человеком, процесс обзора кода хорошо масштабируется в организациях, сопоставимых по размерам с Google.

¹ Там же.

Многие инженеры в отрасли хотят получать дополнительные комментарии (и единодушное одобрение) от группы инженеров. В конце концов, каждый дополнительный рецензент может привнести что-то свое в обзор кода. Но мы выяснили, что увеличение числа рецензентов приводит к уменьшению отдачи: самое важное одобрение — LGTM, и аналогичные одобрения добавляют к этой отметке не так уж много информации. Затраты на привлечение дополнительных рецензентов быстро перевешивают ценность их комментариев.

Процесс обзора кода протекает оптимально, потому что мы доверяем нашим инженерам. Иногда бывает полезно привлечь к обзору конкретного изменения несколько человек, но даже тогда рецензенты должны сосредоточиться на разных аспектах одного и того же изменения.

Автоматизируйте все, что только можно

Обзор кода — это процесс с участием человека, но если в процессе есть этапы, которые можно автоматизировать, попробуйте это сделать. Изучите возможность автоматизации механических задач, и инвестиции в эту автоматизацию окупятся сторицей. Инструменты, используемые в Google в процессе обзора кода, позволяют авторам автоматически отправлять изменения и синхронизировать их в VCS после утверждения (обычно они применяются для наиболее простых изменений).

Одним из самых важных технологических усовершенствований в области автоматизации, сделанных за последние несколько лет, является автоматический статический анализ изменений (глава 20). Инструмент автоматизации обзора кода в Google избавляет авторов от необходимости вручную запускать тесты и применять инструменты форматирования и статического анализа — он автоматически выполняет большую часть механической работы в ходе так называемой *предварительной проверки перед отправкой*. Процедура предварительной проверки запускается, когда изменение впервые посылается рецензенту. Она может обнаружить ошибки и отклонить изменения (и предотвратить отправку электронного письма рецензенту), а также предложить автору исправить найденные недочеты. Такая автоматизация не только ускоряет процесс обзора кода, но также позволяет рецензентам сосредоточиться на более важных задачах, чем форматирование.

Виды обзоров кода

Обзоры кода не похожи друг на друга! Разные виды обзоров кода требуют разного уровня внимания к различным аспектам. В Google изменения обычно попадают в одну (или несколько) из следующих групп:

- разработка новой возможности;
- изменение в поведении, улучшение и оптимизация;
- исправление ошибок и откат к прежним версиям;
- рефакторинг и крупномасштабные изменения.

Обзоры кода новых возможностей

Реже всего проводятся обзоры совершенно нового кода — так называемые *обзоры с нуля*. Самое важное для обзора с нуля — оценить, выдержит ли код проверку временем и насколько легко его будет поддерживать, потому что с течением времени и при росте организации могут измениться предположения, на которых этот код основывается. Конечно, появление совершенно нового кода не должно вызывать удивления. Как упоминалось выше в этой главе, код — это ответственность, поэтому совершенно новый код должен, как правило, решать реальную проблему, а не просто представлять еще одну альтернативу. Мы в Google обычно требуем, чтобы новый код и/или проект подвергались обширной проверке контекста. Обзор кода — не лучшее пространство для обсуждения проектных решений, принятых в прошлом (и к тому же обзор кода совершенно не подходит для представления предлагаемого дизайна API).

Чтобы гарантировать устойчивость кода, в ходе обзора с нуля необходимо убедиться, что API соответствует согласованному дизайну (а для этого может потребоваться пересмотреть проектную документацию), и *полностью* его протестировать, причем все конечные точки API должны быть охвачены некоторой формой юнит-тестирования и все тесты должны завершаться неудачей при изменении предположений, сделанных в коде (глава 11). Код также должен иметь соответствующих владельцев (в одном из самых первых обзоров нового проекта часто исследуется единственный файл OWNERS для нового каталога), содержать достаточно подробные комментарии и при необходимости включать дополнительную документацию. Обзор с нуля может также потребовать включения проекта в систему непрерывной интеграции (глава 23).

Обзоры изменений в поведении, улучшений и оптимизаций

Большинство изменений в Google связаны с модификацией существующего кода. К ним относятся изменения конечных точек API, улучшения существующих реализаций или оптимизация других факторов, таких как производительность. Подобные изменения являются основной работой большинства инженеров-программистов.

По принципу обзора с нуля, сначала нужно узнать, необходимо ли изменение и улучшает ли оно существующий код. Один из лучших способов улучшить общее состояние кодовой базы — удаление мертвого или устаревшего кода.

Любые изменения поведения должны обязательно включать исправления в соответствующие тесты, проверяющие любое новое поведение API. Дополнения к существующей реализации мы тестируем в системе непрерывной интеграции, чтобы убедиться, что изменения не нарушают базовые предположения существующих тестов. Оптимизации должны гарантировать, что они не влияют на тесты, и, возможно, включать критерии производительности, чтобы рецензенты могли проверить их. Некоторые оптимизации могут также требовать наличия тестов производительности.

Обзоры исправлений ошибок и откатов к прежним версиям

Рано или поздно вам придется исправлять ошибки. *При этом вы должны не поддаваться соблазну заняться другими задачами*, чтобы не усложнить обзор кода и не затруднить регрессионное тестирование или откат ваших изменений другими пользователями. Исправление ошибки должно быть сосредоточено исключительно на исправлении конкретной ошибки и (обычно) обновлении соответствующих тестов, чтобы с их помощью можно было заметить ошибку.

Устранение ошибки с помощью исправленного теста часто является срочной необходимостью. Ошибка появилась, потому что существующие тесты были недостаточно точны или код основывался на определенных предположениях, которые не были выполнены. Рецензент, исследующий предложенное исправление ошибки, должен запросить обновленные юнит-тесты при необходимости.

Иногда изменение в кодовой базе с размером, как в Google, приводит к нарушению зависимости, которая либо не обнаруживается тестами, либо обнаруживается в непроверенной части кодовой базы. В Google разрешается «откатывать» такие изменения, как правило, пострадавшим нижестоящим клиентам. Откат — это по существу изменение, отменяющее предыдущее изменение. Такие откаты могут создаваться за считанные секунды, потому что они возвращают код в известное состояние, но они тоже требуют обзора кода.

Также очень важно, чтобы любое изменение, способное спровоцировать потенциальный откат (включающий все изменения!), имело минимально возможный размер и было атомарным, чтобы откат не вызвал нарушений в других зависимостях. Мы в Google видели, насколько быстро разработчики начинают зависеть от нового кода после его принятия, в результате чего откат изменений иногда мешает работе инженера. Чем меньше такие изменения, тем быстрее решаются связанные с ними проблемы благодаря их атомарности.

Обзоры результатов рефакторинга и крупномасштабных изменений

В Google многие изменения генерируются автоматически: автором таких изменений является не человек, а машина. (Подробнее о процессе крупномасштабных изменений (LSC, large-scale change) в главе 22.) Даже сгенерированные машиной изменения требуют обзора. Когда изменение сопряжено с небольшими последствиями, оно проверяется назначенными рецензентами, имеющими право давать одобрение для всей кодовой базы. Но когда изменение считается рискованным или требует особых знаний в предметной области, отдельным инженерам может быть предложено просмотреть автоматически сгенерированные изменения в ходе их обычного рабочего процесса.

На первый взгляд обзор автоматически сгенерированного изменения должен проводиться так же, как любого другого, — рецензент должен проверить правильность и уместность изменения. Однако мы рекомендуем рецензентам ограничить добавление комментариев и отмечать только те проблемы, которые относятся к их коду,

а не к базовому инструменту или процессу, сгенерировавшему изменения. Несмотря на то что конкретное изменение может быть сгенерировано машиной, сам процесс, сгенерировавший его, уже был рассмотрен, и отдельные группы не могут наложить вето на этот процесс, иначе будет невозможно масштабировать такие изменения по всей организации. Если существующее применение процесса или инструмента вызывает беспокойство, рецензенты могут обратиться за дополнительной информацией к группе надзора за крупномасштабными изменениями.

Мы также предлагаем рецензентам автоматических изменений не выходить за рамки своей компетенции. При обзоре новой функции (или изменения), написанной товарищем по команде, часто разумно попросить автора рассмотреть связанные с изменением проблемы и учесть советы, чтобы сохранить изменение небольшим. Но рецензент, запускающий инструмент, может получить сотни изменений, и комментирование даже небольшой их доли только ограничит область применения инструмента.

Заключение

Обзор кода — один из самых важных процессов в Google. Он действует как платформа для взаимодействия инженеров и является основным рабочим процессом разработчика. От обзора кода зависят почти все другие процессы, от тестирования до статического анализа и непрерывной интеграции. Процесс обзора кода должен масштабироваться и соответствовать передовым практикам (ограничению размеров изменений и ускорению обратной связи), которые помогают поддерживать удовлетворенность разработчика и высокую скорость разработки.

Итоги

- Обзор кода имеет множество преимуществ, включая обеспечение правильности, понятности и согласованности изменения с базой кода.
- Всегда сверяйте свои предположения с мнением коллег, чтобы оптимизировать код для читателей.
- Реагируйте на критические отзывы как профессионал.
- Обзоры кода способствуют обмену знаниями внутри организации.
- Автоматизация — важное условие для масштабирования процесса обзора кода.
- Обзор кода является хронологической записью.

Документация

Автор: Том Маншрек

Редактор: Риона Макнамара

Особенно сильным разочарованием большинства инженеров, пишущих, использующих и сопровождающих код, является отсутствие качественной документации. «Какие побочные эффекты имеет этот метод?», «Я столкнулся с ошибкой, выполнив шаг 3!», «Что означает эта аббревиатура?», «Этот документ актуален?» Каждый инженер-программист хотя бы раз высказывал претензии к низкому качеству, недостаточному объему или полному отсутствию документации, и программисты в Google не исключение.

Несмотря на участие технических писателей и руководителей проектов, большую часть документации приходится писать самим инженерам-программистам. Поэтому инженерам нужны соответствующие инструменты и стимулы, чтобы эффективно справляться с этой работой. Облегчить создание качественной документации им поможет внедрение процессов и инструментов, которые масштабируются вместе с организацией и связаны с текущими рабочими процессами.

По уровню развития создание технической документации в конце 2010-х годов напоминало тестирование ПО в конце 1980-х. Все понимают, что необходимо приложить больше усилий для совершенствования процессов документирования, но пока нет повсеместного признания их важности. Ситуация меняется, хотя и медленно. Мы в Google добились определенного успеха в этом направлении, когда документацию стали *рассматривать как код* и включили ее написание и поддержку в повседневный рабочий процесс инженера.

Что считается документацией?

Под словом «документация» мы подразумеваем любой дополнительный текст, который инженер должен написать в процессе своей работы: не только отдельные документы, но также комментарии в коде. (Фактически большая часть документации, которую пишет инженер в Google, имеет вид комментариев в коде.) Далее в этой главе мы обсудим различные виды технических документов.

Зачем нужна документация?

Качественная документация дает огромные преимущества инженерной организации. Благодаря ей код и API становятся более понятными, что способствует уменьшению количества ошибок. Команды действуют более эффективно, когда их цели четко определены в документации. Ручные процессы легче выполнять, когда все шаги четко прописаны. Вливание новых членов в команду или знакомство с новой базой кода требует гораздо меньше усилий, если рабочий процесс ясно задокументирован.

Но поскольку все преимущества наличия документации достаются тем, кто придет потом, они, как правило, не приносят выгоды ее автору, в отличие от тестирования (как мы увидим). Но, как бы то ни было, документ, написанный однажды¹, будет прочитан сотни или даже тысячи раз, и первоначальные усилия на его создание будут компенсированы уменьшением усилий всех будущих читателей. Документация масштабируется и приносит пользу не только ее будущим читателям, но и нынешним коллегам ее автора, что не менее важно. Она помогает ответить на такие вопросы:

- Почему были приняты такие проектные решения?
- Почему код реализован именно так?
- Почему я выбрал именно такую реализацию? (Пару лет спустя спросите вы, глядя на свой код.)

Если документация несет все эти преимущества, почему инженеры не любят писать ее? Одна из причин, как уже упоминалось, заключается в отсутствии немедленных выгод для автора. Но есть и другие причины:

- Инженеры часто рассматривают документирование как отдельный навык, не связанный с программированием. (Мы попробуем показать, что это не совсем так, даже в тех случаях, где это мнение вполне обоснованно.)
- Некоторые инженеры не считают себя хорошими писателями. Но вам не нужна надежная команда по английскому языку², чтобы писать хорошую документацию. Вам просто нужно выйти за привычные рамки и посмотреть на вещи со стороны.
- Писать документацию часто труднее из-за ограниченной инструментальной поддержки или слабой интеграции в рабочий процесс.
- Документация рассматривается как дополнительная нагрузка — что-то еще, что требует поддержки, а не средство, облегчающее сопровождение существующего кода.

Далеко не каждой команде инженеров нужен технический писатель (который все равно не справится со всей документацией). Это означает, что в общем случае инженеры будут писать большую часть документации самостоятельно. Поэтому вместо того чтобы заставлять инженеров становиться техническими писателями, мы должны по-

¹ Конечно, вам придется его поддерживать и периодически пересматривать.

² Английский по-прежнему считается основным языком для большинства программистов, и большая часть технической документации для программистов опирается на понимание английского языка.

думать о том, как упростить написание документации. Рано или поздно организации придется принять решение о том, сколько усилий нужно посвятить документированию.

Документация выгодна разным группам. Авторам документация дает следующие преимущества:

- Помогает сформулировать API и оценить его пригодность. Часто документирование приводит к переоценке проектных решений, которые не вызывали сомнений. Если вы не можете объяснить и определить что-то, то вы, вероятно, недостаточно хорошо это проработали.
- Служит хронологической записью для сопровождающих. Хитросплетений в коде следует избегать в любом случае, но если они есть, хорошие комментарии помогут понять их в коде, написанном пару лет назад.
- Делает код более профессиональным и привлекательным. Среди разработчиков бытует мнение, что хорошо документированный API — это хорошо проработанный API. Это не всегда так, хотя часто эти два аспекта тесно связаны друг с другом: наличие хорошей документации обычно является верным показателем того, насколько просто будет сопровождать продукт.
- Уменьшает поток вопросов от пользователей. Это, пожалуй, самое большое преимущество для тех, кто пишет документацию. Если вам приходится снова и снова что-то объяснять, обычно имеет смысл задокументировать этот момент.

Как бы ни были велики эти преимущества для автора документации, львиная доля выгод от нее, естественно, достается читателю. Руководство по стилю для C++ в Google особо подчеркивает принцип «оптимизации для читателя» (<https://oreil.ly/zCsPc>). Этот принцип применим не только к коду, но также к комментариям в коде и документации, прилагаемой к API. Со временем ценность документации только растет и может принести огромные выгоды для важного кода в масштабах организации.

Документация как код

Инженеры-программисты, которые пишут на одном основном языке программирования, все еще часто используют другие языки для решения узких задач. Инженер может писать сценарии на языке командной оболочки, а для выполнения некоторых задач в командной строке использовать Python или писать большую часть внутреннего кода на C++, а интерфейсную часть — на Java и т. д. Каждый язык — это инструмент в наборе.

Документация — это тоже инструмент, написанный на другом языке (обычно на английском) для решения конкретной задачи. Написание документации мало чем отличается от написания кода. Как и программирование, этот процесс управляется своими правилами, особым синтаксисом и соглашениями по стилю. Цель документации подобна той, что преследует код: обеспечить согласованность, повысить ясность и избежать ошибок. Грамматика в технической документации стандартизирует изложение и предотвращает путаницу в изложенных сведениях. По этой причине Google требует придерживаться определенного стиля в комментариях.

ПРИМЕР: ЭЛЕКТРОННАЯ ЭНЦИКЛОПЕДИЯ GOOGLE

Когда компания Google была намного меньше, в ней было мало технических писателей. В ту пору самым простым способом поделиться информацией была внутренняя электронная энциклопедия GooWiki. Сначала она казалась разумным подходом: все инженеры пользовались одним комплектом документации и могли обновлять ее при необходимости.

Но по мере роста Google проблемы с GooWiki становились все очевиднее. В отсутствие настоящих владельцев многие документы устарели¹. Без четкого процесса добавления новых документов начали появляться дубликаты. Пространство имен в GooWiki было плоским, поэтому ни о какой иерархии в наборах документов не было и речи. Мы нашли в GooWiki от семи до десяти документов (по разным подсчетам) с описанием настройки Borg — нашей производственной вычислительной среды. Только некоторые из них продолжали поддерживаться, а большинство были специфичны для определенных групп.

Со временем появилась еще одна проблема: исправлять документы в GooWiki могли не те, кто их использовал. Новые пользователи, обнаружившие ошибки в документах, либо не могли подтвердить их, либо не имели возможности сообщить о них. И наоборот, авторам, которые могли бы внести исправления, часто не требовалось работать с документами после их написания. Постепенно документация ухудшилась настолько, что ее низкое качество стало все чаще отмечаться в ежегодных опросах разработчиков Google.

Одно из решений, способствовавших улучшению документации, состояло в перемещении важной документации в VCS, аналогичную системе для хранения кода. Документы приобрели владельцев, каноническое местоположение в дереве исходного кода и процессы выявления и исправления ошибок. Благодаря этим мерам документация начала резко улучшаться. Кроме того, процессы написания и поддержки документации стали более похожими на процессы написания и поддержки кода. Появилась возможность сообщать о найденных в документах ошибках в программе отслеживания ошибок и вносить изменения в документы в процессе обзора кода. В конце концов, инженеры начали сами исправлять документы или посылать изменения техническим писателям (которые часто были владельцами).

На первом этапе перемещение документации в VCS столкнулось с рядом противоречий. Многие инженеры были убеждены, что отказ от GooWiki, этого бастиона свободы информации, приведет к снижению качества документов из-за повышенных требований к ним (требований к проведению обзоров, наличию владельцев и т. д.). Но на самом деле документы стали только лучше.

Также помогло внедрение Markdown в роли общепринятого языка разметки для оформления документации, благодаря которому инженеры быстро освоили приемы редактирования документов без специальных знаний HTML и CSS. В итоге Google представила свой фреймворк для встраивания документации в код: g3doc (<https://oreil.ly/YjrTD>), который способствовал дальнейшему улучшению качества документации, находящейся рядом с исходным кодом в среде разработки. Теперь инженеры могут обновлять код и связанную с ним документацию в одном изменении (мы продолжаем попытки распространить эту практику).

Главный прорыв состоит в том, что поддержка документации стала больше походить на поддержку кода: инженеры выявляют ошибки в документах, вносят изменения, отправляют их экспертам для обзора и т. д. Ключевым преимуществом выбранного подхода стало использование существующих рабочих процессов вместо создания новых.

¹ Отказавшись от GooWiki, мы обнаружили, что около 90 % документов не обновлялось и не пересматривалось в течение многих месяцев.

Документы, так же как код, должны иметь владельцев. Документы без владельцев становятся устаревшими и сложными в сопровождении. Чем более ясно выражено владение, тем проще обрабатывать документацию с использованием существующих рабочих процессов: систем отслеживания ошибок, инструментов обзора кода и т. д. Конечно, документы, принадлежащие разным владельцам, могут конфликтовать друг с другом. В этих случаях важно определить *каноническую* документацию: найти главный источник и объединить в него другие связанные документы (или исключить дубликаты).

Документы с прямыми ссылками `go/` (глава 3) часто становятся каноническим источником информации. Еще один способ выделения канонических документов — их размещение в VCS и рядом с соответствующим исходным кодом.

Документация часто тесно связана с кодом и рассматривается *как код* (<https://oreil.ly/GOLBo>). Она должна:

- подчиняться внутренним правилам;
- храниться в VCS;
- иметь ясно обозначенных владельцев, ответственных за ее сопровождение и поддержку;
- подвергаться обзорам в случае изменения (и изменяться вместе с кодом, который она описывает);
- как и код, отражаться в системе отслеживания ошибок;
- периодически переоцениваться (подвергаться своего рода тестированию);
- оцениваться на точность, актуальность и т. д. (эта сфера остается неохваченной инструментами).

Чем чаще инженеры будут рассматривать документацию как «одну из» обязательных задач разработки ПО, тем меньше они будут возмущаться необходимостью тратить время и силы на ее написание и тем весомее будут долгосрочные выгоды от документирования. Кроме того, упрощение процесса документирования может помочь снизить первоначальные затраты на него.

Знание своей аудитории

Одна из главных ошибок при написании документации — документировать только для себя. Это естественное стремление, и написание для себя тоже несет определенную пользу: в конце концов, вам может понадобиться заглянуть в код через несколько лет и понять, что вы имели в виду, когда его писали. Конечно, читающие ваш документ могут иметь примерно тот же уровень знаний и навыков, что и вы. Но если писать только для себя, вы неизбежно будете основываться на определенных предположениях, а к вашему документу, возможно, будет обращаться очень широкая аудитория (все инженеры, внешние разработчики), и потеря даже нескольких читателей — это большие затраты. По мере роста организации ошибки в документации будут становиться более заметными, а ваши предположения — более ошибочными.

Поэтому, прежде чем начать писать, вы должны (формально или неформально) определить аудиторию документа. Проектный документ в первую очередь предназначен для представления доказательств лицам, принимающим решения. Учебное пособие должно содержать четкие инструкции для тех, кто совершенно не знаком с вашей базой кода. В описании API может потребоваться предоставить полную и точную справочную информацию для любых пользователей этого API, будь то эксперты или новички. Всегда старайтесь определить основную аудиторию и пишите для нее. Хорошая документация не должна быть «идеальной». Часто инженеры ошибочно предполагают, что они должны в совершенстве владеть навыками технического писателя. Однако мало кто из инженеров-программистов способен писать документацию на таком уровне. Подходите к написанию документации как к тестированию или любому другому процессу — то есть как инженер. Пишите текст, используя слог и стиль, ожидаемые вашей аудиторией. Если вы умеете читать документы, то сможете их написать. Помните, что вы тоже когда-то находились в вашей аудитории, только *без сегодняшних знаний в предметной области*. Поэтому от вас не требуется быть великим писателем; вы просто должны познакомить читателя с предметной областью. (И пока вы отвечаете за свою документацию, вы можете улучшать ее.)

Виды аудиторий

Как отмечалось выше, ваши документы должны соответствовать уровню навыков и знаний аудитории. Но кто относится к аудитории? Скорее всего, у документа будет несколько аудиторий, разделенных по следующим критериям:

- уровень опыта (опытные программисты или младшие инженеры, которые могут даже не знать — внимание! — языка);
- уровень знания предметной области (члены команды или другие инженеры в организации, которые знакомы только с конечными точками API);
- цель (конечные пользователи, которым нужен ваш API для решения конкретной задачи, желающие быстро найти информацию, или программисты-гуру, отвечающие за внутреннюю реализацию, которую, по вашему мнению, больше никому не придется поддерживать).

Иногда для разных аудиторий документация должна быть написана в разных стилях, но в большинстве случаев документация должна быть настолько универсальной, насколько это возможно. Часто вам придется объяснять сложную тему и экспертам, и новичкам. Описывая предметную область для эксперта, хорошо знающего ее, вы можете позволить себе срезать некоторые углы, но тогда вы запутаете новичка. И наоборот, подробное объяснение, написанное для новичка, несомненно будет раздражать эксперта.

Очевидно, что документирование связано с постоянным поиском баланса, и эта проблема не имеет универсального решения, но мы уверены, что такой подход помогает сократить документы. Пишите достаточно наглядно, объясняя сложные темы

людям, незнакомым с ними, но не отталкивайте и не раздражайте экспертов. Чтобы написать короткий документ, часто требуется сначала написать более длинный (изложив в нем все сведения), а затем выполнить редактирование и удалить повторяющуюся информацию везде, где возможно. Это может показаться утомительным, но имейте в виду, что эти усилия распространяются на всех читателей документации. Как однажды сказал Блез Паскаль, «если бы у меня было больше времени, то я бы написал короче». Стараясь сохранить документ как можно более коротким и ясным, вы сможете гарантировать, что он удовлетворит как эксперта, так и новичка.

Не менее важно уметь различать аудиторию, основываясь на том, как пользователь применит документ.

- *Целеустремленные искатели* — это инженеры, которые *знают, чего хотят*, и для них важно, чтобы документ отвечал их потребностям. Ключевым приемом при создании документации для этой аудитории является *последовательность*. Если вы пишете справочную документацию для этой аудитории — например, в файле кода, — ваши комментарии должны следовать формату справочника, чтобы читатели могли быстро просмотреть их и определить, есть ли в них искомые сведения.
- *Наткнувшиеся случайно* могут не знать точно, чего они хотят. Они могут иметь лишь смутное представление о том, как реализовать необходимое. Ключ к этой аудитории — *ясность*. Добавьте в документацию общий обзор или вводную часть (например, в начале файла) с объяснением назначения кода, который они просматривают. Также полезно указать, для какой аудитории документ не подходит. Многие документы в Google начинаются с «аннотации», например: «Аннотация: если вас не интересуют компиляторы C++ в Google, можете дальше не читать».

Наконец, учитывайте разницу между потребителем (например, пользователем API) и производителем (например, членом команды проекта) — по возможности документы, предназначенные для одной аудитории, должны отличаться от документов, предназначенных для другой. Детали реализации важны членам команды для успешного сопровождения, а конечным пользователям не нужна эта информация. Часто инженеры включают проектные решения в справочник по API своей библиотеки. Описание таких решений более уместно в конкретных (проектных) документах или, в лучшем случае, внутри реализации, скрытой за интерфейсом.

Виды документации

В процессе работы инженеры пишут разные виды документации: проектные документы, комментарии в коде, инструкции и многое другое. Все это считается «документацией». Но важно знать разные виды документации и *не смешивать их*. Документ должен иметь, как правило, одну цель и придерживаться ее. Так же как в API, который должен делать что-то одно и делать это хорошо, не пытайтесь совместить все и вся в одном документе. Разбейте информацию на логические части.

Существует несколько основных видов документов, которые часто приходится писать инженерам-программистам:

- справочная документация, включая комментарии в коде;
- проектные документы;
- учебные руководства;
- концептуальная документация;
- посадочные страницы.

В первые годы существования Google для нас было обычным делом создавать монолитные wiki-страницы с множеством ссылок (многие из которых оказывались неработающими или устаревали со временем), концептуальной информацией о системе, справкой по API и т. д. Такие документы были очень неудобны, потому что не служили ни одной цели (и были настолько длинными, что их никто не хотел читать; некоторые из них занимали до нескольких десятков экранов). Не повторяйте наших ошибок, пишите документ, преследующий одну цель, и если информация, добавляемая на страницу, не связана с ней по смыслу, подберите для нее другой документ или даже создайте новый.

Справочная документация

Справочная документация — это наиболее распространенный вид документации, которую приходится писать инженерам почти каждый день. К ней относятся все документы, описывающие порядок использования кода. Комментарии в коде — самая распространенная форма справочной документации, которую должен вести инженер. Их можно разделить на две основные категории: комментарии с описанием API и комментарии с описанием реализации. Помните о различиях между аудиториями этих двух категорий. Комментарии с описанием API не должны обсуждать детали реализации или проектные решения и не могут предполагать, что пользователь разбирается в API настолько же хорошо, как и автор. Комментарии с описанием реализации, напротив, могут предполагать наличие у читателя глубоких знаний предметной области, но будьте осторожны и не заходите слишком далеко в своих предположениях: вы можете покинуть проект, поэтому должны подробно описать, почему код написан именно так.

Большая часть справочной документации, даже если она оформляется как отдельный документ, генерируется из комментариев внутри кодовой базы. (Желательно, чтобы справочная документация имела только один источник.) Некоторые языки программирования, такие как Java или Python, имеют специализированные фреймворки для обработки комментариев (Javadoc, PyDoc, GoDoc), упрощающие создание справочной документации. Другие языки, такие как C++, не имеют стандартной реализации справочной документации, но поскольку C++ отделяет определение API (в заголовочных файлах `.h`) от реализации (в файлах `.cc`), заголовочные файлы часто являются естественным местом для описания API в C++.

В Google широко используется такой подход, как размещение справочной документации с описанием API на C++ в заголовочных файлах. В других языках, таких как Java, Python и Go, справочная документация встраивается непосредственно в исходный код. Браузер Google Code Search (глава 17) настолько надежен, что избавляет нас от необходимости генерировать отдельную справочную документацию. Пользователи Code Search не только с легкостью находят нужный код, часто им удается найти оригинальное определение в первых результатах поиска. Наличие документации в коде также облегчает ее поиск и сопровождение.

Все мы знаем, насколько важны комментарии в коде для документирования API. Но что отличает «хорошие» комментарии? Выше в этой главе мы определили две основные аудитории справочной документации: целеустремленные искатели и пользователи, наткнувшиеся на документ случайно. Для первых важно, чтобы база кода была прокомментирована последовательно для быстрого поиска искомых сведений. Вторым важно наличие четкого определения назначения API, желательно в начале заголовочного файла. В следующих подразделах мы подробнее рассмотрим некоторые виды комментариев в коде. Далее представлены рекомендации документирования для C++, которые подходят и для других языков.

Комментарии в начале файла

В Google почти все файлы с исходным кодом должны начинаться с комментариев. (Некоторые заголовочные файлы, содержащие только одну вспомогательную функцию и т. д., могут отступать от этого стандарта.) Комментарии в начале файла должны начинаться примерно с такого заголовка:

```
// -----  
// str_cat.h  
// -----  
//  
// Этот заголовочный файл содержит сигнатуры эффективных функций  
// для объединения строк: StrCat() и StrAppend(). Основная работа в этих  
// функциях выполняется с использованием типа AlphaNum, специально  
// разработанного как тип параметров, который эффективно управляет  
// преобразованием строк и позволяет избежать копирования  
// в вышеперечисленных операциях.  
...  

```

Как правило, комментарий в начале файла должен начинаться с общего описания того, что содержится в коде, перечислять основные варианты использования кода и идентифицировать аудиторию (в данном случае это разработчики, желающие использовать инструменты объединения строк). Если API не получается кратко описать в одном-двух абзацах, это обычно является признаком плохо продуманного API. В таких случаях можно попробовать разбить API на отдельные компоненты.

Комментарии к классам

Большинство современных языков программирования являются объектно-ориентированными. Поэтому комментарии к классам важны для определения «объектов»

API в кодовой базе. Все общедоступные классы (и структуры) в Google должны сопровождаться комментарием к классу, описывающему класс (структуру), основные методы и назначение класса (структуры). Как правило, комментарии к классам должны подчеркивать их объектную сущность, например: «Класс Foo, содержащий x, y, z, позволяет выполнять такие-то действия и характеризуется следующими свойствами» и т. п.

В общем случае комментарии к классам должны выглядеть примерно так:

```
// -----  
// AlphaNum  
// -----  
//  
// Класс AlphaNum действует в роли основного типа параметра для StrCat()  
// и StrAppend() и обеспечивает эффективное преобразование числовых,  
// логических и шестнадцатеричных значений (через тип Hex) в строки.
```

Комментарии к функциям

В Google все свободные функции и общедоступные методы классов тоже должны сопровождаться комментариями, описывающими их *назначение*. Комментарии к функциям должны описывать их *активное* состояние и начинаться с глагола в изъявительном наклонении, указывающего, что делает функция и что она возвращает.

В общем случае комментарии к функциям должны выглядеть примерно так:

```
// StrCat()  
//  
// Объединяет заданные строки или числа без использования разделителя,  
// возвращает результат объединения в виде строки.  
...  

```

Обратите внимание, что начальный глагол в комментарии к функции обеспечивает согласованность заголовочного файла. Целеустремленные искатели могут быстро пробежать взглядом по API и прочитав только глаголы, чтобы понять, подходит ли им та или иная функция: «объединяет, удаляет, создает» и т. д.

Некоторые стили документирования (и некоторые генераторы документации) требуют наличия в комментариях к функциям различных шаблонов, таких как «Returns:» (возвращает), «Throws:» (генерирует) и т. д., но мы в Google не считаем их строго необходимыми. Часто такую информацию лучше представить в свободной форме, не разбивая комментарий на разделы:

```
// Создает новую запись с информацией о клиенте, используя заданные имя  
// и адрес, и возвращает числовой идентификатор записи или генерирует  
// исключение `DuplicateEntryError`, если запись для клиента с именем name  
// уже существует.  
int AddCustomer(string name, string address);
```

Обратите внимание, насколько естественно выглядит совместное (в данном случае в одном предложении) описание постусловия, параметров, возвращаемого значения

и исключительных случаев. Добавление явных шаблонных разделов сделало бы комментарий более многословным и повторяющимся, но не более (и, возможно, даже менее) ясным.

Проектная документация

Для начала работы над крупным проектом большинство команд в Google должны иметь одобренный проектный документ. Обычно проектные документы пишутся с использованием шаблона, утвержденного командой. Они предназначены для совместной работы и поэтому часто публикуются в Google Docs, где есть удобные инструменты для сотрудничества. Некоторые команды требуют, чтобы проектные документы рассматривались и обсуждались на общих совещаниях, где эксперты могли бы подвергнуть критике наиболее важные аспекты проекта. Предварительное обсуждение проекта действует как своеобразная форма обзора перед написанием кода.

Поскольку разработка проектной документации осуществляется перед развертыванием новых систем, она подходит для очерчивания круга задач. Канонические шаблоны проектных документов в Google требуют включить в описание будущего проекта такие его аспекты, как безопасность, интернационализация, требования к хранилищу, конфиденциальность и т. д. В большинстве случаев эти аспекты рассматривают эксперты в соответствующих областях.

Хороший проектный документ должен описывать цели проекта, стратегию его реализации и предлагать ключевые проектные решения с акцентом на конкретные компромиссы. Лучший проектный документ должен также охватывать альтернативные проектные решения и обозначать их сильные и слабые стороны.

После утверждения хороший проектный документ действует не только как хронологическая запись, но и как мера успеха в достижении поставленных целей. Большинство команд сохраняют проектные документы вместе с остальными своими документами, чтобы иметь возможность периодически заглядывать в них. Часто бывает полезно пересмотреть проектную документацию перед выпуском продукта, чтобы убедиться, что цели, заявленные в проектном документе, актуальны (а если это не так, то документ или продукт можно соответствующим образом скорректировать).

Тьюториалы

Каждый инженер-программист, присоединяясь к новой команде, должен как можно быстрее войти в курс дела. Наличие тьюториала (учебного руководства), помогающего настроить новый проект, не оценимо. Написание программы «Hello World» — это один из лучших способов взять удачный старт в новом деле. Это касается не только документов, но и кода. Большинство проектов стоит начать с создания документа «Hello World», который ничего не предполагает и помогает инженеру сделать что-то «реальное».

Часто лучшее время для создания учебного руководства, если его пока нет, — период знакомства с новой командой. (Это также лучшее время для поиска ошибок в существующем учебном руководстве, которое вы читаете.) Откройте блокнот (или другой инструмент для заметок) и запишите все шаги, которые нужно выполнить для входа в курс дела, не имея знания предметной области или специальных ограничений. Закончив этот список, вы поймете, какие ошибки допускали и почему, а затем сможете отредактировать свои записи, чтобы сформулировать последовательное учебное руководство. Обязательно описывайте все, что вам пришлось сделать, когда вы еще не знали конкретные настройки, разрешения или особенности предметной области. Если для старта вам были нужны определенные настройки, четко опишите их в начале руководства в виде набора предварительных условий.

Если учебное руководство требует выполнения нескольких шагов в определенном порядке, пронумеруйте эти шаги. В руководстве, ориентированном на *пользователя* (например, внешнего разработчика), нумеруйте каждое действие, которое пользователь должен выполнить сам. Не нумеруйте действия, которыми система отвечает на действия пользователя. Явная нумерация шагов играет важную роль. Ничто так не раздражает, как ошибка на шаге 4 из-за неправильной авторизации.

Пример: плохой туториал

1. Загрузите пакет с нашего сервера: `http://example.com`.
2. Скопируйте сценарий командной оболочки в свой домашний каталог.
3. Запустите сценарий.
4. Система foobar соединится с системой аутентификации.
5. После аутентификации foobar создаст новую базу данных с именем `baz`.
6. Протестируйте `baz`, выполнив команду SQL в командной строке.
7. Введите: `CREATE DATABASE my_foobar_db;`

В предыдущей процедуре шаги 4 и 5 выполняются на стороне сервера. Нужно ли пользователю что-либо делать — неясно, в данном случае никаких действий со стороны пользователя не требуется, поэтому такие побочные эффекты лучше упомянуть в описании шага 3. Также неясно, являются ли шаги 6 и 7 разными действиями. (Здесь это не так.) Объедините все элементарные пользовательские операции в один шаг, чтобы пользователь знал, что делать на каждом этапе процесса. Кроме того, если ваше учебное руководство предлагает пользователю что-то ввести или проверить какой-то вывод, поместите команды и примеры вывода в отдельные строки (например, используя соглашение об оформлении **моноширинным жирным шрифтом**).

Пример: улучшенная версия плохого туториала

1. Загрузите пакет с нашего сервера: `http://example.com`:

```
$ curl -I http://example.com
```

2. Скопируйте сценарий командной оболочки в свой домашний каталог:

```
$ cp foobar.sh ~
```

3. Запустите сценарий в своем домашнем каталоге:

```
$ cd ~; foobar.sh
```

Система foobar соединится с системой аутентификации. Затем foobar создаст новую базу данных с именем **baz** и откроет оболочку для ввода команд.

4. Протестируйте **baz**, выполнив команду SQL в командной строке:

```
baz:$ CREATE DATABASE my_foobar_db;
```

Обратите внимание, что каждый шаг требует конкретных действий со стороны пользователя. Если tutorial фокусируется на каком-то другом аспекте (как, например, документ с описанием «жизненного цикла сервера»), пронумеруйте шаги с точки зрения этого аспекта (перечислите, что делает сервер).

Концептуальная документация

Иногда требуется дать более глубокое описание кода, которое нельзя получить из обычной справочной документации. В таких случаях мы пишем концептуальную документацию, в которой даем обзор API или систем. Примерами концептуальной документации могут служить обзор API популярной библиотеки, описание жизненного цикла данных на сервере и т. д. Практически во всех случаях цель концептуального документа — дополнять, а не заменять справочную документацию. Часто это приводит к дублированию некоторой информации, но с важной целью: повысить ясность. В концептуальном документе необязательно охватывать все худшие случаи (но они обязательно должны быть перечислены в справочной документации). Здесь можно пожертвовать точностью ради ясности.

«Концептуальные» документы являются наиболее сложными документами для написания, поэтому инженеры-программисты часто пренебрегают ими. Эти документы невозможно встроить непосредственно в исходный код из-за отсутствия канонического места для них. Наиболее подходящим местом для «концептуального» объяснения обширного API является начальный комментарий в файле. Но нередко API используют другие API и/или модули, и единственное логичное место для описания такого сложного поведения — отдельный концептуальный документ. Если сравнить документирование с тестированием, комментарии будут похожи на юнит-тесты, а концептуальные документы — на интеграционные тесты.

Даже для узких API часто имеет смысл написать отдельный концептуальный документ. Например, библиотека `Abseil StrFormat` охватывает множество концепций, которые должны понимать ее пользователи, и мы предоставляем документ с описанием концепции ее формата (<https://oreil.ly/TMwSj>), доступный как внутри библиотеки, так и за ее пределами.

Концептуальный документ приносит пользу самой широкой аудитории, от экспертов до новичков. Кроме того, он должен делать упор на *ясность*, поэтому для его создания нередко приходится жертвовать полнотой сведений (описанных в справочной до-

кументации) и (иногда) их точностью. Это не значит, что концептуальный документ должен быть намеренно неточным, но основное внимание в нем должно уделяться типичному использованию кода, а побочные эффекты лучше описать в справочной документации.

Домашние страницы

Большинство инженеров являются членами команд, и у большинства команд есть свои «домашние страницы» где-то в интранете компании. Часто эти страницы запутаны: типичная домашняя страница может содержать несколько интересных ссылок, иногда несколько документов под названием «Сначала прочтите это!», а также некоторую информацию для команды и ее клиентов. Такие документы, первоначально полезные, быстро превращаются в одно сплошное бедствие; они постепенно становятся слишком громоздкими, устаревают, и за их исправление берутся только самые смелые или отчаянные.

К счастью, такие документы только выглядят пугающие, но на самом деле их легко исправить: убедитесь, что домашняя страница четко сообщает о своем назначении, и оставьте на ней *только* ссылки на другие страницы с более подробной информацией. Если домашняя страница выполняет какие-то дополнительные функции кроме регулировки движения, то она явно *выполняет не свою работу*. Если у вас есть отдельный документ с описанием установки, добавьте в домашнюю страницу ссылку на него. Если на домашней странице окажется слишком много ссылок (домашняя страница не должна прокручиваться на несколько экранов), подумайте об их классификации.

Часто домашняя страница служит двум разным целям: играет роль страницы «перехода» для тех, кто пользуется вашим продуктом или API, и является домашней страницей для команды. Не позволяйте странице обслуживать двух хозяев — это может сбить с толку. Создайте отдельную внутреннюю «страницу для команды» и отдельную главную домашнюю страницу. То, что нужно знать команде, часто сильно отличается от того, что нужно знать пользователю API.

Обзоры документации

В Google весь код подвергается рецензированию, и наш процесс обзора кода вам уже понятен. В общем случае документация также нуждается в рецензировании (хотя это не общепринятый подход). Чтобы «проверить» качество вашей документации, отправьте ее на обзор.

Технические документы подвергаются обзорам трех типов, каждый из которых преследует свои цели:

- Технический обзор для проверки точности обычно проводится экспертами в предметной области или другими членами вашей команды часто в ходе обзора кода.

- Обзор с позиции аудитории для проверки ясности проводится лицами, незнакомыми с предметной областью, — новичками в вашей команде или пользователями вашего API.
 - Проверка орфографии и редаKTура проводятся техническими писателями или волонтерами.
-

ПРИМЕР: РУКОВОДСТВО РАЗРАБОТЧИКА БИБЛИОТЕКИ

Как упоминалось выше, мы столкнулись с проблемами, обусловленными тем, что большая часть (почти вся) технической документации находилась в общей электронной энциклопедии (Wiki). Среди них — отсутствие четко обозначенных владельцев, конкурирующая документация, устаревшая информация и трудности с регистрацией ошибок. Но эти проблемы не коснулись руководства по стилю для C++, потому что арбитры стиля взяли на себя заботу о нем. Их владение документом принималось безоговорочно. Кроме того, документ был каноническим: в Google существовало только одно руководство по стилю для C++.

Как уже отмечалось, если документация находится рядом с исходным кодом, она обычно является наиболее авторитетной (надеюсь) и становится канонической. В Google каждый API обычно имеет отдельный каталог *g3doc*, в котором находятся канонические документы (в виде файлов в формате Markdown, доступных для чтения в нашем браузере Code Search). Размещение документации рядом с исходным кодом не только устанавливает фактическое владение, но и делает документацию «частью» кода.

Однако размещение некоторых наборов документов в исходном коде выглядит нелогично. Например, для «Руководства разработчика на C++», созданного в Google, нет очевидного места в исходном коде. Нет и главного каталога «C++». В таких случаях (и если документация выходит за рамки одного API) мы стали создавать отдельные наборы документов. Многие из них объединяли существующие документы в общий набор с общей навигацией и общим внешним видом. Такие документы маркировались как «Руководство разработчика» и, так же как код в кодовой базе, сохранялись в VCS, в специальном разделе для документации, причем этот раздел был организован по темам, а не по API. Поддержкой этих руководств для разработчиков в большинстве случаев занимались технические писатели, потому что они лучше справлялись с описанием тем, выходящих за рамки конкретных API.

Со временем эти руководства для разработчиков стали каноническими. Пользователи, писавшие конкурирующие или дополняющие документы, начали стремиться добавлять свои документы в канонический набор, а затем отказываться от своих конкурирующих документов. В итоге руководство по стилю для C++ стало частью более крупного «Руководства разработчика на C++». Набор документации стал более полным и авторитетным, и его качество улучшилось. Инженеры начали сообщать об ошибках, потому что знали точно, что кто-то сопровождает эти документы. А поскольку документы хранились в VCS и имели конкретных владельцев, инженеры также начали отправлять списки изменений непосредственно техническим писателям.

Внедрение ссылок [go/](#) (глава 3) упростило выделение каноничных документов по разным темам. Например, наше «Руководство разработчика на C++» было опубликовано под ссылкой [go/cpp](#). Благодаря улучшенному внутреннему поиску, ссылкам [go/](#) и объединению документов наборы канонической документации со временем стали еще более авторитетными и надежными.

Конечно, эти виды обзоров имеют довольно размытые границы, но если ваш документ предназначен для широкой аудитории или может публиковаться за пределами организации, то вы наверняка захотите подвергнуть его как можно более разносторонней оценке. (Аналогичный процесс обзора мы использовали для этой книги.) Вышеупомянутые виды обзора полезны для оценки даже узкоспециальных документов. Но если ваш текст оценит всего один рецензент, это все равно лучше, чем отсутствие рецензии.

Важно отметить, что если документация связана с технологическим процессом, ее нужно улучшать со временем. Большинство документов в Google неявно проходят обзор с позиции аудитории, когда читатели кода находят в них ошибки и дают обратную связь (с использованием разных инструментов).

Философия документирования

Внимание: следующий раздел — это скорее рассуждение (и личное мнение) о лучших методах написания технических документов, а не описание «как это делается в Google». Понимание представленных идей, вероятно, поможет вам писать техническую документацию.

Кто, что, когда, где и почему

Большинство технических документов отвечают на вопрос КАК. Как это работает? Как использовать этот API? Как настроить этот сервер? В результате инженеры-программисты склонны сразу переходить к разделу КАК в документе и игнорировать другие вопросы: КТО, ЧТО, КОГДА, ГДЕ и ПОЧЕМУ. Да, они не так важны, как вопрос КАК. Исключением является только проектный документ, потому что он в большей степени посвящен ответу на вопрос ПОЧЕМУ. Но без надлежащей организации технические документы приводят к путанице. Попробуйте ответить на другие вопросы, кроме КАК, в первых двух абзацах документа:

- КТО, как обсуждалось выше, — это аудитория документа. Иногда желательно обозначить ее явно, например: «Этот документ предназначен для новых инженеров проекта Secret Wizard».
- ЧТО определяет назначение документа: «Этот документ является руководством по запуску сервера Frobber в среде тестирования». Иногда раздел ЧТО помогает правильно оформить документ. Информацию, не применимую к ЧТО, вы сможете переместить в отдельный документ.
- КОГДА определяет дату создания, пересмотра или обновления документа. Документы в исходном коде датируются неявно, а некоторые другие схемы публикации автоматически добавляют дату. Обязательно укажите в самом документе, когда он был написан (или пересмотрен в последний раз).
- ГДЕ — это место хранения документа, которое часто подразумевается автором неявно. Обычно предпочтительнее использовать VCS, в идеале — *рядом с исходным*

кодом, который документ описывает. Но можно использовать и другие хранилища. Мы в Google часто используем Google Docs, чтобы упростить совместную работу, особенно с проектными документами. Однако в какой-то момент обсуждение любого общего документа заканчивается, и он становится хронологической записью. В этот момент желательно переместить документ в более надежное место, обеспечивающее права владения, управление версиями и ответственность.

- **ПОЧЕМУ** объявляет цель документа. Кратко опишите, что читатели смогут почерпнуть из документа. Хорошая практика — перечислить разные **ПОЧЕМУ** во введении к документу. Когда вы будете писать заключение, проверьте, сбылись ли ваши первоначальные ожидания (и внесите соответствующие правки).

Начало, середина и конец

Все документы — и все их части — имеют начало, середину и конец. Эти слова могут показаться странными, но большинство документов должны содержать как минимум эти три раздела. Документ, включающий только один раздел, может охватить только одну тему, при этом очень немногие документы действительно описывают единственную тему. Не бойтесь добавлять разделы в документы — они разбивают поток повествования на логические части и помогают читателям быстро оценить, какие темы в документе представлены.

Даже в самом простом документе обычно рассматривается больше одной темы. Наши популярные «Советы недели для C++» очень короткие и основное внимание уделяют одному небольшому совету. Однако даже в них деление на разделы приносит дополнительные выгоды. Традиционно первый раздел описывает проблему, второй рассматривает рекомендуемые решения, а в последнем подводятся итоги. Если бы документ состоял только из одного раздела, читателям было бы трудно вычленив в нем важные моменты.

Большинство инженеров ненавидят избыточность, и тому есть причины. Но в документации избыточность бывает полезна. Порой трудно заметить и запомнить важный момент, спрятанный за стеной текста. С другой стороны, более раннее описание такого момента в более заметном месте может привести к потере его контекста. Типичное решение этой проблемы состоит в том, чтобы кратко представить тот момент во вступительном абзаце, а в остальной части раздела изложить свою точку зрения более подробно. В этом случае избыточность поможет читателю понять важность того, о чем говорится.

Признаки хорошей документации

Типичными признаками хорошей документации являются полнота, точность и ясность. Редко удается совместить все три признака в одном документе: например, более «полный» документ, скорее всего, будет менее ясным. Документируя все варианты использования API, вы получите очень запутанный документ. В документах с описанием языка программирования полная точность в отношении всех вариантов

использования (и перечисление всех побочных эффектов) тоже может нанести ущерб ясности. В других документах попытка прояснить сложную тему может повлиять на точность. В концептуальном документе можно опустить редкие побочные эффекты, потому что его цель — познакомить читателя с API, а не представить скрупулезный обзор всех вариантов поведения кода.

В любом случае, «хорошим» можно считать документ, который *соответствует своему назначению*. Поэтому нежелательно создавать документы с несколькими назначениями. Определите направленность каждого документа (и каждый тип документов) и пишите их соответствующим образом. Пишете концептуальный документ? Тогда вам не нужно расписывать API во всех деталях. Пишете справочник? Тогда уделите внимание точности, пусть и за счет некоторой потери ясности. Пишете домашнюю страницу? Сосредоточьтесь на ее организации и постарайтесь сократить описательный текст до минимума. Все это — слагаемые качества, которое, по общему признанию, трудно измерить.

Как быстро улучшить качество документа? Сосредоточьтесь на потребностях аудитории. Часто чем меньше, тем лучше. Например, многие инженеры допускают распространенную ошибку, добавляя описание проектных решений или деталей реализации в документ по API. Так же как вы отделяете интерфейс API от его реализации, избегайте обсуждения проектных решений в документе по API. Пользователям не нужна эта информация. Лучше опишите эти решения в отдельном документе (обычно проектном).

Устаревшие документы

Проблемы может вызывать не только устаревший код, но и устаревшие документы. Со временем документы устаревают и (нередко) забываются. Старайтесь не забывать документы, и если документ перестал служить своей цели, удалите его или обозначьте как устаревший (и если возможно, укажите, куда обратиться за актуальной информацией). Даже добавить простое примечание «Это больше не работает!» намного полезнее, чем оставить в неизменном виде документ, который выглядит авторитетным, но больше не актуален.

Мы в Google часто включаем упоминание о «свежести документа». Такие упоминания фиксируют дату, когда в последний раз документ пересматривался, и, основываясь на ней, система сопровождения документации отправляет владельцу напоминания по электронной почте, если документ не пересматривался, например, более трех месяцев. Такие даты обновления, как показано в следующем примере, свидетельствующие об ошибочности документов, могут помочь упростить сопровождение набора документации с течением времени:

```
<!--*
# Дата обновления: за дополнительной информацией обращайтесь по ссылке
# go/fresh-source.
freshness: { owner: `username` reviewed: '2019-02-27' }
*-->
```

Владельцы таких документов получают стимул поддерживать актуальность даты обновления (а если документ находится под контролем VCS, еще и стимул пересмотреть код). Это недорогое средство способствует периодической проверке документации. Мы в Google обнаружили, что добавление в документ упоминания о его владельце с подписью «Последний пересмотр...» также способствовало распространению практики пересмотра документации.

Когда привлекать технических писателей?

Когда Google была молодой и растущей компанией, в сфере программной инженерии ощущалась нехватка технических писателей. (Впрочем, их не хватает и сейчас.) Те проекты, которые считались важными, стремились привлечь к работе технического писателя независимо от того, была ли в этом необходимость. Идея заключалась в том, что писатель должен был снять с команды часть бремени написания и ведения документов и (теоретически) позволить важному проекту достичь более высокой скорости разработки. Эта идея себя не оправдала.

Как показал наш опыт, большинство команд инженеров отлично справляются с написанием документации для себя (своих команд), но когда речь заходит о документации для другой аудитории, инженеры ищут помощи технического писателя. Петля обратной связи внутри команды при написании документов более короткая, а знание предметной области и предполагаемые потребности более очевидны. Конечно, технический писатель часто лучше справляется с грамматикой и организацией документов, но поддержка единственной команды — не лучшее использование такого ограниченного и ценного ресурса, как технический писатель, поскольку такой подход не масштабируется. В итоге появился неверный стимул: станьте важным проектом, и вашим инженерам-программистам не придется писать документы. Отгораживание инженеров от написания документов, как оказалось, — это совсем не то, что вам нужно.

Будучи ограниченным ресурсом, технические писатели, как правило, сосредоточены на задачах, которые *не должны* выполняться инженерами-программистами в рамках своих обязанностей. Обычно к таким задачам относится написание документов, выходящих за рамки API. Команда проекта Foo может четко понимать, какая документация нужна, но с трудом представлять, что нужно проекту Bar. Технический писатель лучше справится с ролью человека, не знакомого с предметной областью. Фактически очень полезно оспаривать предположения команды относительно полезности проекта. Это одна из причин, по которой многие, если не большинство технических писателей, в сфере программной инженерии склонны концентрироваться на конкретном типе документации по API.

Заключение

За последние десять лет мы в Google добились значительных успехов в решении проблемы качества документации, но, честно говоря, документация в Google все

еще не является образцом для подражания. Для сравнения, инженеры постепенно осознали, что тестировать необходимо любое изменение в коде, каким бы малым оно ни было. Более того, в разные этапы технологического процесса были включены разнообразные и надежные инструменты тестирования. Однако документации еще далеко до этого уровня.

Впрочем, справедливости ради следует признать, что с документацией не обязательно обращаться так же, как с тестированием. Тесты могут быть атомарными (юнит-тесты) и соответствовать предписанной форме и функциям. Для документов это по большей части невозможно. Тестирование можно автоматизировать, а методики автоматизации ведения документации часто отсутствуют. Документы всегда субъективны; качество документа оценивается читателем, и зачастую спустя время. Тем не менее уже есть понимание важности документации, и процессы, связанные с разработкой документов, постепенно совершенствуются. По мнению автора этой главы, качество документации в Google намного выше, чем в большинстве софтверных компаний.

Чтобы изменить качество технической документации, инженеры и организация в целом должны осознать, что они одновременно являются и проблемой, и решением. Вместо того чтобы закрывать глаза на состояние документации, они должны понять, что создание качественной документации является частью их работы и экономит их время и силы в долгосрочной перспективе. Документирование любого фрагмента кода, который просуществует больше нескольких месяцев, поможет его поддерживать не только другим, но и вам самим.

Итоги

- Документация приносит пользу организации не только в будущем, но и сейчас.
- Работа над документацией должна быть включена в рабочий процесс разработчика.
- Каждый документ должен иметь одно назначение.
- Пишите для других, а не для себя.

Основы тестирования

*Автор: Адам Бендер
Редактор: Том Манирек*

Тестирование всегда было неотъемлемой частью программирования. Написав свою первую программу, вы наверняка опробовали ее с несколькими примерами данных, чтобы убедиться, что она работает так, как ожидалось. Долгое время тестирование ПО напоминало этот процесс, проводилось вручную и было подвержено ошибкам. Однако с начала 2000-х годов подход к тестированию в индустрии ПО претерпел значительные изменения, как того требовал рост размеров и сложности программных систем. Центральное место в этих изменениях заняла автоматизация тестирования.

Автоматическое тестирование способно предотвратить проникновение ошибок в дикую природу кода и их влияние на пользователей. Чем позже в цикле разработки обнаруживается ошибка, тем дороже обходятся ее последствия, причем во многих случаях эта дороговизна растет экспоненциально¹. Однако «ловля жуков» является лишь одним из мотивов для тестирования ПО. Независимо от того, добавляете вы новые функции, выполняете рефакторинг для поддержки кода в здоровом состоянии или проводите переоценку ПО, автоматизированное тестирование поможет быстро выявить ошибки и тем самым позволит с большей уверенностью вносить изменения в ПО.

Компании вынуждены быстро адаптироваться к изменениям в технологиях, условиях рынка и вкусах клиентов. Использование надежной практики тестирования избавит вас от страхов перед изменениями и станет для вас неотъемлемой частью процесса разработки ПО. Чем чаще и быстрее вам потребуется менять свои системы, тем выше будет ваша потребность иметь быстрый способ тестирования изменений.

Сам акт написания тестов способствует улучшению дизайна систем. Как первый клиент кода тест может многое рассказать о его дизайне. Не слишком ли тесно система связана с базой данных? Поддерживает ли API требуемые варианты использования? Обрабатывает ли система все крайние случаи? Написание автоматических тестов помогает ответить на эти вопросы на ранних этапах разработки и делает ПО более модульным и гибким.

¹ См. статью «Defect Prevention: Reducing Costs and Enhancing Quality» (<https://oreil.ly/27R87>).

Много чернил было израсходовано на описание тестирования ПО, и это вполне объяснимо: несмотря на всю свою важность, практика тестирования до сих пор многим кажется загадочной. Мы в Google прошли долгий путь, но все еще сталкиваемся с трудностями, препятствующими распространению наших процессов по всей компании. В этой главе мы поделимся с вами своими знаниями, чтобы вы смогли развить наши идеи.

Почему мы пишем тесты?

Чтобы лучше понять, как получить максимальную отдачу от тестирования, начнем с самого начала. Что мы в действительности подразумеваем, говоря об автоматизированном тестировании?

Самый простой тест определяется:

- единственным тестируемым поведением, обычно методом или API;
- конкретными входными данными — значением, которое передается в API;
- наблюдаемым результатом или поведением;
- управляемой средой, такой как отдельный изолированный процесс.

Выполняя тест, передав входные данные в систему и проверив результат, можно узнать, действует ли система так, как ожидается. Совокупность из сотен или тысяч простых тестов (обычно называемая *набором тестов*) может помочь выявить несоответствия кода предполагаемому дизайну.

Создание и поддержка надежного набора тестов требуют больших усилий. Вместе с ростом кодовой базы растет и набор тестов, что вызывает проблемы в их стабильности и скорости. Игнорирование этих проблем отрицательно сказывается на наборе тестов. Имейте в виду, что главная ценность тестов заключается в доверии к ним со стороны инженеров. Если продуктивность инженеров снижается из-за постоянной поломки тестов и появления неопределенности, инженеры потеряют доверие к тестам и начнут пытаться их обойти. Плохой набор тестов воспринимается хуже, чем полное их отсутствие.

Помимо возможности быстро создавать качественные продукты тестирование обеспечивает безопасность продуктов и услуг. Постепенно ПО все глубже проникает в нашу жизнь, и дефекты в нем могут вызывать не только раздражение: они могут приводить к огромным убыткам, потере имущества или, что хуже всего, к гибели людей¹.

Мы в Google убедились, что тестирование — это далеко не второстепенный вопрос. Внимательное отношение к качеству ПО и тестированию является частью нашей работы. На своем опыте, иногда горьком, мы узнали, что неспособность обеспечить высокое качество продуктов и услуг неизбежно приводит к плохим результатам. Поэтому мы включили тестирование в основу нашей инженерной культуры.

¹ См. статью «Failure at Dhahran» (<https://oreil.ly/lh07Z>).

История Google Web Server

В первые дни существования Google многие инженеры считали тестирование чем-то малозначительным. Команды полагались на умение людей писать ПО без ошибок. Некоторые системы подвергались масштабному интеграционному тестированию, но в основном они напоминали Дикий Запад. Больше всего пострадал один из наших продуктов Google Web Server, также известный как GWS.

GWS — это веб-сервер, отвечающий за обслуживание запросов к Google Search и управление воздушным движением в аэропортах. Еще в 2005 году, когда размер и сложность проекта увеличились, скорость его разработки резко упала. Реализации становились все более громоздкими, а периоды между ними становились все больше. Члены команды испытывали неуверенность при внесении изменений в службу и часто обнаруживали, что что-то не так, только когда какие-то функции переставали работать в продакшене. (В какой-то момент более 80 % изменений в продакшене содержали ошибки, влияющие на пользователя, и их требовалось откатить.)

Для решения этих проблем технический лидер проекта GWS решил использовать автоматизированное тестирование. Согласно этому решению все новые изменения должны были включать тесты, выполняемые непрерывно. Спустя год число чрезвычайных происшествий в проекте *сократилось наполовину*, несмотря на то что в каждом квартале вносилось рекордное число изменений. Даже в условиях беспрецедентного роста числа изменений тестирование способствовало увеличению скорости разработки одного из самых важных проектов в Google. В настоящее время в GWS выполняются десятки тысяч тестов и его новые реализации выходят почти каждый день, но при этом число сбоев, видимых клиенту, остается незначительным.

Изменения в политике разработки GWS ознаменовали перелом в культуре тестирования в Google, так как команды в других подразделениях компании увидели преимущества тестирования и перешли на аналогичную тактику.

Самый важный урок, который мы вынесли из опыта разработки GWS: нельзя избегать дефектов в программном продукте, полагаясь только на способности программистов. Даже если каждый инженер допустит по случайности только одну ошибку, команду таких инженеров захлестнет постоянно растущий список дефектов. Представьте гипотетическую команду из 100 хороших инженеров, каждый из которых допускает только одну ошибку в месяц. В совокупности эта группа выдающихся программистов будет добавлять пять новых ошибок каждый рабочий день. Что еще хуже, исправляя одну ошибку в сложной системе, инженеры могут по неосторожности внести другую ошибку.

Лучшие команды ищут способы использовать коллективную мудрость. Автоматизированное тестирование — один из таких способов. Когда инженер в команде напишет тест, он добавляет его в пул общих ресурсов, после чего другие члены команды могут применить этот тест и получить выгоду от обнаружения проблемы. Сравните этот подход с подходом, основанным на отладке, при котором, обнаружив ошибку, инженер должен потратить время, чтобы найти причину этой ошибки с помощью

отладчика и устранить ее. Затраты времени инженера в этих двух подходах просто несопоставимы, и именно это позволило проекту GWS переломить ситуацию.

Тестирование со скоростью современной разработки

Программные системы становятся все больше и сложнее. Типичное приложение или служба в Google состоит из тысяч или миллионов строк кода, использует сотни библиотек или фреймворков и доставляется через ненадежные сети на все большее число платформ с разными настройками. Что еще хуже, новые версии приложения рассылаются пользователям порой по нескольку раз в день. Наша современность сильно отличается от эпохи коробочных программных продуктов, которые обновлялись пару раз в год.

Механизмы ручного тестирования поведения системы не отвечают стремительному росту числа функций и поддерживаемых платформ в большинстве программных продуктов. Представьте, что вам нужно вручную проверить все функциональные возможности Google Search, такие как поиск авиабилетов, киносеансов, релевантных изображений и, конечно же, веб-поиск (рис. 11.1). Даже если вы найдете решение этой задачи, умножьте получившуюся нагрузку на количество языков, стран и устройств, которые должны поддерживаться в Google Search, и не забудьте учесть такие аспекты, как доступность продукта для людей с ограниченными возможностями и безопасность. Подходы, основанные на оценке качества продукта путем ручного взаимодействия с каждой функцией, просто не масштабируются. Поэтому, когда дело доходит до тестирования, есть только одно решение: автоматизация.

Пиши, запускай, исправляй

В чистом виде автоматическое тестирование состоит из трех этапов: написание тестов, их запуск и исправление выявленных ошибок. Автотест — это небольшой фрагмент кода, обычно отдельная функция или метод, вызывающий изолированную часть более крупной системы, которую требуется протестировать. Тест настраивает среду для тестирования, вызывает систему, обычно с известными входными данными, и проверяет результат. Некоторые тесты очень маленькие и имеют единственный путь выполнения; другие — намного больше и могут включать в себя целые системы, такие как мобильная ОС или веб-браузер.

В примере 11.1 представлен простой тест на Java, не использующий специализированных фреймворков или библиотек. Он не является примером целого набора тестов, но каждый автоматизированный тест по своей сути очень похож на этот простой пример.

В отличие от старых процессов контроля качества, при которых целые команды специально нанятых тестировщиков исследовали новые версии программных систем, современные инженеры играют самую активную и непосредственную роль в написании автоматических тестов для своего кода и их выполнении. Даже в компаниях, где подразделения контроля качества занимают важное место, тесты, написанные разработчиками, являются обычным явлением. При скорости и охвате современной

разработки единственный способ не отстать — организовать написание тестов всеми инженерами.

The image contains two screenshots of Google search results. The top screenshot shows a search for "sfo to london". The results include a sponsored section for "Flights from San Francisco, CA (SFO) to London, United Kingdom (all airports)" with a table of flight options. Below this is a section for "Cheap Flights from San Francisco to London from \$347 - KAYAK" with a list of flight details and a "Show more" link.

The bottom screenshot shows a search for "spaceballs showtimes". The results include a section for "Spaceballs / On TV soon" with a table of showtimes. Below this is a section for "Spaceballs | Fandango" with a rating of 83% and a "Play trailer on YouTube" button. The bottom right of the screenshot shows a movie card for "Spaceballs" with a rating of 83% on Fandango, 7.1/10 on IMDb, and 46% on Metacritic. A description of the movie is also visible.

Airline	Duration	Type	Price
Multiple airlines	10h 10m+	Connecting	from \$353
British Airways	12h 35m+	Connecting	from \$365
Multiple airlines	10h 15m	Nonstop	from \$422
United	10h 30m	Nonstop	from \$432
Delta	10h 15m	Nonstop	from \$628
Virgin Atlantic	10h 15m	Nonstop	from \$628
Air France	10h 15m	Nonstop	from \$629
KLM	10h 15m	Nonstop	from \$629
Lufthansa	10h 30m	Nonstop	from \$692
Austrian	10h 30m	Nonstop	from \$692
Brussels Airlines	10h 30m	Nonstop	from \$692
Norwegian Air UK	10h 10m	Nonstop	from \$710
American	10h 25m	Nonstop	from \$939
British Airways	10h 25m	Nonstop	from \$939
Iberia	10h 25m	Nonstop	from \$939
Other airlines	10h 15m+	Connecting	from \$415

Date	Channel
Today, 9:25 AM	Starz Comedy (West)
Today, 5 PM	Starz Comedy (East)
Today, 8 PM	Starz Comedy (West)
Tomorrow, 10:30 PM	Encore (East)
Fri, 1/31, 1:30 AM	Encore (West)

Rating	IMDb	Metacritic
83%	7.1/10	46%

Рис. 11.1. Скриншоты с комплексными результатами в Google Search

Пример 11.1. Пример теста

```
// Проверяет обработку отрицательных результатов классом Calculator
public void main(String[] args) {
    Calculator calculator = new Calculator();
    int expectedResult = -3;
    int actualResult = calculator.subtract(2, 5); // Из 2 вычесть 5
    assert(expectedResult == actualResult);
}
```

Конечно, написание тестов отличается от написания *хороших тестов*. Обучить десятки тысяч инженеров писать хорошие тесты — довольно сложная задача, и в следующих главах мы расскажем все, что нам удалось узнать о написании хороших тестов.

Написание тестов — это только первый шаг в процессе автоматизированного тестирования. После того как тесты будут написаны, их нужно запускать, причем часто. Фактически автоматизированное тестирование заключается в многократном повторении одного и того же действия и требует внимания человека только при поломке. Мы обсудим непрерывную интеграцию и тестирование в главе 23. Выражая тесты в виде кода, а не последовательности шагов, выполняемых вручную, мы получаем возможность запускать их после каждого изменения кода хоть тысячи раз в день. В отличие от людей-тестировщиков машины никогда не устают и не отвлекаются.

Еще одно преимущество использования тестов в виде кода — их легко адаптировать к различным средам. Тестирование поведения Gmail в Firefox требует не больше усилий, чем его тестирование в Chrome, при наличии в тесте конфигурации для обеих этих систем¹. Код, тестирующий пользовательский интерфейс на английском языке, с не меньшим успехом может тестировать интерфейсы на японском или немецком.

Тестирование продуктов и служб, находящихся в процессе активной разработки, неизбежно будет сталкиваться с проблемами. Эффективность процесса тестирования во многом будет зависеть от реакции инженеров на появление этих проблем. Игнорирование неудачных тестов быстро лишит их любой ценности, которую они давали, поэтому крайне важно отнестись к ним серьезно. Команды, уделяющие приоритетное внимание исправлению сломанного теста в течение нескольких минут после обнаружения сбоя, способны поддерживать высокую степень уверенности в тесте, быстро выяснять причины ошибок и, следовательно, получать большую выгоду от тестов.

Таким образом, здоровая культура автоматизированного тестирования поощряет участие всего коллектива в написании тестов и гарантирует их регулярное выполнение. Что еще важнее, она подразумевает быстрое исправление сломанных тестов, чтобы поддерживать высокую степень доверия к тестированию.

¹ Правильное поведение в разных браузерах и в пользовательских интерфейсах на разных языках — это отдельная история! Но в идеале все конечные пользователи продукта должны получить одинаковый опыт.

Преимущества тестирования кода

Разработчикам из организаций, в которых нет устойчивой культуры тестирования, рассуждения о написании тестов как о средстве увеличения продуктивности и скорости разработки могут показаться странными. В конце концов, написание тестов может уйти столько же времени (и даже больше!), что и на реализацию тестируемой функции. Однако это не так, мы в Google обнаружили, что затраты на тестирование дают ряд важнейших преимуществ, сказывающихся на продуктивности разработчиков:

Уменьшение числа отладок

Тестирование уменьшает число дефектов в коде, причем не только перед его отправкой в репозиторий. Код в Google может меняться десятки раз другими командами и автоматизированными системами сопровождения. Тест, написанный один раз, предотвращает появление дорогостоящих дефектов и избавляет инженеров от раздражающих сеансов отладки в течение всего жизненного цикла проекта. Изменения в проекте или в его зависимостях, вызывающие ошибку при выполнении тестов, будут быстро обнаружены инфраструктурой тестирования и отменены до того, как ошибка попадет в продакшен.

Уверенность в изменениях

Любое ПО рано или поздно меняется. Команды с надежными тестами могут с уверенностью вносить изменения в свой проект, потому что все важные функции в их проектах постоянно проверяются. В тестируемых проектах проще проводить рефакторинг, потому что измененный код, сохранивший существующее поведение (в идеале), не должен требовать изменений в существующих тестах.

Документирование

Документация по ПО заведомо ненадежна. Она может быть слабо связанной с кодом, устаревшей или неполной. Четкие целенаправленные тесты, каждый из которых проверяет что-то одно, действуют подобно документам: чтобы узнать, что делает код в конкретном случае, загляните в тест, проверяющий этот случай. Когда требования меняются и новый код вызывает ошибку при выполнении существующего теста — это сигнал о том, что «документация» устарела. Обратите внимание, что тесты действуют как документы, только если сохранены их четкость и краткость.

Простота обзора

Весь код в Google просматривается, по крайней мере, еще одним инженером перед отправкой в репозиторий (подробнее об обзорах кода в главе 9). Рецензент тратит меньше усилий на проверку работоспособности кода, если код сопровождается тщательно сконструированными тестами, которые проверяют правильность кода, обработку худших случаев и ошибки. Вместо кропотливого и утомительного исследования каждого возможного случая рецензент может просто посмотреть, имеются ли тесты для всех случаев и выполняются ли они успешно.

Продуманность дизайна

Написание тестов для нового кода помогает проверить практичность API, реализуемого этим кодом. Если новый код сложен в тестировании, часто это обусловлено присутствием в коде слишком широких обязанностей или трудноуправляемых зависимостей. Хорошо продуманный код должен быть модульным, не образовывать тесных связей и фокусироваться на конкретных обязанностях. Устранение проблем с дизайном на ранних этапах часто избавляет от лишней работы в последующем.

Частота выхода и высокое качество новых версий

С помощью наборов автоматизированных тестов команды могут уверенно выпускать новые версии своих приложений. Даже крупные проекты в Google с сотнями инженеров выпускают новые версии продуктов каждый день. Это было бы невозможно без автоматического тестирования.

Проектирование набора тестов

Основы нашего подхода к тестированию были заложены давно. С ростом нашей кодовой базы методом проб и ошибок мы многое узнали о проектировании и выполнении наборов тестов.

Уже на самых ранних этапах внедрения тестирования мы обнаружили, что инженеры предпочитали писать тесты, охватывающие всю систему, но эти тесты работали медленно, были ненадежными и сложными в отладке. И инженеры задались вопросами: «Почему мы не тестируем серверы по одному?» и «Зачем тестировать сервер целиком, когда проще тестировать его небольшие модули по отдельности?» В конце концов, стремление избавиться от сложностей привело к тому, что команды стали разрабатывать все меньшие и меньшие тесты, которые работали быстрее и стабильнее и были проще в отладке.

Это вызвало многочисленные споры о значении слова «маленький». Можно ли считать юнит-тест маленьким? А как определить размер интеграционного теста? Мы пришли к выводу, что каждый тест имеет два разных измерения: размер и охват. Размер определяется объемом ресурсов, необходимых для запуска теста, таких как память, процессы и время. Под охватом понимаются конкретные пути в коде, проверяемые тестом. Обратите внимание, что выполнение строки кода не является проверкой правильности его поведения. Размер и охват — взаимосвязанные, но все же разные понятия.

Размер теста

Мы в Google классифицируем тесты по размеру и призываем инженеров всегда писать минимально возможные тесты для проверки некой единицы функциональности. Размер теста определяется не количеством строк кода, а тем, как он выполняется, что ему разрешено делать и сколько ресурсов он потребляет. Фактически наше деле-

ние на маленькие, средние и большие тесты определяется ограничениями, которые инфраструктура тестирования может наложить на тест. Подробнее мы рассмотрим эти ограничения ниже, а пока отметим, что *маленькие тесты* выполняются в одном процессе, *средние тесты* — на одной машине, а *большие тесты* могут охватывать столько машин и процессов, сколько понадобится (рис. 11.2)¹.

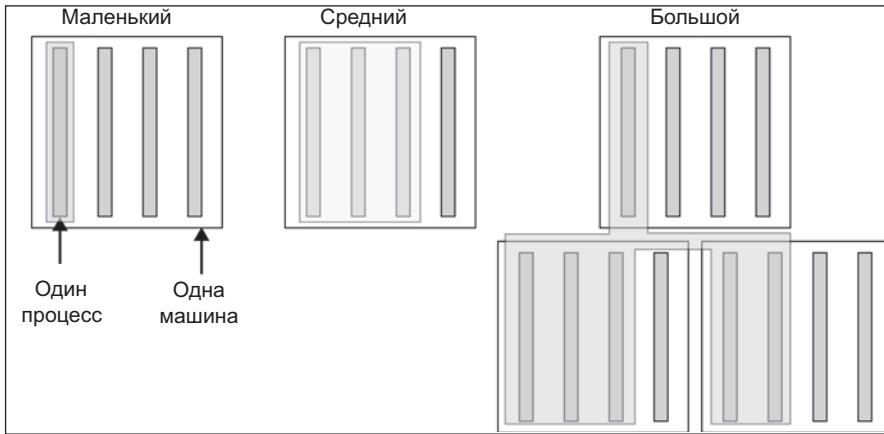


Рис. 11.2. Размеры тестов

Такая классификация, в отличие от более традиционного деления на «юнит-тесты» или «интеграционные тесты», была выбрана, потому что для нас важнее такие качества теста, как скорость выполнения и детерминированность независимо от широты охвата тестирования. Маленькие тесты с любой широтой охвата за счет ограничений, наложенных на них, почти всегда быстрее и более детерминированы, чем тесты, вовлекающие в работу обширную инфраструктуру или потребляющие больше ресурсов. С увеличением размеров тестов многие ограничения ослабляются. Средние тесты обладают большей гибкостью, но они недостаточно детерминированы. Большие тесты сохраняются только для самых сложных и комплексных сценариев тестирования. Давайте подробнее рассмотрим, какие ограничения накладываются на каждый тип тестов.

Маленькие тесты

Маленькие тесты имеют больше всего ограничений. Основное ограничение — они должны выполняться в том же процессе (а во многих языках — в том же потоке), что и тестируемый код. Это значит, что нельзя запустить сервер и подключить к нему

¹ Строго говоря, мы в Google различаем четыре размера тестов: маленький, средний, большой и огромный. Разница между большими и огромными тестами на самом деле очень тонкая и корнями уходит глубоко в историю. Понятие «большой» в этой книге в основном соответствует нашему понятию «огромный».

отдельный процесс с маленьким тестом, а в ходе выполнения маленького теста нельзя запустить стороннюю программу, такую как база данных.

Также при выполнении маленьких тестов запрещены остановки, операции ввода/вывода¹ и любые другие блокирующие вызовы. То есть маленькие тесты не должны обращаться к сети или диску. Для тестирования кода, полагающегося на эти запрещенные операции, необходимо использовать тестовые дублиеры (глава 13), в которых обременительные зависимости замещены внутрипроцессными реализациями.

Цель этих ограничений — отобрать у маленьких тестов доступ к источникам медлительности и недетерминированности. Тест, действующий в единственном процессе и не использующий блокирующие вызовы, сможет эффективно выполняться с максимальной скоростью, на какую только способен процессор. Случайно сделать такой тест медленным или недетерминированным трудно (хотя возможно). Ограничения, накладываемые на маленькие тесты, формируют безопасную среду, которая не позволяет инженерам выстрелить себе в ногу.

Эти ограничения могут показаться чрезмерными, но представьте довольно скромный набор из пары сотен маленьких тестов, многократно запускаемый в течение дня. Если некоторые из этих тестов будут терпеть неудачу случайным (недетерминированным) образом (часто такие тесты называют нестабильными (*flaky tests*), <https://oreil.ly/NxC4A>), на выявление причин неудач потребуется слишком много времени, что отрицательно скажется на продуктивности инженера. В масштабах Google такая проблема может привести к остановке всей инфраструктуры тестирования.

Мы в Google призываем инженеров писать маленькие тесты, когда это возможно, независимо от широты охвата тестирования, потому что такой подход обеспечивает быстрый и надежный запуск всего набора тестов. Подробнее о маленьких тестах и юнит-тестах — в главе 12.

Средние тесты

Ограничения, накладываемые на маленькие тесты, могут оказаться слишком жесткими для многих интересных видов тестирования. Следующая ступенька вверх по лестнице размеров — средние тесты. Средние тесты могут действовать в нескольких процессах, использовать механизмы многопоточного выполнения и производить блокирующие вызовы, включая сетевые вызовы к `localhost`. Единственное ограничение — средним тестам не разрешается посылать сетевые вызовы к любым другим системам, кроме `localhost`. То есть средний тест не должен пересекать границ одной машины.

Разрешение использовать несколько процессов открывает массу новых возможностей для тестирования. Например, тест может запустить экземпляр базы данных, чтобы убедиться, что тестируемый код правильно интегрируется в более реалистичной среде, или проверить взаимодействие веб-интерфейса с серверным кодом. Тесты

¹ Но есть определенное пространство для маневра. Тестам разрешается обращаться к файловой системе при условии использования герметичной реализации в памяти.

для веб-приложений часто используют такие инструменты, как WebDriver (<https://oreil.ly/W27Uf>), которые запускают настоящий браузер и управляют им удаленно из процесса, в котором действует тест.

К сожалению, с увеличением гибкости теста снижаются его скорость выполнения и детерминированность. Тесты, действующие в нескольких процессах или выполняющие блокирующие вызовы, зависят от ОС и других процессов, которые должны работать быстро и детерминированно, чего мы не можем гарантировать. Средние тесты сохраняют некоторую защищенность, запрещая доступ к удаленным машинам по сети, который делает тесты медлительными и недетерминированными. Но «безопасность» в средних тестах почти отсутствует, поэтому инженеры должны работать с ними осторожно.

ПРИМЕР: НЕСТАБИЛЬНЫЕ ТЕСТЫ ОБХОДЯТСЯ ДОРОГО

При наличии нескольких тысяч тестов, многократно выполняющихся в течение дня, каждый из которых может изредка проявлять нестабильность в работе, какой-то из них почти наверняка будет терпеть неудачу (демонстрировать нестабильное поведение). С увеличением количества запусков тестов статистически будет увеличиваться и количество неудачных тестовых прогонов. Если вероятность случайной неудачи для каждого такого теста будет составлять всего 0,1 % и в течение дня выполняется 10 000 тестов, вам придется расследовать 10 случаев отказа в день. Каждое расследование отнимает время, которое можно было бы потратить с большей пользой.

В некоторых случаях можно ограничить влияние нестабильных тестов, автоматически перезапуская их в случае неудачи. Это позволяет обменять время инженера на такты процессора. Такой компромисс имеет смысл при низкой вероятности нестабильного поведения. Но имейте в виду, что повторный запуск тестов откладывает на потом устранение причины нестабильности.

Если нестабильность тестов продолжит увеличиваться, вы столкнетесь с чем-то намного худшим, чем потеря продуктивности инженера, — с потерей уверенности в тестах. Нестабильность не успеет вырасти значительно, прежде чем команда потеряет доверие к набору тестов. И когда это произойдет, инженеры перестанут реагировать на возникающие неудачи, что полностью сведет на нет все выгоды, которые дает набор тестов. Наш опыт показывает, что при приближении к уровню нестабильности в 1 % тесты начинают терять свою ценность. У нас в Google вероятность нестабильности составляет около 0,15 %, что означает тысячи ложных ошибок каждый день. Мы упорно стараемся держать нестабильность под контролем, в том числе выделяя инженерам время для устранения ее причин.

Часто случайные неудачи тестовых прогонов объясняются недетерминированным поведением самих тестов. В ПО есть множество источников недетерминированности: время такта, особенности планирования потоков выполнения, задержки в сети и т. д. Научиться изолировать и стабилизировать влияние случайных эффектов нелегко. Иногда они обусловлены низкоуровневыми проблемами, такими как аппаратные прерывания или особенности работы механизма отображения в браузере. Хорошая инфраструктура автоматизированного тестирования должна помогать инженерам выявлять недетерминированное поведение и ослаблять его влияние на результаты тестирования.

Большие тесты

Наконец, у нас есть большие тесты, которые взаимодействуют не только с `localhost`, но и с другими машинами. Например, такой тест может проверять работу системы, действующей в удаленном кластере.

Как и прежде, с увеличением гибкости теста растут риски. Необходимость взаимодействовать с системой, охватывающей несколько машин, соединенных сетью, значительно увеличивает вероятность медленного и недетерминированного выполнения теста по сравнению с ситуацией, когда тестирование выполняется на одной машине. Мы используем большие тесты в основном для сквозного и комплексного тестирования, в ходе которого чаще проверяются различные конфигурации, а не фрагменты кода, а также для тестирования устаревших компонентов, для которых невозможно использовать дублеров. (Подробнее о больших тестах в главе 14.) Команды в Google часто изолируют большие тесты от маленьких или средних тестов и выполняют их только во время сборки и перед выпуском, чтобы не оказывать влияния на рабочий процесс разработки.

Общие свойства тестов всех размеров

Все тесты должны быть герметичными: тест должен иметь всю необходимую информацию для создания, настройки и удаления его внутренней среды. Тесты должны делать как можно меньше предположений о своей среде, в частности о порядке, в котором они выполняются. Например, они не должны полагаться на общую базу данных. Соблюдить это ограничение все сложнее с увеличением размеров тестов, но необходимо приложить усилия, чтобы обеспечить хороший уровень изоляции теста.

Тест должен содержать *только* ту информацию, которая необходима для проверки некоего поведения. Ясность и простота тестов помогают рецензентам убедиться, что код делает именно то, что должен. Кроме того, чистый код помогает диагностировать обнаруженные ошибки. Мы любим говорить, что «тест должен быть очевиден для исследователя». Поскольку для самих тестов нет тестов, их правильность приходится проверять вручную. Мы настоятельно рекомендуем не использовать в тестах операторы управления потоком, такие как условные выражения и циклы (<https://oreil.ly/fQSuk>), потому что они подвержены ошибкам и затрудняют определение причины неудачного выполнения теста.

Помните, что тесты обычно пересматриваются, только когда что-то ломается. Когда вам предложат исправить сломанный тест, который вы никогда не видели раньше, вы будете благодарны тому, кто в свое время постарался сделать его максимально простым. Код читается гораздо чаще, чем пишется, поэтому пишите тесты так, чтобы их легко было читать!

Выбор размера теста на практике. Точно определив размеры тестов, мы смогли создать инструменты, обеспечивающие соблюдение этих размеров, что, в свою очередь, помогло нам масштабировать наборы тестов с сохранением гарантий в отношении скорости, использования ресурсов и стабильности. Применение классификации

тестов по размеру в Google зависит от языка. Например, все тесты Java мы запускаем с помощью специального диспетчера безопасности, который завершит неудачей любой тест, отмеченный как маленький, если тот попытается сделать что-то запрещенное, например установить сетевое соединение.

Широта охвата тестирования

Мы в Google уделяем большое внимание размерам тестов, однако еще одним их важным свойством, которое необходимо учитывать, является широта охвата тестирования — объем кода, проверяемого данным тестом. Тесты с узким охватом (их обычно называют *юнит-тестами*) предназначены для проверки логики в узкой области кодовой базы, такой как отдельный класс или метод. Тесты со средней шириной охвата (их также часто называют *интеграционными тестами*) предназначены для проверки взаимодействий между небольшим количеством компонентов, например между сервером и его базой данных. Тесты с широким охватом (их нередко называют *функциональными, сквозными* или *общесистемными тестами*) предназначены для проверки взаимодействий между несколькими отдельными частями системы или особенностей поведения кода, которые выражаются несколькими классами или методами.

Важно отметить, что когда мы говорим об узком охвате юнит-тестов, мы имеем в виду *проверяемый* код, а не код, который *выполняется*. Часто классы имеют множество зависимостей от других классов, и эти зависимости, естественно, будут вызываться при тестировании целевого класса. Некоторые политики тестирования (<https://oreil.ly/Lj-t3>) используют дублеров (имитации или фиктивные объекты), чтобы исключить возможность выполнения кода вне тестируемой системы (SUT, system under test), однако мы в Google предпочитаем по возможности сохранять для тестирования действительные зависимости (глава 13).

Узкий охват, как правило, имеют маленькие тесты, а широкий — средние или большие, но это не всегда так. Например, можно написать тест конечной точки сервера, охватывающий обширный код, который реализует синтаксический анализ запросов, их семантическую проверку и бизнес-логику, и этот тест будет считаться маленьким, потому что он использует дублеров для замены всех внепроцессных зависимостей, таких как база данных или файловая система. Точно так же можно написать тест среднего размера, охватывающий единственный метод. Например, при условии, что веб-фреймворк связывает вместе HTML и JavaScript, тестирование компонента пользовательского интерфейса, такого как элемент выбора даты, потребует запуска браузера, чтобы проверить единственный путь в коде.

Мы в Google призываем инженеров писать не только маленькие тесты, но и тесты с как можно более узким охватом. В целом мы стремимся к тому, чтобы примерно 80 % наших тестов составляли юнит-тесты с узким охватом, подтверждающие правильную работу основной бизнес-логики; 15 % — интеграционные тесты со средним охватом, проверяющие взаимодействия между двумя или более компонентами;

и 5 % — сквозные тесты, проверяющие систему целиком. Эти ориентиры показаны на рис. 11.3 в виде пирамиды.



Рис. 11.3. Версия пирамиды тестирования Майка Кона¹; доли тестов в процентах и их состав в разных командах будут немного отличаться

Юнит-тесты формируют основу тестирования, потому что они быстрые, стабильные, сужают охват и снижают когнитивную нагрузку, необходимую для выявления всех возможных поведений класса или функции. Кроме того, они позволяют быстро и безболезненно диагностировать неисправности. Следует помнить о двух антипаттернах — «перевернутая пирамида», или «рожок мороженого» (ice cream cone), и «песочные часы» (рис. 11.4).

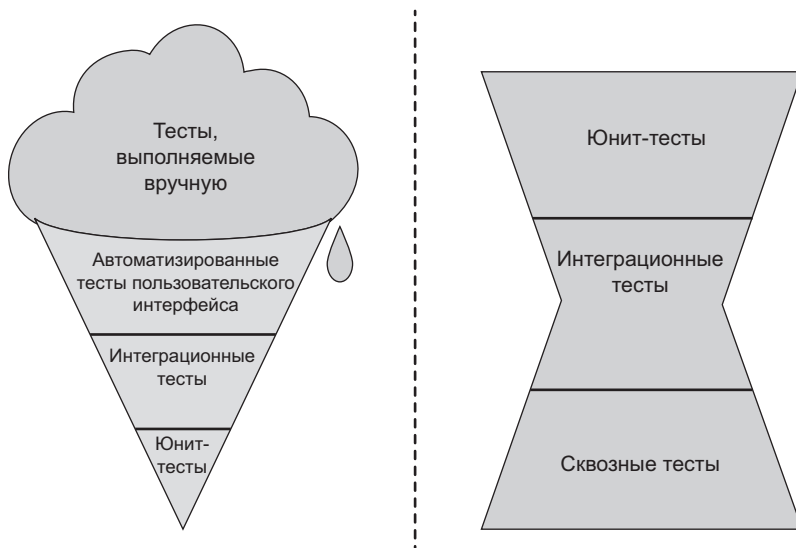


Рис. 11.4. Антипаттерны в наборах тестов

¹ Кон М. Scrum. Гибкая разработка ПО. М.: Вильямс, 2015. — Примеч. пер.

Антипаттерн «перевернутая пирамида» соответствует ситуации, когда инженеры пишут много сквозных, но мало интеграционных или юнит-тестов. Такие наборы тестов часто оказываются медленными, ненадежными и трудными в отладке. Этот антипаттерн часто встречается в проектах, начинающихся с создания прототипов, которые быстро передаются в продакшен без приостановки для решения проблем тестирования.

Антипаттерн «песочные часы» соответствует ситуации, когда в проекте есть много сквозных и юнит-тестов, но мало интеграционных тестов. Это не так плохо, как «перевернутая пирамида», но все же приводит к большому числу сбоев в сквозных тестах, которые быстрее и проще можно было бы обнаружить с помощью набора тестов со средним охватом. Антипаттерн «песочные часы» возникает, когда тесные связи между компонентами затрудняют создание автономных экземпляров некоторых зависимостей.

Рекомендуемое нами сочетание тестов определяется двумя основными целями: повышение продуктивности инженеров и укрепление доверия к продукту. Предпочтительное отношение к юнит-тестам позволяет получить высокую степень уверенности в тестах с самого начала разработки. Большие тесты проверяют общую работоспособность продукта по мере его разработки, и их не следует рассматривать как основной метод выявления ошибок.

Формируя свой набор тестов, вы можете выбрать другое их соотношение. Сделав упор на интеграционное тестирование, вы обнаружите, что тестовые наборы выполняются дольше, но выявляют больше проблем во взаимодействиях между компонентами. Уделив больше внимания юнит-тестам, вы заметите, что тестирование протекает очень быстро и выявляет больше типичных логических ошибок. Но юнит-тесты не в состоянии проверить правильность взаимодействий между компонентами, например соблюдение контракта между двумя системами, разработанными разными командами (<https://oreil.ly/mALqH>). Хороший набор тестов содержит смесь тестов разных размеров и уровней охвата, которые соответствуют локальным архитектурным и организационным реалиям.

Правило Бейонсе

Наши новые сотрудники, проходя первичное обучение, часто спрашивают, какие черты поведения или свойства действительно необходимо тестировать? Ответ прост: тестируйте все, что вы не хотели бы сломать. Другими словами, если вы хотите, чтобы система демонстрировала определенное поведение, единственный способ гарантировать это — написать автоматический тест, проверяющий его. К свойствам кода, которые важно протестировать, относятся: производительность, корректность поведения, доступность и безопасность, а также менее очевидные свойства, такие как обработка ошибок в системе.

Используйте «правило Бейонсе» (https://oreil.ly/X7_-z). В контексте тестирования его можно сформулировать так: «Если вам что-то понравилось, добавьте для него тест».

Правило Бейонсе часто используется инфраструктурными командами, отвечающими за внесение изменений во всей кодовой базе. Если не связанные между собой изменения в инфраструктуре пройдут все ваши тесты, но вызовут нарушения в работе продукта вашей команды, устраните проблемы и добавьте дополнительные тесты.

ТЕСТИРОВАНИЕ ОТКАЗОВ

Отказ — одна из наиболее важных ситуаций, которую должна учитывать система. Отказы неизбежны, но нет ничего хуже, чем просто ждать реальной катастрофы, чтобы выяснить, насколько хорошо система способна справиться с ней. Вместо ожидания отказа напишите автоматизированные тесты, имитирующие типичные сбои: смоделируйте исключения или ошибки в юнит-тестах, симитируйте отказы или задержки в интерфейсе RPC в интеграционных и сквозных тестах. Также можно смоделировать гораздо более масштабные нарушения, влияющие на реальную производственную сети, с использованием таких методов, как Chaos Engineering (<https://oreil.ly/iOO4F>). Предсказуемый и контролируемый ответ на неблагоприятные условия является отличительной чертой надежной системы.

Еще об охвате тестирования

Охват кода — это мера, определяющая, какие строки кода и какими тестами выполняются. Если у вас есть 100 строк кода и ваши тесты выполняют 90 из них, то вы имеете 90%-ный охват тестирования¹. Охват часто считается стандартной метрикой, описывающей качество набора тестов, что несколько ошибочно. С помощью нескольких тестов можно выполнить много строк кода, но так и не убедиться, что каждая строка делает что-нибудь полезное. (Мы рекомендуем измерять охват только для маленьких тестов, чтобы избежать переоценки этой метрики.)

Еще более коварная проблема с оценкой охвата тестирования состоит в том, что, подобно другим метрикам, она быстро становится самоцелью. Обычно команды устанавливают планку ожидаемого охвата, например 80 %. Сначала это выглядит вполне разумно, ведь никто не откажется иметь такую степень охвата. Но на практике получается, что инженеры относятся к этому значению не как к минимальному, а как к максимальному и создают набор тестов, охватывающий не более 80 % кода. В конце концов, зачем делать больше, чем требует стандарт?

Лучший способ повысить качество набора тестов — подумать о тестируемом поведении. Есть ли у вас уверенность, что все будет работать точно так, как ожидают клиенты? Сможете ли вы заметить кардинальные изменения в ваших зависимостях? Ваши тесты стабильны и надежны? Подобные вопросы помогают глобальнее рассуждать о наборе тестов. Разные продукты и команды имеют свои отличительные особенности: некоторые испытывают трудности при взаимодействии с оборудованием, другие — с массивными наборами данных и т. д. Пытаясь ответить на вопрос: «Достаточно ли у нас тестов?» с помощью единственной метрики, вы рискуете упустить

¹ Есть и другие аналогичные метрики охвата тестирования: число путей, ветвей и т. д.

из виду множество аспектов. Метрика охвата может дать некоторое представление об объеме непроверенного кода, но эта оценка не способна заменить критический взгляд на качество тестирования системы.

Тестирование в масштабе Google

Большая часть рекомендаций по тестированию вполне применена к базам кода практически любого размера. Тем не менее мы хотим потратить еще немного времени, чтобы рассказать, что мы узнали, проводя тестирование в огромном масштабе. Чтобы понять, как осуществляется тестирование в Google, необходимо понимать нашу среду разработки, наиболее важной чертой которой является хранение большей части кода в одном монолитном репозитории (monorepo (<https://oreil.ly/qSihi>)). Почти весь код каждого продукта и службы, над которыми мы работаем, хранится в одном месте. В настоящее время в нашем репозитории хранится более двух миллиардов строк кода.

Объем еженедельных изменений в кодовой базе Google достигает 25 миллионов строк. Примерно половина из них вносится десятками тысяч инженеров, работающих с монолитным репозиторием, а другая половина — автоматизированными системами в форме обновлений конфигураций или крупномасштабных изменений (глава 22). Многие из этих изменений инициируются за пределами конкретного проекта. Мы стараемся не ограничивать возможность повторного использования кода нашими инженерами.

Открытость нашей кодовой базы способствует активному совместному владению: любой инженер чувствует ответственность за кодовую базу и может напрямую исправлять ошибки (конечно, с одобрения), вместо того чтобы жаловаться на них, а также вносить изменения в код, принадлежащий кому-то другому.

Еще одна отличительная черта Google заключается в том, что практически ни одна команда не использует возможность создания ветвей в репозитории. Все изменения фиксируются в главной ветви и сразу доступны всем. Кроме того, сборка любого ПО осуществляется с использованием последних зафиксированных изменений, проверенных нашей инфраструктурой тестирования. Когда производится сборка продукта или службы, почти все необходимые зависимости собираются из исходного кода в главной ветви репозитория. Управление тестированием в таком масштабе происходит с помощью системы непрерывной интеграции, одним из ключевых компонентов которой является автоматизированная платформа тестирования (TAP, test automated platform).



Подробнее о нашей философии TAP и непрерывной интеграции в главе 23.

Инженерная среда в Google имеет очень сложную организацию независимо от того, что рассматривать — ее размер, монолитный репозиторий или количество предла-

гаемых продуктов. Каждую неделю в ней проверяются миллионы строк изменений, выполняются миллиарды тестов, создаются десятки тысяч двоичных файлов и сотни обновленных продуктов — попробуйте вообразить что-то еще более сложное!

Недостатки больших наборов тестов

С ростом кодовой базы вам неизбежно придется вносить изменения в существующий код. И плохо написанные автоматизированные тесты могут усложнить выполнение этой задачи. Хрупкие тесты, переопределяющие ожидаемые результаты или полагающиеся на обширные и сложные шаблоны, могут фактически препятствовать добавлению изменений. Такие тесты могут терпеть неудачу даже при внесении изменений, никак не связанных с тестируемым кодом.

Если вам приходилось обнаруживать десятки неработающих тестов после изменения пары строк в функции, значит, вы уже имели возможность почувствовать сопротивление хрупких тестов. Со временем это сопротивление может заставить команду избегать рефакторинга, необходимого для поддержания кодовой базы в здоровом состоянии. В следующих главах мы рассмотрим политики повышения надежности и качества тестов.

К худшим нарушениям в работе тестов относится неправильное употребление фиктивных объектов. База кода в Google так сильно пострадала от злоупотребления фреймворками для создания фиктивных объектов, что заставила некоторых инженеров объявить: «Больше никаких имитаций!» Но правильное понимание ограничений фиктивных объектов может помочь вам избежать их неправильного использования.



Подробнее о работе с фиктивными объектами в главе 13.

Если хрупкие тесты выполняются некорректно, то крупный набор тестов — медленно. Чем дольше работает набор тестов, тем реже он будет запускаться и тем меньше преимуществ он даст. Мы используем некоторые приемы для ускорения выполнения нашего набора тестов, включая параллельное выполнение и использование более быстрого оборудования. Тем не менее эти уловки нивелируются большим количеством медленных тестов.

Медленная работа тестов может быть обусловлена множеством причин, таких как длительная загрузка фрагментов системы, запуск эмулятора перед выполнением, обработка больших наборов данных или ожидание синхронизации разрозненных систем. Нередко тесты сначала работают достаточно быстро, но потом замедляются по мере роста системы. Например, интеграционный тест, проверяющий одну зависимость за пять секунд, при появлении новых зависимостей от десятков сторонних служб станет выполняться пять минут.

Тесты также могут замедляться из-за ненужных ограничений скорости, устанавливаемых такими функциями, как `sleep()` и `setTimeout()`. Эти функций часто используются в качестве наивных эвристик перед проверкой результата недетерминированного поведения. Ожидание по полсекунды то здесь, то там сначала не кажется опасным, но если в широко используемую утилиту встроено «ожидание с последующей проверкой», то довольно скоро в тестовом прогоне появятся минуты простоя. Лучшим решением этой проблемы является активный опрос для определения изменения состояния с частотой, измеряемой микросекундами. Этот прием можно объединить с тайм-аутом на тот случай, если в ходе тестирования не удастся достичь стабильного состояния.

Недетерминированные и медленные наборы тестов препятствуют повышению продуктивности инженера. Столкнувшись с такими тестами, инженеры Google стали искать способы обойти замедление, но некоторые зашли так далеко, что вообще перестали проводить тестирование перед отправкой изменений. Очевидно, что это рискованная практика и ее следует избегать, но если набор тестов приносит больше вреда, чем пользы, то инженеры найдут способ выполнить свою работу, с тестированием или без него.

Но большой набор тестов может быть очень полезен, если относиться к нему с уважением. Мотивируйте инженеров заботиться о своих тестах, вознаграждайте их за надежные тесты в такой же мере, как за отличную реализацию некоторой особенности. Определите соответствующие нормативы производительности и способствуйте рефакторингу медленных или ненадежных тестов. Относитесь к своим тестам как к продакшену. Когда простые изменения начинают занимать значительное время, приложите все силы, чтобы сделать тесты менее хрупкими.

Уделяйте внимание не только культуре, но и инфраструктуре тестирования. Разрабатывайте инструменты статического анализа, пишите документацию и оказывайте любую другую помощь в повышении качества тестов. Сократите количество фреймворков и инструментов, которые вам нужно поддерживать, чтобы выделить время на усовершенствование тестов¹. Если вы не будете уделять внимания простоте управления тестами, инженеры рано или поздно решат, что нет смысла тратить время и силы на их разработку.

История тестирования в Google

Теперь, когда мы обсудили подходы к тестированию в Google, будет полезно узнать, как мы пришли к ним. Как отмечалось выше, инженеры в Google не всегда осознавали ценность автоматического тестирования. На самом деле до 2005 года тестирование

¹ Для каждого поддерживаемого языка программирования в Google есть один стандартный фреймворк тестирования и одна стандартная библиотека с заглушками и фиктивными объектами. Большинство тестов на всех языках для всей кодовой базы выполняет единая инфраструктура.

было очень редким явлением, а не организованной практикой. В большинстве случаев тестирование выполнялось вручную, если вообще выполнялось. Однако с 2005 по 2006 год произошла революция в тестировании, изменившая наш подход к разработке ПО. Ее последствия продолжают ощущаться в компании и по сей день.

Катализатором послужил опыт проекта GWS, который мы обсудили в начале главы. Он показал, насколько мощным может быть автоматизированное тестирование. После усовершенствования проекта GWS в 2005 году наработанные приемы и практики стали распространяться по всей компании. Инструменты, использовавшиеся в ту пору, были довольно примитивными. Однако нашлись добровольцы, объединившиеся в группу Testing Grouplet, которые активно взялись за развитие этих инструментов.

Внедрить идею автоматизированного тестирования в сознание компании помогли три ключевые инициативы: программы ранней ориентации и сертификации практики тестирования, а также новостная рассылка «Тестирование в туалете». Каждая оказывала свое влияние, и вместе они изменили инженерную культуру Google.

Программа ранней ориентации

Несмотря на то что многие инженеры избегали тестирования, пионеры автоматизированного тестирования в Google понимали, что, учитывая темпы роста компании, число новых инженеров быстро превысит число существующих членов команд. И если им удастся привлечь всех новых сотрудников к тестированию, они сумеют изменить культуру. К счастью, в компании существовал и продолжает существовать важный этап, через который проходят все новые инженерные кадры: ориентация.

Программы ранней ориентации в Google в основном касались таких вопросов, как медицинские льготы и особенности работы Google Search, но начиная с 2005 года в них была включена часовая лекция, рассказывающая о значимости автоматизированного тестирования.¹ В ходе лекции обсуждались различные преимущества тестирования, такие как увеличение продуктивности инженера и качества документации, а также улучшение поддержки рефакторинга. Еще в лекции рассказывалось, как написать хороший тест. Для многих нуглеров того периода эта лекция была первым знакомством с тестированием. Но самое главное, что все идеи представлялись в ней так, будто они были стандартной практикой, принятой в компании. Новые сотрудники не знали, что их используют в роли троянских коней с целью внедрить эту идею в ничего не подозревающие команды.

Попадая в свои команды после ориентации, нуглеры начинали писать тесты и задавать вопросы тем, кто тесты не писал. Уже через год-два численность инженеров, обученных тестированию, превзошла общую численность инженеров дотестовой эпохи и многие новые проекты получили правильный старт.

¹ Эта лекция оказалась настолько успешной, что обновленная ее версия проводится до сих пор. На самом деле это одна из самых долгоживущих лекций в программе ориентации.

В настоящее время тестирование широко распространено в отрасли, поэтому большинство новых сотрудников приходят в компанию, уже ожидая встретиться с устойчивой практикой автоматизированного тестирования. В нынешней программе ориентации проводится лекция, укрепляющая ожидания в отношении тестирования и связывающая знания нуглеров о тестировании, полученные за пределами Google, с проблемами, обусловленными большим размером и высокой сложностью нашей кодовой базы.

Программа сертификации практики тестирования

Первоначально большие и наиболее сложные разделы нашей кодовой базы оказались малопригодными для применения передовых практик тестирования. Некоторые проекты имели настолько низкое качество кода, что протестировать их было практически невозможно. Чтобы ясно обозначить цель для проектов, группа Testing Grouplet разработала программу сертификации практики тестирования, которую они назвали «Test Certified». Цель этой программы — дать командам возможность осознать уровень зрелости своих процессов тестирования и, что особенно важно, предоставить рецепты по их улучшению.

Программа была разбита на пять уровней, каждый из которых требовал от команды выполнить определенные действия по совершенствованию практики тестирования. Уровни были определены так, чтобы каждый шаг можно было выполнить в течение квартала, что хорошо укладывалось во внутреннюю систему планирования в Google.

Первый уровень программы охватывал основы: организацию непрерывной сборки, внедрение оценки охвата тестирования, классификацию всех тестов на маленькие, средние или большие, выявление (без обязательного исправления) нестабильных тестов и создание набора быстрых (необязательно всеобъемлющих) тестов, которые можно часто запускать. На каждом последующем уровне добавлялись новые задачи, такие как «отказ от выпуска новых версий при наличии сломанных тестов» или «удаление всех недетерминированных тестов». Пятый уровень требовал автоматизации всех тестов, выполнения всех быстрых тестов перед каждой фиксацией кода в репозитории, отсутствия недетерминированных тестов и охвата тестами каждого поведения. Внутренняя информационная панель, показывающая уровень каждой команды, оказывала сильное социальное давление. Вскоре команды начали соревноваться друг с другом, стремясь подняться на более высокий уровень.

К тому времени, когда программу «Test Certified» заменил автоматизированный подход, внедренный в 2015 году (подробнее об инструменте рН рассказывается ниже), она помогла повысить культуру тестирования более чем в полутора тысячах проектов.

Рассылка «Тестирование в туалете»

Из всех методов, использованных группой Testing Grouplet для совершенствования практики тестирования в Google, наиболее оригинальным оказалась рассылка

«Тестирование в туалете» (TotT). Рассылка TotT имела простую цель: активно пропагандировать тестирование во всей компании. Вопрос в том, как лучше всего организовать пропаганду в компании, сотрудники которой разбросаны по всему миру.

Группа Testing Grouplet рассматривала идею регулярной рассылки по электронной почте, но, учитывая, что сотрудники Google и без того перегружены электронной почтой, эти письма могли затеряться. После небольшого мозгового штурма кто-то в шутку предложил вывешивать листовки в туалетных кабинках. Мы быстро осознали гениальность этой идеи: туалетная кабинка — это место, которое каждый посещает хотя бы раз в день, несмотря ни на что. Как бы то ни было, идея была достаточно дешевой в реализации.

В апреле 2006 года в туалетных кабинках в Google появилась короткая статья о том, как улучшить тестирование кода на Python, написанная небольшой группой добровольцев. Сказать, что реакция была самой разной, — ничего не сказать. Некоторые даже рассматривали это как вторжение в личное пространство и решительно возражали. Списки рассылки заполнились жалобами, но создатели TotT были довольны: даже те, кто жаловался, так или иначе, говорили о тестировании.

В конечном итоге шум утих и рассылка TotT быстро стала частью культуры Google. К настоящему времени инженеры со всей компании опубликовали в рамках этой рассылки несколько сотен статей, охватывающих почти все мыслимые аспекты тестирования (в дополнение к множеству других технических тем). Многие с нетерпением ждут новых листовок, а некоторые инженеры даже добровольно расклеивают листовки в разных помещениях. Мы намеренно ограничиваем размер каждой статьи ровно одной страницей, предлагая авторам сосредоточиться на наиболее важных и действенных советах. В хорошей статье должны описываться инструменты, которые инженер сможет немедленно попробовать.

По иронии судьбы, статьи TotT, появляющиеся в одном из наиболее уединенных мест, оказали огромное общественное влияние. Большинство сторонних посетителей в какой-то момент видят эти статьи, что часто приводит к забавным разговорам о том, что гуглеры постоянно думают только о коде. Кроме того, статьи TotT пользуются большим успехом в блогах, что воодушевило оригинальных авторов. Они начали публиковать их отредактированные версии (<https://oreil.ly/86Nho>), делаясь нашим опытом со всей отраслью.

Несмотря на то что рассылка TotT начиналась как шутка, она пережила испытание временем и оказала самое большое влияние среди всех инициатив по тестированию, запущенных группой Testing Grouplet.

Современная культура тестирования

Современная культура тестирования в Google прошла долгий путь с 2005 года. Нуглеры по-прежнему посещают лекции по тестированию в рамках программы ранней ориентации, и почти каждую неделю публикуются новые статьи TotT. Благодаря

всему этому практика тестирования глубоко укоренилась в повседневном рабочем процессе разработчиков.

Каждое изменение кода в Google должно пройти процедуру обзора, причем каждое изменение должно включать не только код, но и соответствующий тест. Ожидается, что рецензенты оценят качество и правильность обоих. Разумной мерой считается заблокировать изменение, если в нем отсутствуют тесты.

В качестве замены программы сертификации практики тестирования одна из наших команд, занимающаяся вопросами увеличения продуктивности инженеров, недавно выпустила инструмент под названием Project Health (рН). Этот инструмент постоянно собирает десятки показателей о состоянии проекта, включая охват тестирования и задержку на выполнение тестов, и публикует эти сведения для внутреннего использования. рН измеряет состояние проекта по шкале от одного (худшее) до пяти (лучшее). Проект рН-1 считается проблемой для команды. Почти каждая команда, организовавшая непрерывную сборку, автоматически получает свою оценку рН.

Со временем практика тестирования стало неотъемлемой частью культуры Google. У нас есть множество способов повысить ценность тестов для инженеров. Комбинируя курсы обучения, методы мягкой мотивации, наставничество и даже элементы дружеского соревнования, мы укоренили в сотрудниках ясное осознание, что тестирование касается каждого.

Почему мы не требуем обязательного написания тестов?

Группа Testing Grouplet хотела обратиться к высшему руководству с предложением о введении обязательного тестирования, но быстро отказалась от этой мысли. Любые требования о том, как разрабатывать код, даже из самых благих побуждений, будут серьезно противоречить культуре Google и, скорее всего, замедлят прогресс в разработке. Мы верим, что успешные идеи получают самое широкое распространение, нужно только доказать их успешность.

Если инженеры сами решали начать писать тесты, это означает, что они полностью приняли идею тестирования и, вероятно, продолжают поступать правильно без всякого принуждения.

Ограничения автоматизированного тестирования

Не каждый тест следует автоматизировать. Например, тестирование качества результатов поиска предполагает человеческое суждение. Мы проводим целенаправленные внутренние исследования, привлекая экспертов, способных оценить качество поиска, которые выполняют запросы и записывают свои впечатления. Точно так же в автоматическом тестировании трудно уловить нюансы качества звука и видео, поэтому мы привлекаем людей для оценки качества систем телефонии или видеоконференций.

Также есть определенные творческие сферы, тестирование которых лучше всего проводят люди. Например, с поиском сложных уязвимостей в безопасности люди справляются намного лучше автоматизированных систем. Когда инженер обнаружит и поймет причину проблемы, он может добавить тест в автоматизированную систему тестирования, такую как Google Cloud Scanner (https://oreil.ly/6_W_q), где этот тест будет выполняться непрерывно и в большем масштабе.

Этот принципиально творческий подход к тестированию имеет обобщенное название — исследовательское тестирование. Человек рассматривает приложение как головоломку и решает ее, выполнив неожиданную последовательность шагов или передав неожиданные данные. При проведении исследовательского тестирования конкретные проблемы кода изначально неизвестны. Они обнаруживаются постепенно, путем проверки часто упускаемых из виду путей выполнения кода или необычных ответов приложения. Так же как при поиске уязвимостей безопасности, для любой решенной головоломки инженер должен добавить автоматизированный тест, чтобы найденная проблема не повторилась в будущем.

Автоматизированное тестирование понятного поведения кода позволяет сосредоточить дорогостоящие усилия людей-тестировщиков на тех областях, где они могут принести наибольшую пользу, и избавить инженеров от рутины.

Заключение

Внедрение автоматизированного тестирования, управляемого разработчиками, стала одной из самых эффективных практик программной инженерии в Google. Она позволила нам создавать большие системы большими командами быстрее, чем мы думали, и отвечать растущими темпами развития технологий. За последние 15 лет мы превратили тестирование в часть нашей инженерной культуры. За эти годы компания выросла почти в 100 раз, и наша приверженность качеству и тестированию сегодня сильнее, чем когда-либо.

Эта глава была написана с целью рассказать вам, как в Google относятся к тестированию. В следующих нескольких главах мы углубимся в некоторые ключевые темы, которые помогли нам сформировать представление о хороших, стабильных и надежных тестах. Мы подробно обсудим юнит-тестирование как наиболее широко используемый в Google вид тестирования. Поговорим об эффективном тестировании взаимодействий с использованием дублеров, таких как имитации и заглушки, и исследуем проблемы, связанные с тестированием больших и сложных систем, подобных тем, которые мы имеем в Google.

Прочитав следующие три главы, вы получите более полное и ясное представление о стратегиях тестирования, которые мы используем, и, что более важно, о причинах их использования.

Итоги

- Автоматическое тестирование образует основу для изменения ПО.
- Автоматизация — необходимое условие масштабирования тестов.
- Для поддержания достаточного охвата тестирования необходим сбалансированный набор тестов.
- «Если вам что-то понравилось, добавьте для него тест».
- Для изменения культуры тестирования в организациях нужно время.

Юнит-тестирование

*Автор: Эрик Кюфлер
Редактор: Том Манирек*

В предыдущей главе были представлены две основные оси классификации тестов в Google: *размер* и *охват*. Напомним, что под размером подразумевается объем ресурсов, потребляемых тестом, и число элементов инфраструктуры, вовлекаемых в тестирование, а под охватом — объем кода, проверяемого тестом. Размеры тестов четко определены в Google, чего нельзя сказать об уровнях охвата тестирования. Для обозначения тестов с относительно узкой областью охвата, такой как отдельный класс или метод, мы используем термин *юнит-тест*. Юнит-тесты обычно имеют маленькие размеры, но это не всегда так.

После выявления ошибок второй наиболее важной целью тестирования является повышение продуктивности инженеров. По сравнению с более масштабными тестами, юнит-тесты обладают множеством свойств, способствующих увеличению продуктивности.

- Обычно это маленькие по размеру тесты — быстрые и детерминированные, что позволяет разработчикам часто запускать их и получать немедленную обратную связь.
- Их, как правило, легко писать одновременно с кодом, который они тестируют, что дает инженерам возможность сконцентрировать свои тесты на коде, не затраивая более крупную систему.
- Простые в написании, они обеспечивают широкий охват тестирования, который дает инженерам дополнительную уверенность, что они ничего не нарушат, внося изменения.
- Они позволяют быстро найти причину ошибки в случае неудачного тестового прогона благодаря своей простоте и нацеленности на проверку ограниченной части системы.
- Они могут служить документацией и примерами, показывающими инженерам, как использовать тестируемую часть системы и как эта система должна работать.

Благодаря многочисленным преимуществам большинство тестов в Google — юнит-тесты, и мы рекомендуем инженерам стремиться к тому, чтобы юнит-тесты составляли

около 80 % тестовых наборов. Следуя этой рекомендации и ожидая вышеназванных выгод, в течение рядового рабочего дня инженер Google запускает несколько тысяч юнит-тестов (прямо или косвенно).

Тестирование занимает значительную долю рабочего времени инженеров, поэтому в Google уделяется большое внимание *удобству сопровождения тестов*. Тесты должны «просто работать»: после их написания инженер должен забыть о них, пока они не потерпят неудачу из-за реальной ошибки по понятной причине. Основная часть этой главы посвящена изучению удобства сопровождения тестов и методов его достижения.

Важность удобства сопровождения

Представьте такой сценарий: Мэри добавила в продукт новую простую функцию из 20 строк кода. Но в ответ на это автоматизированная система тестирования насыпала полный экран ошибок. И Мэри провела остаток дня, исследуя их. Изменение в коде не содержало фактической ошибки, но нарушало предположения тестов о внутренней структуре кода, что требовало обновления самих тестов. Часто Мэри было трудно понять, чего именно хотели тесты, поэтому ее исправления сделали тесты еще более трудными для понимания в будущем. В итоге работа, которая не должна была занять много времени, растянулась на часы напряженного труда, истощая продуктивность и подрывая моральный дух Мэри.

В данном случае тестирование имело эффект, противоположный ожидаемому, потому что снизило продуктивность, не улучшив при этом качество тестируемого кода. Инженеры в Google сталкиваются с такой проблемой каждый день. У нее нет простого решения, но многие наши сотрудники работают над созданием наборов шаблонов и практик, помогающих преодолеть ее, и мы советуем всем в нашей компании присоединиться к этому благому делу.

Мэри не виновата в проблемах, с которыми ей пришлось столкнуться, и она ничего не могла сделать, чтобы избежать их: плохие тесты должны быть исправлены до того, как попадут в репозиторий, чтобы они не затронули других инженеров. В целом эти проблемы делятся на две категории: *хрупкие* тесты, которые ломаются в ответ на безвредные изменения, и *неясные* тесты, после провала которых трудно определить, в чем причина неудачи, как ее устранить и что именно должны были делать эти тесты.

Как предотвратить хрупкие тесты

Как только что было определено, хрупкий тест — это тест, который может провалить изменение в коде, не содержащее фактических ошибок¹. Такие тесты должны выявляться и исправляться инженерами в рамках повседневной работы. В небольших

¹ Обратите внимание, что хрупкие тесты немного отличаются от *нестабильных тестов*, которые терпят неудачу недетерминированно, без всяких изменений в тестируемом коде.

базах кода, разрабатываемых небольшой командой инженеров, корректировка нескольких тестов не будет большой проблемой. Но если команда регулярно пишет хрупкие тесты, ей придется тратить все больше времени на обслуживание тестов, потому что число сбоев будет расти вместе с числом тестов. Если для каждого изменения инженерам придется вручную настраивать набор тестов, трудно назвать такой набор «автоматизированным»!

Хрупкие тесты вызывают головную боль при работе с базой кода любого размера, но эта боль становится особенно острой в масштабах Google. Любой инженер Google может запускать тысячи тестов в течение дня, а одно крупномасштабное изменение (главу 22) может повлечь запуск сотен тысяч тестов. При таких масштабах ложные отказы даже в небольшой доле тестов могут привести к тратам огромного количества инженерного времени. В разных командах Google разные показатели хрупкости тестов, но мы определили несколько общих практик и шаблонов, помогающих сделать тесты более устойчивыми к изменениям.

Добивайтесь неизменности тестов

Прежде чем перейти к шаблонам, помогающим предотвратить появление хрупких тестов, мы должны ответить на вопрос: насколько часто нужно изменять тест после его написания? Ведь можно заняться более интересной работой, чем изменение тестов. Таким образом, *идеальный тест* — это *неизменяемый тест*: после написания его не нужно изменять, если не изменяются требования SUT.

Как это выглядит на практике? Подумайте о том, какие изменения вносят инженеры в код и как тесты реагируют на эти изменения. По сути, есть четыре вида изменений:

Чистый рефакторинг

Это рефакторинг внутренних компонентов системы без изменения ее интерфейса, например для увеличения производительности, ясности или по любой другой причине, который не требует изменения тестов. Задача тестов в этом случае — убедиться, что в процессе рефакторинга не было изменено поведение системы. Если во время рефакторинга потребовалось изменить тесты, значит, внесенные изменения повлияли на поведение системы (рефакторинг не чистый) или уровень абстракции тестов не соответствует желаемому. Подробнее о чистом рефакторинге и поддержке масштабных изменений в Google читайте в главе 22.

Новые особенности

При добавлении новых особенностей существующее поведение системы должно оставаться неизменным. Инженер должен написать новые тесты для проверки новых особенностей, не меняя уже имеющиеся тесты. Как и в случае с рефакторингом, изменение существующих тестов при добавлении новой особенности свидетельствует о непреднамеренных последствиях включения этой особенности или о некорректности тестов.

Исправление ошибок

Наличие ошибки предполагает, что она не обнаруживалась первоначальным набором тестов, и процесс исправления ошибки должен включать создание отсутствующего теста. Опять же, исправления ошибок обычно не требуют изменения существующих тестов.

Изменение поведения

Может потребовать изменить существующие тесты. Обратите внимание, что такие изменения, как правило, обходятся значительно дороже, чем изменения трех других типов. Пользователи системы, вероятно, будут полагаться на ее текущее поведение, и изменения в поведении потребуют координации с этими пользователями, чтобы избежать путаницы или недопонимания. Изменение теста в этом случае указывает на то, что мы намеренно нарушаем контракт системы, тогда как изменения в предыдущих случаях свидетельствуют о непреднамеренном нарушении контракта. Разработчики низкоуровневых библиотек часто прикладывают значительные усилия, чтобы избежать изменений в поведении и не нарушить работу кода, который использует их библиотеки.

Таким образом, после того как тест написан, мы не должны снова прикасаться к нему при рефакторинге системы, исправлении ошибок или добавлении новых особенностей. Это понимание позволяет нам работать с крупной системой: расширение системы потребует написания лишь небольшого числа новых тестов, прямо связанных с вносимыми изменениями, но не пересмотра тестов, написанных раньше. Только критические изменения в поведении системы могут повлечь изменения тестов, но помните, что обновление тестов системы обходится дешевле обновления всего кода, пользующегося системой.

Тестирование через общедоступные API

Теперь, определив свою цель, давайте взглянем на некоторые практики, помогающие избежать необходимости изменять тесты, если не изменяются требования SUT. Для начала пишите тесты, которые обращаются к SUT точно так же, как ее пользователи, — то есть вызывают ее общедоступный API и не учитывают детали ее реализации (<https://oreil.ly/ijat0>). Если тесты работают так же, как пользователи системы, то изменение, вызывающее нарушение в работе теста, также может привести к нарушению в работе кода пользователя. При этом такие тесты могут служить полезными примерами и документацией для пользователей.

Рассмотрим листинг 12.1, который проверяет транзакцию и сохраняет ее в базе данных.

Заманчиво протестировать этот код, удалив модификаторы `private` и проверив логику реализации напрямую, как показано в листинге 12.2.

Этот тест взаимодействует с процессором транзакций совсем не так, как реальные пользователи: он проверяет внутреннее состояние системы и вызывает методы,

которые не являются частью общедоступного системного API. В результате тест оказывается хрупким и почти любой рефакторинг SUT (например, переименование методов, выделение их во вспомогательный класс или изменение формата сериализации) может привести к поломке теста, даже если эти изменения будут не видны для реальных пользователей класса.

Листинг 12.1. API транзакции

```
public void processTransaction(Transaction transaction) {
    if (isValid(transaction)) {
        saveToDatabase(transaction);
    }
}

private boolean isValid(Transaction t) {
    return t.getAmount() < t.getSender().getBalance();
}

private void saveToDatabase(Transaction t) {
    String s = t.getSender() + "," + t.getRecipient() + "," + t.getAmount();
    database.put(t.getId(), s);
}

public void setAccountBalance(String accountName, int balance) {
    // Запись баланса непосредственно в базу данных
}

public void getAccountBalance(String accountName) {
    // Чтение транзакций из базы данных для определения баланса
}
```

Листинг 12.2. Наивное тестирование реализации API транзакции

```
@Test
public void emptyAccountShouldNotBeValid() {
    assertThat(processor.isValid(newTransaction().setSender(EMPTY_ACCOUNT)))
        .isFalse();
}

@Test
public void shouldSaveSerializedData() {
    processor.saveToDatabase(newTransaction()
        .setId(123)
        .setSender("me")
        .setRecipient("you")
        .setAmount(100));
    assertThat(database.get(123)).isEqualTo("me,you,100");
}
```

Однако того же охвата тестирования можно достичь путем тестирования только основного общедоступного API класса, как показано в листинге 12.3¹.

Листинг 12.3. Тестирование общедоступного API

```
@Test
public void shouldTransferFunds() {
    processor.setAccountBalance("me", 150);
    processor.setAccountBalance("you", 20);

    processor.processTransaction(newTransaction()
        .setSender("me")
        .setRecipient("you")
        .setAmount(100));

    assertThat(processor.getAccountBalance("me")).isEqualTo(50);
    assertThat(processor.getAccountBalance("you")).isEqualTo(120);
}

@Test
public void shouldNotPerformInvalidTransactions() {
    processor.setAccountBalance("me", 50);
    processor.setAccountBalance("you", 20);

    processor.processTransaction(newTransaction()
        .setSender("me")
        .setRecipient("you")
        .setAmount(100));

    assertThat(processor.getAccountBalance("me")).isEqualTo(50);
    assertThat(processor.getAccountBalance("you")).isEqualTo(20);
}
```

Тесты, использующие только общедоступные API, по определению обращаются к SUT так же, как ее пользователи. Такие тесты более реалистичные и менее хрупкие, потому что используют явные контракты: если такой тест ломается, это означает, что нормальная работа существующего кода, использующего систему, тоже будет нарушена. Тестирование только с использованием явных контрактов позволяет свободно выполнять рефакторинг любых внутренних компонентов системы, не забываясь о необходимости вносить изменения в тесты.

В юнит-тестировании не всегда очевидно, что относится к «общедоступному API», и эта проблема сводится к определению понятия «юнит». Юниты могут быть такими же маленькими, как отдельные функции, или большими, как набор из нескольких взаимосвязанных пакетов. Когда в этом контексте мы говорим «общедоступный API», то подразумеваем API, предоставляемый одним юнитом третьим сторонам. Это не

¹ Иногда этот подход называют «Принципом использования парадной двери» (<https://oreil.ly/8zSZg>).

всегда согласуется с понятием видимости в некоторых языках программирования. Например, классы в Java могут объявляться как «общедоступные», чтобы их можно было использовать в других пакетах в том же юните, но они не предназначены для использования кодом за пределами юнита. Некоторые языки, такие как Python, вообще не имеют механизмов ограничения видимости (и часто полагаются на такие соглашения, как добавление символов подчеркивания в начало имен закрытых методов), а системы сборки, такие как Bazel (<https://bazel.build>), могут дополнительно ограничивать тех, кому позволено использовать API, объявленный общедоступным в исходном коде.

Определение границ юнита и соответственно общедоступного API — это больше искусство, чем наука. Тем не менее есть несколько универсальных правил:

- Если метод или класс предназначен только для поддержки одного или двух других классов (то есть это «вспомогательный класс»), его, вероятно, не следует рассматривать как юнит и нужно тестировать не напрямую, а через эти другие классы.
- Если пакет или класс спроектирован для использования извне без разрешения его владельца, он почти наверняка является юнитом, который следует тестировать напрямую, причем тесты должны обращаться к юниту так же, как сторонний код.
- Если пакет или класс доступен только его владельцу, но предлагает возможности, полезные в различных контекстах (то есть это «библиотека поддержки»), его следует рассматривать как юнит и тестировать непосредственно. Обычно в этом случае возникает некоторая избыточность в тестировании, потому что код библиотеки поддержки будет охватываться не только собственными тестами, но и тестами кода, который ее использует. Но такая избыточность полезна: без нее может появиться пробел в охвате тестирования, если какой-то код, использующий библиотеку (и его тесты), будет удален.

Мы долго убеждали инженеров, что тестирование через общедоступные API лучше тестирования деталей реализации. Конечно, им было проще написать тест, проверяющий только что написанный фрагмент кода, не выясняя, как этот код влияет на систему в целом. Тем не менее мы считаем, что дополнительные предварительные усилия окупаются многократно за счет снижения затрат на поддержку. Вход через общедоступные API не устраняет хрупкость теста полностью, но позволяет гарантировать, что тесты будут ломаться только в случае значительных изменений в системе.

Тестируйте состояние, а не взаимодействия

Другая причина появления зависимости тестов от деталей реализации заключается не в том, какие методы вызывает тест, а в том, как проверяются результаты этих вызовов. Есть два основных способа убедиться, что поведение SUT соответствует ожиданиям. *Тестируя состояние*, вы наблюдаете за поведением самой системы после обращения к ней. *Тестируя взаимодействия*, вы хотите убедиться, что в ответ на вызов система предприняла ожидаемую последовательность действий (<https://oreil.ly/3S8AL>). Многие тесты объединяют проверки состояния и взаимодействий.

Тестирование взаимодействий, как правило, более хрупкое, чем тестирование состояния, по той же причине, почему тестирование закрытого метода более хрупкое, чем тестирование общедоступного метода: при тестировании взаимодействия проверяется, как система достигла своего результата, а не сам результат. В листинге 12.4 показан тест, использующий тестового дублера (глава 13), чтобы проверить, как система взаимодействует с базой данных.

Листинг 12.4. Хрупкий тест взаимодействия

```
@Test
public void shouldWriteToDatabase() {
    accounts.createUser("foobar");
    verify(database).put("foobar");
}
```

Тест проверяет, был ли сделан вызов определенного API базы данных. Есть несколько ситуаций, когда события могут начать развиваться по неправильному пути:

- Если ошибка в SUT приводит к удалению записи из базы данных вскоре после ее добавления, тест выполнится успешно, даже если мы хотим, чтобы он не был пройден.
- Если в ходе рефакторинга SUT будет вызывать другой API для создания записи, тест завершится ошибкой, даже если мы хотим, чтобы он выполнялся успешно.

Гораздо меньше проблем возникает, если тестировать состояние системы, как показано в листинге 12.5.

Листинг 12.5. Тестирование состояния

```
@Test
public void shouldCreateUsers() {
    accounts.createUser("foobar");
    assertThat(accounts.getUser("foobar")).isNotNull();
}
```

Этот тест точнее выражает наш интерес: состояние SUT после взаимодействия с ней.

Распространенная причина проблем при тестировании взаимодействий — чрезмерная зависимость системы от фреймворков с фиктивными объектами. Эти фреймворки позволяют легко создавать и использовать тестовые дублеры, которые регистрируют и проверяют каждый вызов, направленный им. Эта стратегия прямо ведет к появлению хрупких тестов взаимодействий, и поэтому мы предпочитаем использовать реальные объекты вместо их имитаций, если эти реальные объекты действуют быстро и детерминированно.



Подробнее о тестовых дублерах, фреймворках с фиктивными объектами и их альтернативах в главе 13.

Создание ясных тестов

Рано или поздно, даже предприняв все возможное, чтобы избежать хрупкости, мы столкнемся со сбоями в тестировании. Неудача — это хорошо: неудачи при тестировании дают полезные сигналы инженерам.

Сбой в тесте может произойти по одной из двух причин¹:

- SUT содержит ошибку или недочет. Это именно то, для чего предназначены тесты: сообщать об ошибках, чтобы вы могли их исправить.
- Сам тест содержит ошибку. В этом случае с SUT все в порядке, но тест был построен неверно. Если это уже существующий тест — не тот, который вы только что написали, — это означает, что данный тест хрупкий. В предыдущем разделе обсуждалось, как предотвратить появление хрупких тестов, но их редко получается полностью устранить.

Когда тест провален, инженер в первую очередь должен определить, по какой из этих причин произошел сбой, а затем диагностировать истинную проблему. Скорость, с какой инженер может сделать это, зависит от *ясности* теста. Ясный тест — это тест, диагностика сбоя которого позволяет инженеру сразу определить цель теста и причину сбоя. Также ясные тесты документируют SUT и могут использоваться в качестве основы для новых тестов.

Ценность ясности теста растет с течением времени. Тесты живут долго и должны отвечать изменениям в системе по мере ее развития. Может так получиться, что тест, потерпевший неудачу, был написан несколько лет назад инженером, который больше не работает в команде и не оставил ничего, что помогло бы выяснить назначение теста или способы его исправления. Если назначение неясного кода можно определить по коду, который его вызывает, и нарушениям при его удалении, то цель неясного теста понять невозможно, так как удаление теста вызовет только (потенциально) пробел в охвате тестирования.

В худшем случае такие неясные тесты просто удаляются, если инженерам не удается понять, как их исправить. Удаление тестов не только создает пробел в охвате тестирования, но также указывает на то, что тест имел нулевую ценность, возможно, в течение всего периода своего существования.

Чтобы набор тестов масштабировался и оставался полезным с течением времени, важно, чтобы каждый тест в наборе был максимально ясным. В этом разделе мы обсудим методы и способы увеличения ясности тестов.

¹ Это те же две причины «ненадежности» тестов. Либо SUT имеет недетерминированную ошибку, проявляющуюся от случая к случаю, либо тест имеет недостатки, вынуждающие его терпеть неудачу в случаях, когда он должен выполняться безошибочно.

Тесты должны быть краткими и полными

Достичь ясности тестов помогают два их основных качества: полнота и краткость (<https://oreil.ly/lqwyG>). Тест считается *полным*, если его тело содержит всю информацию, которая может понадобиться читателю, чтобы понять, как этот тест получает свой результат. Тест считается *кратким*, если он не содержит никакой другой отвлекающей или не относящейся к тесту информации. В листинге 12.6 показан тест, который не является ни полным, ни кратким:

Листинг 12.6. Неполный и громоздкий тест

```
@Test
public void shouldPerformAddition() {
    Calculator calculator = new Calculator(new RoundingStrategy(),
        "unused", ENABLE_COSINE_FEATURE, 0.01, calculusEngine, false);
    int result = calculator.calculate(new TestCalculation());
    assertThat(result).isEqualTo(5); // Окуда взялось это число?
}
```

Тест содержит в вызове конструктора много информации, не относящейся к делу, а некоторые важные части теста скрыты внутри вспомогательного метода. Вы можете сделать тест более полным, пояснив назначение аргументов вспомогательного метода, и более кратким, использовав другой вспомогательный метод, чтобы скрыть ненужные детали создания калькулятора.

Листинг 12.7. Полный и краткий тест

```
@Test
public void shouldPerformAddition() {
    Calculator calculator = new Calculator();
    int result = calculator.calculate(new Calculation(2, Operation.PLUS, 3));
    assertThat(result).isEqualTo(5);
}
```

Идеи, которые мы обсудим позже, в частности совместное использование кода, тоже будут связаны с полнотой и краткостью. В частности, нередко имеет смысл нарушить принцип DRY (don't repeat yourself — «не повторяйся»), если это поможет написать более ясный тест. Помните: *тело теста должно содержать всю информацию, необходимую для его понимания, без любой отвлекающей или не имеющей отношения к тесту информации.*

Тестируйте поведение, а не методы

Многие инженеры инстинктивно пытаются подогнать структуру своих тестов к структуре тестируемого кода, чтобы для каждого метода в коде имелся соответствующий метод тестирования. Поначалу этот паттерн может быть удобен, но по мере усложнения тестируемого метода тест также усложняется и читать его становится все труднее. Например, взгляните на фрагмент кода в листинге 12.8, который выводит результаты транзакции:

Листинг 12.8. Фрагмент реализации транзакции

```
public void displayTransactionResults(User user, Transaction transaction) {
    ui.showMessage("You bought a " + transaction.getItemName());
    if (user.getBalance() < LOW_BALANCE_THRESHOLD) {
        ui.showMessage("Warning: your balance is low!");
    }
}
```

Часто тест охватывает оба сообщения, которые могут выводиться методом.

Листинг 12.9. Тест, проверяющий метод

```
@Test
public void testDisplayTransactionResults() {
    transactionProcessor.displayTransactionResults(
        newUserWithBalance(
            LOW_BALANCE_THRESHOLD.plus(dollars(2))),
        new Transaction("Some Item", dollars(3)));

    assertThat(ui.getText()).contains("You bought a Some Item");
    assertThat(ui.getText()).contains("your balance is low");
}
```

В подобных ситуациях изначально тест мог проверять только первое сообщение. Инженер мог расширить тест после добавления вывода второго сообщения (нарушив идею неизменности тестов, которую мы обсуждали выше). Но это изменение создало бы плохой прецедент: по мере усложнения и расширения возможностей тестируемого метода его юнит-тест становился бы все более запутанным и сложным.

Проблема заключается в том, что тестирование, основанное на повторении структуры методов, может способствовать появлению неясных тестов, потому что часто единственный метод решает несколько разных задач и может иметь несколько худших случаев. Поэтому пишите тесты не для каждого метода, а для каждого *поведения*¹. Поведение — это гарантированная реакция системы на серию входных данных в определенном состоянии². Выразить поведение можно словами «дано», «если» и «тогда» (<https://oreil.ly/I9IvR>): «Дано: банковский счет пуст. Если предпринимается попытка снять с него деньги, тогда транзакция должна быть отклонена». Методы и поведения связаны отношением многие-ко-многим: большинство нетривиальных методов реализуют множество поведений, а некоторые поведения полагаются на взаимодействие нескольких методов. Предыдущий пример можно переписать, используя подход, основанный на тестировании поведения (листинг 12.10).

Дополнительный шаблонный код введен для разделения теста (<https://oreil.ly/hcoon>). Тесты, проверяющие поведение, по ряду причин получаются более ясными, чем тесты, проверяющие методы. Во-первых, они читаются почти как текст на естественном языке,

¹ См. <https://testing.googleblog.com/2014/04/testing-on-toilet-test-behaviors-not.html> и <https://dannorth.net/introducing-bdd>.

² Кроме того, *особенность* (или продукт) можно выразить в виде набора поведений.

что позволяет легко понять их без скрупулезного анализа. Во-вторых, они более ясно выражают причинно-следственные связи в ограниченном объеме теста (<https://oreil.ly/dAd3k>). Наконец, краткость и описательность теста помогают понять, какие функциональные возможности он тестирует, и побуждают инженеров писать более оптимизированные новые методы, вместо того чтобы пытаться вставить новый код в существующие методы.

Листинг 12.10. Тестирование поведения

```
@Test
public void displayTransactionResults_showsItemName() {
    transactionProcessor.displayTransactionResults(
        new User(), new Transaction("Some Item"));
    assertThat(ui.getText()).contains("You bought a Some Item");
}

@Test
public void displayTransactionResults_showsLowBalanceWarning() {
    transactionProcessor.displayTransactionResults(
        newUserWithBalance(
            LOW_BALANCE_THRESHOLD.plus(dollars(2))),
        new Transaction("Some Item", dollars(3)));
    assertThat(ui.getText()).contains("your balance is low");
}
```

Структурируйте тесты, чтобы подчеркнуть поведение

Подход, основанный на тестировании поведения, влияет на структуру тестов. Поведение состоит из трех частей: компонент «дано» определяет настройки системы, компонент «если» определяет выполняемое в системе действие и компонент «тогда» проверяет результат. Тесты тем яснее, чем более явно следуют этой структуре. Некоторые фреймворки, такие как Cucumber (<https://cucumber.io>) и Spock (<http://spockframework.org>), напрямую используют структуру дано/если/тогда. В других языках можно использовать пустые строки и комментарии, чтобы выделить структуру, как показано в листинге 12.11.

Листинг 12.11. Хорошо структурированный тест

```
@Test
public void transferFundsShouldMoveMoneyBetweenAccounts() {
    // Дано: два счета, на которых находятся $150 и $20
    Account account1 = newAccountWithBalance(usd(150));
    Account account2 = newAccountWithBalance(usd(20));

    // Если предпринята попытка перевести $100 с первого счета на второй
    bank.transferFunds(account1, account2, usd(100));

    // Тогда суммы на счетах должны отражать факт перевода
    assertThat(account1.getBalance()).isEqualTo(usd(50));
    assertThat(account2.getBalance()).isEqualTo(usd(120));
}
```

Такое подробное описание не всегда необходимо, особенно в тривиальных тестах: часто комментарии можно опустить и использовать только пустые строки, чтобы выделить разделы. Однако если тесты сложные, явные комментарии сделают их более простыми для понимания. Следующий шаблон позволяет читать тесты на трех уровнях детализации:

1. Для начала читатель смотрит на имя метода теста (обсуждается ниже), чтобы получить примерное представление о тестируемом поведении.
2. Если информации недостаточно, читатель знакомится с формальным описанием поведения в комментариях, выраженным через дано/если/тогда.
3. Наконец, читатель исследует код, чтобы точно узнать, как поведение выражается.

Этот шаблон чаще всего нарушается чередованием вызовов системы и инструкций проверки (то есть слиянием блоков «если» и «тогда»). Такое слияние делает тест менее ясным, потому что затрудняет различие между выполняемым действием и ожидаемым результатом.

Чередование блоков «если» и «тогда» допустимо, когда необходимо проверить каждый шаг в многоступенчатом процессе. Длинные блоки для удобства чтения можно разделить союзом «и». В листинге 12.12 показано, как может выглядеть тест относительно сложного поведения.

Листинг 12.12. Чередование блоков «если» и «тогда» в тесте

```
@Test
public void shouldTimeOutConnections() {
    // Дано: два пользователя
    User user1 = newUser();
    User user2 = newUser();

    // И пустой пул соединений с 10-минутным тайм-аутом
    Pool pool = newPool(Duration.minutes(10));

    // Если к пулу попытаются подключиться оба пользователя
    pool.connect(user1);
    pool.connect(user2);

    // Тогда пул должен содержать два соединения
    assertThat(pool.getConnections()).hasSize(2);

    // Если прошло 20 минут
    clock.advance(Duration.minutes(20));

    // Тогда в пуле не должно остаться соединений
    assertThat(pool.getConnections()).isEmpty();

    // И оба пользователя должны быть отключены
    assertThat(user1.isConnected()).isFalse();
    assertThat(user2.isConnected()).isFalse();
}
```

Проявляйте особое внимание при написании таких тестов и убедитесь, что по случайности не пытаетесь протестировать несколько разных поведений. Каждый тест должен охватывать только одно поведение, и подавляющее большинство юнит-тестов должны включать только один блок «если» и один блок «тогда».

Давайте тестам имена, описывающие тестируемое поведение

Тесты, проверяющие методы, обычно называются в честь тестируемых методов (например, тест для метода `updateBalance` обычно получает имя `testUpdateBalance`). В имени теста, проверяющего поведение, мы можем передать намного больше полезной информации. Имя теста играет важную роль в расследовании причин неудач тестирования, поскольку оно отображается в отчетах о сбоях. Также имя — самый простой способ выразить цель теста.

Чтобы имя теста подсказывало тестируемое поведение, в этом имени нужно указать не только действия, предпринимаемые системой, но и ожидаемый результат тестирования (<https://oreil.ly/8eqqv>). Имена тестов могут включать дополнительную информацию, например состояние системы или ее окружения перед выполнением действий с ней. Некоторые языки и фреймворки позволяют вкладывать тесты друг в друга и давать им строковые имена, как показано в примере фреймворка Jasmine (<https://jasmine.github.io>):

Листинг 12.13. Некоторые примеры паттернов вложенных имен

```
describe("multiplication", function() {
  describe("with a positive number", function() {
    var positiveNumber = 10;
    it("is positive with another positive number", function() {
      expect(positiveNumber * 10).toBeGreaterThan(0);
    });
    it("is negative with a negative number", function() {
      expect(positiveNumber * -10).toBeLessThan(0);
    });
  });
  describe("with a negative number", function() {
    var negativeNumber = 10;
    it("is negative with a positive number", function() {
      expect(negativeNumber * 10).toBeLessThan(0);
    });
    it("is positive with another negative number", function() {
      expect(negativeNumber * -10).toBeGreaterThan(0);
    });
  });
});
```

Другие языки требуют кодировать эту информацию в имени метода, как в следующем паттерне именования.

Листинг 12.14. Некоторые примеры паттернов именования методов

```
multiplyingTwoPositiveNumbersShouldReturnAPositiveNumber  
multiply_postiveAndNegative_returnsNegative  
divide_byZero_throwsException
```

Такие имена слишком многословны для использования в коде, но подходят для описания назначения тестов, поскольку нам никогда не придется писать код, вызывающий тесты.

Любые политики именования приемлемы, если они неуклонно используются в одном тестовом классе. Если вам трудно придумать подходящее имя для теста, попробуйте начать со слова «should» (должен). Использование этого слова с именем тестируемого класса позволяет прочитать имя теста как предложение. Например, имя теста `shouldNotAllowWithdrawalsWhenBalanceIsEmpty` для класса `BankAccount` можно прочитать как «BankAccount не должен разрешать снимать средства с пустого счета». Чтение имен тестов в наборе должно давать достаточно полное представление о поведении, реализуемом SUT. Такие имена также помогают отследить сосредоточенность теста на одном поведении: если в имени теста потребовался союз «и», то есть большая вероятность, что этот тест в действительности тестирует несколько поведений, и его следует разбить на несколько тестов!

Не включайте логику в тесты

Корректность ясных тестов видна с первого взгляда. Это происходит потому, что тест обрабатывает только определенный набор входных данных. В свою очередь, код должен обрабатывать любой входной сигнал, и чтобы гарантировать правильную работу логики, мы должны писать сложные проверки. Но тестовый код лишен таких возможностей — если у вас возникает чувство, что было бы неплохо написать тест для теста, значит, с ним что-то не так!

Сложность часто имеет вид *логики*. Логика определяется посредством императивных элементов языка программирования, таких как операторы, циклы и условные выражения. Чтобы понять, к какому результату приведет код, содержащий логику, недостаточно просто прочитать его с экрана — необходимо провести вычисления. Чтобы сделать тест сложным для понимания, не нужно использовать много логики. Например, взгляните на тест в листинге 12.15 и оцените, выглядит ли он корректным (<https://oreil.ly/yJDqh>).

Листинг 12.15. Логика скрывает ошибку в тесте

```
@Test  
public void shouldNavigateToAlbumsPage() {  
    String baseUrl = "http://photos.google.com/";  
    Navigator nav = new Navigator(baseUrl);  
    nav.goToAlbumPage();  
    assertThat(nav.getCurrentUrl()).isEqualTo(baseUrl + "/albums");  
}
```

Здесь не так много логики — всего одна операция конкатенации строк. Но если упростить тест, удалив оператор, ошибка сразу станет очевидной.

Листинг 12.16. Отсутствие логики раскрывает ошибку в тесте

```
@Test
public void shouldNavigateToPhotosPage() {
    Navigator nav = new Navigator("http://photos.google.com/");
    nav.goToPhotosPage();
    assertThat(nav.getCurrentUrl())
        .isEqualTo("http://photos.google.com//albums"); // Вот она!
}
```

Когда видна вся строка, сразу становится заметно, что тест ожидает наличия двух символов слеша в URL вместо одного. Если в коде будет допущена аналогичная ошибка, этот тест не сможет обнаружить ее. Дублирование базового URL — невысокая плата за увеличенную описательность и наглядность теста (см. обсуждение тестов DAMP и DRY далее в этой главе).

Распознать ошибки в операциях конкатенации строк, особенно в сложных конструкциях, таких как циклы и условные выражения, очень трудно. Вывод: старайтесь в тестах писать прямолинейный код и не бойтесь повторения его фрагментов, если это сделает тест более описательным и наглядным. Идеи повторения и совместного использования кода мы обсудим далее в этой главе.

Пишите понятные сообщения об ошибках

Еще один аспект, способствующий увеличению ясности, связан не с написанием самого теста, а с сообщением, которое видит инженер, когда тест терпит неудачу. Инженер должен иметь возможность диагностировать проблему, просто прочитав сообщение об ошибке в журнале или отчете, даже не заглядывая в сам тест. Хорошее сообщение об ошибке содержит почти ту же информацию, что и имя теста: оно должно четко отражать желаемый результат, фактический результат и любые параметры, имеющие отношение к результату.

Вот пример плохого сообщения об ошибке:

```
Test failed: account is closed1
```

Непонятно, то ли тест потерпел неудачу, потому что учетная запись была закрыта, то ли ожидалось, что учетная запись будет закрыта, а тест потерпел неудачу, потому что это не так. Сообщение об ошибке должно четко отличать ожидаемое состояние от фактического и давать больше информации о результате:

```
Expected an account in state CLOSED, but got account2:
  <{name: "my-account", state: "OPEN"}
```

¹ Ошибка тестирования: учетная запись закрыта. — *Примеч. пер.*

² Ожидалась учетная запись в состоянии ЗАКРЫТО, а была получена учетная запись. — *Примеч. пер.*

Хорошие библиотеки могут помочь писать информативные сообщения об ошибках. Рассмотрим утверждения в листинге 12.17, в первом из которых используются классические утверждения из JUnit, а во втором — из Truth (<https://truth.dev>), библиотеки утверждений, разработанной в Google.

Листинг 12.17. Использование утверждений из библиотеки Truth

```
Set<String> colors = ImmutableSet.of("red", "green", "blue");
assertTrue(colors.contains("orange")); // JUnit
assertThat(colors).contains("orange"); // Truth
```

Первое утверждение получает только логическое значение, поэтому может выводить лишь обобщенное сообщение об ошибке, например `ожидалось <true>`, но было получено `<false>`, которое не очень информативно в случае неудачи теста. Второе утверждение явно получает проверяемый предмет и может выводить гораздо более информативные сообщения (<https://oreil.ly/RFUEN>): `AssertionError: <[red, green, blue]> должно содержать <orange>`.

Не во всех языках есть такие средства, но всегда должна быть возможность вручную добавить важную информацию в сообщение об ошибке. Например, тестовые утверждения в Go выглядят так:

Листинг 12.18. Тестовое утверждение в Go

```
result := Add(2, 3)
if result != 5 {
    t.Errorf("Add(2, 3) = %v, want %v", result, 5)
}
```

Повторное использование тестов и кода: DAMP, не DRY

Последний аспект написания ясных тестов и предотвращения хрупкости связан с повторным использованием кода. При разработке ПО принято следовать принципу DRY («не повторяйся»), согласно которому ПО проще в обслуживании, если каждая идея реализована в одном определенном месте и повторение кода сведено к минимуму. Этот подход, в частности, значительно упрощает внесение изменений, потому что инженеру достаточно изменить только один фрагмент кода, а не искать точно такие же фрагменты по всему коду. Однако следование этому принципу может сделать код менее ясным, требуя от читателей переходить по цепочкам ссылок, чтобы понять, что делает код.

В коде неясность является небольшой платой за простоту его изменения и сопровождения. Но следование принципу DRY не дает большой выгоды в тестовом коде. Хорошие тесты должны сохранять стабильность и *желательно* терпеть неудачу при изменении SUT. Сложность в тестах обходится дорого: работоспособность кода (по мере его усложнения) гарантирует набор тестов, но у самих тестов нет гаранта их работоспособности, поэтому с их усложнением увеличивается риск появления оши-

бок. Как упоминалось выше, если тесты становятся слишком сложными и возникает желание их протестировать, значит, в тесте есть проблемы.

Разрабатывая тестовый код, следуйте принципу DAMP (<https://oreil.ly/5VPs2>), то есть используйте «описательные и осмысленные фразы» (descriptive and meaningful phrases). Небольшое дублирование кода в тестах допустимо, если это дублирование делает тест более простым и понятным. Рассмотрим тесты, чрезмерно близко следующие принципу DRY.

Листинг 12.19. Тесты, чрезмерно близко следующие принципу DRY

```
@Test
public void shouldAllowMultipleUsers() {
    List<User> users = createUsers(false, false);
    Forum forum = createForumAndRegisterUsers(users);
    validateForumAndUsers(forum, users);
}

@Test
public void shouldNotAllowBannedUsers() {
    List<User> users = createUsers(true);
    Forum forum = createForumAndRegisterUsers(users);
    validateForumAndUsers(forum, users);
}

// Множество других тестов...

private static List<User> createUsers(boolean... banned) {
    List<User> users = new ArrayList<>();
    for (boolean isBanned : banned) {
        users.add(new User()
            .setState(isBanned ? State.BANNED : State.NORMAL)
            .build());
    }
    return users;
}

private static Forum createForumAndRegisterUsers(List<User> users) {
    Forum forum = new Forum();
    for (User user : users) {
        try {
            forum.register(user);
        } catch (BannedUserException ignored) {}
    }
    return forum;
}

private static void validateForumAndUsers(Forum forum, List<User> users) {
    assertThat(forum.isReachable()).isTrue();
}
```



```
for (User user : users) {
    assertThat(forum.hasRegisteredUser(user))
        .isEqualTo(user.getState() == State.BANNED);
}
}
```

Проблемы в этом коде очевидны. Во-первых, тестовые методы краткие, но не полные: важные детали скрыты за вызовами вспомогательных методов, которые читатель не может увидеть, не прокрутив содержимое редактора. Кроме того, эти вспомогательные методы содержат логику, которую трудно понять с первого взгляда (вы смогли заметить ошибку?). Тест станет намного понятнее, если переписать его, следуя принципу DAMP:

Листинг 12.20. Тесты должны следовать принципу DAMP

```
@Test
public void shouldAllowMultipleUsers() {
    User user1 = newUser().setState(State.NORMAL).build();
    User user2 = newUser().setState(State.NORMAL).build();

    Forum forum = new Forum();
    forum.register(user1);
    forum.register(user2);

    assertThat(forum.hasRegisteredUser(user1)).isTrue();
    assertThat(forum.hasRegisteredUser(user2)).isTrue();
}

@Test
public void shouldNotRegisterBannedUsers() {
    User user = newUser().setState(State.BANNED).build();

    Forum forum = new Forum();
    try {
        forum.register(user);
    } catch (BannedUserException ignored) {}

    assertThat(forum.hasRegisteredUser(user)).isFalse();
}
```

В этих тестах больше повторяющегося кода и методы длиннее, но смысл каждого отдельного теста объяснен в пределах тела метода. При чтении этих тестов инженер может быть уверен, что тесты делают именно то, что заявляют, и не скрывают ошибок.

Принцип DAMP служит не заменой, а дополнением к принципу DRY. Он не запрещает использовать вспомогательные методы и исключать повторяющиеся шаги, если эти меры улучшают описательность и осмысленность тестов. В оставшейся части этого раздела мы рассмотрим общие паттерны совместного использования кода в нескольких тестах.

Общие значения

Часто наборы тестов определяют множество общих значений, используемых отдельными тестами для проверки различных случаев применения этих значений. Листинг 12.21 иллюстрирует такие тесты.

Листинг 12.21. Общие значения с неоднозначными именами

```
private static final Account ACCOUNT_1 = Account.newBuilder()
    .setState(AccountState.OPEN).setBalance(50).build();

private static final Account ACCOUNT_2 = Account.newBuilder()
    .setState(AccountState.CLOSED).setBalance(0).build();

private static final Item ITEM = Item.newBuilder()
    .setName("Cheeseburger").setPrice(100).build();

// Сотни других тестов...

@Test
public void canBuyItem_returnsFalseForClosedAccounts() {
    assertThat(store.canBuyItem(ITEM, ACCOUNT_1)).isFalse();
}

@Test
public void canBuyItem_returnsFalseWhenBalanceInsufficient() {
    assertThat(store.canBuyItem(ITEM, ACCOUNT_2)).isFalse();
}
```

Эта стратегия позволяет делать тесты очень краткими, но может стать источником проблем при росте набора тестов. Иногда бывает трудно понять, почему для теста выбрано определенное значение. К счастью, имена тестов в листинге 12.21 поясняют, какие сценарии тестируются, но читатель все равно вынужден прокрутить код, чтобы убедиться, что ACCOUNT_1 и ACCOUNT_2 подходят для этих сценариев. Для ясности можно использовать описательные имена констант (например, CLOSED_ACCOUNT и ACCOUNT_WITH_LOW_BALANCE), но тогда потребуется дополнительно проверить детали тестируемого значения, а простота повторного использования констант может побудить инженеров задействовать их, даже когда имена этих констант не совсем точно соответствуют тесту.

Инженеры предпочитают определять и использовать общие константы, потому что объявлять отдельные значения в каждом тесте довольно утомительно. Лучший способ достижения этой цели — конструирование данных с использованием вспомогательных методов (<https://oreil.ly/Jc4VJ>) (листинг 12.22), которые требуют, чтобы автор теста указал только необходимые значения, и устанавливают разумные значения по умолчанию¹

¹ Часто полезно также добавить немного случайности в выбор значений по умолчанию, не заданных явно. Это поможет гарантировать, что два разных экземпляра не будут оцениваться

для всех остальных констант. Такое конструирование легко реализовать в языках, поддерживающих именованные параметры, а в других языках можно использовать, например, паттерн *Строитель* для их эмуляции (часто с помощью таких инструментов, как AutoValue (<https://oreil.ly/cVYK6>)):

Листинг 12.22. Определение общих значений с помощью вспомогательных методов

```
# Вспомогательный метод обортывает конструктор, определяя произвольные
# значения по умолчанию для всех его параметров
def newContact(
    firstName="Grace", lastName="Hopper", phoneNumber="555-123-4567"):
    return Contact(firstName, lastName, phoneNumber)

# Тесты вызывают вспомогательный метод и явно определяют только необходимые
# параметры
def test_fullNameShouldCombineFirstAndLastNames(self):
    def contact = newContact(firstName="Ada", lastName="Lovelace")
    self.assertEqual(contact.fullName(), "Ada Lovelace")

// Некоторые языки, такие как Java, не поддерживающие именованные параметры,
// могут эмулировать их, возвращая изменяемый объект "builder", который
// представляет конструируемое значение
private static Contact.Builder newContact() {
    return Contact.newBuilder()
        .setFirstName("Grace")
        .setLastName("Hopper")
        .setPhoneNumber("555-123-4567");
}

// В этих языках тесты могут вызывать методы объекта builder, чтобы
// переопределить только необходимые параметры, а затем вызвать
// build(), чтобы создать фактический экземпляр builder
@Test
public void fullNameShouldCombineFirstAndLastNames() {
    Contact contact = newContact()
        .setFirstName("Ada")
        .setLastName("Lovelace")
        .build();
    assertThat(contact.getFullName()).isEqualTo("Ada Lovelace");
}
```

Использование вспомогательных методов для построения таких значений позволяет каждому тесту создавать значения, которые не приведут к конфликту между тестами.

как равные, и не позволит инженерам жестко задавать в коде зависимости от этих значений по умолчанию.

Общие настройки

Еще один прием для совместного использования кода в тестах — определение общей логики настройки и инициализации. Многие фреймворки тестирования позволяют определять методы, которые будут выполняться перед каждым тестом в наборе. При правильном использовании эти методы могут сделать тесты более ясными и краткими и избавить их от повторений кода и неуместной логики инициализации. Однако неправильное использование этих методов может повредить полноте теста и скрыть важные детали в отдельном методе инициализации.

Наилучший вариант использования методов, выполняющих настройку, — создать тестируемый объект вместе с его зависимостями. Этот прием удобен, когда для большинства тестов в наборе не важны конкретные аргументы, применяемые для создания тестируемых объектов, и допускается использовать значения по умолчанию. Эта же идея применима и к заглушкам возвращаемых значений в тестовых дублерах (глава 13).

Один из недостатков методов настройки — они могут ухудшить ясность тестов, если эти тесты зависят от конкретных значений, используемых при настройке. Например, тест в листинге 12.23 выглядит неполным, потому что он вынуждает читателя выяснять, откуда взялась строка "Donald Knuth".

Листинг 12.23. Зависимость от значений, определяемых в методах настройки

```
private NameService nameService;
private UserStore userStore;

@Before
public void setUp() {
    nameService = new NameService();
    nameService.set("user1", "Donald Knuth");
    userStore = new UserStore(nameService);
}

// [... сотни других тестов ...]

@Test
public void shouldReturnNameFromService() {
    UserDetails user = userStore.get("user1");
    assertThat(user.getName()).isEqualTo("Donald Knuth");
}
```

Тесты, которые явно зависят от конкретных значений, должны указывать эти значения непосредственно, при необходимости переопределяя значения по умолчанию, создаваемые методом настройки. Конечно, это приведет к повторению кода, как показано в листинге 12.24, но в результате тест получается гораздо более описательным и осмысленным.

Листинг 12.24. Переопределение значений, создаваемых в методе настройки

```
private NameService nameService;
private UserStore userStore;

@Before
public void setUp() {
    nameService = new NameService();
    nameService.set("user1", "Donald Knuth");
    userStore = new UserStore(nameService);
}

@Test
public void shouldReturnNameFromService() {
    nameService.set("user1", "Margaret Hamilton");
    UserDetails user = userStore.get("user1");
    assertThat(user.getName()).isEqualTo("Margaret Hamilton");
}
```

Общие вспомогательные методы и проверки

Последний распространенный способ совместного использования кода в тестах — «вспомогательные методы», которые вызываются из тестовых методов. Выше мы сказали, что некоторые вспомогательные методы полезны для конструирования общих тестовых значений, но другие виды вспомогательных методов могут быть опасными.

Одной из распространенных разновидностей вспомогательных методов являются методы, выполняющие общий набор утверждений (проверок), относящихся к SUT. Например, метод `validate`, вызываемый в конце каждого тестового метода, выполняющего ряд фиксированных проверок, заставляет тесты обращать меньше внимания на тестирование поведения. Читая такие тесты, вы не сможете определить цель каждого из них. При появлении ошибки метод `validate` может затруднить локализацию тестов и распространит проблему на большое количество тестов в наборе.

Однако более сфокусированные методы проверки все же могут быть полезны. Желательно, чтобы такие вспомогательные методы проверяли *один конкретный факт* о своих входных данных, а не целый ряд условий. Такие методы особенно полезны, когда для проверки концептуально простого условия требуется использовать циклы или условную логику, которая снизила бы ясность теста, если была бы включена в тело тестового метода. Например, вспомогательный метод в листинге 12.25 может пригодиться в тесте, охватывающем несколько разных случаев доступа к учетной записи.

Листинг 12.25. Концептуально простой тест

```
private void assertUserHasAccessToAccount(User user, Account account) {
    for (long userId : account.getUsersWithAccess()) {
        if (user.getId() == userId) {
```

```
        return;  
    }  
}  
fail(user.getName() + " cannot access " + account.getName());  
}
```

Определение тестовой инфраструктуры

Методы, которые мы обсудили выше, охватывают совместное использование кода в одном тестовом классе или наборе. Иногда бывает полезно определить общий код для использования в нескольких наборах тестов. Мы называем такой код *тестовой инфраструктурой*. Она подходит для интеграционного или сквозного тестирования и при хорошей организации может значительно облегчить написание юнит-тестов.

Разработка тестовой инфраструктуры требует более тщательного подхода, чем разработка общего кода, который используется в единственном наборе тестов. Во многих отношениях код тестовой инфраструктуры больше похож на код продакшена, чем на тестовый, потому что он может вызываться из множества разных наборов и нередко его сложно изменить, не нарушив работоспособность тестов. В большинстве случаев предполагается, что инженеры не должны вносить изменения в общую тестовую инфраструктуру, чтобы протестировать свои функции. Тестовая инфраструктура должна рассматриваться как отдельный продукт и соответственно *всегда должна иметь собственные тесты*.

Значительная часть тестовой инфраструктуры, которую использует большинство инженеров, представлена в виде известных сторонних библиотек, например JUnit (<https://junit.org>). Таких библиотек великое множество, и их стандартизация внутри организации должна проводиться как можно раньше и как можно шире. Например, много лет назад фреймворк Mockito был объявлен в Google единственной платформой, которая должна использоваться в новых тестах на Java, и было запрещено использовать другие фреймворки в новых тестах. Тогда этот запрет вызвал недовольство у сотрудников, знакомых с другими фреймворками, но в настоящее время он воспринимается как верный шаг, облегчивший понимание и работу наших тестов.

Заключение

Юнит-тесты — один из самых мощных инструментов, которые помогают инженерам-программистам убедиться, что системы сохранят работоспособность даже после непредвиденных изменений. Но широкие возможности подразумевают большую ответственность, и в результате неосторожного использования юнит-тестирования можно получить систему, которую будет трудно обслуживать и изменять.

Юнит-тестирование в Google еще далеко от совершенства, но мы поняли, что следование методикам, изложенным в этой главе, на несколько порядков увеличивает ценность тестов. Мы надеемся, что вы будете применять эти методики в своей работе!

Итоги

- Стремитесь к неизменности тестов.
- Тестируйте код через общедоступные API.
- Тестируйте состояние, а не взаимодействия.
- Пишите полные и краткие тесты.
- Тестируйте поведение, а не методы.
- Структурируйте тесты так, чтобы они подчеркивали поведение.
- Подбирайте для тестов имена, отражающие тестируемое поведение.
- Не вставляйте логику в тесты.
- Пишите ясные сообщения об ошибках.
- Используйте принцип DAMP вместо DRY при использовании общего кода в тестах.

Тестирование с дублерами

Авторы: Эндрю Тренк и Диллон Блай

Редактор: Том Манирек

Юнит-тесты поддерживают продуктивность разработчиков и уменьшают количество дефектов в коде. Их легко писать для простого кода, но с увеличением сложности кода растёт и сложность написания тестов.

Представьте тестирование функции, которая отправляет запрос внешнему серверу и сохраняет ответ в базе данных. Выполнить несколько тестов такой функции можно, потратив не так много времени. Но на выполнение сотни или тысячи подобных тестов может уйти несколько часов, а наборы таких тестов могут стать нестабильным из-за случайных сбоев в сети или взаимовлияния тестов друг с другом.

В таких случаях применяются *тестовые дублеры* (<https://oreil.ly/vbpiU>) — объекты или функции, способные заменить в тесте действительную реализацию, подобно тому как каскадер-дублер может заменить актёра на съёмочной площадке. Использование тестовых дублеров часто называют имитированием, но мы постараемся избегать этого термина в этой главе, потому что, как будет показано далее, этот термин также используется для обозначения других аспектов, имеющих отношение к тестовым дублерам.

Самой очевидной разновидностью тестового дублера является упрощённая реализация, которая действует подобно реальной, например как база данных в памяти. Другие виды тестовых дублеров позволяют проверить конкретные детали системы, например упрощают вызов редкой ошибки или гарантируют, что вызов тяжёлой функции будет происходить без фактического выполнения реализации этой функции.

В двух предыдущих главах мы познакомились с понятием *маленьких тестов* и обсудили, почему они должны составлять основу набора тестов. Однако код содержит взаимодействия между несколькими процессами или машинами, тестировать которые легче с помощью тестовых дублеров, позволяющих написать множество маленьких тестов, быстрых и стабильных в выполнении.

Влияние тестовых дублеров на разработку ПО

Использование тестовых дублеров вносит некоторые сложности в разработку ПО и требует определенных компромиссов. В этой главе мы подробно обсудим следующие понятия:

Тестируемость

Чтобы использовать тестовые дублиры, база кода должна быть спроектирована с учетом *возможности ее тестирования* — тесты должны иметь возможность заменять реальные реализации тестовыми дублирами. Например, код, обращающийся к базе данных, должен быть достаточно гибким, чтобы тест для этого кода имел возможность использовать тестового дублира вместо реальной базы данных. Если база кода изначально не поддерживает возможность тестирования, вам потребуется приложить массу усилий, чтобы реорганизовать код для поддержки тестовых дублиров.

Применимость

Правильное *применение* тестовых дублиров может значительно увеличить скорость разработки, а при неправильном их использовании (особенно в большой кодовой базе) вы получите хрупкие, сложные и неэффективные тесты, которые будут снижать продуктивность инженеров. Если тестовые дублиры не могут использоваться в конкретном случае, инженер должен применить реальные реализации.

Достоверность

Под *достоверностью* понимается близость поведения тестового дублира к поведению реальной реализации, которую он заменяет. Если в этих поведеньях есть существенные отличия, тесты, использующие такого дублира, скорее всего, не принесут большой пользы. Представьте, насколько полезен тестовый дублир, который имитирует базу данных, игнорирует любые попытки записи данных в него и всегда возвращает пустые результаты. Но идеальная достоверность может оказаться невозможной: тестовые дублиры часто должны быть намного проще реальных реализаций. Во многих ситуациях допустимо использовать дублиров даже без идеальной достоверности. Юнит-тесты, использующие дублиров, часто должны дополняться более масштабными тестами, исследующими реальную реализацию.

Тестовые дублиры в Google

Мы в Google видели бесчисленные примеры увеличения продуктивности инженеров и качества ПО за счет правильного использования тестовых дублиров. Изначально у нас почти не было руководств по эффективному применению тестовых дублиров, но по мере обнаружения общих паттернов и антипаттернов в базах кода многих команд мы разработали свои передовые практики в этой области.

Один из уроков, которые мы усвоили, — чрезмерное использование фреймворков фиктивных объектов, которые позволяют легко создавать тестовые дублиры (мы обсудим такие фреймворки подробнее в этой главе), может быть опасно. Когда мы начали использовать фреймворки фиктивных объектов, они казались универсальным инструментом для создания узкоспециализированных тестов для изолированных фрагментов кода без конструирования зависимостей. Через несколько лет, накопив богатый опыт написания тестов, мы начали понимать стоимость таких тестов: легкие

в написании, они требовали постоянных усилий для поддержки и редко находили ошибки. В результате маятник качнулся в другую сторону: инженеры стали избегать фреймворков с фиктивными объектами, стараясь писать реалистичные тесты.

Несмотря на то что согласие по практикам, обсуждаемым в этой главе, в целом достигнуто в Google, фактическое их применение в разных командах значительно различается. Это обусловлено противоречивостью мнений инженеров и инерцией в базах кода, не соответствующих этим практикам, и стремлением команд делать то, что быстрее всего окупится в краткосрочной перспективе.

Базовые понятия

Прежде чем углубиться в обсуждение приемов эффективного использования тестовых дублеров, обсудим некоторые базовые понятия, связанные с ними, и заложим фундамент для исследования передовых практик, рассмотренных в этой главе.

Пример тестового дублера

Представьте сайт электронной коммерции, который должен обрабатывать платежи по кредитным картам. Такой сайт может содержать код, примерно похожий на код в листинге 13.1.

Листинг 13.1. Служба обработки кредитных карт

```
class PaymentProcessor {
    private CreditCardService creditCardService;
    ...
    boolean makePayment(CreditCard creditCard, Money amount) {
        if (creditCard.isExpired()) { return false; }
        boolean success =
            creditCardService.chargeCreditCard(creditCard, amount);
        return success;
    }
}
```

Использовать в тесте реальную службу обработки кредитных карт невозможно (представьте, какую комиссию придется заплатить за транзакции, выполняемые в ходе тестирования!), зато можно использовать тестового дублера, *эмулирующего* поведение реальной системы. В листинге 13.2 демонстрируется чрезвычайно простой тестовый дублер.

Листинг 13.2. Простейший тестовый дублер

```
class TestDoubleCreditCardService implements CreditCardService {
    @Override
    public boolean chargeCreditCard(CreditCard creditCard, Money amount) {
        return true;
    }
}
```

Этот тестовый дублер не кажется очень полезным, но его использование в тесте позволит нам проверить логику в методе `makePayment()`. Для иллюстрации в листинге 13.3 приводится тест, проверяющий поведение метода в случае предъявления кредитной карты с истекшим сроком действия (тест не зависит от поведения службы обработки кредитных карт).

Листинг 13.3. Использование дублера в тесте

```
@Test public void cardIsExpired_returnFalse() {
    boolean success = paymentProcessor.makePayment(EXPIRED_CARD, AMOUNT);
    assertThat(success).isFalse();
}
```

Далее в этой главе мы обсудим особенности использования дублеров в более сложных ситуациях.

Швы

Код считается пригодным для тестирования (<https://oreil.ly/yssv2>), если позволяет применить к нему юнит-тесты. Организация *швов* (<https://oreil.ly/pFSff>) — это способ подготовки кода к тестированию, добавление в него возможности применения тестовых дублеров. Швы дают возможность использовать при тестировании другие зависимости, отличные от используемых в продакшене.

Внедрение зависимостей (<https://oreil.ly/og9p9>) — распространенный метод организации швов. Если класс использует внедрение зависимостей, любые другие необходимые ему классы (то есть *зависимости*) будут передаваться ему извне, а не создаваться непосредственно в нем, что позволит заменять эти зависимости в тестах.

Листинг 13.4 иллюстрирует внедрение зависимости. Конструктор в этом примере не создает экземпляр `CreditCardService`, а получает его в виде параметра.

Листинг 13.4. Внедрение зависимости

```
class PaymentProcessor {
    private CreditCardService creditCardService;

    PaymentProcessor(CreditCardService creditCardService) {
        this.creditCardService = creditCardService;
    }
    ...
}
```

Код, вызывающий этот конструктор, должен создать соответствующий экземпляр `CreditCardService`. По аналогии с кодом, передающим реализацию `CreditCardService`, которая взаимодействует с внешним сервером, тест может передать тестового дублера.

Листинг 13.5. Передача тестового дублера

```
PaymentProcessor paymentProcessor =
    new PaymentProcessor(new TestDoubleCreditCardService());
```

Чтобы уменьшить объем шаблонного кода, который приходится писать для вызова конструкторов вручную, используйте автоматизированные фреймворки внедрения зависимостей, позволяющие конструировать графы объектов, например Guice (<https://github.com/google/guice>) и Dagger (<https://google.github.io/dagger>) — интегрированные среды автоматического внедрения зависимостей в ПО на Java.

В языках с динамической типизацией, таких как Python или JavaScript, можно динамически заменять отдельные функции или методы объекта. В этих языках внедрение зависимостей не так важно, потому что они позволяют использовать реальные реализации зависимостей в тестах, переопределяя только те функции или методы в зависимостях, которые неуместны для тестов.

Написание кода, пригодного к тестированию, требует затрат, поэтому определите, насколько пригодным для тестирования должен быть ваш код, как можно раньше. Код, написанный без учета тестирования, как правило, приходится реорганизовывать или переписывать перед добавлением соответствующих тестов.

Фреймворки фиктивных объектов

Фреймворк фиктивных объектов (mocking framework) — это программная библиотека, которая упрощает создание тестовых дублеров. Он позволяет заменить объект его *подделкой* (моком) — тестовым дублером, поведение которого определяется непосредственно в тесте. Использование фреймворков фиктивных объектов помогает сократить объем шаблонного кода, потому что избавляет вас от необходимости определять новый класс всякий раз, когда требуется использовать тестовый дублер.

В листинге 13.6 показано использование Mockito (<https://site.mockito.org>) — фреймворка для Java. Mockito создает тестовый дублер для `CreditCardService`, который возвращает определенное значение.

Листинг 13.6. Фреймворки фиктивных объектов

```
class PaymentProcessorTest {
    ...
    PaymentProcessor paymentProcessor;

    // Тестовый дублер для CreditCardService создается единственной строкой кода
    @Mock CreditCardService mockCreditCardService;
    @Before public void setUp() {
        // Передать тестовый дублер в SUT
        paymentProcessor = new PaymentProcessor(mockCreditCardService);
    }

    @Test public void chargeCreditCardFails_returnFalse() {
        // Определить поведение для тестового дублера: вызов его метода
        // chargeCreditCard() всегда должен возвращать false. Использование
        // "any()" в аргументах метода требует от дублера возвращать
        // false независимо от значений аргументов
    }
}
```

```
when(mockCreditCardService.chargeCreditCard(any(), any()))
    .thenReturn(false);
boolean success = paymentProcessor.makePayment(CREDIT_CARD, AMOUNT);
assertThat(success).isFalse();
}
}
```

Фреймворки фиктивных объектов существуют для большинства основных языков программирования. Мы в Google используем Mockito для Java, компонент googlemock в Googletest (<https://github.com/google/googletest>) для C++ и unittest.mock (<https://oreil.ly/clzvH>) для Python.

Хотя фреймворки фиктивных объектов полезны, применять их следует с осторожностью, потому что злоупотребление ими часто усложняет поддержку кодовой базы. Некоторые из проблем, вызванных чрезмерным использованием таких фреймворков, мы рассмотрим в этой главе.

Приемы использования тестовых дублеров

Существуют три основных приема использования тестовых дублеров. Этот раздел кратко описывает их, чтобы вы могли получить общее представление об их особенностях и отличиях. А в последующих разделах мы более детально рассмотрим способы их эффективного применения.

Инженер, знакомый с отличительными чертами этих приемов, сможет грамотно их применить, когда столкнется с необходимостью использовать тестовые дублеры.

Имитации

Имитация (<https://oreil.ly/rymnl>) — это легковесная реализация API, которая действует подобно реальной реализации, но не подходит для использования в продакшене (например, база данных в памяти). В листинге 13.7 показано использование имитации.

Листинг 13.7. Простая имитация

```
// Имитации создаются быстро и просто
AuthorizationService fakeAuthorizationService =
    new FakeAuthorizationService();
AccessManager accessManager = new AccessManager(fakeAuthorizationService);

// Пользователь с неизвестным идентификатором не должен получить доступ
assertFalse(accessManager.userHasAccess(USER_ID));

// После добавления идентификатора в службу авторизации пользователь должен
// получить доступ
fakeAuthorizationService.addAuthorizedUser(new User(USER_ID));
assertThat(accessManager.userHasAccess(USER_ID)).isTrue();
```

Имитации часто идеально сочетаются с тестовыми дублерами. Но иногда для объекта, который необходимо использовать в тесте, нет подходящей имитации, а ее написание может быть затруднено из-за необходимости ее соответствия реальному объекту не только сейчас, но и в будущем.

Заглушки

Установка *заглушки* (стаба) (<https://oreil.ly/gmShS>) — это процесс присвоения функции некоторого поведения, которое иначе недоступно. С ее помощью вы указываете, что именно должна вернуть функция (то есть «заглушаете» фактические возвращаемые значения своими).

Листинг 13.8 иллюстрирует применение заглушки. Вызовы метода `when(...).thenReturn(...)` из фреймворка Mockito определяют поведение метода `lookupUser()`.

Листинг 13.8. Заглушка

```
// Передать тестовый дублер, созданный фреймворком фиктивных объектов
AccessManager accessManager = new AccessManager(mockAuthorizationService);

// Доступ должен быть закрыт, если функция поиска пользователя вернула null
when(mockAuthorizationService.lookupUser(USER_ID)).thenReturn(null);
assertThat(accessManager.userHasAccess(USER_ID)).isFalse();

// Доступ должен быть открыт, если функция поиска пользователя вернула
// непустое значение
when(mockAuthorizationService.lookupUser(USER_ID)).thenReturn(USER);
assertThat(accessManager.userHasAccess(USER_ID)).isTrue();
```

Заглушки, как правило, реализуются с применением фреймворков фиктивных объектов, чтобы уменьшить объем шаблонного кода, который пришлось бы написать, чтобы определить новые классы, возвращающие жестко заданные значения.

Использование заглушек — простой и удобный метод, но он имеет свои ограничения, о которых мы поговорим в этой главе.

Тестирование взаимодействий

Тестирование взаимодействий (<https://oreil.ly/zGfFn>) — это проверка того, как вызывается функция, без фактического вызова ее реализации. Тест должен завершиться неудачей, если функция вызвана неправильно: вообще не была вызвана, вызывалась слишком много раз или была вызвана с неправильными аргументами.

В листинге 13.9 показано тестирование взаимодействий. С помощью метода `verify(...)` из фреймворка Mockito проверяется, произошел ли ожидаемый вызов функции `lookupUser()`.

Подобно заглушкам, тестирование взаимодействий обычно реализуется с применением фреймворков фиктивных объектов. Это сокращает объем шаблонного кода

и избавляет вас от создания вручную новых классов, определяющих, как часто вызывается функция и какие аргументы ей переданы.

Листинг 13.9. Тестирование взаимодействий

```
// Передать тестовый дублер, созданный фреймворком фиктивных объектов
AccessManager accessManager = new AccessManager(mockAuthorizationService);
accessManager.userHasAccess(USER_ID);
```

```
// Тест потерпит неудачу, если accessManager.userHasAccess(USER_ID) не
// вызвал mockAuthorizationService.lookupUser(USER_ID)
verify(mockAuthorizationService).lookupUser(USER_ID);
```

Тестирование взаимодействий иногда называют *имитированием* (<https://oreil.ly/IfMoR>). Мы постараемся избегать этого термина в этой главе, чтобы не связывать имитации с фреймворками фиктивных объектов (подделок), которые можно использовать не только для тестирования взаимодействий, но и для создания заглушек.

Как вы увидите далее в этой главе, тестирование взаимодействия полезно в определенных ситуациях, но по возможности его следует избегать, потому что его чрезмерное использование может привести к созданию хрупких тестов.

Реальные реализации

Тестовые дублеры могут быть эффективными инструментами тестирования, но мы больше предпочитаем использовать реальные реализации тестируемых зависимостей системы — то есть реализации, которые используются в самом коде, потому что тесты более достоверны, когда выполняют код, который будет действовать в продакшене.

Преимущества использования реальных реализаций в тестовом коде открывались нам в течение многих лет, по мере того как мы наблюдали тенденцию захламления тестов повторяющимся кодом из-за чрезмерного использования фреймворков фиктивных объектов, который не обновлялся синхронно с изменениями в реальной реализации и затруднял рефакторинг. Мы рассмотрим эту тему подробнее в этой главе.

Предпочтительное использование реальных реализаций в тестах называется *классическим тестированием* (<https://oreil.ly/OWw7h>). Соответственно в *имитационном тестировании* предпочтение отдается использованию фреймворков фиктивных объектов. Несмотря на то что некоторые инженеры в софтверной индустрии практикуют имитационное тестирование (включая создателей первых фреймворков фиктивных объектов (https://oreil.ly/_QWy7)), мы в Google обнаружили, что этот стиль тестирования трудно масштабировать. При проектировании SUT инженеры должны следовать строгим правилам (<http://jmock.org/oopsla2004.pdf>), а инженеры Google по умолчанию должны писать тестовый код, более подходящий для классического стиля тестирования.

ПРИМЕР: @DONOTMOCK

Мы в Google видели достаточно тестов, которые чрезмерно полагаются на фреймворки фиктивных объектов, чтобы осознать необходимость создания аннотации `@DoNotMock` в Java, которая доступна как часть инструмента статического анализа `ErrorProne` (<https://github.com/google/error-prone>). Эта аннотация позволяет владельцам API объявить, что «этот тип не должен замещаться фиктивной реализацией, потому что существуют лучшие альтернативы».

Если инженер попытается использовать фреймворк фиктивных объектов, чтобы создать экземпляр класса или интерфейса, отмеченного аннотацией `@DoNotMock`, как показано в примере 13.10, он увидит сообщение об ошибке, предписывающее использовать более подходящую стратегию тестирования с применением реальной реализации или имитации. Эта аннотация чаще всего используется для объектов-значений, которые достаточно просты в использовании «как есть», а также для API, имеющих хорошо продуманные имитации.

Листинг 13.10. Аннотация `@DoNotMock`

```
@DoNotMock("Используйте SimpleQuery.create() вместо фиктивной реализации.")
public abstract class Query {
    public abstract String getQueryValue();
}
```

Почему это должно беспокоить владельцев API? Просто потому, что использование в тестах фреймворков фиктивных объектов сильно ограничивает возможности владельца API вносить изменения в свою реализацию с течением времени. Как будет показано далее в этой главе, каждый раз, когда используется фреймворк фиктивных объектов, он дублирует поведение API.

Решив изменить свой API, владелец может обнаружить, что поведение его API подделывается в тысячах или даже десятках тысяч тестов во всей кодовой базе Google! Эти тестовые дублиры с большой вероятностью будут демонстрировать поведение, нарушающее контракт настоящего API, например возвращать `null` из метода, который никогда не может вернуть `null`. Если бы тесты использовали реальную реализацию или имитацию, владелец API смог бы внести изменения в свою реализацию, не исправляя предварительно тысячи тестов, затронутых этим изменением.

Предпочтительность реализма перед изоляцией

Использование реальных реализаций зависимостей делает тестирование системы более реалистичным, потому что тест выполняет код реальных реализаций. Напротив, тест, который использует тестовые дублиры, изолирует SUT от ее зависимостей, потому что тест не выполняет код фактических зависимостей SUT.

Мы предпочитаем реалистичные тесты, потому что они дают больше уверенности в правильной работе SUT. Если юнит-тесты слишком сильно полагаются на тестовые дублиры, инженеру может потребоваться запустить интеграционные тесты или вручную проверить правильность выполнения функции, чтобы достичь такого же уровня уверенности в тесте. Выполнение этих дополнительных задач замедлит разработку

или, если инженеры вообще откажутся выполнять эти задачи из-за слишком большой трудоемкости по сравнению с запуском юнит-тестов, в код проникнут ошибки. Замена всех зависимостей класса тестовыми дублерами приводит к тому, что тестированию подвергается только реализация, которую автор поместил непосредственно в класс. Однако хороший тест не должен зависеть от реализации — он должен быть написан с точки зрения тестируемого API, а не с точки зрения структуры реализации. Использование реальной реализации может вызвать сбой теста (или нескольких тестов), если в ней есть ошибка. И это хорошо! Тесты *не должны* завершаться успехом в подобных случаях, потому что такой «успех» означает, что тестируемый код не будет работать правильно в продакшене. Но вооруженные хорошими инструментами, такими как система непрерывной интеграции, разработчики обычно легко находят изменение в реализации, вызвавшее сбой.

Как определить, когда лучше использовать реальную реализацию

Реальная реализация предпочтительнее, если она работает быстро и детерминированно и имеет простые зависимости. Например, для *объектов-значений* всегда лучше использовать реальную реализацию (<https://oreil.ly/UZiXP>). К ним относятся классы, представляющие денежные суммы, даты, географические адреса или коллекции, такие как список или ассоциативный массив.

Однако использование в тесте реальных реализаций с более сложным кодом часто неосуществимо. Не всегда можно точно сказать, что лучше использовать — реальную реализацию или тестовый дублер, поэтому, не забывая о существующих компромиссах, вы должны руководствоваться следующими соображениями.

Время выполнения

Одним из наиболее важных качеств юнит-тестов является скорость: они должны постоянно запускаться во время разработки и быстро сообщать о работоспособности кода (а также быстро выполняться при запуске в системе непрерывной интеграции). В них тестовые дублеры могут быть полезны, если реальная реализация выполняется медленно.

Но что значит «медленно» с точки зрения юнит-тестов? Если реальная реализация добавляет одну миллисекунду ко времени выполнения каждого отдельного теста, мало кто назовет ее медленной. А если она добавляет 10 миллисекунд, 100 миллисекунд, 1 секунду или больше?

На этот вопрос нет точного ответа. Ответ зависит от того, насколько сильно инженеры ощущают потерю продуктивности во время выполнения реализации и сколько тестов используют реализацию (дополнительная секунда на тест выглядит приемлемой, когда имеется пять тестов, но не 500). Можно сначала использовать в юнит-тесте реальную реализацию, и если она станет слишком медленной, обновить тесты и использовать в них тестовый дублер.

Распараллеливание тестов тоже может помочь уменьшить общее время их выполнения. Наша тестовая инфраструктура в Google упрощает деление тестов на наборы, которые будут выполняться на нескольких серверах. Этот подход увеличивает затраты процессорного времени, но экономит время разработчика. Подробнее об этом подходе в главе 18.

Также помните еще об одном компромиссе: использование реальной реализации может увеличить время тестирования, если тестам придется выполнять сборку реальной реализации и всех ее зависимостей. Хорошо масштабируемая система сборки, такая как Bazel (<https://bazel.build>), может ослабить эту проблему за счет кеширования неизменяемых артефактов.

Детерминированность

Тест считается *детерминированным* (<https://oreil.ly/brxJl>), если для данной версии SUT выполнение теста всегда приводит к одному и тому же результату: тест либо всегда выполняется успешно, либо всегда терпит неудачу. Напротив, тест считается *недетерминированным* (<https://oreil.ly/5pG0f>), если его результат меняется при неизменности SUT.

Отсутствие детерминированности в тестах (<https://oreil.ly/710FU>) может порождать их нестабильное поведение, при котором тесты могут завершаться неудачей, даже если SUT не изменялась. Нестабильность вредит набору тестов, потому что разработчики начинают терять доверие к результатам тестирования и игнорировать сбои (глава 11). Если использование реальной реализации редко приводит к появлению случайных ошибок, эти ошибки не мешают работе инженеров. Но если нестабильное поведение тестов начинает проявляться слишком часто, значит, пришло время заменить реальную реализацию тестовым дублером, чтобы повысить достоверность теста.

Реальная реализация может быть сложнее тестового дублера и больше него способна вызвать недетерминированное поведение в тесте. Например, реальная реализация, использующая многопоточность, иногда может вызывать сбой теста, если выходные данные SUT зависят от порядка выполнения потоков.

Распространенной причиной отсутствия детерминированности в тесте является негерметичный код (https://oreil.ly/aes__), то есть код, зависящий от внешних служб, неподконтрольных тесту. Например, тест, пытающийся получить веб-страницу с HTTP-сервера, может завершиться неудачей, если сервер перегружен или изменилось содержимое веб-страницы. Чтобы тест не зависел от внешнего сервера, используйте тестовые дублеры или герметичный экземпляр сервера, жизненный цикл которого управляется тестом. Подробнее о герметичных экземплярах в следующей главе.

Еще недетерминированным считается тестовый код, опирающийся на системные часы. Если выходные данные SUT зависят от текущего времени, используйте тестовые дублеры, которые жестко кодируют определенное время.

Создание зависимостей

При использовании реальной реализации необходимо создать все ее зависимости. Например, для тестирования объекта нужно построить полное дерево его зависимо-

стей, в которое войдут все объекты, от которых он зависит, все объекты, от которых зависят эти объекты-зависимости, и т. д. Тестовый дублер часто не имеет зависимостей, поэтому создать его проще, чем реальную реализацию.

Представьте, что вы создаете объекты внутри теста. Может потребоваться немало времени, чтобы выяснить, как создать каждый отдельный объект. Кроме того, тесты требуют постоянного обслуживания, потому что они должны обновляться при изменении сигнатур конструкторов этих объектов:

```
Foo foo = new Foo(new A(new B(new C()), new D()), new E(), ..., new Z());
```

В таких случаях может показаться заманчивым использование тестового дублера. Например, вот все, что нужно для создания тестового дублера с помощью фреймворка Mockito:

```
@Mock Foo mockFoo;
```

Да, создать тестовый дублер просто, но использование реальной реализации дает значительные преимущества, которые мы обсудили выше в этом разделе. Кроме того, у чрезмерного использования тестовых дублеров есть существенные недостатки, которые мы рассмотрим далее в этой главе. Таким образом, только компромиссы помогают нам сделать выбор между реальной реализацией и тестовым дублером.

Вместо создания вручную объектов в тестах идеальным решением было бы применение того же кода, который используется в самой системе, например с помощью фабричного метода или автоматического внедрения зависимостей. Для поддержки тестов такой код должен быть достаточно гибким, чтобы можно было использовать тестовые дублеры вместо жестко заданных реализаций, используемых в продакшене.

Имитации

В случае невозможности использовать реальную реализацию в тесте лучшим вариантом часто оказывается использование имитаций. Имитация предпочтительнее других видов тестовых дублеров, потому что она действует подобно реальной реализации: SUT может даже не определить, с чем она взаимодействует — с реальной реализацией или с имитацией. В листинге 13.11 показана имитация файловой системы.

Листинг 13.11. Имитация файловой системы

```
// Эта имитация реализует интерфейс FileSystem. Этот же интерфейс
// используется реальной реализацией
public class FakeFileSystem implements FileSystem {
    // Ассоциативный массив, отображающий имена файлов в их содержимое.
    // Файлы хранятся в памяти, а не на диске, потому что в тестах
    // нежелательно выполнять операции ввода/вывода с диском
    private Map<String, String> files = new HashMap<>();
```

```
@Override
public void writeFile(String fileName, String contents) {
    // Добавить имя файла и его содержимое в ассоциативный массив
    files.add(fileName, contents);
}

@Override
public String readFile(String fileName) {
    String contents = files.get(fileName);
    // Реальная реализация сгенерирует это исключение, если файл
    // не будет найден, поэтому имитация тоже генерирует его
    if (contents == null) { throw new FileNotFoundException(fileName); }
    return contents;
}
}
```

Почему имитации так важны?

Имитации могут быть мощным инструментом тестирования: они быстро выполняются и позволяют эффективно тестировать код, избавляя его от недостатков использования реальных реализаций.

Единственная имитация может радикально улучшить качество тестирования API. А если реализовать большое количество имитаций для всех видов API, то они могут значительно повысить скорость разработки.

В софтверной компании, в которой имитации используются редко, скорость разработки будет низкой, потому что инженеры используют в тестах реальные реализации, из-за которых тесты получаются медленными и нестабильными, или тестовые дублеры (тестирование взаимодействий, заглушки), которые, как будет показано далее в этой главе, могут привести к созданию неясных и хрупких тестов.

Когда следует писать имитации?

Для создания имитации требуется много усилий и опыта. Имитация тоже требует обслуживания: всякий раз, когда меняется поведение реальной реализации, необходимо соответствующим образом обновить ее имитацию. По этой причине команда, которой принадлежит реальная реализация, должна сама написать и поддерживать ее имитацию.

Рассматривая вопрос о создании имитации, команда должна определить, перевесит ли увеличение продуктивности, обусловленное использованием имитации, затраты на ее написание и обслуживание. Если имитация будет использоваться лишь в паре тестов, тогда она не будет стоить времени, потраченного на ее создание, но если таких тестов сотни, разработка имитации даст существенный прирост продуктивности.

Чтобы уменьшить количество имитаций, которые необходимо поддерживать, инженеры создают их только для самого основного кода, который невозможно ис-

пользовать в тестах. Например, если базу данных нельзя использовать в тестах, то следует создать имитацию самого API базы данных, а не каждого класса, который этот API вызывает.

Поддержка имитации может оказаться обременительным делом, если ее реализацию необходимо повторить на разных языках программирования. Представьте имитацию службы, имеющей клиентские библиотеки для обращения к ней из разных языков. Одно из решений для ее поддержки — создать единую реализацию имитируемой службы и использовать в тестах клиентские библиотеки для отправки запросов. Это более сложный подход по сравнению с размещением имитации в памяти, потому что он требует тестировать взаимодействия между процессами. Однако он может стать разумным компромиссом, если тесты выполняются быстро.

Достоверность имитаций

Пожалуй, самой важной характеристикой имитации является ее *достоверность* — мера соответствия поведения реальной реализации поведению ее имитации. Если поведение имитации не соответствует поведению реальной реализации, тест с такой имитацией бесполезен — один и тот же тестовый код может выполняться успешно с имитацией и завершиться неудачей с реальной реализацией.

Идеальная достоверность возможна далеко не всегда. В конце концов, имитация используется в тестах, потому что реальная реализация не подходит для тестирования по каким-то причинам. Например, имитация базы данных обычно недостоверно имитирует реальную базу данных с точки зрения хранения данных на жестком диске, потому что будет хранить все данные в памяти.

Однако в первую очередь имитация должна поддерживать контракты API реальной реализации. Для любых входных данных имитация должна возвращать тот же результат и так же изменять свое состояние, как и реальная реализация. Например, если реальная реализация `database.save(itemId)` успешно сохраняет элемент, указанного идентификатора которого пока не существует, и генерирует ошибку в противном случае, то имитация должна действовать так же.

То есть имитация должна иметь идеальную достоверность *только с точки зрения теста*. Например, имитация API хеширования не должна гарантировать точное совпадение значения хеша для данного ввода со значением, сгенерированным реальной реализацией. В тестах, скорее всего, будет проверяться не конкретное значение хеша, а его уникальность для данного ввода. Если контракт API-хеширования не гарантирует возврат конкретных значений, то имитация будет соответствовать контракту, даже если не обладает идеальной достоверностью.

В числе других примеров, когда идеальная достоверность не имеет большого значения для имитаций, можно назвать задержки и потребление ресурсов. Однако имитация не может использоваться для явной проверки этих показателей (например, в тесте производительности, проверяющем задержку вызова функции) — в таких случаях лучше использовать реальную реализацию.

Имитация не должна обладать всеми функциональными возможностями соответствующей реальной реализации, особенно если этого не требует большинство тестов. Например, в таких случаях, как обработка редких ошибок, лучше, чтобы имитация быстро вышла из строя. Для этого можно вызывать сбой теста при попытке обратиться к возможностям, не реализованным в имитации. Полученная ошибка сообщит инженеру, что имитация не подходит для данной ситуации.

Имитации должны тестироваться

Имитация должна иметь *собственные* тесты, чтобы инженер мог убедиться, что она соответствует API реальной реализации. Имитация без тестов может первоначально показывать реалистичное поведение, но по мере развития реальной реализации поведение реальной реализации и имитации может начать отличаться.

Один из подходов к разработке тестов для имитаций предполагает написание тестов для общедоступного API и проверку с их помощью реальной реализации и имитации (такие тесты еще называют *контрактными* (<https://oreil.ly/youVIX>)). Тесты, использующие реальную реализацию, скорее всего, будут выполняться медленно, но этот их недостаток не имеет существенного значения, потому что эти тесты должны запускаться только владельцами имитации.

Что делать в отсутствие имитации

Если требуемая имитация недоступна, обратитесь к владельцам API с предложением создать ее. Владельцы могут быть не знакомы с идеей имитаций или не осознавать преимуществ, которые имитации дают пользователям API.

Если владельцы API не хотят или не могут создать имитацию, попробуйте написать ее сами. Для этого можно заключить все вызовы API в один класс, а затем создать имитационную версию класса, которая не взаимодействует с API. Это может оказаться намного проще, чем создавать имитацию для всего API, потому что на практике часто используется только подмножество API. Некоторые команды в Google даже делятся своими имитациями с владельцами API, что позволяет другим командам тоже использовать эти имитации.

Наконец, можно просто использовать реальную реализацию (с ее достоинствами и недостатками, о которых мы говорили выше в этой главе) или прибегнуть к другим методам тестирования с дублерами (о достоинствах и недостатках которых мы расскажем далее в этой главе).

В некоторых случаях имитацию можно рассматривать как оптимизацию: если тесты, использующие реальные реализации, выполняются слишком медленно, можно создать имитацию, чтобы заставить их работать быстрее. Но если выгоды от ускорения тестирования не перевешивают затрат на создание и поддержку имитации, то лучше оставить в тесте реальную реализацию.

Заглушки

Как обсуждалось выше в этой главе, заглушки помогают запрограммировать в тесте определенное поведение функции, которое иначе недоступно. Часто это простой и быстрый способ заменить реальную реализацию в тесте. Например, код в листинге 13.12 использует заглушку для эмуляции ответа сервера, обслуживающего кредитные карты.

Листинг 13.12. Использование заглушки для эмуляции ответа сервера

```
@Test public void getTransactionCount() {
    transactionCounter = new TransactionCounter(mockCreditCardServer);
    // Использовать заглушку для возврата трех транзакций
    when(mockCreditCardServer.getTransactions()).thenReturn(
        newList(TRANSACTION_1, TRANSACTION_2, TRANSACTION_3));
    assertThat(transactionCounter.getTransactionCount()).isEqualTo(3);
}
```

Опасности злоупотребления заглушками

Заглушки очень просты в использовании, поэтому у вас может возникнуть соблазн использовать их взамен любой нетривиальной реальной реализации. Однако злоупотребление заглушками в тестах может привести к значительным потерям продуктивности инженеров, которые должны будут поддерживать эти тесты.

Тесты становятся менее ясными

Чтобы создать заглушку, нужно написать дополнительный код, определяющий поведение заглушаемых функций. Этот дополнительный код отвлекает от главной цели теста, и его трудно понять, если вы не знакомы с реализацией SUT.

Если вы ловите себя на мысли, что пытаетесь вникнуть в особенности SUT, чтобы понять, почему некоторые функции в тесте заглушены, это верный признак того, что заглушка не подходит для этого теста.

Тесты становятся хрупкими

Заглушки способствуют проникновению деталей реализации вашего кода в тест. Когда детали реализации в коде изменятся, вам придется обновить тесты, чтобы отразить эти изменения. В идеале хороший тест должен изменяться только при изменении поведения API, обращенного к пользователю, и на тест не должны влиять изменения в реализации API.

Тесты становятся менее эффективными

При использовании заглушек нет гарантий, что они будут действовать подобно реальной реализации. Например, следующая инструкция жестко кодирует часть контракта метода `add()` («Если передать методу числа 1 и 2, он вернет число 3»):
`when(stubCalculator.add(1, 2)).thenReturn(3);`

Заглушка — плохой выбор, если SUT зависит от контракта реальной реализации, потому что вынуждает дублировать в тесте детали контракта и не гарантирует его правильность (то есть достоверность заглушки).

Кроме того, заглушки не позволяют сохранить состояние, что может затруднить тестирование некоторых аспектов кода. Например, после вызова метода `database.save(item)` реальной реализации или имитации можно получить объект `item` обратно, вызвав `database.get(item.id())`, если, конечно, оба метода обращаются к внутреннему состоянию, но сделать то же самое с заглушкой нет никакой возможности.

Пример злоупотребления заглушками

В листинге 13.13 приводится тест, злоупотребляющий заглушками.

Листинг 13.13. Злоупотребление заглушками

```
@Test public void creditCardIsCharged() {
    // Передать в вызов конструктора тестовые дублеры,
    // созданные фреймворком фиктивных объектов
    paymentProcessor =
        new PaymentProcessor(mockCreditCardServer, mockTransactionProcessor);

    // Настроить заглушки для этих тестовых дублеров
    when(mockCreditCardServer.isServerAvailable()).thenReturn(true);
    when(mockTransactionProcessor.beginTransaction()).thenReturn(transaction);
    when(mockCreditCardServer.initTransaction(transaction)).thenReturn(true);
    when(mockCreditCardServer.pay(transaction, creditCard, 500))
        .thenReturn(false);
    when(mockTransactionProcessor.endTransaction()).thenReturn(true);

    // Вызвать SUT
    paymentProcessor.processPayment(creditCard, Money.dollars(500));

    // Здесь невозможно определить, действительно ли метод pay() выполнил
    // транзакцию; тест может проверить только, вызывался ли метод pay()
    verify(mockCreditCardServer).pay(transaction, creditCard, 500);
}
```

В листинге 13.14 представлена улучшенная версия того же теста, не использующая заглушки. Обратите внимание, что тест получился короче и в нем нет деталей реализации (например, порядка использования процессора транзакций). Никаких специальных настроек не потребовалось, потому что сервер обработки кредитных карт знает, как себя вести.

Листинг 13.14. Реорганизованная версия теста без заглушек

```
@Test public void creditCardIsCharged() {
    paymentProcessor =
        new PaymentProcessor(creditCardServer, transactionProcessor);
```



```
// Вызвать SUT
paymentProcessor.processPayment(creditCard, Money.dollars(500));

// Запросить состояние сервера кредитных карт, чтобы определить,
// был ли выполнен платеж
assertThat(creditCardServer.getMostRecentCharge(creditCard))
    .isEqualTo(500);
}
```

Конечно, крайне нежелательно, чтобы такой тест взаимодействовал с внешним сервером кредитных карт, поэтому здесь использована имитация сервера. Если готовой имитации сервера нет, можно использовать его реальную реализацию, настроенную на взаимодействие с герметичным сервером кредитных карт, правда, это увеличит время выполнения тестов. (С герметичными серверами мы познакомимся в следующей главе.)

Когда использование заглушек оправданно?

Заглушки лучше использовать, когда нужно получить от функции определенное возвращаемое значение, чтобы привести SUT в определенное состояние, как показано в листинге 13.12, который требует, чтобы SUT вернула непустой список транзакций. Поскольку поведение функции определяется прямо в тесте, заглушка может эмулировать широкий спектр возвращаемых значений или ошибок, которые невозможно получить при использовании реальной реализации или имитации.

Чтобы более ясно отразить свою цель, каждая заглушка должна иметь прямую связь с утверждениями в тесте. Соответственно, тест должен использовать минимум заглушек, потому что большое количество заглушек в тесте может ухудшить его ясность и будет свидетельствовать о злоупотреблении заглушками или о том, что SUT слишком сложна и ее следует реорганизовать.

Обратите внимание, что даже в случаях, когда применение заглушек оправданно, предпочтительнее все же использовать реальные реализации или имитации, потому что они не раскрывают детали реализации и дают больше гарантий правильности кода по сравнению с заглушками. Но заглушки остаются разумной методикой тестирования, если используются ограниченно и не усложняют тесты.

Тестирование взаимодействий

Как обсуждалось выше в этой главе, тестирование взаимодействий помогает проверить, как вызывается функция, без фактического вызова ее реализации.

Применение фреймворков фиктивных объектов упрощает тестирование взаимодействий. Однако чтобы тесты были полезными, удобочитаемыми и устойчивыми к изменениям, тестирование взаимодействий следует использовать только при явной необходимости.

Тестируйте состояние, а не взаимодействия

Вместо взаимодействий лучше тестировать состояние (<https://oreil.ly/k3hSR>).

При тестировании состояния вы вызываете SUT и проверяете, вернула ли она правильное значение или правильно ли она изменила какое-то другое состояние. В листинге 13.15 показан тест, проверяющий состояние.

Листинг 13.15. Тестирование состояния

```
@Test public void sortNumbers() {
    NumberSorter numberSorter = new NumberSorter(quickSort, bubbleSort);

    // Вызвать SUT
    List sortedList = numberSorter.sortNumbers(newList(3, 1, 2));

    // Проверить, отсортирован ли список. Алгоритм сортировки не важен,
    // если возвращен правильный результат
    assertThat(sortedList).isEqualTo(newList(1, 2, 3));
}
```

В листинге 13.16 похожим образом протестированы взаимодействия. Обратите внимание, что в этом тесте невозможно определить, действительно ли числа отсортированы, потому что тестовые дублеры не знают, как выполнять сортировку, — они могут только сообщить, что SUT пыталась отсортировать числа.

Листинг 13.16. Тестирование взаимодействий

```
@Test public void sortNumbers_quickSortIsUsed() {
    // Передать в вызов конструктора тестовые дублеры,
    // созданные фреймворком фиктивных объектов
    NumberSorter numberSorter =
        new NumberSorter(mockQuickSort, mockBubbleSort);

    // Вызвать SUT
    numberSorter.sortNumbers(newList(3, 1, 2));

    // Убедиться, что numberSorter.sortNumbers() использовала алгоритм
    // быстрой сортировки (quickSort). Тест потерпит неудачу, если
    // mockQuickSort.sort() не будет вызвана (например, если будет
    // использована mockBubbleSort) или вызвана с неверными аргументами
    verify(mockQuickSort).sort(newList(3, 1, 2));
}
```

Мы в Google заметили, что тестирование состояния более масштабируемо: оно повышает надежность тестов, упрощает внесение изменений в код и легко поддерживается с течением времени.

Основная проблема тестирования взаимодействий состоит в том, что оно не дает уверенности, что SUT работает должным образом, и может только подтвердить, что определенные функции вызываются как следует. Это вынуждает нас строить пред-

положения о поведении кода, например: «Если вызывается `database.save(item)`, то мы предполагаем, что элемент `item` сохранен в базе данных». Тестирование состояния для нас предпочтительнее, потому что оно фактически проверяет сделанное предположение (например, сохраняет элемент в базе данных и затем получает его, чтобы узнать, действительно ли элемент сохранен).

Еще один недостаток тестирования взаимодействий — использование деталей реализации SUT. Например, чтобы убедиться, что функция была вызвана, вы должны сообщить тесту, какую именно функцию система должна вызвать. Так же как в случае с заглушками, дополнительный код делает тесты хрупкими из-за проникновения в тесты деталей реализации кода. Некоторые инженеры в Google в шутку называют тесты, которые злоупотребляют тестированием взаимодействий, *детекторами изменений* (<https://oreil.ly/zkMDu>), потому что они терпят неудачу в ответ на любое изменение в коде, даже если поведение SUT остается неизменным.

Когда тестирование взаимодействий оправданно?

Вот несколько случаев, когда тестирование взаимодействий оправданно:

- Иногда выполнить тестирование состояния невозможно, потому нельзя использовать реальную реализацию или имитацию (например, если реальная реализация слишком медленная или имитация отсутствует). В таких случаях как запасной вариант можно выполнить тестирование взаимодействий и убедиться, что определенные функции действительно вызываются. Это не идеальное решение, но оно хотя бы дает некоторую уверенность в том, что SUT работает должным образом.
- Различия в количестве или порядке вызовов функций могут стать причиной нежелательного поведения. В этом случае может пригодиться тестирование взаимодействий, потому что с помощью тестирования состояния трудно проверить такое поведение. Например, если ожидается, что функция кеширования сократит количество обращений к базе данных, тестирование взаимодействий покажет, действительно ли число обращений к базе данных не превысило ожидаемого количества. Если в тесте использовать фреймворк Mockito, его код будет выглядеть примерно так:

```
verify(databaseReader, atMostOnce()).selectRecords();
```

Тестирование взаимодействий не может полностью заменить тестирование состояния. Если невозможно выполнить тестирование состояния в юнит-тесте, настоятельно рекомендуем дополнить свой набор тестами с более широким охватом, выполняющими тестирование состояния. Например, если у вас есть юнит-тест, проверяющий использование базы данных посредством тестирования взаимодействий, добавьте к нему интеграционный тест, который проверит состояние на реальной базе данных. Расширенное тестирование является важной стратегией снижения рисков, и мы обсудим ее в следующей главе.

Передовые практики тестирования взаимодействий

Следование перечисленным ниже практикам тестирования взаимодействий помогает ослабить влияние вышеупомянутых недостатков.

Используйте тестирование взаимодействий только для функций, изменяющих состояние

Функции зависимостей, вызываемые SUT, относятся к одной из двух категорий:

Изменяющие состояние

Функции с побочными эффектами, наблюдаемыми за пределами SUT. Например: `sendEmail()`, `saveRecord()`, `logAccess()`.

Не изменяющие состояние

Функции, не имеющие побочных эффектов, которые возвращают информацию о мире за пределами SUT, но ничего не изменяют. Например: `getUser()`, `findResults()`, `readFile()`.

Как правило, тестирование взаимодействий следует выполнять только для функций, изменяющих состояние. Тестирование взаимодействий для функций, не изменяющих состояние, обычно излишне, поскольку SUT будет использовать возвращаемое значение функции для выполнения другого действия, которое можно проверить. Само взаимодействие не влияет на правильность кода, если оно не имеет побочных эффектов.

Тестирование взаимодействий для функций, не изменяющих состояние, делает тесты хрупкими, потому что эти тесты нужно обновлять при каждом изменении паттерна взаимодействий, и менее удобочитаемыми, потому что дополнительные инструкции затрудняют определение, какие инструкции важны для проверки правильности кода. Тестирование взаимодействий, изменяющих состояние, напротив, намного полезнее, потому что помогает проверить, что делает код для изменения состояния где-то еще.

Листинг 13.17 демонстрирует тестирование взаимодействий для функций, изменяющих и не изменяющих состояние.

Листинг 13.17. Взаимодействия, изменяющие и не изменяющие состояние

```
@Test public void grantUserPermission() {
    UserAuthorizer userAuthorizer =
        new UserAuthorizer(mockUserService, mockPermissionDatabase);
    when(mockPermissionService.getPermission(FAKE_USER)).thenReturn(EMPTY);

    // Вызвать SUT
    userAuthorizer.grantPermission(USER_ACCESS);

    // addPermission() изменяет состояние, поэтому для нее разумно выполнить
    // тестирование взаимодействий, чтобы убедиться, что она вызывается
    verify(mockPermissionDatabase).addPermission(FAKE_USER, USER_ACCESS);

    // getPermission() не изменяет состояние, поэтому эту строку кода можно
    // опустить. Вывод о бессмысленности тестирования взаимодействий можно
    // сделать по наличию заглушки для getPermission() выше в этом тесте
    verify(mockPermissionDatabase).getPermission(FAKE_USER);
}
```

Избегайте чрезмерного уточнения деталей

В главе 12 мы обсудили, почему полезнее тестировать поведение, а не методы. Мы выяснили, что тест должен проверять только один аспект поведения метода или класса, а не несколько сразу.

Тот же принцип желательно применять при выполнении тестирования взаимодействий и избегать чрезмерного уточнения таких деталей, как проверяемые функции и аргументы, чтобы повысить ясность и краткость кода, а также его устойчивость к изменениям в поведении SUT, которое выходит за рамки отдельного теста.

Листинг 13.18 иллюстрирует тестирование взаимодействий с чрезмерным уточнением деталей. Целью теста является проверка присутствия имени пользователя в тексте приветствия. Этот тест провалится, если изменится поведение, не связанное с полученным текстом.

Листинг 13.18. Тест взаимодействий с чрезмерным уточнением

```
@Test public void displayGreeting_renderUserName() {
    when(mockUserService.getUserName()).thenReturn("Fake User");
    userGreeter.displayGreeting(); // Вызов SUT

    // Тест провалится, если изменится какой-либо из аргументов setText()
    verify(userPrompt).setText("Fake User", "Good morning!", "Version 2.1");

    // Тест провалится, если setIcon() не будет вызвана, несмотря на то что
    // это поведение не имеет прямого отношения к тесту, потому что
    // не связано с проверкой имени пользователя
    verify(userPrompt).setIcon(IMAGE_SUNSHINE);
}
```

В листинге 13.19 показан подход к тестированию взаимодействий с более осторожным уточнением деталей в отношении аргументов и функций. Тестируемое поведение разделено на отдельные тесты, и каждый тест проверяет минимум из того, что необходимо проверить, чтобы убедиться в правильности тестируемого поведения.

Листинг 13.19. Тесты взаимодействий без чрезмерного уточнения деталей

```
@Test public void displayGreeting_renderUserName() {
    when(mockUserService.getUserName()).thenReturn("Fake User");
    userGreeter.displayGreeting(); // Вызов SUT
    verify(userPrompt).setText(eq("Fake User"), any(), any());
}

@Test public void displayGreeting_timeIsMorning_useMorningSettings() {
    setTimeOfDay(TIME_MORNING);
    userGreeter.displayGreeting(); // Вызов SUT
    verify(userPrompt).setText(any(), eq("Good morning!"), any());
    verify(userPrompt).setIcon(IMAGE_SUNSHINE);
}
```

Заключение

Мы узнали, что тестовые дублеры имеют решающее значение в ускорении разработки, потому что они помогают всесторонне тестировать код и выполнять тесты быстро. Но их неправильное использование может привести к значительному уменьшению продуктивности инженеров из-за ухудшения ясности, увеличения хрупкости и снижения эффективности тестов. Вот почему для инженеров важно иметь представление об эффективных практиках применения тестовых дублеров.

Часто невозможно точно сказать, когда лучше использовать реальную реализацию, а когда тестовый дублер или какую методику тестирования с дублерами использовать. Инженер должен оценить некоторые компромиссы при выборе правильного подхода в конкретном случае.

Тестовые дублеры отлично подходят для обхода зависимостей, которые трудно использовать в тестах, но если вы хотите максимально повысить доверие к тестовому коду, задействуйте эти зависимости в тестах. Следующая глава рассматривает крупномасштабное тестирование, в котором зависимости используются, даже если они делают тесты медленными или недетерминированными.

Итоги

- В тестах предпочтительнее использовать реальную реализацию.
- Если реальную реализацию невозможно использовать в тестах, примените ее имитацию.
- Злоупотребление заглушками делает тесты неясными и хрупкими.
- Желательно избегать тестирования взаимодействий: этот подход к тестированию делает тесты хрупкими, потому что в них проникают детали реализации SUT.

Крупномасштабное тестирование

*Автор: Джозеф Грейс
Редактор: Том Манирек*

В предыдущих главах мы рассказали, как в Google была создана культура тестирования и как маленькие юнит-тесты стали фундаментальной частью рабочего процесса инженера. Но как обстоят дела с другими видами тестирования? В Google используется большое количество более масштабных тестов, которые стали частью политики снижения рисков в программной инженерии. Но эти тесты создают дополнительные проблемы, и часто довольно сложно сделать их ценным активом, а не пожирателями ресурсов. В этой главе мы обсудим, что мы подразумеваем под «более масштабными тестами», когда мы их выполняем, и какие практики используем для повышения их эффективности.

Что такое большие тесты?

Как упоминалось выше, в Google используются свои определенные понятия о размере теста. Маленькие тесты ограничиваются одним потоком выполнения или одним процессом, средние — одной машиной. У больших тестов нет таких ограничений. Но в Google также есть понятие *широты охвата* теста. Юнит-тест всегда имеет более узкую область охвата, чем интеграционный. А тесты с самой большой областью охвата (иногда называемые сквозными, или системными, тестами) обычно вовлекают в работу несколько реальных зависимостей и почти не используют тестовые дублиеры.

Большие тесты в отличие от других видов тестов:

- могут работать медленно; время ожидания для наших больших тестов по умолчанию составляет от 15 минут до 1 часа, но у нас также есть тесты, которые выполняются несколько часов или даже дней;
- могут быть негерметичными; большие тесты могут использовать ресурсы совместно с другими тестами и обмениваться трафиком;
- могут быть недетерминированными; если большой тест негерметичен, то почти невозможно гарантировать его детерминированное поведение из-за влияния других тестов или данных, вводимых пользователем.

Так зачем нужны большие тесты? Подумайте о процессе программирования. Как убедиться, что программы действительно работают? Возможно, в процессе разработ-

ки вы пишете и запускаете юнит-тесты, но запускаете ли вы настоящий двоичный файл и пробуете ли его применить? А когда вы передаете свой код другим, как они проверяют его? Запускают свои юнит-тесты или пробуют его применить?

Кроме того, как узнать, что ваш код продолжает работать после внесения изменений? Предположим, вы создаете сайт, который использует Google Maps API. Чтобы выявить проблемы его совместимости с новой версией API, вы не станете создавать юнит-тесты, а запустите этот сайт и попробуете его применить к новой версии API, чтобы увидеть, не появились ли какие-нибудь нарушения в его работе.

Юнит-тесты могут дать уверенность в правильном поведении отдельных функций, объектов и модулей, а большие тесты дают уверенность в том, что вся система работает так, как задумано. Кроме того, автоматизированные тесты поддерживают возможность масштабирования, которая отсутствует в ручном тестировании.

Достоверность

Основная цель создания больших тестов — увеличение *достоверности* тестирования. Достоверность — это свойство теста, позволяющее ему отражать реальное поведение SUT.

Рассуждая о достоверности, полезно рассмотреть возможные градации тестовых сред. Как показано на рис. 14.1, юнит-тесты объединяют тестовый код и небольшую часть тестируемого кода в отдельную единицу (юнит), которая позволяет проверить код, но этот код выполняется в тесте совсем не так, как в продакшене. Продакшен, естественно, является самой точной средой для тестирования. Существует также целый спектр промежуточных вариантов сред для тестирования. Ключом к крупномасштабному тестированию является правильное определение его масштаба, потому что увеличение достоверности сопровождается ростом затрат и (в случае продакшена) увеличением риска отказа.



Рис. 14.1. Увеличение достоверности с ростом масштаба тестирования

Достоверность тестов также может измеряться по соответствию их содержимого действительности. Многие большие тесты, написанные вручную, отклоняются инженерами, если данные в них выглядят нереалистичными. Тестовые данные, скопированные из продакшена, намного ближе к реальности, но их применение подразумевает генерирование реалистичного тестового трафика перед запуском нового кода. Это особенно сложно сделать для искусственного интеллекта, в котором «начальные» данные часто страдают внутренней смещенностью. И поскольку большинство данных

для юнит-тестов готовится вручную, эти тесты охватывают лишь узкий диапазон возможных сценариев и имеют тенденцию соответствовать предубеждениям автора. Неохваченные сценарии из-за пропущенных данных снижают достоверность тестов.

Распространенные недостатки юнит-тестов

Большие тесты могут понадобиться там, где маленькие тесты не справляются со своими задачами. В следующих подразделах мы перечислим некоторые области, в которых юнит-тесты не обеспечивают достаточного для снижения рисков охвата тестирования.

Неточные дублеры

Один юнит-тест обычно охватывает один класс или модуль. Для избавления от тяжеловесных или труднотестируемых зависимостей в таких тестах часто используются тестовые дублеры (глава 13). Но когда зависимости заменяются дублерами, возникает вероятность появления несоответствий между поведением реальной реализации и поведением дублеров.

Почти все юнит-тесты в Google пишутся теми же инженерами, которые пишут тестируемый код. Когда возникает необходимость использовать дублеры в юнит-тестах и когда в роли этих дублеров выступают фиктивные объекты, инженер, пишущий юнит-тест, определяет фиктивный объект и его предполагаемое поведение. Но, как правило, этот инженер *не* пишет зависимости, для которых создает фиктивные объекты, и может неправильно понимать поведение реальных реализаций. Сходство между зависимостью и ее представлением в тесте дублер определяет поведенческим контрактом, и если инженер ошибается в определении фактического поведения зависимости, контракт становится недействительным.

Кроме того, фиктивные объекты могут устаревать. Если юнит-тест, основанный на использовании фиктивного объекта, не виден автору реальной реализации, то, когда реальная реализация изменится, автор теста не получит сигнала о том, что тест (и проверяемый код) следует обновить, чтобы учесть изменения.

Обратите внимание, что если команды создают фиктивные объекты для имитации своих собственных служб, то эта проблема обычно не возникает (глава 13).

Проблемы в конфигурациях

Юнит-тесты охватывают содержимое двоичного файла, который, как правило, не является полностью автономным с точки зрения выполнения. Обычно для двоичного файла определяется какая-то конфигурация развертывания или запускающий сценарий. При этом реальные экземпляры двоичных файлов, обслуживающие конечных пользователей, имеют свои файлы или базы данных с конфигурациями.

Если в этих файлах возникнут какие-то проблемы или состояние, определяемое ими, потеряет совместимость с состоянием, указанным в двоичном файле теста, пострадают конечные пользователи. Юнит-тесты не смогут самостоятельно выявить

такую несовместимость¹. Всегда храните конфигурацию в VCS вместе с кодом, чтобы при отказе изменения в конфигурации можно было идентифицировать как источник ошибки и встроить их проверку в большие тесты.

В Google изменения в конфигурации часто оказываются главной причиной масштабных сбоев. Мы поняли это не сразу и допустили ряд ошибок. Например, в 2013 году случился глобальный перерыв в работе Google из-за сбоя в конфигурации сети, которая никогда не проверялась. Конфигурации обычно пишутся на языках конфигурации, а не на языках программирования, на которых пишется код. Также они часто имеют более короткий цикл передачи в эксплуатацию, чем двоичные файлы, и их сложнее проверить. Все это увеличивает вероятность отказа. Но в этом случае (и в других) конфигурация хранилась в VCS, и мы смогли быстро выявить причину сбоя и устранить проблему.

Проблемы, возникающие под нагрузкой

Мы в Google требуем, чтобы юнит-тесты были маленькими и быстрыми, вписывались в нашу стандартную инфраструктуру тестирования, а также многократно запускались в ходе рабочего процесса инженера. Но для оценки производительности, устойчивости к нагрузкам и стресс-тестирования часто требуется отправлять большие объемы трафика в двоичный файл. Эти объемы трудно смоделировать в типовой модели юнит-тестирования. А в нашем понимании большие объемы — это тысячи и даже миллионы запросов в секунду (в рекламе и торгах в режиме реального времени (<https://oreil.ly/brV5->))!

Неожиданное поведение, входные данные и побочные эффекты

Юнит-тесты ограничены воображением инженера, пишущего их. То есть они могут проверять только ожидаемое поведение и входные данные. Однако пользователи чаще сталкиваются с непредвиденными проблемами (если бы это было не так, проблемы едва ли достигли бы конечных пользователей). Это говорит о том, что для проверки непредвиденного поведения необходимы разные методы тестирования.

Здесь важную роль играет закон Хайрама (<http://hyrumslaw.com>): даже если мы могли бы установить строгое соответствие системы указанному контракту, любое наблюдаемое поведение системы будет зависеть не только от заявленного контракта, но и от действий пользователя. Маловероятно, что одни лишь юнит-тесты смогут протестировать все наблюдаемые поведения, не указанные в общедоступном API.

Непредсказуемое поведение и «эффект вакуума»

Юнит-тесты имеют ограниченную область охвата (особенно при широком использовании тестовых дублеров) и не могут обнаружить изменение поведения, возникающее за ее пределами. А поскольку юнит-тесты предназначены для быстрой и надежной работы, они намеренно не включают хаос реальных зависимостей, сети и данных.

¹ Подробнее об этом в разделе «Непрерывная поставка» (глава 23) и в главе 25.

Юнит-тест подобен явлению из теоретической физики: он находится в вакууме, надежно изолирован от хаоса реального мира, имеет высокую скорость и надежность, но игнорирует некоторые категории дефектов.

Недостатки больших тестов

В предыдущих главах мы обсудили основные свойства удобного для разработчика теста. В частности, тест должен обладать следующими характеристиками:

Надежность

Он должен быть стабильным и обеспечивать полезный сигнал успех/неудача.

Высокая скорость

Он должен быть достаточно быстрым, чтобы не прерывать рабочий процесс инженера.

Масштабируемость

Все тесты должны эффективно выполняться как перед отправкой кода в репозиторий, так и в будущем.

Хорошие юнит-тесты обладают всеми этими свойствами. Большие тесты, напротив, часто не соответствуют этим требованиям. Они не всегда надежны, потому что полагаются на инфраструктуру, выполняются медленно и плохо масштабируются из-за более высоких требований к ресурсам и времени, а также они слабо изолированы и могут конфликтовать друг с другом.

Кроме того, большие тесты имеют две другие проблемы. Первая проблема связана с владением. Юнит-тестом явно владеет инженер (и команда), которому принадлежит модуль. Большой тест охватывает несколько модулей и, следовательно, может иметь нескольких владельцев. В долгосрочной перспективе неясно, кто отвечает за поддержку такого теста и диагностику проблем, когда тест терпит неудачу. Без четкого владения тест приходит в негодность.

Вторая проблема больших тестов — это стандартизация (или ее отсутствие). В отличие от юнит-тестов, большие тесты страдают отсутствием стандартизации с точки зрения инфраструктуры и процессов, с помощью которых они пишутся, выполняются и отлаживаются. Реализация больших тестов зависит от архитектурных решений, из-за чего возникают различия в типах требуемых тестов. Например, регрессионные A/B-тесты для Google Ads пишутся и выполняются совершенно иначе, чем аналогичные тесты для Google Search, и иначе, чем для Google Drive. Они используют разные платформы, разные языки, разные инфраструктуры, разные библиотеки и разные фреймворки тестирования.

Поскольку большие тесты могут выполняться множеством разных способов, их нередко игнорируют во время крупномасштабных изменений (глава 22). В инфраструктуре отсутствует стандартный способ выполнения больших тестов, и невозможно требовать от людей, выполняющих крупномасштабные изменения, знания особен-

ностей тестирования в каждой команде. Тесты, проверяющие интеграцию команд, могут потребовать объединения несовместимых инфраструктур. Из-за отсутствия стандартизации мы также не можем обучить сотрудников какому-то единому подходу к тестированию, что вызывает в инженерах устойчивое непонимание мотивов разработки больших тестов.

Большие тесты Google

Обсуждая историю развития тестирования в Google (глава 11), мы упоминали, как проект GWS внедрил автоматизированное тестирование в 2003 году и как это стало переломным моментом. Однако и до этого момента у нас имелись автоматизированные тесты, но в основном только большие и огромные. Например, в AdWords сквозной тест был создан еще в 2001 году. Он использовался для проверки разных сценариев использования продукта. В 2002 году в Google Search был написан аналогичный «регрессионный тест» для кода индексации, а в проекте AdSense (который еще даже не был официально запущен) был создан свой вариант теста AdWords.

Также примерно в 2002 году появились другие модели «больших» тестов. Пользовательский интерфейс Google Search в значительной степени опирался на ручную проверку качества — версии сквозных тестов, которые выполнялись вручную. В Gmail создали свою версию «локальной демонстрационной» среды для локального тестирования вручную — сценарий, генерирующий локальную среду для сквозного тестирования Gmail, включающую несколько сгенерированных тестовых учетных записей с почтовыми данными.

Когда был выпущен C/J Build — наш первый фреймворк непрерывной сборки, он не проводил различий между юнит-тестами и другими тестами до того, как произошли два важных события. Во-первых, компания Google сосредоточилась на юнит-тестах, чтобы сформировать правильную пирамиду тестирования и добиться распространения юнит-тестирования. Во-вторых, TAP, заменившая C/J Build в роли нашей официальной системы непрерывной сборки, поддерживала тесты только с определенными характеристиками: герметичные, создаваемые отдельно для каждого изменения и выполняемые в кластере сборки/тестирования в течение ограниченного времени. Требованиям TAP удовлетворяли большинство юнит-тестов, но это не устранило потребность в других видах тестирования, и большие тесты продолжали заполнять пробелы в охвате. Мы поддерживали C/J Build еще несколько лет специально для больших тестов, пока не появились более новые системы.

Большие тесты и время

На протяжении всей книги мы рассматриваем влияние времени на программную инженерию, потому что Google продолжает создавать ПО более 20 лет. Как большие тесты зависят от измерения времени? Мы знаем, что с увеличением ожидаемого срока службы кода некоторые действия приобретают особую значимость, а тестирование

должно сопровождать развитие систем всегда, но виды тестов меняются в течение этого срока.

Как отмечалось выше, юнит-тесты приобретают смысл, если ожидаемая продолжительность жизни ПО превышает несколько часов. Для кода со сроком жизни, измеряемом минутами (некоторые небольшие сценарии), более распространено ручное тестирование, и в роли SUT, которая обычно выполняется локально, *выступает* продакшен (особенно в одноразовом сценарии). Для кода с более длительным сроком жизни тоже используется ручное тестирование, но оно применяется к другой системе, отличной от продакшена, потому что часто реальный экземпляр такой системы размещается в облаке.

Остальные большие тесты полезны для более долгоживущего ПО и требуют сложного сопровождения.

Кстати, зависимость вида тестирования от срока действия тестируемого кода является одной из причин появления антипаттерна «рожок мороженого» (глава 11):



Рис. 14.2. Антипаттерн тестирования «рожок мороженого»

Когда в начале разработки используется в основном ручное тестирование (если инженеры считают, что код просуществует всего несколько минут), ручные тесты накапливаются и начинают доминировать в наборе тестов. Например, часто используется такой подход: изменить сценарий (или приложение), запустить и протестировать его, а затем добавлять в него новые возможности и тестировать их вручную. Созданный таким способом прототип позднее становится работоспособным и начинает использоваться другими инженерами, но для него нет автоматических тестов.

Хуже того, если код с трудом поддается юнит-тестированию (в первую очередь из-за особенностей его реализации) и единственные автоматизированные тесты, которые

можно для него написать, являются сквозными, это значит, что мы случайно создали «устаревший код».

В долгосрочной перспективе *очень важно* получить правильную пирамиду тестирования: написать юнит-тесты в первые дни разработки, а затем добавить автоматические интеграционные тесты и полностью избавиться от сквозного тестирования вручную. Мы смогли сделать обязательным юнит-тестирование перед отправкой кода в репозиторий, но в долгосрочной перспективе также важно заполнить пробел между юнит-тестами и ручными тестами.

Большие тесты в масштабе Google

Казалось бы, большие тесты больше соответствуют крупномасштабному ПО, но даже если это так, сложность создания, запуска, сопровождения и отладки этих тестов растет быстрее, чем сложность юнит-тестов.

В системе, состоящей из микросервисов или отдельных серверов, схема взаимодействий имеет вид графа, число узлов которого равно N . Каждый раз, когда в этот граф добавляется новый узел, число различных путей через него растет в геометрической прогрессии.

На рис. 14.3 изображена воображаемая SUT. Она включает социальную сеть с пользователями (граф), через которую распространяются поток сообщений и рекламные объявления от рекламодателей, пересылаемые в потоке сообщений. Эта система включает две группы пользователей, два пользовательских интерфейса, три базы данных, конвейер индексирования и шесть серверов. В графе есть 14 ребер. Тестирование всех возможных путей в этом графе уже затруднено, а что произойдет, если добавить в него дополнительные услуги, конвейеры и базы данных: фотографии и изображения, анализ фотографий с помощью машинного обучения и т. д.?

Количество различных сценариев для сквозного тестирования может возрасти экспоненциально или даже комбинаторно в зависимости от структуры SUT, и этот рост не масштабируется. Как следствие, с ростом системы мы вынуждены искать альтернативные, более масштабируемые политики тестирования, чтобы сохранить управляемость.

Ценность масштабируемых тестов тоже увеличивается с ростом системы из-за потребности в достоверности: по мере движения к более глубоким слоям, когда службы тестируются с использованием дублеров, имеющих более низкую достоверность ($1 - \epsilon$), вероятность появления ошибок при их объединении изменяется экспоненциально в зависимости от глубины слоя N . Вернемся к примеру SUT: если заменить серверы пользователей и рекламы дублерами с менее высокой достоверностью (например, с 10%-ной неточностью), вероятность ошибки составит 99 % (то есть $1 - (0,1 \times 0,1)$). И это только с двумя менее достоверными дублерами.

Соответственно, важно реализовать большие тесты так, чтобы они хорошо работали в интересующем масштабе, но поддерживали достаточно высокую достоверность.

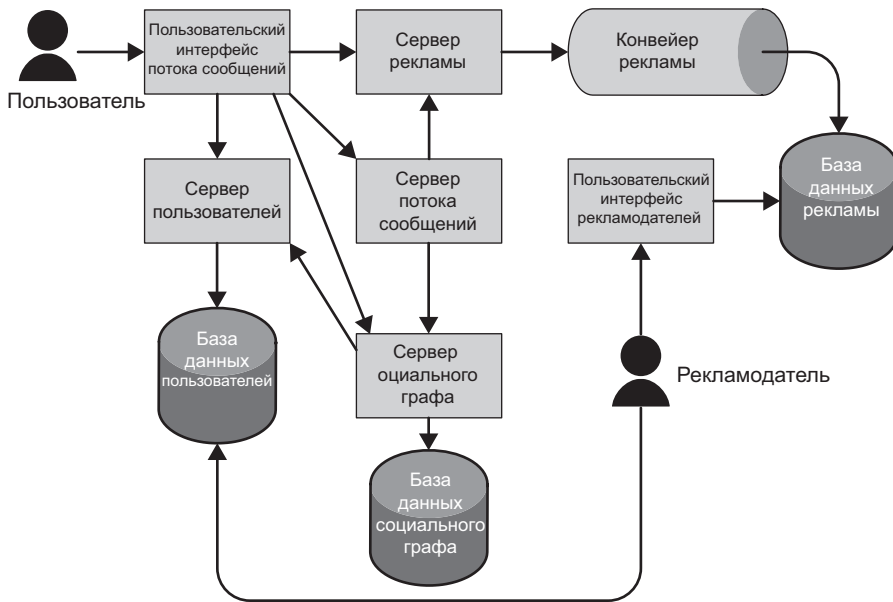


Рис. 14.3. Пример чрезвычайно маленькой SUT: социальная сеть с поддержкой рекламы

СОВЕТ: «МИНИМАЛЬНО ВОЗМОЖНЫЙ ТЕСТ»

Даже в интеграционном тестировании чем меньше тест, тем лучше, — несколько больших тестов лучше одного огромного. И поскольку область охвата теста часто зависит от области охвата SUT, поиск способов уменьшения SUT помогает сделать тесты меньше.

Один из способов достичь уменьшения тестов, например в тестировании пользовательского пути, который может потребовать участия многих внутренних систем, состоит в том, чтобы связать тесты в «цепочку», представляющую общий сценарий (рис. 14.4). Для этого выходные данные одного теста нужно передать на вход другого путем сохранения этих данных в хранилище.

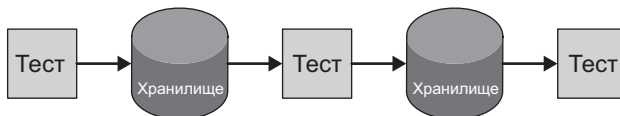


Рис. 14.4. Цепочка тестов

Структура большого теста

Большие тесты не связаны ограничениями, которые накладываются на маленькие тесты, и могут включать все что угодно, но они тоже подчиняются некоторым общим

закономерностям. Обычно большие тесты образуют рабочий процесс со следующими этапами:

- получение SUT;
- генерирование необходимых тестовых данных;
- выполнение операций с SUT;
- проверка результатов.

SUT

Одним из ключевых компонентов больших тестов является вышеупомянутая SUT (рис. 14.5). Типичный юнит-тест сосредоточен на одном классе или модуле. Более того, код теста выполняется в том же процессе (или виртуальной машине Java [JVM], если написан на Java), что и тестируемый код. Большие тесты, в свою очередь, включают один или несколько тестируемых процессов и код теста, который часто (но не всегда) выполняется в своем собственном процессе.

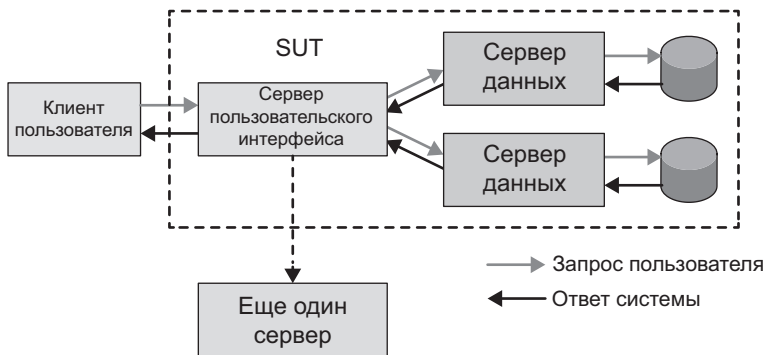


Рис. 14.5. Пример SUT

Мы в Google используем разные виды SUT, область охвата которых влияет на охват соответствующих больших тестов (чем больше SUT, тем больше тест, проверяющий ее). Каждую разновидность SUT можно оценить по двум основным факторам:

Герметичность

Степень изоляции от использования другими компонентами среды (помимо теста) и взаимодействий с ними. SUT с высокой герметичностью меньше подвержена проблемам, связанным с конкурентным выполнением и нестабильностью инфраструктуры.

Достоверность

Степень точности отражения продакшена. SUT с высокой достоверностью будет состоять из двоичных файлов, напоминающих реальные реализации (ис-

пользующие схожие конфигурации и инфраструктуру и имеющие общую с SUT топологию).

Часто эти два фактора вступают в прямой конфликт друг с другом.

Вот несколько примеров SUT:

SUT с единственным процессом

Вся SUT упакована в один двоичный файл (даже если реальная версия состоит из нескольких файлов). Код теста может быть упакован в тот же двоичный файл. Такая комбинация кода теста и SUT может считаться маленьким тестом, если весь код выполняется в одном потоке, но она недостоверно отражает топологию и конфигурацию продакшена.

SUT с единственной машиной

SUT состоит из одного или нескольких двоичных файлов (как и продакшен), а код теста находится в своем двоичном файле. И система, и тест выполняются на одной машине. Этот подход применяется для тестов среднего размера. В этом случае желательно использовать реальную конфигурацию запуска для каждого двоичного файла для более высокой достоверности.

SUT с несколькими машинами

SUT распределена по нескольким машинам (подобно развертыванию продакшена в облаке). Этот вариант дает еще более высокую достоверность, чем вариант с единственной машиной, но делает тесты большими и, следовательно, нестабильными из-за возможных ошибок в сети.

Общая среда (промежуточная среда и продакшен)

Вместо запуска отдельной SUT тест использует общую среду. Этот вариант имеет самую низкую стоимость, потому что часто общая среда уже существует, но тест может конфликтовать с процессами, выполняемыми параллельно, и копирование тестируемого кода в среду требует времени. Тестирование в продакшене также увеличивает риск отрицательного влияния на конечного пользователя.

Гибриды

Комбинация предыдущих вариантов: запуск части SUT при ее взаимодействии с общей средой. Обычно тестируемый объект запускается явно, но его серверная часть общедоступна. Такая большая компания, как Google, не может запустить несколько копий всех взаимосвязанных серверов, поэтому нам приходится использовать гибридизацию в той или иной форме.

Преимущества герметичных SUT

SUT может быть основным источником ненадежности и низкой скорости выполнения большого теста. Использование продакшена в тесте не требует дополнительных накладных расходов на развертывание SUT, но тестирование не начинается, пока тестируемый код не будет полностью размещен в среде. Поскольку тесты не могут

блокировать распространение кода в продакшене, само тестирование в этом случае осуществляется слишком поздно.

Наиболее распространенная альтернатива — создание гигантской общей промежуточной среды для выполнения тестов. Обычно этот подход является частью продвижения новой версии, но, опять же, выполнение теста возможно, только когда тестируемый код находится в среде. Некоторые команды позволяют инженерам «резервировать» временное окно в промежуточной среде для развертывания и тестирования нового кода, но этот вариант не масштабируется с ростом числа инженеров или тестируемых служб, потому что вероятность конфликтов между ними тоже возрастает.

Другой подход — поддержка облачных или машинно-герметичных SUT. Такие среды позволяют избежать конфликтов и не требуют резервировать время для развертывания кода.

ПРИМЕР: РИСКИ ТЕСТИРОВАНИЯ В ПРОДАКШЕНЕ И WEBDRIVER TORSO

Выше мы сказали, что тестирование в продакшене — рискованное предприятие. Однажды во время такого тестирования произошел забавный инцидент с Webdriver Torso. Мы искали возможность проверить правильность отображения видео в продакшене YouTube: создали сценарии, автоматически генерирующие тестовые видеоролики, выгрузили их и приступили к проверке качества. Для этого мы использовали общедоступный канал на YouTube под названием Webdriver Torso, принадлежащий Google. Но этот канал был общедоступным, как и большинство видеороликов.

Впоследствии этот канал был упомянут в статье на Wired (<https://oreil.ly/1KxVn>), что привело к появлению множества публикаций о нем в СМИ и последующим попыткам разгадать его тайну. Наконец, один блогер (https://oreil.ly/ko_kv) связал происходящее с Google. Тогда мы рассказали правду, немало повеселившись, в том числе о рикроллинге и «пасхалках», так что все закончилось как нельзя лучше. Этот инцидент заставил нас задуматься о возможности обнаружения конечными пользователями тестовых данных, которые мы добавляем в продакшен, и быть готовыми к этому.

Ограничение размера SUT

В тестировании есть опасные границы, за которые иногда не стоит выходить. Тесты, проверяющие не только внешние, но и внутренние интерфейсы, становятся слишком болезненными, потому что, как известно, тесты пользовательского интерфейса ненадежны и дороги по двум причинам:

- оформление пользовательских интерфейсов часто меняется (без изменений в их базовом поведении), что делает тесты пользовательского интерфейса хрупкими;
- пользовательские интерфейсы часто имеют асинхронное поведение, которое сложно тестировать.

Иногда полезно проводить сквозное тестирование пользовательского интерфейса вплоть до внутренней реализации службы, но при этом тесты будут иметь мульт-

типичативную стоимость обслуживания. Поэтому если внутренняя реализация имеет общедоступный API, проще разделить тесты на связанные группы по границе, разделяющей пользовательский интерфейс и API, и использовать общедоступный API для сквозного тестирования. Такой подход уместен и для пользовательского веб-интерфейса, и для интерфейса командной строки (CLI, command-line interface), и для обычных или мобильных приложений.

Еще одна особая граница отделяет сторонние зависимости. Сторонние системы могут не иметь общедоступной среды для тестирования, а в некоторых случаях отправка трафика таким системам обходится слишком дорого. Поэтому мы не рекомендуем использовать настоящий сторонний API в автоматизированных тестах и проводить границу тестирования по сторонним зависимостям.

Для решения проблем, связанных с размером теста, мы уменьшаем SUT, заменяя ее базы данных базами данных в памяти и исключая один из серверов, находящийся за рамками SUT (рис. 14.6). Такая SUT больше подходит для тестирования на одной машине.

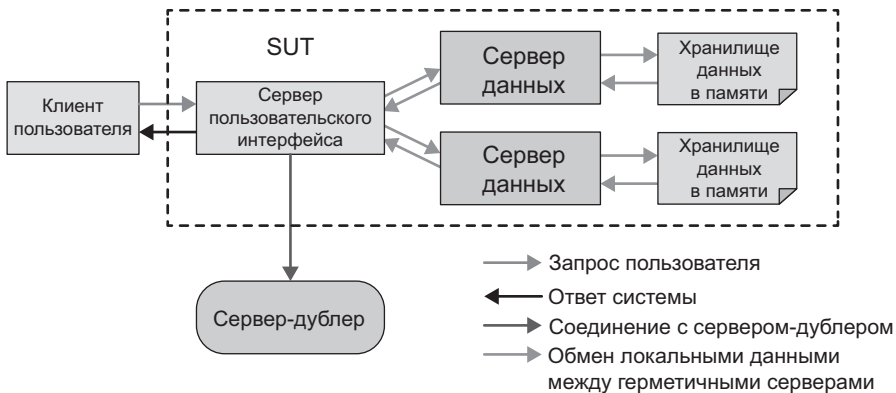


Рис. 14.6. SUT уменьшенного размера

Ключом к улучшению качества тестирования является компромисс между достоверностью и стоимостью/надежностью, а также определение разумных границ. Организовав возможность запускать несколько двоичных файлов и тестов на тех же машинах, где выполняются компиляция, компоновка и юнит-тестирование, мы получаем самые простые и стабильные интеграционные тесты.

Запись/воспроизведение

В предыдущей главе мы обсудили приемы использования тестовых дублеров и способы отделения тестируемого класса от его сложных для тестирования зависимостей. Однако можно создавать дублеры для целых серверов и процессов, используя фиктивные версии, заглушки или имитации серверов или процессов с эквивалентным API. Правда, при этом нет никакой гарантии, что такой тестовый дублер будет соответствовать контракту подменяемого реального сервера или процесса.

Тестовый дублер помогает справиться с проблемой зависимостей SUT, но как узнать, действительно ли он отражает реальное поведение зависимости? За пределами Google все большую популярность приобретает фреймворк тестирования контрактов (<https://oreil.ly/RADVJ>), в котором клиент определяет шаблон поведения службы, сообщая, что в ответ на некие входные данные он ожидает получить конкретный результат. Затем служба использует эту пару входных/выходных данных в тесте, который показывает, соответствует ли результат ожиданиям. Есть два общедоступных инструмента для тестирования контрактов: Pact Contract Testing (<https://docs.pact.io>) и Spring Cloud Contracts (<https://oreil.ly/szQ4j>), но сильная зависимость Google от Protocol Buffers не позволяет использовать их внутри компании.

В самом популярном в Google подходе (<https://oreil.ly/-wvYi>) (для которого существует общедоступный API) большой тест при запуске генерирует меньший тест, записывая трафик к внешним службам и воспроизводя этот трафик при выполнении меньших тестов. Большой тест, или «Режим записи», выполняется непрерывно после отправки кода в репозиторий, его главная цель — сгенерировать журналы трафика (для этого он должен завершиться успехом). Меньший тест, или «Режим воспроизведения», используется во время разработки и предварительного тестирования.

Один из интересных аспектов этого механизма записи/воспроизведения заключается в необходимости использования специальных средств, чтобы определить, какой результат послать в ответ на те или иные входные данные. Эти средства похожи на заглушки и фиктивные объекты, которые тоже используют сопоставление аргументов для выбора результирующего поведения.

Что происходит с новыми тестами или тестами, в которых поведение клиента существенно меняется? В этих случаях вход может перестать соответствовать содержимому записанного файла трафика, поэтому тест в режиме воспроизведения завершится неудачей и инженер должен будет запустить тест в режиме записи, чтобы сгенерировать новый трафик (важно, чтобы тесты записи были простыми, быстрыми и стабильными).

Тестовые данные

Тестам нужны данные, а большим тестам нужны данные двух типов:

Начальные данные

Данные, предварительно сгенерированные в SUT и отражающие состояние системы в начале теста.

Тестовый трафик

Данные, посылаемые тестом в SUT во время выполнения.

Сгенерировать начальное состояние больших и изолированных SUT часто на несколько порядков сложнее, чем настроить юнит-тест, поскольку начальное состояние включает:

Конфигурационные данные

Иногда данные, используемые для настройки среды, хранятся в специальных таблицах базы данных. Действительные двоичные файлы службы обращаются к базе данных и могут не запуститься, если искомые данные в ней отсутствуют.

Реалистичные базовые данные

Чтобы приблизить SUT к реальности, может потребоваться реалистичный по количеству и качеству набор базовых данных. Например, большому тесту социальной сети наверняка понадобится реалистичный социальный граф, в котором должно быть достаточно много тестовых пользователей с реалистичными профилями, а также достаточное количество взаимосвязей между этими пользователями.

API для заполнения

API для передачи начальных данных могут быть очень сложными. Иногда возможна прямая запись данных в хранилище, но она не задействует триггеры и проверки, выполняемые фактическими двоичными файлами во время записи.

Данные могут быть сгенерированы разными способами:

Вручную

Для подготовки данных для нескольких служб в большой SUT вручную может потребоваться много усилий.

Копирование

Данные можно скопировать из продакшена. Например, при тестировании географической карты проверка изменений проводится на скопированных данных из продакшена.

Выборка

При копировании можно получить слишком много данных. Выборка данных поможет уменьшить их объем и тем самым сократить время тестирования и облегчить анализ результатов. «Интеллектуальная выборка» — это метод копирования минимального количества данных, необходимых для достижения максимального охвата.

Проверка

После запуска SUT и передачи трафика мы должны проверить ее поведение. Сделать это можно несколькими способами:

Вручную

Так же как при тестировании локального двоичного файла, проверка вручную предполагает взаимодействие человека с SUT. Это может быть регрессионное тестирование в соответствии с согласованным планом или исследовательское тестирование, при котором проверяются различные сценарии взаимодействия для выявления возможных новых сбояв.

Обратите внимание, что регрессионное тестирование вручную не масштабируется линейно: чем больше размер системы и чем больше сценариев взаимодействия с ней, тем больше человеко-часов необходимо затратить на ее тестирование вручную.

С использованием утверждений

Так же как в юнит-тестах, утверждения в более крупных тестах явно проверяют предполагаемое поведение системы. Например, в интеграционном тесте для Google Search, проверяющем поиск по строке *xuzzy*, утверждение может выглядеть так:

```
assertThat(response.Contains("Colossal Cave"))
```

A/B-сравнение (отличий)

A/B-тестирование предполагает запуск двух копий SUT, отправку им одних и тех же данных и сравнение полученных результатов. Предполагаемое поведение в этом случае не определяется явно: инженер должен вручную отыскать все отличия и убедиться, что они ожидаемы.

Типы больших тестов

Теперь объединим разные подходы к организации SUT, сбору данных и проверкам и перейдем к созданию больших тестов. Каждый тест имеет разные особенности, касающиеся смягчаемых рисков, трудозарат на разработку, отладку и сопровождение, а также ресурсов, необходимых для запуска.

Ниже мы привели список разных видов больших тестов, используемых в Google, и описали, как они конструируются, для чего служат и какие ограничения имеют:

- функциональное тестирование одного или нескольких двоичных файлов;
- тестирование в браузерах и на мобильных устройствах;
- тестирование производительности, нагрузочное тестирование и стресс-тесты;
- тестирование конфигурации развертывания;
- исследовательское тестирование;
- A/B-тестирование различий (регрессионное);
- приемочное тестирование пользователем (UAT, user acceptance testing);
- зонды и канаречный анализ;
- восстановление после аварии и проектирование хаоса;
- оценка действий пользователей.

Как управлять таким большим количеством комбинаций и, соответственно, широким спектром тестов? Частью проектирования ПО является составление плана тестирования, а ключевой частью плана тестирования является стратегическое описание необходимых видов тестирования и тестов каждого вида. Это описание определяет основные риски и подходы, помогающие их смягчить.

В Google есть специальная должность «инженер-тестировщик», и один из навыков хорошего инженера-тестировщика — умение наметить стратегию тестирования наших продуктов.

Функциональное тестирование одного или нескольких двоичных файлов

Тесты этого типа имеют следующие характеристики:

- SUT: единственная герметичная машина или изолированное развертывание в облаке;
- данные: подготавливаются вручную;
- проверка: с использованием утверждений.

Как мы уже видели, юнит-тесты не способны протестировать сложную систему с полной достоверностью, потому что они оформлены совсем не так, как настоящий код. Многие сценарии функционального тестирования взаимодействуют с двоичным файлом иначе, чем юнит-тесты с классами внутри этого двоичного файла. Для функционального тестирования необходимы отдельные SUT, и по этой причине они считаются каноническими большими тестами.

Тестирование взаимодействий между несколькими двоичными файлами сложнее, чем тестирование одного двоичного файла. В среде с микросервисами, в которой службы развертываются в виде нескольких двоичных файлов, функциональный тест может охватывать реальные взаимодействия между двоичными файлами, обращаясь к SUT, состоящей из всех необходимых двоичных файлов, и взаимодействуя с ней через общедоступный API.

Тестирование в браузерах и на мобильных устройствах

Тестирование веб-интерфейсов и мобильных приложений — это вид функционального тестирования одного или нескольких взаимодействующих двоичных файлов. Для конечных пользователей общедоступным API является само приложение. Чтобы обеспечить большой охват тестирования, недостаточно выполнить юнит-тестирование кода реализации и нужно использовать тесты, взаимодействующие с приложением через его внешний интерфейс.

Тестирование производительности, нагрузочное тестирование и стресс-тесты

Тесты этого типа имеют следующие характеристики:

- SUT: изолированное развертывание в облаке;
- данные: подготавливаются вручную или импортируются из продакшена;
- проверка: различий (параметров производительности).

Иногда можно провести тестирование производительности, нагрузочное тестирование и стресс-тестирование небольшого модуля, но чаще такое тестирование требует одновременной отправки трафика внешнему API. Это означает, что тесты этого типа

являются многопоточными и их охват обычно ограничивается тестируемым двоичным файлом. Тем не менее только они могут выявить снижение производительности в новых версиях и гарантировать, что система справится с ожидаемыми пиками трафика.

С увеличением масштаба нагрузочного теста увеличивается объем входных данных, и в какой-то момент становится трудно генерировать высокую нагрузку, чтобы вызвать ошибку. Способность справляться с высокой нагрузкой — это свойство системы, а не отдельных ее компонентов. Поэтому важно, чтобы тесты были максимально приближены к продакшену. Каждой SUT необходим тот же объем ресурсов, что и продакшену, чтобы уменьшить шум от повторения топологии продакшена.

Одним из способов устранения шума в тестах производительности является изменение топологии развертывания — распределения различных двоичных файлов в сети компьютеров. Машина, на которой выполняется двоичный файл, может влиять на характеристики производительности. Если в тесте, оценивающем различия в производительности, базовая версия работает на быстром компьютере (или на компьютере с быстрой сетью), а новая версия — на медленном, то результат может выглядеть как снижение производительности. Поэтому лучше запускать обе версии на одном и том же компьютере. Если одна и та же машина не может использоваться для тестирования обеих версий двоичного файла, калибруйте результаты, выполнив несколько прогонов и удалив пики и провалы.

Тестирование конфигурации развертывания

Тесты этого типа имеют следующие характеристики:

- SUT: единственная герметичная машина или изолированное развертывание в облаке;
- данные: нет;
- проверка: с использованием утверждений (без сбоев).

Часто источником дефектов является не код, а конфигурация: файлы данных, базы данных, определения параметров и т. п. Большие тесты могут проверить интеграцию SUT с файлами конфигурации, потому что эти файлы читаются во время запуска двоичного файла.

На самом деле такие тесты являются проверкой «на дым» — они не требуют большого количества дополнительных данных или уточнений. Если SUT запускается, тест считается пройденным, в противном случае — не пройденным.

Исследовательское тестирование

Тесты этого типа имеют следующие характеристики:

- SUT: продакшен или общая промежуточная среда;
- данные: импортируются из продакшена или из известного теста;
- проверка: вручную.

Исследовательское тестирование¹ — это разновидность ручного тестирования, в котором все внимание сосредоточено не на поиске изменений в поведении (при повторном тестировании известных сценариев использования), а на выявлении сомнительного поведения (при использовании новых сценариев). Обученные пользователи/тестировщики взаимодействуют с продуктом через общедоступный API, пытаются найти новые сценарии использования системы, в которых ее поведение отличается от ожидаемого или интуитивно предполагаемого, а также уязвимости в системе безопасности.

Исследовательское тестирование полезно как для новых, так и для давно действующих систем. Оно позволяет выявить неожиданное поведение и побочные эффекты. Благодаря тому что тестировщики следуют разными путями, доступными в системе, они могут увеличить охват системы и при выявлении ошибки создать новые автоматизированные функциональные тесты. Эта разновидность тестирования напоминает ручное «нечеткое тестирование» — своеобразную версию интеграционного функционального тестирования.

Ограничения

Ручное тестирование не масштабируется линейно; то есть для выполнения ручных тестов требуется, чтобы человек тратил свое время. Поэтому для любых дефектов, обнаруженных в ходе исследовательского тестирования, должны быть написаны автоматические тесты, которые можно будет запускать гораздо чаще.

Охота за ошибками

Одним из распространенных подходов, которые мы используем для ручного исследовательского тестирования, является охота за ошибками (bug bashes, <https://oreil.ly/zRLyA>). Команда инженеров и связанные с ней специалисты (менеджеры, менеджеры по продукту, инженеры-тестировщики — все, кто знаком с продуктом) планируют «встречу», на которой тестируют продукт вручную и озвучивают рекомендации относительно конкретных областей и/или отправных точек, заслуживающих внимания. Цель таких встреч в том, чтобы опробовать разные варианты взаимодействия для документирования случаев сомнительного поведения продукта и явных ошибок.

Регрессионное А/В-тестирование

Тесты этого типа имеют следующие характеристики:

- SUT: два изолированных развертывания в облаке;
- данные: обычно импортируются из продакшена или методом выборки;
- проверка: А/В-сравнение.

Юнит-тесты охватывают ожидаемые пути выполнения в небольшом фрагменте кода. Но ни для какого продукта невозможно предсказать все или большинство возможных вариантов сбоев. Кроме того, как гласит закон Хайрама, действительный общедо-

¹ *Whittaker J. A. Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design.* New York: Addison-Wesley Professional, 2009.

ступный API — это не публично объявленные, а все видимые пользователю аспекты продукта. Поэтому неудивительно, что А/В-тесты являются наиболее распространенной формой крупномасштабного тестирования в Google. Идея А/В-сравнения зародилась еще в 1998 году, и уже с 2001 года в Google его начали проводить для большинства наших продуктов, начиная с Ads, Search и Maps.

В процессе выполнения А/В-тесты посылают трафик общедоступному API и сравнивают ответы старой и новой версий (что особенно важно для миграций). Любые отклонения в поведении идентифицируются как ожидаемые или непредвиденные (регрессия). В этом случае SUT состоит из двух наборов реальных двоичных файлов: один представляет версию-кандидат, а другой — действующую базовую версию. Третий двоичный файл — это тест, который посылает трафик и сравнивает результаты.

Есть и другие варианты этого подхода. А/А-тестирование (сравнение системы с самой собой) выявляет недетерминированное поведение, шум и нестабильность для их исключения из результатов А/В-тестирования. А/В/С-тестирование проводит сравнение последней рабочей версии, базовой сборки и новых изменений, чтобы показать не только влияние самих изменений, но и их последствия, которые проявятся в следующей версии.

А/В-тесты — это недорогой и легко автоматизируемый способ выявления неожиданных побочных эффектов в любой действующей системе.

Ограничения

Тестирование различий сопряжено с несколькими сложностями, которые необходимо преодолеть:

Оценка результатов

Кто-то должен понимать результаты, чтобы распознавать ожидаемые различия.

При тестировании различий не всегда ясно, являются ли различия хорошими или плохими (и действительно ли базовая версия действует правильно), поэтому А/В-тестирование включает шаги, выполняемые вручную.

Шум

Все, что вносит дополнительный шум в процесс тестирования различий, требует ручного исследования результатов. Это необходимо для устранения шума и является одним из факторов, осложняющих создание хороших тестов различий.

Охват

Создание трафика, пригодного для тестирования различий, сложно готовить вручную. Тестовые данные должны охватывать широкий круг сценариев использования, чтобы помочь выявить редкие различия.

Настройки

Настройка и поддержка даже одной SUT довольно сложны. Создание сразу двух таких систем может удвоить сложность, особенно если они содержат взаимозависимости.

Приемочное тестирование пользователем

Тесты этого типа имеют следующие характеристики:

- SUT: единственная герметичная машина или изолированное развертывание в облаке;
- данные: подготавливаются вручную;
- проверка: с использованием утверждений.

Юнит-тест пишет автор тестируемого кода. Это обстоятельство увеличивает вероятность, что недопонимание *предполагаемого* поведения продукта отразится не только на коде, но и на юнит-тестах: тест поможет убедиться, что код работает так, «как реализовано», а не «как задумано».

Если у кода есть заказчик или представитель заказчика (коллегиальный орган или менеджер по продукту), полезно создать приемочные тесты — автоматические тесты, проверяющие продукт посредством общедоступного API на соответствие общего поведения определенным сценариям использования (<https://oreil.ly/IOaOq>). Есть несколько общедоступных фреймворков (например, Cucumber и RSpec), позволяющих создавать такие тесты на удобном для пользователя языке «выполняемых спецификаций».

В Google для разработки автоматизированных приемочных тестов редко используются языки спецификаций. Многие продукты Google созданы инженерами-программистами, поэтому нет необходимости применять языки выполняемых спецификаций, если те, кто определяет предполагаемое поведение продукта, свободно владеют языками программирования.

Зонды и канареечный анализ

Тесты этого типа имеют следующие характеристики:

- SUT: продакшен;
- данные: продакшен;
- проверка: с использованием утверждений и метрик A/B-тестирования.

Зонды и канареечный анализ позволяют убедиться в нормальном функционировании продакшена. Они являются формой мониторинга, но структурно очень похожи на другие большие тесты.

Зонды — это функциональные тесты, которые используют утверждения, касающиеся характеристик работы продакшена. Обычно эти тесты выполняют хорошо известные и детерминированные операции, осуществляющие доступ только для чтения, поэтому утверждения остаются действительными даже при изменении данных в продакшене. Например, зонд может обратиться к службе Google Search по адресу www.google.com и убедиться, что она возвращает результат, не проверяя его содержимого. В этом отношении зонды являются проверками «на дым», но при этом помогают выявлять основные проблемы продукта на ранних этапах разработки.

Канареечный анализ действует аналогично зонду, но выполняется, когда новая версия отправляется в продакшен. Если новая версия внедряется в производство поэтапно, есть возможность протестировать обновленные (канареечные) службы с помощью зондов и сравнить показатели работы канареечных и базовых частей продакшена, чтобы убедиться, что обновленные версии функционируют нормально.

Зонды должны использоваться в любой действующей системе. Если процесс развертывания в продакшене включает этап, в ходе которого двоичный файл развертывается на ограниченном подмножестве реальных серверов (этап канареечного опробования), то канареечный анализ должен использоваться на этом этапе.

Ограничения

Любые проблемы, обнаруженные этими тестами (в продакшене), уже затронули конечных пользователей.

Зонд, выполняющий действие, которое изменяет данные (запись), изменит состояние продакшена. Это может привести к одному из трех результатов: недетерминированное поведение среды и ошибки проверки утверждений, невозможность изменить данные в будущем или видимые пользователю побочные эффекты.

Восстановление после аварии и проектирование хаоса

Тесты этого типа имеют следующие характеристики:

- SUT: продакшен;
- данные: продакшен и подготовленные пользователем (для внедрения ошибки);
- проверка: вручную и с использованием метрик A/B-тестирования.

Эти тесты проверяют реакцию системы на неожиданные изменения или сбои.

В течение многих лет в Google ежегодно проводятся «учения» под названием DiRT (Disaster recovery testing — «тестирование восстановления после отказа», <https://oreil.ly/17ffL>), во время которых мы внедряем в нашу инфраструктуру ошибки почти в планетарном масштабе. Мы моделируем самые разные ситуации — от пожаров в вычислительном центре до злонамеренных атак. Однажды мы смоделировали землетрясение, которое полностью изолировало нашу штаб-квартиру в Маунтин-Вью (штат Калифорния) от остальных подразделений компании. Это помогло выявить не только технические недостатки, но и проблемы управления, когда все ключевые лица, принимающие решения, оказались недоступны¹.

DiRT-тесты требуют четкой координации действий сотрудников всей компании. Проектирование хаоса, напротив, — это скорее «непрерывное тестирование» технической инфраструктуры. Инженерия хаоса, ставшая популярной благодаря

¹ Во время проведения этого теста практически ничего нельзя было делать, поэтому многие сотрудники пошли в одно из наших многочисленных кафе. Так мы создали DDoS-атаку на наши команды!

Netflix (<https://oreil.ly/BCwdM>), включает разработку программ, которые постоянно вводят ошибки в системы и наблюдают за их последствиями. В целом инструменты тестирования хаоса предназначены для контроля восстановления функциональности. Цель проектирования хаоса — помочь командам перестать уповать на стабильность и надежность своих продуктов и поднять отказоустойчивость кода на новый уровень. Команды в Google выполняют тысячи тестов хаоса каждую неделю, используя нашу систему Catzilla.

Спровоцированные отказы и тесты с негативным влиянием используются для повышения уровня отказоустойчивости в действующих реальных системах, для которых цена и риски таких испытаний допустимы.

Ограничения

Любые проблемы, обнаруженные этими тестами (в продакшене), уже затронули конечных пользователей.

Тестирование восстановления после отказа довольно дорого в управлении, поэтому мы проводим скоординированные учения не очень часто. Имитируя аварии такого уровня, мы фактически причиняем массу неприятностей своим сотрудникам и негативно влияем на их работу.

Оценка действий пользователей

- SUT: продакшен;
- данные: продакшен;
- проверка: вручную и с использованием метрик A/B-тестирования.

Тестирование в продакшене позволяет собирать данные о поведении пользователей. У нас есть несколько альтернативных способов сбора метрик, характеризующих востребованность продукта и проблемы с новыми особенностями.

Внутренняя оценка

Существуют технологии для создания ограниченного количества развертываний и экспериментов, позволяющие открыть доступ к новым особенностям только узкому кругу пользователей, например нашим сотрудникам (для внутренней оценки), и получить обратную связь в реальном окружении.

Проведение экспериментов

Доступ к новым экспериментальным возможностям открывается подмножеству пользователей без их ведома. Затем поведение экспериментальной группы сравнивается с поведением контрольной группы по некоторым укрупненным показателям. Например, мы провели на базе YouTube эксперимент, связанный с изменением способа обработки лайков к видеороликам (отключив возможность оставлять дизлайки), и только ограниченный круг пользователей видел это изменение.

Это важная возможность для Google (<https://oreil.ly/OAvqF>). Одна из первых историй, которую нуглеры слышат при поступлении к нам на работу, связана с экспериментом по изменению цвета затемнения фона в объявлениях AdWords в Google Search. В результате этого эксперимента пользователи из экспериментальной группы выполнили значительно больше переходов по объявлениям, чем пользователи контрольной группы.

Экспертная оценка

Эксперты получают результаты некоторого изменения, выбирают «лучшие» результаты и объясняют свой выбор. Затем на основе их отзывов определяется, является ли изменение положительным, нейтральным или отрицательным. Например, Google уже давно привлекает экспертов для оценки качества обработки поисковых запросов (мы даже опубликовали рекомендации по оцениванию). В некоторых случаях отзывы с рейтинговыми оценками помогают определить уместность внесения исследуемых изменений в алгоритм. Экспертная оценка имеет решающее значение для недетерминированных систем, таких как системы машинного обучения, в которых нет четкого критерия качества.

Большие тесты и рабочий процесс разработчика

Мы поговорили о том, что такое большие тесты, зачем, когда и как часто их нужно проводить, но мы почти не затронули вопрос «кто». Кто пишет тесты? Кто проводит тесты и исследует причины сбоев? Кто владеет тестами? И как сделать тестирование менее утомительным?

Стандартная инфраструктура юнит-тестирования может быть непригодной для больших тестов, и все же важно интегрировать большие тесты в рабочий процесс разработчика. Один из способов сделать это — создать механизмы автоматического выполнения тестов до и после отправки изменений в репозиторий. Многие большие тесты в Google не вписываются в структуру TAP, поскольку они негерметичные, ненадежные и ресурсоемкие. Но мы должны гарантировать их выполнение, иначе рискуем пропустить важный сигнал и усложнить устранение проблем. Поэтому мы отдельно выполняем непрерывную сборку после отправки кода в репозиторий и рекомендуем выполнять эти тесты перед отправкой, чтобы получить обратную связь от автора.

Также в рабочий процесс можно включить A/B-тесты. Перед отправкой в репозиторий они могут проверять факт проведения обзора кода и утверждение различий в пользовательском интерфейсе. Один из таких наших тестов автоматически блокирует выпуск новой версии, если в коде есть неутвержденные различия.

Иногда тесты настолько велики или требовательны к ресурсам, что их выполнение перед отправкой кода вызывает слишком много сложностей. Поэтому такие тесты выполняются после отправки и дополнительно включаются в процесс выпуска новой версии. При этом в монолитный репозиторий могут проникнуть ошибки и потребуются найти и откатить изменения, которые к ним привели. Мы ищем компромиссы

между сложностью для разработчика, задержками в передаче изменений и надежностью непрерывной сборки.

Разработка больших тестов

Структура больших тестов довольно стандартная, но их создание все равно сопряжено с трудностями, особенно для новичков.

При разработке таких тестов нужно иметь под рукой ясные библиотеки, документацию и примеры. Юнит-тесты легче писать благодаря поддержке используемого языка программирования (когда-то фреймворк JUnit считался экзотикой, но в настоящее время получил широкое распространение). Мы повторно используем одни и те же библиотеки утверждений для функциональных и интеграционных тестов и со временем создали библиотеки для взаимодействия с SUT, помогающие запускать A/B-тесты, генерировать тестовые данные и организовывать рабочие процессы тестирования.

Большие тесты дороги в сопровождении с точки зрения как ресурсов, так и времени, но не все большие тесты одинаковы. Одна из причин популярности A/B-тестов состоит в том, что они не требуют больших затрат на выполнение этапа проверки. Аналогично тестирование в продакшене обходится дешевле, чем создание изолированных и герметичных SUT. Различия в затратах на разные виды тестов с учетом сопровождения инфраструктуры и кода могут оказаться весьма существенными.

Однако все эти затраты необходимо рассматривать вместе. Если затраты на согласование различий вручную или поддержку и безопасность тестирования в продакшене перевешивают выгоды от тестов, то такие тесты не будут эффективными.

Выполнение больших тестов

Поскольку наши большие тесты не вписываются в структуру TAP, мы организовали альтернативный процесс непрерывной сборки и предварительного тестирования. Одна из основных задач, стоящих перед нашими инженерами, заключается в поиске способа запуска и повтора нестандартного теста.

Мы постарались организовать тестирование так, чтобы дать инженерам хорошо знакомый способ запуска тестов. Наша инфраструктура тестирования перед отправкой предлагает общий API для выполнения больших тестов и тестов TAP, а наша инфраструктура обзора кода отображает оба набора результатов. Но многие большие тесты делаются на заказ, и к ним прилагается специальная документация, описывающая требования к их выполнению. Это обстоятельство может поставить в тупик инженеров, не знакомых с инфраструктурой.

Ускорение выполнения тестов

Инженеры не любят ждать выполнения медленных тестов. Чем медленнее работает тест, тем реже инженеры запускают его и тем больше времени требуется на устранение ошибок в случае сбоя.

Лучший способ ускорить тест — сузить область его охвата или разбить его на тесты меньшего размера, которые будут выполняться параллельно. Но есть и другие приемы, помогающие ускорить большие тесты.

Некоторые наивные тесты используют временные задержки перед появлением недетерминированного действия, и этот прием часто используется в больших тестах. Однако эти тесты не ограничивают многопоточное выполнение, поэтому мы стремимся к тому, чтобы тест реагировал так же, как реальные пользователи, которые не любят долго ждать. Вот некоторые возможные подходы для этого:

- выполнять опрос в цикле для определения момента изменения состояния или появления события в течение временного окна с частотой, близкой к микросекундам. Этот прием можно объединить с определением тайм-аута на тот случай, если тест в течение длительного времени не достигнет стабильного состояния;
- реализовать обработчик событий;
- подписаться на получение уведомлений о событиях.

Обратите внимание, что тесты, которые полагаются на задержки и тайм-ауты, начнут давать сбой, когда возрастет нагрузка на серверы, выполняющие эти тесты, что приведет к необходимости запускать тесты чаще, что еще больше увеличит нагрузку.

Уменьшите задержки и тайм-ауты внутри системы

Реальные системы обычно настраиваются с учетом топологии распределенного развертывания, но SUT можно развернуть на одном компьютере (или, по крайней мере, в кластере близко расположенных машин). Если в коде имеются жестко заданные тайм-ауты или (особенно) операторы `sleep` для учета задержек, их следует настроить и сократить их число при выполнении тестов.

Оптимизируйте время сборки теста

Один из недостатков нашего монолитного репозитория — необходимость собирать все зависимости большого теста, но у этого требования есть исключения. Если SUT состоит из основной части, которую проверяет тест, и некоторых других двоичных зависимостей, то можно использовать в тесте предварительно собранные версии этих двоичных зависимостей. Наша система сборки (основанная на монолитном репозитории) плохо поддерживает эту модель, но этот подход точнее соответствует продакшену, в котором разные службы имеют разные версии.

Устранение нестабильности

Нестабильность — серьезный недостаток для юнит-теста, который для большого теста может стать причиной его непригодности. Команда должна рассматривать устранение нестабильности в больших тестах как высокоприоритетную задачу. Но как устранить нестабильность из больших тестов?

Устранение нестабильности следует начинать с сокращения области охвата теста. Герметичная SUT не подвержена рискам, связанным с одновременной работой нескольких пользователей в продакшене или общей промежуточной среде. Если она

выполняется на единственной машине, то не страдает от проблем распределенного развертывания. Смягчить другие проблемы, вызванные нестабильностью, помогут тесты и другие методы, сбалансированные со скоростью выполнения тестов.

Использование методов реактивного или событийно-ориентированного программирования может не только ускорить выполнение тестов, но и устранить причины их нестабильного поведения. Для ограничения задержек по времени необходимо использовать тайм-ауты, которые можно встроить в код теста. Увеличение числа внутренних системных тайм-аутов может уменьшить нестабильность, тогда как их сокращение, наоборот, может вызвать нестабильное поведение тестов, если система проявляет недетерминированное поведение. Важно найти правильный компромисс, определяющий допустимое поведение системы для конечных пользователей (например, указать максимально допустимый тайм-аут — n секунд), и обеспечить надежную обработку нестабильного поведения при выполнении теста.

Превышение допустимого числа внутренних системных тайм-аутов может приводить к сложным ошибкам. Реальные системы часто стараются оградить конечных пользователей от катастрофических сбоев за счет обработки внутренних проблем. Например, если Google не может показывать рекламу в течение определенного времени, мы не возвращаем код ошибки 500, а просто не показываем рекламу. Но для тестировщика это выглядит так, будто код, отображающий рекламу, потерял работоспособность. В таких случаях важно сделать факт сбоя очевидным и упростить настройку внутренних тайм-аутов для тестирования.

Увеличение понятности тестов

Сложно интегрировать в рабочий процесс инженера тест, который дает непонятные результаты. Даже юнит-тесты могут вызвать путаницу — когда мое изменение нарушает работу вашего теста, мне будет трудно понять причину отказа, если я не знаком с вашим кодом, — а для больших тестов такая путаница может оказаться непреодолимой. Тесты, использующие утверждения, должны давать четкий и ясный сигнал об успехе/неудаче и выводить понятные сообщения об ошибке, чтобы помочь инженеру определить причины отказа. Тесты, предусматривающие участие человека, такие как А/В-тесты, требуют специального обращения, чтобы инженеры не проигнорировали их на этапе отправки.

Как это реализуется на практике? Хороший большой тест, терпящий неудачу, должен:

Вывести сообщение, четко поясняющее причину сбоя

Худший сценарий — когда на экран выводятся туманное сообщение об ошибке, такое как `Ошибка утверждения`, и трассировка стека. Хорошее сообщение учитывает, что тестировщик не знает код, и ясно описывает контекст: `B test_ReturnsOneFullPageOfSearchResultsForAPopularQuery` ожидалось 10 результатов поиска, а получен 1. Тесты производительности или А/В-тесты должны четко объяснять, что измеряется и почему результаты тестирования считаются подозрительными.

Минимизировать усилия, необходимые для определения основных причин несоответствия

Трассировка стека бесполезна для больших тестов, потому что цепочка вызовов может пересекать границы нескольких процессов. В таких случаях следует сгенерировать трассировку по всей цепочке вызовов или использовать инструменты автоматизации, которые помогут локализовать ошибку. Для этого тест должен создать какой-то артефакт. Например, в Google используется фреймворк Dapper (<https://oreil.ly/FXzbv>), связывающий идентификатор запроса со всеми запросами в цепочке RPC, что позволяет сопоставить все журналы, фиксирующие прохождение запроса с указанным идентификатором, и тем самым упростить исследование трассировки.

Предоставить сведения о поддержке и контактную информацию

Тот, кто выполняет тест, должен иметь возможность связаться с владельцами теста или с лицами, сопровождающими тест, чтобы получить помощь.

Владение большими тестами

Большие тесты должны иметь официальных владельцев — инженеров, которые смогут адекватно проанализировать изменения в тесте и помочь разобраться в неудачном завершении теста. В отсутствие официального владельца:

- инженерам будет все труднее изменять и обновлять тест;
- устранение причин сбоев теста будет занимать все больше времени.

И тест перестанет использоваться.

Интеграционным тестом для компонентов в конкретном проекте должен владеть руководитель проекта. Тест, проверяющий функциональные возможности (охватывающие определенную бизнес-функцию, которая поддерживается множеством служб), должен принадлежать «владельцу функции»: либо инженеру-программисту, ответственному за комплексную реализацию функции, либо менеджеру по продукту или «инженеру по тестированию», которому принадлежит описание бизнес-сценария. Владелец теста должен обладать полномочиями по обеспечению общей работоспособности теста, и для этого ему нужно дать необходимые инструменты и стимулы.

На основе структурированной информации о владельцах можно автоматизировать тестирование. Вот некоторые наши подходы к автоматизации тестов:

Обычная информация о владении кодом

Часто большой тест — это отдельный артефакт кода, который находится в определенном месте в кодовой базе. В этом случае мы можем использовать информацию о владельцах (глава 9), уже имеющуюся в репозитории, чтобы подсказать инструментам автоматизации, кто владеет данным конкретным тестом.

Аннотации к тесту

В один тестируемый класс или модуль может быть добавлено несколько методов-тестов, у каждого из которых может быть свой владелец функции. В таких случаях мы добавляем структурированные аннотации для каждого языка, определяющие владельца теста, чтобы в случае сбоя того или иного метода определить, к кому можно обратиться за помощью.

Заключение

В набор тестов должны входить большие тесты, гарантирующие точное соответствие SUT требованиям тестирования и помогающие выявлять проблемы, которые не обнаруживаются юнит-тестами. Поскольку большие тесты сложные и медленные, необходимо позаботиться о надлежащем владении такими тестами, содержании их в актуальном состоянии и выполнении при необходимости (например, перед развертыванием системы в рабочей среде). Мы стараемся делать большие тесты как можно меньше (не в ущерб точности), чтобы упростить их обслуживание. Для большинства программных проектов необходимо разрабатывать комплексную стратегию тестирования, определяющую риски системы, и большие тесты, помогающие их смягчить.

Итоги

- Большие тесты выявляют проблемы, которые не обнаруживаются юнит-тестами.
- Большие тесты состоят из SUT, данных, действий и проверок.
- Хороший программный проект включает стратегию тестирования, определяющую риски системы, и большие тесты, помогающие их смягчить.
- При создании больших тестов необходимо прикладывать дополнительные усилия, чтобы тесты не создавали ненужных сложностей в рабочем процессе инженера.

Устаревание

*Автор: Хайрам Райт
Редактор: Том Манирек*

Обожаю дедлайны. Обожаю свист, с которым они, словно встречный ветер, проносятся мимо.

Дуглас Адамс

Все системы стареют. Несмотря на то что ПО является цифровой сущностью и сами биты не разрушаются, появление новых технологий, библиотек, методов, языков и других достижений приводит к устареванию существующих систем. Старые системы требуют постоянного обслуживания, специфических знаний и, как правило, больше усилий при взаимодействии с ними, потому что они постепенно все сильнее отстают от окружающей экосистемы. Часто лучше избавиться от устаревших систем, чем позволять им бесконечно находиться рядом с системами, которые призваны их заменить. Но все возрастающее число устаревших систем, которые продолжают эксплуатироваться, ясно показывает, что на практике не так просто от них отказаться. Мы называем процесс упорядоченной миграции и последующего удаления устаревших систем *устареванием*, или *прекращением поддержки*.

Устаревание — это еще одна тема, которая в большей мере относится к программной инженерии, чем к программированию, потому что требует размышлений об управлении системой с течением времени. В долгосрочных программных экосистемах правильное планирование и прекращение поддержки сокращают затраты и повышают скорость разработки, устраняя избыточность и сложность, которые со временем накапливаются в системе. С другой стороны, неправильное проведение мероприятий по выводу системы из эксплуатации может обходиться дороже бездействия. Устаревшие системы требуют дополнительных усилий для их сопровождения, однако их устаревание можно запланировать на этапе проектирования, чтобы потом было проще вывести их из эксплуатации и удалить. Устаревание может затрагивать как отдельные вызовы функций, так и целые программные стеки. Далее мы сосредоточимся в основном на устаревании кода.

Мы в Google все еще учимся, как лучше прекращать поддержку программных систем и удалять их. В этой главе вы найдете уроки, которые мы извлекли из устаревания больших и интенсивно используемых внутренних систем. Иногда вывод из эксплу-

атации идет как по маслу, а иногда — нет, и удаление устаревших систем остается для нас сложной задачей.

Эта глава в основном посвящена устареванию технических систем, не предназначенных для взаимодействий с конечными пользователями. Да, внешний API — это лишь одна из разновидностей продукта, и у внутреннего API тоже могут быть потребители, считающие себя конечными пользователями. Многие из описанных ниже принципов применимы к устареванию общедоступного продукта, но в этой главе мы сосредоточимся на технических и политических аспектах устаревания и удалении устаревших систем, владельцы которых имеют представление об их использовании.

Почему необходимо заботиться об устаревании

Обсуждение устаревания мы начнем с утверждения: *код является обязательством, а не ценностью*. В конце концов, если считать код ценностью, зачем тогда тратить время на отключение и удаление устаревших систем? Код имеет свои издержки, меньшая часть из которых приходится на процесс создания системы и большая часть — на поддержку системы в течение всего срока ее службы. Эти постоянные издержки, такие как эксплуатационные ресурсы, необходимые для поддержания работоспособности системы, и усилия по постоянному обновлению ее кодовой базы с развитием окружающих экосистем, требуют поиска компромисса между поддержанием работоспособности устаревшей системы и отказом от нее.

Возраст системы сам по себе не является определяющим фактором ее устаревания. Система может быть спроектирована для работы в течение нескольких лет и быть образцом хорошего ПО. Например, система верстки LaTeX совершенствовалась в течение десятилетий, и хотя изменения в нее вносятся до сих пор, они немногочисленны и редки. Большой возраст системы еще не означает, что она устарела.

Устаревание в большей степени касается систем, которые явно устарели, и для них появилась замена с сопоставимыми функциональными возможностями. Новая система может эффективнее использовать ресурсы, иметь лучшие показатели безопасности, обладать большей устойчивостью или просто не содержать ошибок старой системы. Наличие двух систем для решения одной и той же задачи порой не кажется большой проблемой, но со временем затраты на их обслуживание могут существенно возрасти. Зависимости, использующие устаревшую систему, могут помешать пользователям применить новую систему.

Если старой и новой системам придется взаимодействовать друг с другом, это потребует сложного кода для выполнения преобразований. По мере развития системы могут становиться зависимыми друг от друга, что усложнит удаление одной из них. Новая система, которая поддерживает совместимость со старой системой, не может развиваться. Затраты на удаление старой системы могут окушаться, потому что новая система сможет развиваться быстрее.

Несмотря на всю пользу устаревания, мы в Google понимаем, что есть определенные ограничения на объем работ по прекращению поддержки с точки зрения команд и их клиентов. Например, всем нравятся свежееуложенные дороги, но если управление коммунальным хозяйством закроет на ремонт *все* дороги одновременно, никто не сможет никуда поехать. Сосредоточив свои усилия, дорожные бригады могут быстрее выполнять определенные работы и не мешать транспортному сообщению. Точно так же важно тщательно выбирать проекты для прекращения поддержки и обязательно доводить процесс их удаления до конца.

Выше мы отметили, что «код является обязательством, а не ценностью». Если это так, то почему мы потратили большую часть этой книги на обсуждение эффективных способов создания программных систем, которые могут существовать десятилетиями? Зачем тратить силы и время на создание большого объема кода, если в общем балансе он попадет в статью затрат?

Код *сам по себе* не имеет никакой ценности — его ценность заключена в его *функциональности*. Но функциональность является ценностью, только если она отвечает потребностям пользователя: код, реализующий необходимую функциональность, является просто средством достижения цели. Если ту же функциональность можно получить, написав одну строку простого и понятного кода вместо 10 000 строк сложного и запутанного кода, мы бы предпочли первый вариант. Сам код несет только затраты — чем он проще при том же объеме функциональности, тем лучше.

Мы должны сосредоточиться не на объеме кода, который можем написать, и не на размерах кодовой базы, а на максимизации количества функциональных возможностей, приходящихся на единицу кода. Для этого проще всего не писать больше кода в надежде получить больше функциональности, а удалять лишний код и системы, которые уже не нужны. Политика и процедуры прекращения поддержки кода позволяют нам работать в этом направлении.

Почему устаревание вызывает такие сложности?

Применим закон Хайрама к устареванию: чем больше пользователей системы, тем выше вероятность, что они будут использовать систему неожиданными и непредвиденными способами, и тем сложнее будет прекратить ее поддержку. Необычные и непредвиденные сценарии использования появляются, потому что система «случайно дает нужный результат», а не «гарантирует» его. В этом контексте удаление системы можно рассматривать как бесповоротное изменение: мы не меняем поведение, а полностью удаляем его! Такое радикальное изменение позволит избавиться от целого ряда неожиданных зависимостей.

Однако отказ от развития системы обычно невозможен, пока не будет доступна новая система с той же (или лучшей!) функциональностью. Новая система может быть лучше, но она также имеет отличия от старой системы: в конце концов, если бы она в точности повторяла устаревшую систему, то не принесла бы пользователям никакой выгоды от перехода на нее (хотя могла бы принести выгоду команде, эксплуатиру-

ющей ее). Такое функциональное отличие означает, что полное соответствие между старой и новой системами встречается редко и каждый сценарий использования старой системы должен оцениваться в контексте новой.

Причиной неприятия отказа от старой системы может стать эмоциональная привязанность к ней, особенно если принимающий решение о прекращении поддержки системы сам приложил руку к ее созданию. В Google мы иногда сталкиваемся с аргументом: «Мне нравится этот код!», когда пытаемся убедить инженеров сломать что-то, что они строили годами. Это понятный аргумент, но он ведет к саморазрушению: если система устарела, она несет одни лишь издержки для организации и должна быть удалена. Один из способов решить проблему сохранения удаленного кода в Google — возможность поиска его хронологической записи (глава 17).

В Google есть старая шутка о том, что всегда есть два способа сделать что-то: устаревший и тот, что еще не готов. Она отражает печальную реальность работы в сложной и быстро развивающейся технологической среде, где новое решение всегда «почти» готово.

Инженеры в Google уже привыкли работать в этой среде, но даже им требуется обратиться к документации, указателям и экспертам, чтобы определить, использовать ли старую систему со всеми ее недостатками или перейти на новую со всеми ее неопределенностями.

Наконец, выделение средств и выполнение работ по удалению устаревших систем могут сопровождаться политическими сложностями. Комплектование команды и трата времени на удаление устаревших систем стоят реальных денег, в то время как затраты на бездействие и оставление системы без присмотра трудно заметить и измерить. Также может быть трудно убедить заинтересованные стороны, что работы по остановке развития и удалению системы необходимы для разработки новых функций. Методы исследования, подобные описанным в главе 7, могут помочь найти конкретные доказательства того, что прекращение поддержки системы имеет смысл.

Учитывая трудности, связанные с прекращением поддержки и удалением устаревших программных систем, пользователям зачастую проще развивать систему *на месте*, чем полностью ее заменять. Такое развитие не останавливает процесс устаревания, а разбивает его на более мелкие и управляемые этапы, что может принести дополнительные выгоды. Мы в Google имели возможность убедиться, *насколько дорого* обходится миграция на совершенно новые системы, и эта дороговизна часто недооценивается. Постепенный отказ от устаревшей системы с помощью рефакторинга на месте может поддерживать работоспособность существующих систем, помогая пользователям извлекать дополнительную выгоду.

Планирование прекращения поддержки системы на этапе ее проектирования

Прекращение поддержки программной системы можно запланировать на этапе ее создания. Выбор языка программирования, архитектуры ПО, состава команды

и даже политики и культуры компании — все это определяет, насколько легко удалять системы в этой компании.

Идея проектировать системы так, чтобы впоследствии их можно было объявить устаревшими, возможно, выглядит слишком радикальной, но она получила широкое распространение в других инженерных дисциплинах. Рассмотрим, например, атомную электростанцию, которая представляет собой сложнейшее инженерное сооружение. При ее проектировании необходимо учитывать ее вывод из эксплуатации по истечении срока службы вплоть до выделения средств на эти цели¹. Необходимость вывода атомной электростанции из эксплуатации существенно влияет на многие решения, принимаемые инженерами.

К сожалению, программные системы редко проектируются настолько же тщательно. Инженеров-программистов больше заботят создание и запуск новых систем, а не обслуживание существующих. Корпоративная культура многих компаний, в том числе Google, ориентируется на быстрое создание и ввод в эксплуатацию новых продуктов, что часто мешает учитывать возможность прекращения поддержки продукта при его проектировании. И несмотря на популярное представление о программистах как о роботах, управляющих данными, психологически сложно планировать возможную гибель своих творений, над созданием которых мы так усердно трудимся.

Итак, что следует учитывать при разработке систем, чтобы в будущем их проще было объявить устаревшими? Вот пара вопросов из числа тех, которые мы в Google рекомендуем рассматривать своим командам инженеров:

- Насколько легко потребители смогут мигрировать с нашего продукта на его потенциальную замену?
- Есть ли возможность постепенно заменять компоненты нашей системы?

Многие из этих вопросов касаются особенностей предоставления услуг системой и использования зависимостей. Подробнее о порядке управления зависимостями — в главе 16.

Отметим, что решение о долгосрочной поддержке проекта принимается, когда организация только решает его запустить. После создания программной системы остаются только три возможных варианта: поддерживать ее, максимально осторожно вывести из эксплуатации или позволить ей самой прекратить работу, когда какое-то внешнее событие сделает ее неработоспособной. Все эти варианты допустимы, и выбор любого из них будет зависеть от организации. Новый стартап с одним проектом спокойно уничтожит свою систему, когда компания обанкротится, но крупные компании должны бережнее относиться к своим инвестициям и репутации при рассмотрении возможности удаления старых проектов. Как упоминалось выше, мы в Google все еще учимся, как лучше прекращать поддержку наших внутренних и внешних продуктов.

¹ «Design and Construction of Nuclear Power Plants to Facilitate Decommissioning», (<https://oreil.ly/heo5Q>) Technical Reports Series No. 382, IAEA, Vienna (1997).

Не стоит начинать проект, который организация не намерена поддерживать в течение ожидаемого срока своей работы. Даже если организация решит отказаться от проекта и удалить его, она все равно понесет какие-то затраты, но их можно уменьшить за счет тщательного планирования и инвестиций в инструменты и политику.

Подходы к прекращению поддержки

Прекращение поддержки — это не один, а целая совокупность процессов, основанных на решениях, от «возможно, мы когда-нибудь отключим эту систему» до «эта система остановится завтра, и клиентам лучше быть готовыми к этому». В целом мы делим эту совокупность на две отдельные группы: рекомендуемая и принудительная миграции.

Рекомендуемая миграция

Рекомендуемая миграция имеет место, когда для миграции не назначен крайний срок, и вывод из эксплуатации не является приоритетной задачей для организации (и когда компания не желает продолжать выделять ресурсы). Такой подход к прекращению поддержки также можно назвать *желательной* миграцией: инженеры знают, что для системы есть замена, и хотя они понимают, что клиенты рано или поздно мигрируют, инженеры не намерены в ближайшем будущем оказывать клиентам помощь в миграции или удалять старую систему. В этом подходе к прекращению поддержки отсутствует элемент принуждения: мы верим, что клиенты перейдут на новую систему, но не можем заставить их сделать это. Наши друзья, занимающиеся проблемами надежности, ответят: «Надежда — это не стратегия».

Рекомендуемая миграция — хороший инструмент для рекламы новой системы и стимулирования пользователей, которые первыми начнут ее применять. Однако новая система не должна продвигаться в период бета-тестирования: она должна быть готова к использованию, нагрузкам и поддержке новых пользователей в течение неопределенного срока. Конечно, любая новая система будет испытывать трудности роста, но когда старая система окончательно устареет, новая система станет важной частью инфраструктуры организации.

Мы в Google наблюдали сценарий, когда рекомендованный переход на новую систему давал очевидные преимущества. В этих случаях простое уведомление пользователей о существовании новой системы и предоставление им инструментов, упрощающих миграцию, способствовали продвижению новой системы. Однако выгоды должны быть существенными. Пользователи будут неохотно мигрировать на новую систему для получения незначительных выгод, и даже новые системы со значительными улучшениями не получают полного признания, если использовать только рекомендательные меры по миграции.

Рекомендуя произвести миграцию, авторы подтолкнут пользователей в желаемом направлении, но не думайте, что авторы выполняют основную работу, связанную с миграцией. Часто у автора возникает соблазн просто объявить о прекращении под-

держки старой системы и уйти, не прикладывая никаких дополнительных усилий. Это может (немного) уменьшить количество новых применений устаревшей системы, но редко приводит к тому, что команды активно отказываются от нее. Существующие варианты использования старой системы создают своего рода концептуальное (или техническое) притяжение к ней и поглощают большую долю новых ее вариантов, сколько бы мы ни говорили: «Пожалуйста, используйте новую систему». Старая система по-прежнему будет требовать обслуживания и ресурсов, если активно не стимулировать ее пользователей к миграции.

Принудительная миграция

Активное стимулирование пользователей перейти на новую систему называется *принудительной* миграцией. При ней назначается крайний срок удаления устаревшей системы, и если пользователи продолжают зависеть от нее после этой даты, они обнаружат, что их собственные системы перестали работать.

Как это ни парадоксально, но лучший способ масштабирования принудительной миграции — это ее локализация в пределах одной команды экспертов, ответственной за полное удаление устаревшей системы. У этой команды есть стимул помогать другим командам мигрировать, накопленный опыт и возможность предлагать инструменты миграции. Многие миграции можно выполнить с применением инструментов, которые мы обсудим в главе 22.

Принудительная миграция должна содержать план исполнения. Это не означает, что планы не могут измениться, но такой подход дает возможность команде, осуществляющей процесс миграции, отключать несогласных пользователей после того, как они будут предупреждены о необходимости миграции. Без этого плана команды клиентов могут просто проигнорировать предупреждение, предпочтя потратить время не на миграцию, а на разработку новых особенностей или на выполнение другой более важной работы.

В то же время принуждение к миграции без привлечения персонала, оказывающего необходимую помощь, может показаться клиентам злонамеренным действием, что обычно препятствует завершению процесса прекращения поддержки старой системы. Клиенты могут рассматривать прекращение поддержки системы как нарушение договоренностей, требующее от них отложить решение своих приоритетных задач и заняться другой работой ради сохранения работоспособности служб. Это очень похоже на «бег на месте» и создает трения между специалистами, обслуживающими инфраструктуру, и их клиентами. Поэтому мы настоятельно рекомендуем активно поддерживать принудительное прекращение поддержки системы силами специализированной команды.

Также стоит отметить, что даже с учетом имеющейся власти принудительное прекращение поддержки может столкнуться с политическими препятствиями. Представьте, что вы пытаетесь принудить к миграции последнего оставшегося пользователя старой системы, от которого зависит инфраструктура всей вашей организации. Готовы ли вы

разрушить эту инфраструктуру и опосредованно оказать негативное воздействие на всех, кто от нее зависит, только для того, чтобы установить произвольно выбранный крайний срок? Вряд ли отказ от поддержки действительно произойдет, если этот клиент наложит вето.

Монолитный репозиторий и граф зависимостей в Google дают полное представление об использовании систем в нашей организации. Но все равно некоторые команды могут не знать, что зависят от устаревшей системы, а обнаружение этих зависимостей аналитическим путем может быть затруднено. Такие зависимости также можно выявить динамически с помощью тестов, увеличивая частоту и продолжительность временного отключения старой системы, которое покажет, что ломается при отключении, и предупредит команды о необходимости подготовиться к приближающемуся сроку миграции. Мы в Google время от времени меняем названия символов, предназначенных только для реализации, чтобы узнать, какие пользователи зависят от них.

В Google команда предупреждает о запланированных отключениях за месяцы и недели. Подобно учениям DiRT, эти предупреждения помогают обнаружить неизвестные зависимости между работающими системами. Поэтапный подход к миграции позволяет командам запланировать изменения, связанные с возможным удалением системы, или согласовать с командой, обеспечивающей миграцию, дату прекращения поддержки. (Тот же подход применим и к выявлению статических зависимостей в коде, но семантической информации, предоставляемой инструментами статического анализа, часто бывает достаточно для обнаружения всех зависимостей от устаревшей системы.)

Предупреждение о прекращении поддержки

Как при рекомендуемой, так и при принудительной миграции часто бывает полезно отметить системы как устаревшие, чтобы предупредить пользователей о прекращении поддержки старой системы и подтолкнуть их к миграции на новую систему. Вы можете просто отметить продукт как устаревший и надеяться, что со временем он выйдет из употребления, но помните: «Надежда — это не стратегия». Предупреждения об устаревании могут помочь предотвратить использование устаревшей системы в новых продуктах, но редко приводят к миграции существующих систем.

Такие предупреждения имеют свойство накапливаться. Если они генерируются в промежуточных зависимостях (например, библиотека A зависит от библиотеки B, которая зависит от библиотеки C, и C генерирует предупреждение, которое появляется при сборке A), то могут быстро потерять актуальность и настанет момент, когда пользователи начнут их игнорировать. В сфере медицины эту реакцию называют «усталостью от предупреждений» (<https://oreil.ly/uYUef>).

Любое предупреждение о прекращении поддержки должно обладать двумя свойствами: действенностью и релевантностью. Под *действенностью* подразумевается, что любой инженер сможет, опираясь на предупреждение, предпринять на практике

конкретное действие. Например, инструмент может предупредить, что вызов данной функции следует заменить вызовом более нового аналога, или в электронном письме могут быть описаны шаги по перемещению данных из старой системы в новую. В каждом случае предупреждение должно содержать последовательность шагов, которые инженер сможет выполнить, чтобы больше не зависеть от устаревшей системы¹.

Предупреждение может быть действенным и при этом раздражающим. Для большей пользы предупреждение должно быть еще и *релевантным*, то есть появляться в тот момент, когда пользователь выполняет действие, имеющее прямое отношение к устаревшей зависимости. Предупреждение об использовании устаревшей функции лучше всего выводить во время написания кода, когда предпринимается попытка использовать ее, а не спустя несколько недель после отправки зависимого кода в репозиторий. Точно так же электронное письмо с инструкциями по переносу данных лучше всего отправлять за несколько месяцев до удаления старой системы, а не перед выходными, после которых система будет удалена.

Важно не поддаваться стремлению предупреждать об устаревании всего на свете. Сами по себе предупреждения — это неплохо, но наивные инструменты часто генерируют такой поток предупреждающих сообщений, что он может ошеломить ничего не подозревающего инженера. Мы в Google очень либерально относимся к объявлению старых функций устаревшими, но используем такие инструменты, как ErrorProne (<https://errorprone.info>) или clang-tidy, чтобы гарантировать целенаправленное отображение предупреждений. Как обсуждается в главе 20, мы выводим предупреждения только в новых строках кода, чтобы предупредить сотрудников, что в новом коде не следует использовать устаревший символ. Более назойливые предупреждения, например сообщающие об устаревших элементах в графе зависимостей, добавляются только в процессе принудительной миграции. В любом случае инструменты играют важную роль в предоставлении соответствующей информации соответствующим сотрудникам в нужное время и позволяют добавлять больше предупреждений и не утомлять получателей.

Управление процессом прекращения поддержки

С точки зрения управления и выполнения проекты по планомерному прекращению поддержки похожи на любые другие проекты программной инженерии, несмотря на то что могут восприниматься как мероприятия совсем другого типа. Мы не будем тратить время на выявление сходства в управлении разными проектами и поговорим в основном об особенностях прекращения поддержки.

Владение процессом

Мы в Google поняли, что без явных владельцев процесс прекращения поддержки вряд ли достигнет значительного прогресса, сколько бы предупреждений не гене-

¹ См. пример на <https://abseil.io/docs/cpp/tools/api-upgrades>.

рировала система. Наличие явных владельцев проекта, управляющих процессом прекращения поддержки, может показаться неэффективной тратой ресурсов, но еще дороже не завершать прекращение поддержки или делегировать усилия по миграции пользователям системы. Во втором случае мы получим рекомендуемую миграцию, которая никогда не завершится естественным путем, а в первом возьмем на себя обязательство продолжать поддерживать старые системы. Централизация усилий по прекращению поддержки, как показывает опыт, помогает гарантировать *снижение* затрат на миграцию, делая их более прозрачными.

Заброшенные проекты часто превращаются в проблему, когда возникает необходимость установить их владельца и согласовать с ним меры по стимулированию к миграции. В каждой организации разумного размера есть проекты, которые все еще активно используются, но ими явно никто не владеет и никто их не поддерживает. Иногда проекты оказываются в этом состоянии из-за того, что они устарели: первоначальные владельцы перешли в проект-преемник, оставив устаревший проект в качестве зависимости другого проекта в надежде, что он со временем исчезнет.

Такие проекты редко исчезают сами по себе. Как показывает наш опыт, несмотря на самые радужные надежды, для таких проектов по-прежнему требуется участие специалистов по прекращению поддержки и выводу их из эксплуатации. Удаление должно быть основной целью отдельной команды, поскольку при наличии других задач работа по прекращению поддержки почти всегда будет восприниматься командой как низкоприоритетная и редко будет получать необходимое внимание. Важные, но не срочные задачи по очистке требуют 20 % рабочего времени и предоставляют инженерам доступ к другим частям кодовой базы.

Основные этапы

При создании новой системы основные этапы проекта, как правило, предельно ясны, например: «Запустить функцию `frobuzzer` к следующему кварталу». Следуя практике последовательной разработки, команды постепенно создают и предоставляют новые функциональные возможности пользователям, которые выигрывают от каждой новой возможности. Конечной целью может быть запуск всей системы, но последовательные этапы помогают команде почувствовать прогресс и избавляют от необходимости ждать окончания процесса, чтобы принести пользу организации.

Напротив, единственным важным этапом процесса прекращения поддержки часто может считаться полное удаление устаревшей системы. Команда может чувствовать, что ничего не добилась, пока не выключит свет и не уйдет домой. Конечно, удаление системы — значимый шаг для команды, но вне команды его никто не заметит, потому что к этому моменту у системы уже нет пользователей. Руководители проектов по прекращению поддержки не должны поддаваться искушению сделать факт удаления системы единственной измеримой вехой в работе команды, особенно если учесть, что иногда оно не требуется.

Так же как создание новой системы, прекращение поддержки старой системы должно предусматривать конкретные этапы, полезность которых можно оценить. Метрики для оценки прогресса в направлении прекращения поддержки могут быть специфическими, но для поддержания морального духа в команде, как и в любом проекте, важно отмечать промежуточные достижения, такие как удаление ключевого субкомпонента и т. п.

Инструменты для миграции

Большая часть инструментов для управления процессом прекращения поддержки подробно обсуждается в других главах этой книги, например инструменты для процесса крупномасштабных изменений (глава 22) и для процесса обзора кода (глава 19). Поэтому мы не будем подробно обсуждать особенности инструментов здесь, а лишь кратко опишем, как они могут пригодиться при управлении прекращением поддержки устаревшей системы. Их можно разделить на инструменты обнаружения, миграции и предотвращения отката.

Обнаружение

На ранних этапах процесса прекращения поддержки, а точнее на всем его протяжении, полезно знать, *как* и *кем* используется устаревшая система. Основная начальная работа по прекращению поддержки заключается в определении, кто использует старую систему и какими неожиданными способами. В зависимости от видов использования и после получения новой информации мы можем пересмотреть решение о прекращении поддержки. Кроме того, мы используем эти инструменты, чтобы контролировать работу.

Такие инструменты, как Code Search (глава 17) и Kythe (глава 23), позволяют статически определить, какие клиенты и как используют ту или иную библиотеку, и показывают, от какого неожиданного поведения клиенты зависят. Поскольку зависимости времени выполнения обычно требуют использования некоторой статической библиотеки или тонкого клиента, этот метод позволяет получить большую часть информации, необходимой для запуска и выполнения процесса прекращения поддержки. Журналирование и телеметрия среды выполнения в продакшене помогают обнаружить проблемы, связанные с динамическими зависимостями.

Наконец, наш глобальный набор тестов, как оракул, показывает, все ли ссылки на старый символ были удалены. Как обсуждалось в главе 11, тесты — это механизм предотвращения нежелательных изменений в поведении системы по мере развития организации. Прекращение поддержки является важной частью этого развития, и клиенты несут ответственность за достаточность тестов, гарантирующих, что удаление устаревшей системы не нанесет им вреда.

Миграция

Большая часть работы, связанной с прекращением поддержки, в Google выполняется с применением набора инструментов, который используется для генерации и обзора

кода, упоминавшегося выше. Процесс крупномасштабных изменений и его инструменты особенно полезны при масштабном обновлении кодовой базы для включения ссылок на новые библиотеки или службы времени выполнения.

Предотвращение отката

Наконец, часто упускаемый из виду элемент инфраструктуры прекращения поддержки — это инструменты, препятствующие использованию в новом коде объектов, которые находятся на пути к удалению. Даже в случае рекомендуемой миграции полезно предупредить пользователей о том, что при написании нового кода вместо устаревшей системы они должны использовать новую. Без предотвращения отката процесс прекращения поддержки может превратиться в бесконечную игру, в которой пользователи постоянно добавляют новые варианты использования знакомой им системы (или находят примеры такого использования в кодовой базе), а команда, занимающаяся прекращением поддержки, постоянно переносит эти новые варианты использования в новую систему. Это контрпродуктивный и деморализующий процесс.

Чтобы предотвратить откат на микроуровне, мы используем фреймворк статического анализа Tricorder, который уведомляет пользователей о том, что они добавляют обращения к устаревшей системе, и предлагает соответствующую замену. Владельцы устаревших систем могут добавлять аннотации компилятора к устаревшим символам (например, `@deprecated` в Java), а Tricorder будет обнаруживать попытки использования устаревших символов в новом коде во время проверки. В ограниченных случаях инструмент также предлагает исправления, которые можно применить щелчком мыши.

На макроуровне, чтобы предотвратить появление новых зависимостей от устаревшей системы, мы используем белые списки в нашей системе сборки. Автоматизированные инструменты периодически проверяют эти белые списки и корректируют их по мере миграции зависящих систем.

Заключение

Прекращение поддержки в чем-то напоминает грязную работу по уборке улицы после того, как по ней прошел цирковой парад, но эти усилия улучшают общую программную экосистему, сокращая накладные расходы на обслуживание и уменьшая когнитивную нагрузку на инженеров. Масштабируемое обслуживание сложных программных систем — это больше, чем просто создание и запуск ПО: оно подразумевает возможность удаления устаревших или неиспользуемых систем.

Полный процесс прекращения поддержки включает успешное решение социальных и технических проблем с помощью политических мероприятий и инструментов. Организованное и управляемое прекращение поддержки часто не рассматривается как источник выгоды для организации, но оно необходимо для ее устойчивого развития в долгосрочной перспективе.

Итоги

- Программные системы требуют постоянных затрат на обслуживание, которые следует сопоставлять с затратами на их удаление.
- Удалить систему часто труднее, чем создать ее с нуля, потому что пользователи нередко используют систему непредвиденными способами.
- Развитие существующей системы обычно обходится дешевле, чем ее замена на новую систему, если учитывать затраты на прекращение поддержки.
- Трудно достоверно оценить затраты на прекращение поддержки: помимо прямых затрат на обслуживание, связанных с сохранением старой системы, существуют экосистемные затраты, связанные с наличием множества аналогичных систем, между которыми нужно организовать взаимодействие. Старая система может неявно тормозить разработку новых возможностей.

ЧАСТЬ IV

Инструменты

Управление версиями и ветвями

Автор: Титус Винтерс

Редактор: Лиза Кэри

Пожалуй, ни один инструмент программной инженерии не получил такого широкого распространения, как системы управления версиями (VCS). Трудно представить софтверную компанию, которая официально не использовала бы VCS для управления исходным кодом и координации работы инженеров.

В этой главе мы поговорим о том, почему управление версиями стало нормой в программной инженерии, и рассмотрим плюсы и минусы разных подходов к организации VCS. Мы уверены, что управление версиями должны использовать все инженеры без исключения, но дополнительно рекомендуем выбирать только те правила и процессы, которые подходят вашей организации. В частности, мы считаем, что «разработка в главной ветви», популяризированная ассоциацией DevOps¹ (подразумевающая наличие одного репозитория без отдельных веток для разрабатываемых версий), является наиболее легко масштабируемым политическим подходом, и мы постараемся объяснить, почему это так.

Что такое управление версиями?



Многим читателям этот раздел может показаться скучным: в конце концов, управление версиями достаточно распространено. Если хотите, можете пропустить этот раздел и сразу перейти к разделу «Источник истины».

VCS — это система, которая отслеживает изменения (версии) в файлах. Она поддерживает метаданные с информацией о наборе управляемых файлов, а совокупность

¹ Ассоциация DevOps Research Association, которую приобрела компания Google в период между созданием рукописи этой главы и ее публикацией, подробно рассказывала об этом в ежегодном отчете «*State of DevOps Report*» и книге Николь Форсгрэн и др. «Ускоряйся! Наука DevOps. Как создавать и масштабировать высокопроизводительные цифровые организации» (Интеллектуальная литература, 2020). — *Примеч. пер.* Насколько мы можем судить, именно она популяризовала терминологию *разработки в главной ветви*.

копий файлов и метаданных называется репозиторием¹. VCS помогает координировать деятельность команд, позволяя нескольким разработчикам одновременно работать с одним и тем же набором файлов. Ранние VCS разрешали редактировать файл только одному человеку в каждый конкретный момент — такая блокировка позволяла установить очередность (согласовать понимание, «что новее»). Современные системы гарантируют, что изменения в *коллекции* файлов, отправленных вместе, будут обрабатываться как единое целое (*атомарно*, то есть логическое изменение затронет все файлы в наборе). Такие системы, как CVS (популярная в 1990-х), в которых процедура фиксации не поддерживает атомарности, подвержены искажениям и потерям изменений. Атомарность устраняет вероятность непреднамеренного затирания предыдущих изменений, но требует отслеживания времени синхронизации последней версии — попытка сохранить изменения отклоняется, если какой-либо из отправляемых файлов изменен раньше момента последней синхронизации локального разработчика. Поэтому в VCS рабочая копия управляемых файлов разработчика снабжается своими метаданными. В зависимости от архитектуры VCS копия репозитория может быть полной или содержать уменьшенное количество метаданных. Уменьшенная копия репозитория называется «клиентом» или «рабочей областью».

Все это выглядит пугающе сложным. Так зачем нужна VCS? Какая особенность помогла ей стать одним из немногих почти универсальных инструментов в разработке ПО и программной инженерии?

Представьте работу без VCS. Небольшая распределенная группа разработчиков, работающих над небольшим проектом, чтобы скоординировать свои действия, может пересылать версии проекта друг другу. Этот подход лучше всего работает, когда редактирование выполняется не одновременно (инженеры работают в разных часовых поясах или, по крайней мере, в разное рабочее время). Если же сотрудники не знают, какая версия самая последняя, они должны как-то ее отследить. Любой, кто пытался сотрудничать с коллегами в несетевой среде, вероятно, вспомнит ужасы копирования файлов с именами вроде `Presentation v5 - final - redlines - Josh's version v2`. И как мы увидим, когда нет единого согласованного источника истины, сотрудничество вызывает большие сложности и возрастает вероятность появления ошибок.

Организация общего хранилища требует чуть больше инфраструктуры (для доступа к нему), но обеспечивает простую и очевидную координацию. Для начала можно организовать работу на общем диске, но со временем возникнет угроза перезаписи работы коллеги. Кроме того, непосредственная работа с общим хранилищем означает, что любая задача, которая не поддерживает непрерывную сборку, начнет мешать всем в команде — если я внесу изменения в какую-то часть системы одновременно с вами и вы начнете сборку, то вы потерпите неудачу. Очевидно, что такой подход плохо масштабируется.

¹ Формальное представление о том, что является репозиторием, почти не зависит от выбора VCS, но терминология может существенно отличаться в разных источниках.

На практике невозможность блокировать файлы от записи и организовать слияние изменений, внесенных разными людьми, неизбежно приведет к конфликтам и перезаписи работы коллеги. Такая система, скорее всего, потребует внешней координации, помогающей решить, кто будет работать с тем или иным файлом. Если вы захотите предусмотреть блокировку файлов в ПО, то заново изобретете раннюю VCS, такую как RCS. Когда вы поймете, что предоставление разрешения на запись в файл кому-то одному — слишком грубое решение, и захотите организовать отслеживание на уровне строк, то снова подойдете к изобретению VCS. Если необходимость в каком-то структурированном механизме для управления сотрудничеством выглядит почти неизбежной, то почему бы не использовать готовый инструмент.

Почему важно управлять версиями?

В настоящее время управление версиями используется практически повсеместно, но так было не всегда. Самые первые VCS появились в 1970-х (SCCS) и 1980-х годах (RCS) — много лет спустя после первых упоминаний программной инженерии как отдельной дисциплины. Команды участвовали в «разработке многопользовательского многоверсионного ПО» (<https://arxiv.org/pdf/1805.02742.pdf>) еще до того, как в отрасли появилось формальное понятие управления версиями. Управление версиями возникло как ответ на новые проблемы цифрового сотрудничества. Потребовались десятилетия эволюции и популяризации, чтобы надежное и последовательное использование VCS превратилось в норму¹. Итак, почему это стало настолько важным и почему кто-то еще может противиться идее применения VCS, несмотря на то что она выглядит очевидным решением?

Напомню, что программная инженерия — это программирование во времени. Мы различаем (по числу усилий) непосредственное создание исходного кода и поддержку программного продукта с течением времени. Это базовое различие объясняет смысл внедрения VCS: на самом фундаментальном уровне VCS — это основной инструмент управления взаимосвязью между исходным кодом и временем. Управление версиями можно представить как расширение стандартной файловой системы. Если файловая система отображает имена файлов в их содержимое, то VCS отображает пары имя файла — время в содержимое и метаданные, необходимые для отслеживания последних точек синхронизации и истории версий. Управление версиями заставляет явно учитывать время. В большинстве случаев VCS также позволяет вводить

¹ Я провел несколько публичных выступлений, в которых использовал «внедрение управления версиями» как канонический пример развития норм программной инженерии с течением времени. В 1990-х годах управление версиями считалось передовой практикой, но применялось не повсеместно. В начале 2000-х еще часто можно было встретить профессиональные группы, не использующие VCS. В настоящее время применение таких инструментов, как Git, стало обычным явлением даже среди студентов колледжей, работающих над личными проектами. Отчасти такой рост популярности управления версиями, вероятно, связан с совершенствованием инструментов (едва ли кто-то захочет вернуться к RCS), но в гораздо большей степени он обусловлен накопленным опытом и изменением норм.

дополнительную переменную в это отображение (имя ветви), разрешая тем самым существование параллельных отображений:

VCS(имя_файла, время, имя ветви) => содержимое файла

По умолчанию в переменной «имя ветви» передается общепринятое и стандартное имя, соответствующее главной ветви, такое как `head`, `default` или `trunk`.

Остающиеся (незначительные) сомнения в отношении неуклонного использования VCS обусловлены объединением понятий программирования и программной инженерии. Мы обучаем программированию, учим программистов, проводим собеседования при приеме на работу, уделяя основное внимание проблемам и методам программирования. Даже в такой компании, как Google, привыкли не требовать от новых сотрудников опыта работы с кодом дольше пары недель или навыков работы в команде. Учитывая такую практику, управление версиями не выглядит чем-то важным, поскольку оно решает проблемы, с которыми новый сотрудник может и не столкнуться (например, «отмена» не для отдельного файла, а для всего проекта), что усложняет распространение неочевидных преимуществ VCS.

Тот же результат можно наблюдать в других софтверных компаниях, руководство которых рассматривает работу технических специалистов как «разработку ПО» (сесть и написать код), а не как «программную инженерию» (создать код и поддерживать его работоспособность и ценность в течение долгого времени). Из-за восприятия программирования как основной задачи и слабого понимания взаимосвязи между кодом и временем легко посчитать чрезмерной роскошью «возврат к предыдущей версии, чтобы исправить ошибку».

Помимо раздельного хранения версий и доступа к ним VCS позволяет преодолеть разрыв между процессами, выполняемыми в одиночку и в команде. С практической точки зрения именно по этой причине так важно использовать VCS в программной инженерии — она позволяет увеличивать масштабы команд и организаций, даже если кнопка «отмена» используется нечасто. Разработка, по своей сути, — это процесс ветвления и слияния, целью которого является координация усилий не только нескольких разработчиков, но и одного разработчика в разные моменты. VCS снимает вопрос «какая версия более свежая?» Современные VCS автоматизируют операции, в которых легко допустить ошибку, например определение набора изменений, который был применен. VCS — это инструмент координации действий нескольких разработчиков или одного разработчика в разные моменты.

Управление версиями настолько глубоко проникло в программную инженерию, что даже правовая и нормативная практики стали ему соответствовать. VCS позволяет официально регистрировать каждое изменение в каждой строке кода, что становится все более востребованным для аудита. При совмещении внутренней разработки и использования стороннего исходного кода VCS помогает отследить происхождение каждой строки кода.

Помимо технических и нормативных аспектов отслеживания действий во времени и обработки операций синхронизации/ветвления/слияния управление версиями способствует появлению некоторых нетехнических черт поведения. Процедура фиксации изменений в VCS и получение журнала фиксаций заставляет инженеров остановиться и подумать: что изменилось с момента последней фиксации? Довольны ли они состоянием исходного кода? Момент размышлений, связанных с фиксацией, подведением итогов и отметкой задачи как выполненной, для многих может иметь свою ценность. Начало процесса фиксации изменения — идеальное время для выполнения операций из контрольного списка: статического анализа (глава 20), проверки охвата тестирования, выполнения тестов, анализа их результатов и т. д.

Подобно любым другим процессам, управление версиями имеет свои издержки: кто-то должен настраивать, сопровождать и использовать VCS. Но не пугайтесь: почти всегда эти издержки невелики. Как ни странно, опытные инженеры-программисты инстинктивно используют управление версиями для любых проектов, которые существуют дольше одного-двух дней, даже когда работают в одиночку. Это доказывает, что ценность управления версиями (включая снижение риска возникновения ошибок) перевешивает издержки. Но мы признаем, что контекст имеет значение, и призываем руководителей самостоятельно принимать решения о внедрении VCS. Всегда стоит рассматривать альтернативы, даже в таком фундаментальном вопросе, как управление версиями.

Честно говоря, в современной программной инженерии трудно представить задачу, решаемую без немедленного внедрения VCS. Понимая ценность и необходимость управления версиями, вы, вероятно, теперь задаетесь вопросом: какой тип управления версиями предпочтительнее для вас?

Централизованные и распределенные VCS

На самом упрощенном уровне все современные VCS эквивалентны: если система поддерживает атомарную фиксацию изменений в пакете файлов, то остальная часть системы является просто пользовательским интерфейсом. Написав грудку простых сценариев командной оболочки, можно реализовать общую семантику (не рабочий процесс) любой современной VCS. Поэтому решение, какая VCS «лучше», в первую очередь зависит от пользовательского опыта — основные возможности разных VCS одинаковы, но у каждой системы свой пользовательский опыт, именование, пограничные особенности и производительность. Выбор VCS похож на выбор формата файловой системы: если форматы достаточно современные, то различия между ними не имеют особого значения, гораздо важнее, каким содержимым вы наполните эту систему и как будете его использовать. Однако архитектурные отличия VCS могут упростить или усложнить принятие решений о конфигурации, политике использования и масштабировании кода. С учетом этих различий VCS делятся на два типа: централизованные и децентрализованные.

Централизованные VCS

В централизованных VCS используется модель с единым центральным репозиторием (вероятно, хранящимся на общем вычислительном ресурсе организации). Разработчик может хранить на своей локальной машине файлы, извлеченные из репозитория, но операции (добавление файлов, синхронизация, обновление существующих файлов и т. д.), влияющие на состояние этих файлов, должны выполняться на центральном сервере. Любой код, фиксируемый разработчиком, передается в центральный репозиторий. Первые реализации VCS были централизованными.

Самые ранние реализации VCS, появившиеся в 1970-х — начале 1980-х годов, такие как RCS, были ориентированы на блокировки и предотвращение возможности изменения файлов несколькими пользователями. Вы могли скопировать содержимое репозитория к себе, но для редактирования файла требовалось установить блокировку в VCS, чтобы гарантировать, что правки сможете внести только вы. По окончании редактирования необходимо было снять блокировку. Эта модель работала нормально, когда любое изменение происходило быстро и блоки на конкретный код ставил только один разработчик. Модель позволяла с легкостью вносить небольшие правки, например в файлах конфигурации, и сотрудничать в небольшой команде, члены которой работали в разные часы или с разными файлами. Однако в больших командах блокировка может стать предметом конкуренции¹.

В ответ на проблему масштабирования появились VCS, ставшие популярными в 1990-х — начале 2000-х годов. Они не использовали вышеописанные блокировки, а отслеживали синхронно добавленные изменения, требуя, чтобы новые правки основывались на самой последней версии каждого зафиксированного файла. CVS, которую можно считать усовершенствованной версией RCS, работала сразу с пакетами файлов и позволяла нескольким разработчикам посылать измененные файлы одновременно: если изменения были выполнены на основе базовой версии, хранящейся в репозитории, CVS разрешала автору изменений выполнить фиксацию. Система Subversion продвинулась еще дальше и обеспечила истинную атомарность фиксаций, отслеживание версий и лучшее отслеживание редких операций (таких, как переименование, использование символических ссылок и т. д.). Модель централизованного репозитория и клиента до сих пор продолжает использоваться в Subversion и в большинстве коммерческих VCS.

¹ Забавно: чтобы проиллюстрировать это, я посмотрел, какие изменения (ожидаемые/неотправленные) внесены гуглерами в один из самых популярных файлов в моем последнем проекте. На момент написания этой главы в состоянии ожидания находились 27 изменений, 12 — от членов моей команды, 5 — от членов соседних команд и 10 — от инженеров, с которыми я никогда не встречался. Блокировки работают, как ожидалось. Из этого следует, что технические системы или политики, требующие непосредственной координации, не подходят для круглосуточной и распределенной разработки ПО.

Распределенные VCS

Начиная с середины 2000-х годов, многие популярные VCS превратились в распределенные системы управления версиями (DVCS, distributed version control system). Примерами таких систем могут служить Git и Mercurial. Основное концептуальное отличие DVCS от более традиционных централизованных VCS (Subversion, CVS) заключено в ответах на вопрос: «Где фиксировать изменения?» или, возможно, «Какие копии этих файлов считаются репозиторием?»

В DVCS нет ограничений центрального репозитория: если у вас есть копия (клон) репозитория, то у вас есть репозиторий, в котором можно фиксировать изменения, а также все метаданные, необходимые для получения разнообразной информации, например истории изменений. Стандартный рабочий процесс с использованием такой системы включает клонирование существующего репозитория, внесение изменений, их локальную фиксацию, а затем передачу набора фиксаций в другой репозиторий, который может быть или не быть источником копии. Наличие центрального репозитория является чисто концептуальным вопросом политики, а не фундаментальным условием технологии VCS или лежащих в ее основе протоколов.

Не объявляя один конкретный репозиторий источником истины, модель DVCS позволяет улучшить автономную работу и сотрудничество. Ни один из репозиториев не может быть «впереди» или «позади», потому что изменения фактически не проецируются на линейную шкалу времени. Однако, учитывая типичные *сценарии использования*, централизованная и распределенная модели в значительной степени взаимозаменяемы: если централизованная VCS четко определяет центральный репозиторий с помощью технологии, то большинство экосистем DVCS точно так же определяют центральный репозиторий, но уже политическими методами. То есть большинство проектов DVCS построено на одном концептуальном источнике истины (например, на конкретном репозитории в GitHub). Модели DVCS, как правило, предполагают более распределенный сценарий использования и широко распространены в проектах с открытым исходным кодом.

В настоящее время доминирующей VCS является Git, которая реализует модель DVCS¹. Если вы затрудняетесь в выборе VCS, используйте ее — поступая так же, как другие, вы найдете определенные преимущества. Если вы ожидаете необычные сценарии использования, соберите информацию о нескольких системах и сопоставьте их достоинства и недостатки.

В Google сложилось неоднозначное отношение к DVCS: наш основной репозиторий основан на собственной (массивной) централизованной VCS. Периодически предпринимаются попытки интегрировать в нее дополнительные стандартные возможности, чтобы соответствовать рабочему процессу, к которому наши инженеры (особенно нуглеры) привыкли, работая во внешних проектах. К сожалению, все попытки перейти на использование более распространенных инструментов, таких как Git, оказались невозможными из-за огромного размера кодовой базы и большого числа пользова-

¹ См. результаты опроса разработчиков на сайте Stack Overflow (<https://oreil.ly/D173D>), 2018 г.

телей, не говоря уже об эффектах закона Хайрама, связывающих нас с конкретной VCS и ее интерфейсом¹. И это неудивительно: большинство существующих VCS не способны справиться с поддержкой 50 000 инженеров и десятков миллионов фиксаций². Модель DVCS, которая часто (но не всегда) предполагает передачу истории и метаданных, требует большого количества данных, чтобы развернуть репозиторий.

Централизация и облачное хранилище для кодовой базы в нашем рабочем процессе не очень хорошо масштабируется. Модель DVCS основана на идее загрузки всей кодовой базы и ее локального использования. Однако с течением времени и по мере роста организации каждый разработчик работает с постоянно уменьшающимся процентом файлов в репозитории и ограниченным кругом их версий. С ростом количества файлов и инженеров передача всей кодовой базы превращается в напрасную трату ресурсов. Единственная потребность в локальности возникает при сборке, но распределенные (и воспроизводимые) системы сборки (глава 18) лучше подходят и для решения этой задачи, чем DVCS.

Источник истины

Централизованные VCS (Subversion, CVS, Perforce и др.) включают в свой дизайн понятие источника истины: все, что было последним зафиксировано в главной ветви, является текущей версией. Когда разработчик извлекает проект, по умолчанию ему передается именно эта версия из главной ветви. Ваши изменения станут частью «текущей версии», когда вы зафиксируете их поверх прежней текущей версии.

В отличие от централизованных VCS, в распределенных системах отсутствует *врожденное* понятие единого источника истины. Теоретически можно передавать теги фиксации и запросы на включение изменений (PR, pull request) без централизации или координации, но это способствует беспрепятственному распространению разрозненных ветвей разработки и, как следствие, риску концептуального возврата к `Presentation v5 - final - redlines - Josh's version v2`. По этой причине применение DVCS требует определения более четких правил и норм, чем применение централизованной VCS.

В хорошо управляемых проектах, использующих DVCS, одна конкретная ветвь в одном конкретном репозитории объявляется источником истины, что исключает хаотичное появление других ветвей. Мы часто наблюдаем это на практике, в решениях с распределенным размещением на основе DVCS, таких как GitHub или GitLab, — пользователи могут клонировать репозиторий проекта и создавать свои ветви, но

¹ Больше всего неприятностей доставляет использование монотонно увеличивающихся номеров версий вместо хешей фиксации. Многие системы и сценарии были созданы в экосистеме разработчиков Google, которые предполагали, что нумерация фиксаций всегда будет совпадать с их очередностью во времени, — отменить такие скрытые зависимости сложно.

² К слову, на момент публикации статьи «Мопогеро» в репозитории хранилось около 86 Тбайт данных и метаданных без учета ветвей с версиями. Создать копию такого репозитория непосредственно на рабочей станции разработчика было бы... сложновато.

изменения становятся частью «текущей версии» только после их фиксации в главной ветви первичного репозитория.

Нас не удивляет, что централизация и понятие источника истины нашли применение даже в мире DVCS. Чтобы проиллюстрировать, насколько важна идея источника истины, представим, что может случиться при его отсутствии.

Сценарий: нет четко определенного источника истины

Представьте, что ваша команда решила придерживаться философии DVCS до такой степени, что отказалась определить конкретные ветвь и репозиторий в качестве единого источника истины.

В некотором смысле это напоминает модель `Presentation v5 - final - redlines - Josh's version v2` — после извлечения кода из репозитория своего товарища по команде вы не сможете четко определить, какие изменения зафиксированы, а какие — нет. Это неплохо, потому что модель DVCS отслеживает слияние отдельных «заплат» с гораздо более высокой степенью детализации, чем специальные схемы именования, но одно дело, когда DVCS «знает», *какие* изменения зафиксированы, и совсем другое, когда каждый инженер ошибочно уверен, что он в курсе *всех* прошлых и текущих изменений.

Подумайте, как в таких условиях обеспечить включение в сборку новой версии всех функций, написанных каждым разработчиком за последние несколько недель. Какие существуют (децентрализованные и масштабируемые) механизмы для этого? Можно ли разработать политику, которая в корне лучше, чем простое соглашение? Не повлечет ли рост команды экспоненциального роста человеческих усилий для поддержания инфраструктуры? Сохранится ли нормальная работа инфраструктуры с увеличением числа разработчиков в команде? Насколько мы можем судить, ответы на все вопросы — нет. Без центрального источника истины кто-то должен будет вести список функций, потенциально готовых к включению в следующую версию. В итоге этот учет будет воспроизводить модель централизованного источника истины.

Теперь представьте, что к команде присоединяется новый разработчик. Откуда он получит самую свежую копию кода?

DVCS поддерживают множество рабочих процессов и моделей использования. Но если вам нужна система, для управления которой не требуется экспоненциального увеличения человеческих усилий с ростом команды, то вам придется организовать единое хранилище (и одну ветвь), которое фактически будет служить источником истины.

В этом понятии источника истины есть некоторая относительность: источником истины для одного и того же проекта в разных организациях могут служить разные репозитории. Это важно: для инженеров Google и RedHat разумно иметь разные источники истины для исправлений ядра Linux, которые, в свою очередь, отличаются от источника истины, который использует сам Линус (создатель ядра Linux). DVCS отлично работает, когда организации и их источники истины имеют иерархическую

структуру (и невидимы за пределами организации), — это, пожалуй, наиболее ценное достоинство модели DVCS. Инженер из RedHat может фиксировать свои наработки в локальном репозитории — источнике истины — и оттуда периодически передавать изменения вверх по иерархии, при этом у Линуса будет совершенно иное представление об источнике истины. До тех пор пока есть определенность в том, куда направлять изменения, вы можете не беспокоиться о проблеме хаотического масштабирования в модели DVCS.

Во всех этих размышлениях мы придаем особое значение главной ветви. Конечно, выбор главной ветви в вашей VCS — это необходимость по умолчанию, и кроме этого, организация может устанавливать свои правила. Может так случиться, что ветвь по умолчанию будет оставлена и вся работа фактически сосредоточится в какой-то другой ветви — кроме дополнительной необходимости указывать имя ветви в большом количестве операций, в этом подходе нет ничего принципиально неправильного — просто он нестандартный. В обсуждениях управления версиями предполагается (часто негласно), что применение технологии VCS должно быть согласовано со сводом правил и соглашений в организации.

Ни одна тема в сфере управления версиями не содержит больше правил и соглашений, чем обсуждение особенностей использования ветвей и управления ими. Мы рассмотрим управление ветвями более подробно в следующем разделе.

Управление версиями и управление зависимостями

Обсуждения правил управления версиями и управления зависимостями (глава 21) имеют массу концептуальных сходств. Различия этих видов управления в основном проявляются в двух аспектах: правила использования VCS обычно более детализированы и в основном затрагивают управление своим кодом, в то время как управление зависимостями требует больше нестандартных решений и касается проектов, управляемых другими организациями. Далее в книге мы подробно обсудим эти аспекты.

Управление ветвями

Возможность отслеживать различные исправления в VCS открывает множество подходов к управлению версиями. В совокупности эти подходы называются *управлением ветвями*, растущими от главной ветви.

Незавершенная работа похожа на ветвь

В любом обсуждении правил управления ветвями внутри организации должно учитываться, что всякая незавершенная работа эквивалентна ветви. Это особенно сильно проявляется в DVCS, где разработчики могут совершать множество локальных промежуточных фиксаций, прежде чем передать изменения в источник истины. Это также верно для централизованных VCS: незафиксированные локальные изменения ничем не отличаются от изменений, зафиксированных в отдельной ветви, просто их

труднее найти и сравнить. Некоторые централизованные системы даже стараются сделать такие ветви более явными. Например, в Perforce каждому изменению присваиваются два номера: один определяет неявную точку ветвления, в которой было создано изменение, а другой — точку, в которой оно было зафиксировано (рис. 16.1). Пользователи Perforce могут проверять, у кого есть незавершенные изменения в файле, исследовать ожидающие фиксации изменения других пользователей и делать многое другое.

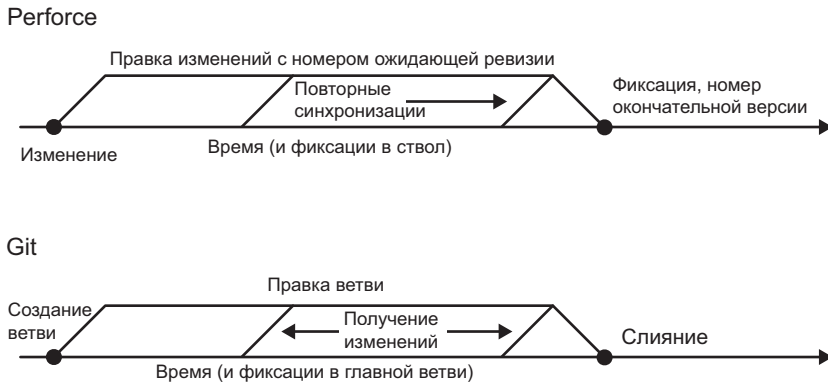


Рис. 16.1. Два номера ревизии в Perforce

Идея «сходства незавершенной работы с ветвью» особенно актуальна при рефакторинге. Представьте, что разработчику говорят: «Переименуй `widget` в `oldWidget`». В зависимости от политики управления ветвями и понимания, что считается ветвью и какие ветви имеют значение, переименовать `widget` можно по-разному:

- в главной ветви репозитория, служащего источником истины;
- во всех ветвях в репозитории, служащем источником истины;
- во всех ветвях в репозитории, служащем источником истины, и найти всех разработчиков, имеющих незавершенные изменения в файлах, которые содержат ссылки на `widget`.

Продолжая рассуждения в этом направлении, можно прийти до варианта «переименовать везде, даже в незавершенных изменениях», и тогда станет ясно, почему коммерческие централизованные VCS, как правило, отслеживают, «у каких инженеров этот файл открыт для редактирования?» (Мы не считаем масштабируемым такой способ *выполнения* рефакторинга, но относимся к нему с пониманием.)

Ветви разработки

В эпоху, предшествовавшую распространению юнит-тестирования (глава 11), когда любое изменение было сопряжено с немалым риском что-то сломать в другом

месте, имело смысл с особым трепетом относиться к *главной ветви*. Технический руководитель мог сказать: «Мы не будем фиксировать изменения в главной ветви, пока они не пройдут полный цикл тестирования. Наша команда будет использовать специализированные ветви для разработки».

Ветвь разработки (*dev branch*) — это промежуточная точка между «это сделано, но не зафиксировано» и «это текущая версия, на которой будет основана новая работа». Проблема, которую такие ветви пытаются решить (нестабильность продукта), очевидна, но мы обнаружили, что она решается лучше более широким использованием тестов, непрерывной интеграцией (глава 23) и качественным набором практик, таких как обзор кода.

Мы считаем, что политика управления версиями, широко использующая ветви разработки как средство обеспечения стабильности продукта, изначально ошибочна, поскольку те же самые изменения все равно придется фиксировать в главной ветви. Маленькие слияния делаются проще, чем большие. Слияние, выполняемое автором изменений, происходит проще, чем пакетное слияние несвязанных изменений (что в конечном итоге происходит, если команда использует ветвь разработки). Если предварительное тестирование перед слиянием выявит новые проблемы, определить, чьи изменения являются причиной ошибки, легче, если изменения принадлежат одному инженеру. Слияние большой ветви разработки предполагает участие в предварительном тестировании большого количества изменений, что затрудняет локализацию сбоев. Выяснить причину проблемы будет сложно, а исправить ее — еще сложнее.

Помимо отсутствия опыта слияния ветвей и врожденных проблем этого процесса учитывайте значительные риски потери масштабирования и снижения продуктивности инженеров. Когда в течение длительных периодов разрабатывается сразу несколько ветвей, координация операций слияния становится более дорогостоящей (и рискованной), чем разработка в главной ветви.

Как появилась зависимость от ветвей разработки?

Легко увидеть, как организации попадают в эту ловушку: они видят, что «слияние этой долгоживущей ветви разработки снижает стабильность», и приходят к выводу, что «слияние ветвей сопряжено с риском». Вместо того чтобы улучшить тестирование и отказаться от политики разработки на основе ветвей, они создают новые ветви на основе действующих ветвей. Команды, работающие над долгоживущей ветвью разработки, могут регулярно синхронизировать эту ветвь с главной ветвью разработки. Но с увеличением масштабов организации также растет число ветвей разработки и приходится прикладывать все больше усилий для координации слияния ветвей, которое не масштабируется. Один несчастный инженер начинает координировать сборку, слияния и управление содержимым и становится главным координатором слияния всех разрозненных ветвей в организации. На регулярно проводимых встречах делаются попытки «выработать

стратегию слияния на неделю»¹. Команды, слияние ветвей которых отложено на будущее, вынуждены повторно синхронизироваться и тестировать свой код после каждого крупного слияния.

Все эти усилия по слиянию и повторному тестированию — это *чистые накладные расходы*. Альтернативный подход требует другой парадигмы: разработки в главной ветви, сильной зависимости от тестирования и непрерывной интеграции, сохранения зеленого индикатора на этапе сборки и отключения незаконченных/непроверенных функций во время выполнения. Каждый несет свою долю ответственности за синхронизацию с главной ветвью и фиксацию изменений. Никаких встреч по «политике слияния», никаких крупных дорогих слияний, никаких жарких дискуссий о выборе версии библиотеки для использования — есть только одна действующая версия и один источник истины. В конце концов, выпускаться будет только одно исправление: ограничение единственным источником истины — это просто «сдвиг влево» с целью определить, что нужно включить в это исправление.

Ветви версий

Если период между версиями (срок действия версий) продукта превышает несколько часов, разумно создать ветвь версии, которая определяет код, вошедший в сборку версии продукта. Если между текущей и следующей версиями продукта обнаружатся какие-либо критические недостатки, необходимые исправления можно будет добавить из главной ветви в ветвь версии (выполнить минимальное целевое слияние).

По сравнению с ветвями разработки, ветви версий обычно безобидны: проблема не в технологии ветвей, а в их использовании. Основное отличие ветвей разработки от ветвей версий — ожидаемое конечное состояние: ожидается, что ветвь разработки снова объединится с главной ветвью и, возможно, будет разветвлена другой командой. Ветвь версии, напротив, останется в фиксированном состоянии.

В наиболее эффективных технологических организациях, выявленных компанией Google DevOps Research and Assessment (DORA), ветви версий практически не создаются. Компании, организовавшие у себя процесс непрерывного развертывания, позволяющий выпускать обновления из основной ветви несколько раз в день, скорее всего, не будут создавать ветви версий: для них намного проще добавить исправление и выполнить повторное развертывание. Они считают создание ветвей ненужными затратами. Очевидно, ветви версий больше применимы к организациям, которые развертывают продукты в цифровом виде (например, веб-службы и приложения), чем к компаниям, которые продвигают свои продукты в материальном виде. Для последних важно, что именно было предложено клиентам.

¹ Недавно проведенный неофициальный опрос в Twitter показывает, что около 25 % инженеров-программистов участвовали в «регулярных» встречах по определению политики слияния.

В том же исследовании DORA отмечается сильная положительная корреляция между «разработкой в главной ветви», «отсутствием долгоживущих ветвей разработки» и хорошими техническими результатами. Суть этих идей кажется очевидной: ветви снижают продуктивность. Во многих случаях мы воспринимаем сложные политики ветвления и слияния как поддержку безопасности — попытку сохранить стабильность главной ветви. Как не раз было показано в этой книге, есть и другие способы достижения этой цели.

Управление версиями в Google

Подавляющее большинство исходного кода в Google хранится в едином (монолитном) репозитории, которым пользуются примерно 50 000 инженеров. Там находятся почти все проекты, принадлежащие Google, за исключением крупных проектов с открытым исходным кодом, таких как Chromium и Android. К ним относятся такие общедоступные продукты, как Google Search, Gmail, наши рекламные продукты, Google Cloud Platform, а также инфраструктура для поддержки и развития всех этих продуктов.

Мы полагаемся на разработанную у нас централизованную VCS под названием Piper, которая действует как микросервис в нашем продакшене. Она позволяет использовать стандартные в Google технологии хранения, связи и вычислений как услуги для организации глобально доступной VCS, в которой хранится более 80 Тбайт данных и метаданных. Каждый день с монолитным репозиторием Piper одновременно работают тысячи инженеров. Сотрудники и полуавтоматические процессы, которые используют управление версиями (улучшают что-то, хранящееся в VCS), каждый день выполняют от 60 000 до 70 000 фиксаций. В Piper распространены двоичные артефакты, потому что они не требуют передачи полного репозитория и к ним не применяются обычные затраты. Из-за ориентации на масштабирование с самого начала развития компании Google операции в экосистеме VCS остаются недорогими с точки зрения продуктивности: создание нового клиента в главной ветви, добавление файла и фиксация (непроверенного) изменения занимают всего 15 секунд. Такая малая задержка и хорошо понятное масштабирование значительно упрощают работу инженера.

Поскольку Piper — внутренний продукт, у нас есть возможность настраивать его и применять любые политики управления версиями. Например, мы используем дробное владение в монолитном репозитории: на каждом уровне иерархии файлов есть файлы OWNERS с фамилиями инженеров, которым разрешено одобрять изменения в поддереве (в дополнение к владельцам, перечисленным в файлах OWNERS на более высоких уровнях дерева). В среде с множеством репозиториях аналогичного владения можно достичь за счет создания отдельных репозиториях, имеющих доступ к файловой системе и управляющих возможностью фиксации, или с помощью «ловушек фиксаций» в Git (процедур, запускаемых во время фиксации) для проверки разрешений. Управляя своей VCS, мы можем сделать владение и одобрение

более явными и гарантировать их соблюдение на этапе выполнения фиксации. Это гибкий подход: владение определяется простым текстовым файлом, не привязанным к физическому разделению репозитория, который легко обновить при переходе команд или реструктуризации организации.

Единственная версия

Невероятные возможности масштабирования одной только системы Pipeit не могли бы обеспечить такого уровня сотрудничества, на которое мы рассчитываем. Как уже говорилось выше, управление версиями — это еще и политика. В дополнение к нашей VCS, еще одной ключевой особенностью политики управления версиями в Google является идея так называемой «единственной версии». Она дополняет идею «единственного источника истины», которую мы рассмотрели выше, и гарантирует, что разработчик всегда знает, какая ветвь и какой репозиторий являются источником истины для него. То есть фактически эта идея требует, чтобы «для каждой зависимости в репозитории имелась только одна версия»¹. Для сторонних пакетов это означает, что в репозитории в устойчивом состоянии может находиться только одна версия этого пакета². Для внутренних пакетов это означает невозможность ветвления без переупаковки или переименования, то есть исключение любой возможности смешивания оригинала и его новой версии (производной) в одном проекте. Это мощная особенность нашей экосистемы, и в ней очень мало пакетов с ограничениями вида «если вы подключили пакет А, то не можете подключить пакет В».

Идея наличия единственной копии в единственной ветви в единственном репозитории, играющей роль источника истины, интуитивно понятна, но имеет некоторые тонкости применения. Рассмотрим сценарий, в котором есть монолитный репозиторий (что, казалось бы, соответствует требованию наличия единственного источника истины), в главную ветвь которого попали несколько версий библиотек.

Сценарий: несколько доступных версий

Представьте, что команда обнаруживает ошибку в коде общей инфраструктуры (например, Abseil или Guava), но вместо ее исправления на месте решает создать новую версию этой инфраструктуры и внести в нее изменения, исправляющие ошибку, — без переименования библиотеки или любых ее символов. Эта команда сообщает соседним командам: «Привет, мы создали улучшенную версию Abseil, пользуйтесь ею». Несколько других команд создают библиотеки, полагающиеся на эту новую производную версию.

¹ Например, во время операции обновления в репозитории могут быть зарегистрированы две версии, но если разработчик добавляет новую зависимость в существующий пакет, у него не должно быть возможности выбирать версию зависимости.

² Часто нам это не удается, потому что внешние пакеты иногда имеют закрепленные копии своих зависимостей, входящих в их версию исходного кода. Подробнее о возможных последствиях в главе 21.

Как будет показано в главе 21, это решение создает опасную ситуацию. Если какой-то проект в кодовой базе будет зависеть сразу от обеих версий Abseil, исходной и производной, то в лучшем случае сборка не выполнится, а в худшем случае — возникнут трудные для понимания ошибки времени выполнения, возникающие из-за связывания кода с двумя несовместимыми версиями одной и той же библиотеки. Производная версия фактически добавила раскраску/разделение в базу кода. Набор переходных зависимостей для любой заданной цели должен включать в себя ровно одну копию библиотеки. Любая ссылка, добавленная из раздела исходной версии в соответствующий раздел производной версии, почти наверняка нарушит работоспособность кода. В конечном итоге такая простая операция, как «добавление новой зависимости», может потребовать выполнения всех тестов для всей кодовой базы ради соблюдения требований этого разделения. Это дорого, утомительно и плохо масштабируется.

Иногда можно применить грубый прием жесткого кодирования и добиться, чтобы выполняемый файл работал правильно. В Java, например, есть относительно стандартная практика, называемая *затенением* (<https://oreil.ly/RuWX3>), при которой вы меняете имена внутренних зависимостей библиотеки, чтобы скрыть их зависимости от остальной части приложения. При работе с функциями это технически разумно, даже если теоретически это выглядит немного грубо. При работе с типами, которые можно передавать из одного пакета в другой, решения на основе затенения не работают ни в теории, ни на практике. Насколько нам известно, для типов не действуют никакие технологические уловки, позволяющие нескольким изолированным версиям библиотеки действовать в одном двоичном файле, например попытка использовать несколько версий любой библиотеки, предлагающей тип словаря (или любую конструкцию высокого уровня), обречена на неудачу. Практика затенения и связанные с ней подходы устраняют основную проблему: необходимость использования нескольких версий одной и той же зависимости. (Подробнее о том, как минимизировать опасность при нескольких доступных версиях, в главе 21.)

Любая система политик, допускающая использование нескольких версий в одной кодовой базе, способствует проникновению в код дорогостоящих несовместимостей. Возможно, вам удастся на время решить эту проблему (нам иногда это удается), но в целом любая ситуация с несколькими версиями может привести к большим проблемам.

Правило «единственной версии»

Глядя на этот пример и описание модели единого источника истины, можно понять и оценить всю глубину, казалось бы, простого правила управления версиями и ветвями:

У разработчиков не должно быть выбора, «от какой версии компонента им зависеть».

Для простоты это правило можно назвать «правилом единственной версии». На практике «единственная версия» не является неизменной¹, но формулировка ограничения *в выборе* версий для добавления новой зависимости задает верное направление в работе с версиями.

Отдельному разработчику отсутствие выбора может показаться малопонятным препятствием. Однако для организаций это отсутствие выбора — важный компонент эффективного масштабирования. Согласованность имеет огромное значение на всех уровнях организации, от обсуждения общей согласованности в компании до обеспечения возможности согласованного использования «узких мест».

(Почти) полный отказ от долгоживущих ветвей

В наше правило о единственности версии заложено несколько глубоких идей. Главная из них: количество ветвей разработки должны быть минимальным, и эти ветви должны существовать очень короткое время. Это следует из большого количества работ, опубликованных за последние двадцать лет, от описаний процессов гибкой разработки до результатов исследований DORA по разработке в главной ветви и даже уроков Phoenix Project² по «сокращению незавершенного производства». Представление незавершенной работы как ветви разработки еще больше усиливает требование выполнять работу небольшими шагами и регулярно проводить фиксацию изменений в главную ветвь.

В качестве контрпримера: в сообществе разработчиков, которое в значительной степени зависит от долгоживущих ветвей разработки, нетрудно представить возврат к возможности выбора.

Представьте такой сценарий: команда, занимающаяся развитием инфраструктуры, работает над новым виджетом, который лучше старого. Волнение нарастает: из других проектов, запущенных недавно, команде поступают вопросы: «Можем ли мы использовать ваш новый виджет?» Но не стоит торопиться «разрешать» зависимость от нового виджета, если он существует только в параллельной ветви. Помните: у новых проектов не должно быть выбора при добавлении зависимостей. Этот новый виджет должен быть зафиксирован в главной ветви, недоступен для использования, пока не будет полностью готов, и по возможности скрыт от других разработчиков с помощью правил видимости. Либо две версии виджета должны быть спроектированы так, чтобы они могли сосуществовать в одних и тех же программах.

¹ Например, бывает невозможно обновить внешнюю или стороннюю библиотеку и весь код, использующий ее, в одном атомарном изменении. Поэтому часто приходится добавлять новую версию библиотеки, запрещать новым пользователям добавлять зависимости от старой и постепенно переводить код с использования старой библиотеки на новую.

² Бер К., Ким Дж., Спаффорд Дж. Проект «Феникс». Роман о том, как DevOps меняет бизнес к лучшему. М.: Эксмо, 2014. — *Примеч. пер.*

Отметим, что уже есть свидетельства того, насколько правило единственной версии важно для отрасли. В книге «Ускоряйся!»¹ и в самых последних отчетах «State of DevOps» ассоциации DORA указывается, что существует тесная взаимосвязь между разработкой в главной ветви и эффективностью softверных организаций. Компания Google — не единственная организация, которая обнаружила это. Конечно, мы не всегда получали ожидаемые результаты, развивая эту политику, просто нам казалось, что никакие другие подходы не позволяют достичь желаемого результата. Однако результаты, полученные в DORA, безусловно, соответствуют нашему опыту.

Наши политики и инструменты для крупномасштабных изменений (глава 22) придают дополнительную важность идее разработки в главной ветви: обширные (поверхностные) изменения, которые применяются ко всей кодовой базе, уже стали массовым (часто утомительным) явлением при изменении всего, что находится в главной ветви. Наличие неограниченного количества дополнительных ветвей разработки, которые могут потребовать синхронного рефакторинга, повлекло бы гигантские затраты на выполнение изменений такого рода, обусловленные необходимостью поиска в постоянно расширяющемся наборе скрытых ветвей. А в модели распределенной VCS может даже оказаться невозможным идентифицировать все такие ветви.

Конечно, наш опыт не универсален. Вы можете оказаться в необычной ситуации, требующей создания долгоживущей ветви разработки, существующей параллельно главной ветви (и регулярно объединяющейся с ней).

Такие сценарии должны быть редкими, и их следует расценивать как дорогостоящие. Из примерно 1000 команд, которые работают в монолитном репозитории Google, лишь несколько имеют подобные ветви разработки². Обычно эти ветви создаются по очень конкретной (необычной) причине, например: «У нас необычное требование к длительному сохранению совместимости». Часто такой причиной является обеспечение совместимости по данным между версиями: код, читающий и записывающий файлы в некотором формате, должен продолжать использовать согласованное представление о формате и после внесения изменений в него. В других случаях причиной может стать сохранение совместимости API, например если при выпуске новой версии нужно гарантировать сохранение работоспособности более старой версии клиента микросервисов при работе с новым сервером (или наоборот). Такая гарантия может оказаться очень сложным требованием, и вы не должны легкомысленно давать долгосрочные гарантии, работая над активно развивающимся API, чтобы период времени их действия не начал расти вопреки вашим желаниям. Зависимость кода от времени в любой ее форме обходится дорого. Внутренние службы Google редко дают такие гарантии³. Мы также здорово выигрываем, ограничивая

¹ Форсгрэн Н., Хамбл Дж., Ким Дж. Ускоряйся! Наука DevOps. Как создавать и масштабировать высокопроизводительные цифровые организации. Интеллектуальная литература, 2020. — Примеч. пер.

² Точно подсчитать довольно трудно, но таких команд меньше 10.

³ Другое дело — облачные интерфейсы.

потенциальную асимметрию версий, обусловленную нашим «горизонтом сборки»: все службы в продакшене должны повторно собираться и развертываться не реже, чем один раз в шесть месяцев (обычно гораздо чаще).

Мы уверены, что есть и другие ситуации, когда могут понадобиться долгоживущие ветви разработки. Просто постарайтесь использовать их как можно реже. Если вы решите использовать у себя другие инструменты и методы, описанные в этой книге, то многие из них будут оказывать давление на долгоживущие ветви разработки. Инструменты и средства автоматизации, прекрасно работающие с главной ветвью и терпящие неудачу (или требующие больших усилий) при работе с ветвью разработки, могут помочь разработчикам не отставать от жизни.

А что насчет ветвей версий?

Многие команды в Google подходят к использованию ветвей избирательно. Если вы собираетесь выпускать новые версии ежемесячно и продолжать работу над следующей версией, вполне разумно создать ветвь версии. Точно так же, если вы собираетесь поставлять устройства клиентам, важно точно знать, какая версия находится в «эксплуатации». Будьте внимательны и рассудительны при создании таких ветвей и не планируйте объединять их с главной ветвью. Наши команды определяют всевозможные политики в отношении ветвей, учитывая, что лишь немногие достигли частой периодичности выпуска, которую обещает методология непрерывного развертывания (глава 24), избавляющая от необходимости или желания создавать ветви версий. В целом, как показывает наш опыт, ветви версий не влекут за собой больших затрат (или, по крайней мере, никаких заметных затрат, кроме дополнительных затрат на поддержку VCS).

Монолитные репозитории

В 2016 году мы опубликовали (часто цитируемый и широко обсуждаемый) документ о подходе компании Google к монолитным репозиториям¹. Главное преимущество монолитных репозиториях — простота следования правилу единственной версии: нарушить это правило в репозитории сложнее, чем следовать ему. У нас нет процесса, определяющего, какие версии являются официальными или какие репозитории наиболее важны. Инструменты сборки (глава 23) тоже не требуют определения важных репозиториях. Повысить эффективность внедрения новых инструментов и оптимизаций помогает согласованность. В целом инженеры могут видеть действия коллег и использовать эту информацию при выборе решений в своем коде и проекте своей системы.

Учитывая все это и нашу веру в преимущества правила единственной версии, возникает вопрос: является ли использование монолитного репозитория самым верным

¹ Potvin R., Levenberg J. Why Google stores billions of lines of code in a single repository. *Communications of the ACM*, 59 No. 7 (2016): 78–87.

подходом. Для сравнения: сообщество открытого ПО с успехом использует подход, основанный на применении кажущегося бесконечным множества некоординируемых и несинхронизируемых репозиториях проектов.

Если отвечать кратко, то мы не думаем, что подход на основе монолитного репозитория, как мы его описали, является идеальным для всех. Продолжая параллель между форматом файловой системы и VCS, легко представить выбор между использованием 10 дисков для одной большой логической файловой системы или 10 небольших файловых систем, доступ к которым осуществляется раздельно. В мире файловых систем оба подхода имеют свои плюсы и минусы. С технической точки зрения оценка выбора файловой системы может зависеть от устойчивости к сбоям, ограничений размеров файлов, характеристик производительности и т. д. Оценки с точки зрения удобства, скорее всего, в большей степени будут зависеть от возможности ссылаться на файлы через границы файловой системы, добавлять символические ссылки и синхронизировать файлы.

Аналогично определяется предпочтительность монолитных репозиториях или коллекций более мелких репозиториях. Всякое конкретное решение о том, как хранить исходный код (то есть файлы), легко оспорить, и в некоторых случаях особенности организации и рабочего процесса будут иметь большее значение при выборе хранилища кода, чем другие аргументы. Это решение вы должны принять самостоятельно.

Важна не сосредоточенность кода на монолитном репозитории, а стремление максимально придерживаться принципа единственной версии: у разработчиков не должно быть выбора при добавлении зависимости в библиотеку, которая уже используется в организации. Нарушение правила единственной версии приводит к необходимости обсуждения политики слияния, появлению ромбовидных зависимостей, потере времени и сил.

Инструменты программной инженерии, включая VCS и системы сборки, предоставляют механизмы для комбинирования разрозненных и монолитных репозиториях, чтобы обеспечить согласование фиксаций и анализ графа зависимостей как в монолитном репозитории. Подмодули Git, система сборки Bazel с внешними зависимостями и подпроекты CMake позволяют современным разработчикам синтезировать нечто, имитирующее поведение монолитного репозитория без свойственных ему затрат и недостатков¹. Например, с разрозненными мелкими репозиториями легче работать с точки зрения масштабирования (в Git часто возникают проблемы с производительностью после нескольких миллионов фиксаций и наблюдается склонность к замедлению процедуры клонирования при наличии в репозитории больших двоичных артефактов) и размера хранилища (метаданные VCS могут постепенно накапливаться, особенно если в VCS имеются двоичные артефакты). Мелкие репозитории в объединенном виртуальном монолитном репозитории (VMR, Virtual-MonoRepo) могут упростить изоляцию экспериментальных или секретных

¹ Мы не думаем, что это легко реализовать, но идея хранения зависимостей в разных репозиториях и организации виртуального монолитного репозитория явно витает в воздухе.

проектов, обеспечивая соблюдение принципа единственной версии и предоставляя доступ к общим утилитам.

Таким образом, если все проекты в организации имеют одинаковые требования к секретности, законности, конфиденциальности и безопасности, используйте монолитный репозиторий¹. В противном случае лучше *отдать предпочтение* функциональности монолитного репозитория, но реализовать ее другим способом. Если у вас есть возможность управлять отдельными репозиториями и придерживаться принципа единственной версии или ваша рабочая нагрузка достаточно разобщена, чтобы использовать отдельные репозитории, примените коллекцию репозитория. В противном случае — создайте что-то вроде VMR.

В конце концов, выбор формата файловой системы действительно не так важен, как то, что вы в нее пишете.

Будущее управления версиями

Компания Google — не единственная организация, публично обсуждающая преимущества подхода на основе монолитного репозитория. Microsoft, Facebook, Netflix и Uber тоже публично заявили, что используют этот подход. Ассоциация DORA много писала об этом. Маловероятно, что эти успешные компании заблуждаются или их опыт неприменим к организациям меньшего размера.

Большинство аргументов против монолитных репозитория сводятся к техническим ограничениям, связанным с организацией единственного репозитория. Если клонирование репозитория происходит быстро и не требует больших затрат, разработчики с большей вероятностью будут придерживаться небольших и изолированных изменений (и избегать ошибок, связанных с фиксацией в неправильной ветви для разработки). Если клонирование репозитория (или выполнение любой другой стандартной операции VCS) потребует от разработчика впустую тратить свое время, то легко понять, что организация будет избегать использования такого большого репозитория. К счастью, мы смогли не попасть в эту ловушку, уделив должное внимание созданию масштабируемой VCS.

Если посмотреть на серьезные улучшения Git за последние несколько лет, то становится очевидным, как много работы было сделано для поддержки крупных репозитория: поверхностное клонирование, разреженные ветви, оптимизация и многое другое. Мы полагаем, что этот процесс продолжится и стремление к «небольшому хранилищу» уменьшится.

Другой важный аргумент против монолитных репозитория: они не соответствуют процессу разработки ПО с открытым исходным кодом. Да, это правда, но многие практики в мире открытого исходного кода обусловлены приоритетом свободы, от-

¹ Или у вас есть желание и возможность настроить и поддерживать VCS на протяжении всего срока службы кодовой базы или организации. В противном случае — лучше избегать такого варианта, потому что он влечет много накладных расходов.

сутствием координации и нехваткой вычислительных ресурсов. Раздельные проекты в мире открытого исходного кода фактически являются отдельными организациями, которые могут видеть код друг друга, управлять объемом вычислительных ресурсов, координировать работу и централизовать управление.

Менее распространенная, но, пожалуй, более обоснованная проблема, связанная с использованием монолитного репозитория, заключается в том, что с ростом организации уменьшается вероятность соответствия каждого фрагмента кода одним и тем же юридическим и нормативным требованиям, а также требованиям к секретности и конфиденциальности. Одним из естественных преимуществ разрозненных репозитория является возможность иметь разные группы авторизованных разработчиков, правила видимости, разрешения и т. д. Вшить все то же самое в монолитный репозиторий можно, но это потребует дополнительных расходов на настройку и обслуживание.

В то же время отрасль, похоже, снова и снова изобретает средства связи между репозиториями. Иногда они помещаются в VCS (как, например, подмодули в Git) или в систему сборки. Если набор репозитория поддерживает согласованное представление о том, «что такое главная ветвь» и «какое изменение произошло первым», и имеет механизмы описания зависимостей, то можно легко представить объединение разрозненной коллекции физических репозитория в один общий виртуальный монолитный репозиторий. Несмотря на то что система Pipe очень хорошо зарекомендовала себя, инвестиции в масштабируемый виртуальный монолитный репозиторий и инструменты для управления им, а также использование готовых настроек и политик для каждого репозитория могли бы оказаться более выгодным вложением.

Мы уверены, что как только кто-то в сообществе открытого исходного кода создаст достаточно большую группу совместимых и взаимозависимых проектов и опубликует их представление в виртуальном монолитном репозитории, практика разработки с открытым исходным кодом начнет меняться. Мы видим такую тенденцию в инструментах, *способных* синтезировать виртуальные монолитные репозитории, а также в работе, проделанной (например) крупными разработчиками дистрибутивов Linux, публикующими взаимно совместимые версии тысяч пакетов. Благодаря юнит-тестам, непрерывной интеграции и автоматическому подбору версий для новых исправлений владельцы могут обновлять главную ветвь своего пакета (конечно, без нарушения правил), и мы думаем, что эта модель завоеует популярность в мире открытого ПО. В конце концов, это просто вопрос эффективности: подход на основе (виртуального) монолитного репозитория с правилом единственной версии сокращает сложность разработки ПО на целое (сложное) измерение — время.

Мы ожидаем, что в ближайшие 10–20 лет управление версиями и зависимостями будет развиваться в следующем направлении: VCS сосредоточатся на *поддержке* более крупных репозитория с улучшенным масштабированием производительности, но при этом избавят от необходимости использовать большие репозитории, предложив более совершенные механизмы объединения данных в границах проектов и организаций. Кто-то, возможно, группа управления пакетами или разработчик дис-

трибутивов Linux, станет автором создания стандартного виртуального монолитного репозитория. С помощью утилит этот монолитный репозиторий обеспечит легкий доступ к совместимому набору зависимостей как к одному целому. В целом мы понимаем, что номера версий — это временные метки и что разрешение асимметрии версий добавляет сложность дополнительного измерения (времени), что требует больших затрат, которых мы можем научиться избегать. Но переход начнется с создания чего-то логически похожего на монолитный репозиторий.

Заключение

VCS являются врожденным компонентом совместной работы, основанной на общих вычислительных ресурсах и компьютерных сетях. Они развивались в соответствии с понятными нам нормами программной инженерии.

Ранние VCS использовали упрощенные механизмы блокировки на уровне файлов. Но по мере роста типичных проектов и команд, занимающихся программной инженерией, проблемы масштабирования этого подхода становились все более очевидными, и наше понимание управления версиями изменилось. С развитием модели разработки открытого ПО со множеством распределенных участников VCS стали более децентрализованными. Мы ожидаем сдвига в технологии VCS, которая предполагает постоянную доступность сети и уделяет больше внимания хранению данных и созданию облака, чтобы избежать передачи ненужных файлов и артефактов. Этот сдвиг становится все более важным для крупных, долгоживущих проектов программной инженерии, даже при том что он означает изменение подхода, использовавшегося в простых проектах с одним разработчиком и с одной машиной. Сдвиг в сторону облачных технологий конкретизирует подход DVCS: даже если разрешена распределенная разработка, все равно требуется использовать централизованный источник истины.

Текущая децентрализация в DVCS — это естественная реакция технологии на потребности отрасли (особенно сообщества открытого ПО). Однако конфигурация DVCS должна строго контролироваться и сочетаться с политиками управления ветвями, принятыми в организации. Часто это может приводить к неожиданным проблемам с масштабированием: для безупречной работы в автономном режиме требуется гораздо больше локальных данных. Неспособность обуздать потенциальную сложность свободного ветвления может привести к неограниченному росту накладных расходов на разработку и развертывание кода. Однако сложные технологии необязательно должны использоваться комплексно: как мы видели, простота политики ветвления в моделях разработки на основе монолитного репозитория и главной ветви обычно способствует плодотворной работе.

Излишняя свобода выбора ведет к издержкам. Мы поддерживаем представленное здесь правило единственной версии: разработчики в организации не должны иметь выбора, куда выполнять фиксации или от какой версии существующего компонента зависеть. Мы знаем лишь несколько допустимых политик, дающих выбор: они могут раздражать отдельных разработчиков, но улучшают конечный результат.

Итоги

- Используйте управление версиями для любых проектов разработки ПО, размер которых превышает размер «игрушечного проекта с одним разработчиком, который никогда не будет обновляться».
- Возможность выбора «от какой версии зависеть» всегда порождает проблемы масштабирования.
- Правило единственной версии на удивление важно для эффективности организации. Устранение возможности выбора, куда производить фиксации или от чего зависеть, значительно упрощает работу.
- Некоторые языки программирования поддерживают затенение, отдельную компиляцию, сокрытие компоновщика и т. д. Но эти приемы не помогают создать что-то полезное и формируют технический долг.
- Исследования («State of DevOps», «Ускоряйся!») показали, что существует тесная взаимосвязь между разработкой в главной ветви и эффективностью софтверных организаций. Долгоживущие ветви разработки — не лучшая идея по умолчанию.
- Используйте любую VCS, подходящую для ваших нужд. Чтобы хранить отдельные проекты в отдельных репозиториях, разумно организовать зависимости между репозиториями так, чтобы они ссылались на «главную ветвь». С каждым годом появляется все больше VCS и систем сборки, которые позволяют иметь как небольшие, разрозненные репозитории, так и единую «виртуальную» главную ветвь для всей организации.

Code Search

Авторы: Александр Нойбек и Бен Сент-Джон

Редактор: Лиза Кэри

Code Search — это инструмент для просмотра и поиска кода в Google, который состоит из пользовательского интерфейса и множества серверных компонентов. Его появление, как и многих других инструментов разработки в Google, было обусловлено масштабом кодовой базы. Первоначально Code Search создавался как комбинация внутренних инструментов типа `grep`¹ с ранжированием и внешнего пользовательского интерфейса Code Search². Его место как ключевого инструмента для разработчиков Google было закреплено благодаря его интеграции со службами Kythe и Grok³, добавляющими перекрестные ссылки и возможность перехода к определениям символов.

Эта интеграция сместила фокус с поиска кода на просмотр кода, и при создании более поздней версии Code Search разработчики отчасти руководствовались принципом: «Позволить получить ответ на вопрос о коде одним щелчком мыши». Теперь ответы на такие вопросы, как: «Где определяется этот символ?», «Где он используется?», «Как его подключить?», «Когда он был добавлен в кодовую базу?» и даже: «Сколько тактов процессора он потребляет?», можно получить одним или двумя щелчками мыши.

В отличие от IDE или редакторов кода, Code Search оптимизирован для чтения и изучения больших объемов кода. По этой причине он во многом полагается на облачные компоненты, осуществляющие поиск и разрешающие перекрестные ссылки.

В этой главе мы подробно расскажем об инструменте Code Search: как гуглеры используют его, почему мы решили создать отдельный веб-инструмент для поиска кода и как он решает проблемы, связанные с поиском и просмотром кода в масштабе репозитория Google.

¹ Изначально GSearch работал на ПК Джеффа Дина, который однажды взбесил всю компанию, уйдя в отпуск и выключив свой компьютер!

² Закрыт в 2013 году. См. https://en.wikipedia.org/wiki/Google_Code_Search (<https://oreil.ly/xOk-e>). Описание на русском языке доступно по адресу: https://ru.wikipedia.org/wiki/Google_Code_Search. — *Примеч. пер.*

³ Эта служба, ныне известная как Kythe (<http://kythe.io>), предоставляет перекрестные ссылки (кроме всего прочего): сообщает точки использования определенного символа, например имени функции, с полной информацией о сборке, чтобы дать возможность отличить его от других похожих символов.

Пользовательский интерфейс Code Search

Центральным элементом пользовательского интерфейса Code Search (рис. 17.1) является поле поиска. Так же как поле поиска в веб-интерфейсе поисковой системы Google, оно предлагает «подсказки», которые разработчики могут использовать для быстрого перехода к файлам, символам или каталогам. В ответ на сложные запросы поле поиска возвращает страницу результатов с фрагментами кода. Сам поиск можно рассматривать как мгновенный «поиск в файлах» (действующий подобно команде `grep` в Unix) с ранжированием по релевантности и некоторыми особенностями, характерными для программного кода, такими как правильная подсветка синтаксиса, оценка области видимости и выделение комментариев и строковых литералов. Code Search также имеет интерфейс командной строки и может использоваться другими инструментами через API RPC. Его используют при постобработке или, если набор результатов слишком велик, для проверки вручную.

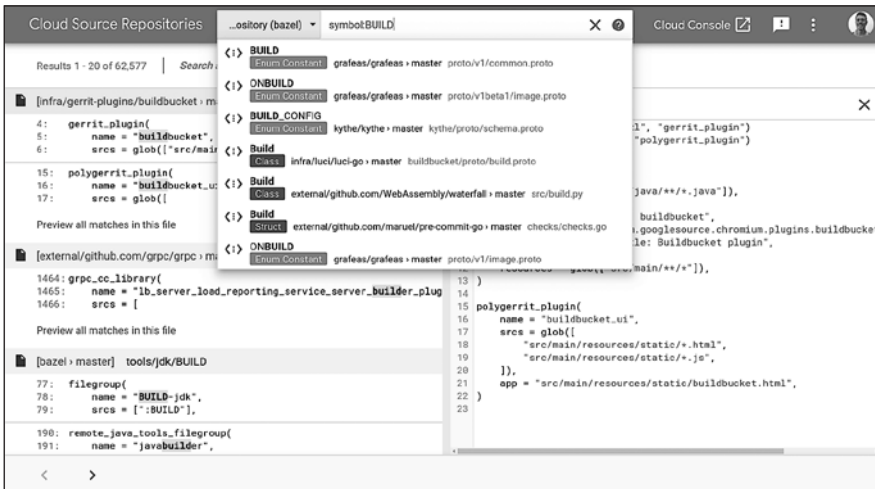


Рис. 17.1. Пользовательский интерфейс Code Search

При просмотре одного файла на большинстве символов можно щелкнуть и быстро получить информацию об этом символе. Например, щелчок на имени функции перенесет вас к ее определению, щелчок на имени импортируемого файла — к фактическому файлу с исходным кодом, а щелчок на идентификаторе ошибки в комментарии — к соответствующему отчету об ошибке. Эта возможность обеспечивается инструментами индексирования на основе компилятора, такими как Kythe (<http://kythe.io>). Если щелкнуть на имени символа в точке его определения, откроется панель, в которой перечислены все места, где он используется. Точно так же при наведении указателя мыши на локальные переменные внутри функции будут подсвечены все появления этой переменной в реализации.

Благодаря интеграции с Piper (главу 16) Code Search может также отображать историю файла. То есть с его помощью можно увидеть более старые версии файла, изменения и их авторов и переходить к ним в Critique (глава 19), сравнивать версии файлов, просматривать историю фиксаций изменений и даже видеть удаленные файлы.

Как гуглеры используют Code Search?

Аналогичные функции доступны и в других инструментах, однако гуглеры по-прежнему активно используют интерфейс Code Search для поиска и просмотра файлов и для изучения кода¹. Если инженер хочет найти ответы на вопросы о коде, то достичь этой цели легче всего с Code Search².

Где?

Около 16 % запросов Code Search связаны с поиском, где в кодовой базе находится конкретная информация, например определение функции, конфигурация, вызовы API или файл в репозитории. На вопрос «где?» можно ответить с помощью поисковых запросов или семантических ссылок, таких как «перейти к определению символа». Такие вопросы часто возникают при решении крупных задач, таких как рефакторинг или сбор мусора, или при обсуждении проектов с другими инженерами.

Code Search предоставляет два вида помощи при таком поиске: ранжирование результатов и расширенный язык запросов. Ранжирование учитывает общие случаи, а язык запросов помогает конкретизировать поиск (например, ограничивать доступ в коде, исключать языки, рассматривать только функции), чтобы находить редкие случаи.

Пользовательский интерфейс Code Search позволяет с легкостью делиться результатами поиска с коллегами. Так, при выполнении обзора кода можно включить ссылку, такую как: «Рассматривали ли вы возможность использования этого специализированного хеш-массива: `cool_hash.h`?» Такие ссылки также можно использовать в документации, в отчетах об ошибках и в результатах вскрытия, и они считаются в Google каноническим способом ссылки на код. Ссылаться можно даже на старые версии кода, поэтому ссылки остаются действительными с течением времени и по мере развития кодовой базы.

¹ Наличие вездесущего браузера кода оказывает благотворное влияние: инженеры стремятся писать код, который легко просматривать. Они используют не слишком глубокие иерархии, которые не требуют множества щелчков мышью для перехода от точки вызова к фактической реализации, и выбирают именованные типы вместо более обобщенных сущностей, таких как строки или целые числа, потому что это облегчает поиск их применений.

² Sadowski C., Stolee K. T., Elbaum S. How Developers Search for Code: A Case Study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. <https://doi.org/10.1145/2786805.2786855>.

Что?

Примерно четверть всех случаев использования Code Search — это просмотр файлов и поиск ответа на вопрос: «Что делает та или иная часть кодовой базы?» Такое применение в большей степени относится к исследованиям — чтению и изучению исходного кода перед внесением своих изменений или для оценки чужих.

Чтобы упростить такие исследования, в Code Search поддерживается возможность быстрой навигации через иерархию вызовов и между связанными файлами (например, между заголовочными файлами и файлами реализации, тестирования и сборки). Code Search должен ответить на множество вопросов, возникающих у разработчиков в процессе просмотра файлов.

Как?

Примерно треть всех случаев использования Code Search — это изучение примеров решения похожих задач. К этому моменту разработчик, как правило, уже нашел нужный ему API (например, для чтения файла из удаленного хранилища) и хочет увидеть, как его использовать в конкретной ситуации (например, как настроить удаленное соединение и обработать определенные типы ошибок). Code Search также часто используется с целью поиска библиотек для конкретных задач (например, для эффективного вычисления идентификационных меток целочисленных значений) и последующего выбора наиболее подходящей реализации. Для такого рода применения типично сочетание поиска и просмотра перекрестных ссылок.

Почему?

Вместе с выяснением, *что* делает код, многие пытаются выяснить, *почему* код ведет себя не так, как ожидалось. Около 16 % случаев использования Code Search связаны с попытками найти ответ на вопрос: «Почему был добавлен тот или иной фрагмент кода?» или «Почему фрагмент ведет себя определенным образом?» Вопрос, который часто возникает при отладке: «Почему в этих обстоятельствах возникает ошибка?»

Важной здесь является возможность определения и изучения точного состояния кодовой базы в указанный момент. При отладке проблемы в продакшене нередко приходится исследовать кодовую базу в состоянии, в котором она находилась несколько недель или месяцев назад, а при поиске причин сбоев в тестах для нового кода обычно изучаются изменения, давность которых составляет всего несколько минут. Code Search поддерживает оба варианта исследований.

Кто и когда?

Около 8 % случаев использования Code Search — это попытки получить ответы на вопросы о том, кто или когда добавил определенный фрагмент кода в VCS. Например, Code Search позволяет узнать, когда и кем была добавлена конкретная строка (подобно команде `blame` в Git), и перейти к соответствующему обзору кода. Панель

истории также может помочь определить, к кому лучше обратиться с вопросом о коде или с просьбой выполнить обзор изменений¹.

Зачем понадобился отдельный веб-инструмент?

За пределами Google большинство вышеупомянутых исследований проводится в локальной IDE. Так зачем понадобился еще один инструмент?

Масштаб

База кода в Google настолько велика, что локальная копия полной кодовой базы — необходимое условие для большинства IDE — просто не поместится ни на одном компьютере. Но даже если удастся преодолеть этот фундаментальный барьер, создание локальных индексов поиска и перекрестных ссылок при запуске IDE на компьютере каждого разработчика неизбежно приведет к снижению скорости разработки. А без использования индексов поиск (например, с помощью `grep`) может выполняться очень медленно. Использование централизованного поискового индекса предполагает выполнение этой работы только один раз и заранее, что приносит пользу всем. Например, индекс для поиска по коду обновляется с каждым отправленным изменением, что обеспечивает постоянную стоимость поддержки индекса².

В обычном веб-поиске быстро меняющиеся текущие события смешиваются с элементами, меняющимися более медленно, такими как страницы Википедии. Это же свойство можно распространить на поиск в коде и сделать индексацию инкрементальной, снизив ее стоимость и позволив мгновенно находить изменения в кодовой базе. При отправке изменений требуется переиндексировать только затронутые файлы, что позволяет параллельно и независимо обновлять глобальный индекс.

К сожалению, быстро обновить индекс перекрестных ссылок подобным способом невозможно. К нему неприменим прием инкрементального обновления, потому что любое изменение в нем потенциально может затрагивать всю кодовую базу, а на практике часто влияет на тысячи файлов. Многие (в Google почти все) двоичные файлы приходится повторно собирать³ (или, по крайней мере, повторно анали-

¹ Отметим, что с учетом частоты фиксаций изменений, выполняемых автоматизированными инструментами, снижается ценность поиска «виноватых» авторов изменений.

² Для сравнения: модель «каждый разработчик имеет свою IDE и выполняет индексацию в своем рабочем пространстве» масштабируется почти квадратично: разработчики создают примерно постоянный объем кода в единицу времени, поэтому база кода масштабируется линейно (даже при фиксированном количестве разработчиков). Линейное количество IDE каждый раз выполняет линейно растущее количество работы — это никак нельзя назвать рецептом хорошего масштабирования.

³ Для извлечения узлов и ребер семантического графа из исходного кода Kythe (<https://www.kythe.io>) использует процесс сборки. Сначала она конструирует частичные графы перекрестных ссылок для каждого правила сборки. Затем из частичных графов создается один глобальный граф и оптимизируется для обработки наиболее типичных запросов (перейти к определению, найти все места использования, выбрать все объявления из файла). Стои-

зировать), чтобы определить их полную семантическую структуру. Для ежедневного (текущая частота) создания индекса перекрестных ссылок нужны огромные вычислительные ресурсы. Несоответствие между индексом поиска и ежедневным индексом перекрестных ссылок является причиной редких, но повторяющихся проблем у пользователей.

Отсутствие необходимости настраивать глобальное представление кода

Возможность мгновенно и эффективно просматривать всю кодовую базу позволяет с легкостью находить подходящие библиотеки и хорошие примеры. IDE, которые создают индексы в момент запуска, требуют настройки проектов или видимых объемов кода, чтобы сократить время запуска и избежать появления шума в таких инструментах, как автодополнение. Веб-интерфейс Code Search не требует настройки (например, описания проекта, среды сборки) и позволяет легко и быстро находить код, что повышает эффективность разработки. Также нет опасности пропустить зависимости, поскольку, например, при обновлении API сокращается количество проблем слияния и управления версиями библиотек.

Специализация

Возможно, это покажется необычным, но одно из преимуществ Code Search заключается в том, что он не является средой разработки, что позволяет оптимизировать его для просмотра и изучения кода, а не для редактирования, которое обычно является основной задачей IDE (это видно по сочетаниям клавиш, меню, щелчкам мышью и организации экранного пространства). Благодаря отсутствию текстового курсора каждому щелчку мышью на символе в Code Search можно придать свое значение (например, показать все места использования символа или перейти к его определению), отличное от намерения переместить текстовый курсор. Это преимущество настолько велико, что разработчики часто открывают несколько вкладок Code Search рядом с редактором.

Интеграция с инструментами разработки

Как основной инструмент просмотра исходного кода, Code Search является платформой для размещения информации о коде. Это освобождает создателей других инструментов от необходимости реализовать пользовательский интерфейс для отображения результатов своей работы, которые покажет всем Code Search. Регулярно в кодовой базе Google выполняется множество разных видов анализа, и их результаты обычно отображаются в Code Search: например, Code Search выявляет и выделяет при просмотре файлов «мертвый» (неиспользуемый) во множестве языков.

мость обоих этапов — извлечения и постобработки — примерно равна стоимости полной сборки. К примеру, создание индекса Kythe для Chromium занимает около шести часов в распределенной среде, поэтому строить индекс на рабочих станциях разработчиков слишком дорого. Из-за таких затрат индекс Kythe строится только раз в день.

Кроме того, ссылки на исходные файлы в Code Search считаются каноническим способом «адресации» и используются во многих инструментах разработчика (рис. 17.2). Например, мы привыкли, что записи в файлах журналов включают имя файла и номер строки, где находится инструкция журналирования. Средство просмотра журналов использует ссылки Code Search, чтобы связать инструкцию журналирования с фактическим кодом. В зависимости от доступной информации это может быть прямая ссылка на файл определенной версии или результат простого поиска по имени файла и номеру строки. Если существует только один подходящий файл, щелчок на ссылке откроет его на строке с соответствующим номером. В противном случае будут показаны фрагменты с нужной строкой во всех найденных файлах.



Рис. 17.2. Интеграция Code Search со средством просмотра журналов

Аналогичным образом стековые фреймы связываются с исходным кодом в инструменте просмотра отчетов о сбоях или в записях журнала (рис. 17.3). В зависимости от языка программирования для создания ссылки используется поиск по имени файла или символа. Поскольку состояние репозитория, в котором находился исходный элемент в момент создания двоичного файла, известно, то поиск фактически можно ограничить одной версией. Ссылки долго останутся действительными, даже если соответствующий код будет изменен или удален.

Наконец, практические учебные пособия Codelab и другая документация могут включать ссылки на API и примеры реализации. Такие ссылки могут создаваться поисковыми запросами, отыскивающими конкретные классы или функции, и остаются действительными даже после изменения структуры файла. Кроме того, самая свежая реализация в главной ветви позволяет с легкостью встраивать фрагменты кода в страницы документации (рис. 17.4), но засоряя исходный файл дополнительными аннотациями.



Рис. 17.3. Интеграция Code Search с инструментом просмотра отчетов о сбоях



Рис. 17.4. Интеграция Code Search с документацией

Открытый API

Code Search имеет открытый API поиска, перекрестных ссылок и подсветки синтаксиса. Разработчики могут использовать все эти возможности в своих инструментах без необходимости повторно реализовать их. Кроме того, для редакторов и IDE, таких как vim, emacs и IntelliJ, в Code Search есть плагины поиска и перекрестных ссылок. Эти плагины частично компенсируют невозможность индексирования кодовой базы локально и способствуют увеличению продуктивности разработчиков.

Влияние масштаба на дизайн

В предыдущем разделе мы рассмотрели некоторые аспекты пользовательского интерфейса Code Search и выяснили причины создания отдельного инструмента для просмотра кода. В следующих разделах мы подробнее обсудим проблемы его реали-

зации: масштабирование и особенности поиска и просмотра фрагментов в большой кодовой базе. Затем мы расскажем, как мы решили эти проблемы и на какие уступки нам пришлось пойти при создании Code Search.

Наибольшую сложность для масштабируемого¹ поиска представляет размер исследуемого корпуса кода. Для поиска в небольшом репозитории, занимающем пару мегабайт, подойдет метод перебора с помощью `grep`. Чтобы выполнить поиск в корпусе с объемом в несколько сотен мегабайт, можно создать простой локальный индекс и ускорить поиск на порядок или даже больше. А для поиска в гигабайтах или даже терабайтах исходного кода необходимо привлекать облачные решения с несколькими серверами. Ценность централизованного решения растет с увеличением числа разработчиков, использующих его, и расширением кодовой базы.

Задержка обработки поисковых запросов

Быстрый и отзывчивый пользовательский интерфейс мы воспринимаем как само собой разумеющееся, но добиться низкой задержки поиска очень непросто. Чтобы оправдать затраченные на это усилия, их можно сравнить с временем, которое позже сэкономят пользователи. Ежедневно Code Search обрабатывает более миллиона поисковых запросов от разработчиков Google. Если время обработки каждого запроса увеличить всего на одну секунду, то в сумме получится время, соответствующее бездействию примерно 35 инженеров каждый день. Для сравнения: создать и поддерживать серверную часть сможет группа инженеров с численностью в десять раз меньше. То есть при 100 000 запросов в день (которые посылают менее 5000 разработчиков) уменьшение задержки в одну секунду станет своего рода точкой окупаемости.

На самом деле потеря продуктивности растет не в прямой линейной зависимости от увеличения задержки. Пользовательский интерфейс считается отзывчивым, если задержки не превышают 200 мс (<https://oreil.ly/YUN0b>). Но уже через секунду внимание разработчика часто начинает переключаться. Если пройдет еще 10 секунд, разработчик, скорее всего, полностью переключится на что-то другое, что, как принято считать, очень негативно влияет на продуктивность. Лучший способ удержать разработчика в продуктивном «рабочем» состоянии — обеспечить сквозную задержку менее 200 мс для всех частых операций и вложить средства в создание подходящих для этого серверных компонентов.

Большая часть запросов к Code Search посылается в процессе навигации по кодовой базе. В лучшем случае «следующий» файл должен находиться на расстоянии одного щелчка мышью от «предыдущего» (это может быть подключаемый файл или файл с определениями символов). Но еще эффективнее, если инструмент поиска не требует указывать имя искомого файла (или символа) полностью и предлагает возможные варианты завершения вводимого текста запроса. Этот подход становится более актуальным с ростом кодовой базы (и дерева файлов).

¹ Запросы обрабатываются независимо, поэтому большее количество серверов может обслужить больше пользователей.

Обычный переход к конкретному файлу в другой папке или другом проекте требует от пользователя выполнить несколько действий. А при использовании механизма поиска для этого может быть достаточно нажать всего пару клавиш. Чтобы сделать поиск таким эффективным, движку поиска можно предоставить дополнительную информацию о контексте (например, о просматриваемом в данный момент файле). Контекст может ограничить поиск рамками определенного проекта или повлиять на ранжирование файлов. В пользовательском интерфейсе Code Search¹ есть возможность заранее определить несколько контекстов и быстро переключаться между ними по мере необходимости. Открытые в редакторах файлы неявно используются в качестве контекста для определения релевантности результатов поиска по степени близости к этим файлам.

Еще одним критерием качества инструмента поиска можно считать мощность языка поисковых запросов (например, описание искомых файлов с использованием регулярных выражений). Мы обсудим этот вопрос в разделе, посвященном компромиссам, далее в этой главе.

Задержка индексирования

Часто разработчики не замечают, что индексы устарели. Их интересует только узкая область кода, и они не знают, есть ли более свежий код. Однако когда они пишут или оценивают некое изменение, отсутствие синхронизации может вызвать большую путаницу. Обычно неважно, является ли изменение небольшим исправлением, результатом рефакторинга или совершенно новым фрагментом кода — разработчики просто ожидают получить его согласованное представление.

Когда разработчик пишет код, он ожидает мгновенной индексации измененного кода. Невозможность найти вновь добавленные файлы, функции или классы вызывает недовольство и нарушает нормальный рабочий процесс разработчиков, привыкших полагаться на перекрестные ссылки. Другой пример — когда удаленный код сразу исчезает из результатов поиска, это не только удобно, но и помогает учитывать новое состояние при последующих сеансах рефакторинга на основе операций поиска и замены. При работе с централизованной VCS разработчику может потребоваться мгновенная индексация отправленного кода, если предыдущее изменение больше не является частью набора файлов, измененного локально.

С другой стороны, иногда бывает полезно вернуться назад к предыдущей версии кода. Расхождение между индексом и выполняющимся кодом во время расследования инцидента может иметь особенно негативные последствия, скрыть реальные причины проблемы и обратить внимание инженера на детали, не имеющие отношения к инциденту. Эта проблема усложняет использование перекрестных ссылок, потому что текущая технология построения индекса в масштабе Google занимает часы, а из-за высокой сложности хранится только одна «версия» индекса. Конечно,

¹ Пользовательский интерфейс Code Search тоже имеет классическое дерево файлов, поэтому навигация по нему тоже возможна.

можно внести некоторые исправления и организовать сопоставление нового кода со старым индексом, но эту задачу еще предстоит решить.

Реализация в Google

Конкретная реализация Code Search в Google адаптирована к уникальным характеристикам кодовой базы компании, и в предыдущем разделе мы изложили конструктивные ограничения, мешающие созданию надежного и гибкого индекса. В следующем разделе мы расскажем, как команда Code Search реализовала свой инструмент.

Поисковый индекс

Огромный размер базы кода в Google — это проблема для Code Search. Первое время мы использовали подход, основанный на триграммах. Затем Расс Кокс выпустил упрощенную версию этого инструмента с открытым исходным кодом (<https://github.com/google/codesearch>). В настоящее время Code Search индексирует около 1,5 Тбайт контента и обрабатывает около 200 запросов в секунду со средней задержкой на стороне сервера менее 50 мс и средней задержкой индексации (время между фиксацией кода и появлением его в индексе) менее 10 секунд.

Давайте примерно оценим требования к ресурсам для достижения такой производительности при использовании метода простого перебора на основе `grep`. Библиотека RE2, которую мы используем для сопоставления регулярных выражений, обрабатывает около 100 Мбайт/с данных, находящихся в ОЗУ. Учитывая временное окно в 50 мс, для обработки 1,5 Тбайт данных потребуется 300 000 процессорных ядер. Поскольку в большинстве случаев достаточно простого поиска подстроки, сопоставление регулярных выражений можно заменить специальным поиском строк, способным при определенных условиях обрабатывать данные со скоростью 1 Гбайт/с¹, что уменьшает потребность в процессорных ядрах в 10 раз. До сих пор мы рассматривали только требования к ресурсам для обработки одного запроса в течение 50 мс. А если взять за основу 200 запросов в секунду, 10 из которых будут обрабатываться одновременно в указанном окне 50 мс, то мы вновь возвращаемся к необходимости иметь 300 000 процессорных ядер для самого простого поиска по строкам.

Эта оценка не учитывает возможности прекращения поиска в момент, когда будет найдено определенное количество результатов, или существование более эффективной оценки ограничений выбора файлов, а также дополнительных затрат на сетевые взаимодействия, ранжирование и параллельную работу десятков тысяч машин. Однако она достаточно точно отражает масштаб и наглядно объясняет, почему команда Code Search постоянно работает над совершенствованием индексации. С течением времени на смену реализации на основе триграмм пришла реализация на основе массива суффиксов, а затем и текущая реализация, основанная на раз-

¹ См. <https://blog.scalyr.com/2014/05/searching-20-gbsec-systems-engineering-before-algorithms> и http://volnitsky.com/project/str_search.

реженных n -граммах. Это последнее решение более чем в 500 раз эффективнее метода перебора и способно обрабатывать запросы с регулярными выражениями с молниеносной скоростью.

Одна из причин, по которой мы отказались от решения на основе массива суффиксов в пользу решения на основе n -грамм на основе лексем, заключалась в возможности использовать первичный стек индексации и поиска. В решении на основе массива суффиксов создание и распространение нестандартных индексов сами по себе являются проблемой. Используя «стандартную» технологию, мы извлекаем выгоду из всех достижений в сфере построения обратного индекса, кодирования и обслуживания, сделанных основной командой поиска. Мгновенное индексирование — это еще одна особенность, присущая стандартным стекам поиска, и сама по себе представляет большую проблему при применении в масштабных решениях.

Применение стандартных технологий — это компромисс между простотой реализации и производительностью. Несмотря на то что реализация Code Search в Google основана на стандартных обратных индексах, фактическое извлечение, сопоставление и ранжирование настраиваются и оптимизируются в весьма широких пределах. Без этого некоторые из продвинутых особенностей Code Search были бы просто невозможны. Для индексации истории изменений в файлах мы разработали специальную схему сжатия, при использовании которой индексирование полной истории увеличивало потребление ресурсов всего в 2,5 раза.

Первые версии Code Search обрабатывали все данные в оперативной памяти. С увеличением размера индекса мы переместили обратный индекс (<https://oreil.ly/OtETK>) во флеш-память. Флеш-память как минимум на порядок дешевле оперативной памяти, но время доступа к ней как минимум на два порядка дольше. То есть индексы, показывающие хорошую производительность в оперативной памяти, могут не подходить для работы с флеш-памятью. Например, первоначальный индекс на основе триграмм требовал извлекать из флеш-памяти большое количество обратных индексов огромного размера. В схемах с n -граммами количество обратных индексов и их размеры можно уменьшить за счет увеличения основного индекса.

Для поддержки локальных рабочих пространств (содержимое которых немного отличается от содержимого глобального репозитория) у нас есть несколько машин, выполняющих поиск простым методом перебора. Данные из рабочей области загружаются при первом запросе и затем синхронизируются при обнаружении изменения файлов. Когда у нас заканчивается память, мы удаляем с машин самое старое рабочее пространство. Поиск не изменявшихся документов производится с помощью индекса истории. Как следствие, поиск неявно ограничен состоянием репозитория, с которым синхронизируется рабочее пространство.

Ранжирование

Для небольшой кодовой базы ранжирование не приносит особой пользы, потому что результатов поиска не много. Но чем больше размер кодовой базы, тем больше

результатов будет получено в ходе поиска и тем важнее становится ранжирование. В кодовой базе в Google любая короткая подстрока встречается тысячи, а то и миллионы раз. Без ранжирования пользователю придется проверить все полученные результаты, чтобы найти правильный, или уточнять запрос¹ до тех пор, пока в наборе результатов не останется несколько файлов. Оба варианта ведут к пустой трате времени разработчика.

Ранжирование обычно начинается с оценки, которая отражает набор характеристик каждого файла (сигналы) в некоторое число: чем выше оценка, тем выше результат. Цель поиска — как можно эффективнее найти первые N результатов. Обычно различают два типа сигналов: зависящие только от документа («не зависящие от запроса») и зависящие от поискового запроса и его соответствия документу («зависящие от запроса»). Примерами сигналов, не зависящих от запроса, могут служить длина имени файла или язык программирования, на котором написан код в файле, а примерами сигналов, зависящих от запроса, могут служить совпадения с определением функции или строковым литералом.

Сигналы, не зависящие от запроса

К числу наиболее важных сигналов, не зависящих от запроса, можно отнести количество просмотров файла и количество ссылок на файл. Количество просмотров файла показывает, какие файлы разработчики с большей вероятностью захотят найти. Например, служебные функции в базовых библиотеках просматриваются часто. Неважно, является ли библиотека стабильной или продолжает активно развиваться. Самый большой недостаток этого сигнала — создаваемая им петля обратной связи. Увеличение оценки для часто просматриваемых документов увеличивает вероятность, что разработчики увидят их, и уменьшает шансы других документов попасть в первые N строк. Эта проблема известна как *компромисс между эксплуатацией и исследованием* (exploitation versus exploration). Существуют разные решения этой проблемы (например, расширенные поисковые A/B-эксперименты или курирование обучающих данных). На практике можно завышать рейтинг результатов: они будут игнорироваться, если не имеют отношения к делу, и приниматься во внимание, если требуется обобщенный пример. Однако эта проблема актуальна для новых файлов, еще не имеющих достаточно информации для хорошего сигнала².

Мы также учитываем количество ссылок на файл, как в оригинальном алгоритме ранжирования страниц (<https://oreil.ly/k3CJx>), заменив веб-ссылки операторами `include` и `import`, присутствующими в большинстве языков. Эту идею можно распространить вверх, до поиска зависимостей (ссылки на уровне библиотеки или модуля), и вниз,

¹ В отличие от веб-поиска, добавление большего количества знаков в запрос Code Search всегда уменьшает набор результатов (за исключением редких случаев, связанных с особенностями регулярных выражений).

² Этот недостаток можно исправить, если использовать степень новизны как сигнал, как это реализовано, например, в веб-поиске в отношении новых страниц. Но мы пока отложили это исправление.

до определений функций и классов. Эту глобальную релевантность часто называют «приоритетом» документа.

При ранжировании на основе количества ссылок следует учитывать две проблемы. Во-первых, нужно уметь надежно извлекать ссылочную информацию. Раньше Code Search извлекал операторы `include` и `import` с помощью простых регулярных выражений, а затем применял эвристику для преобразования их в полные пути к файлам. Но с ростом сложности кодовой базы эта эвристика стала все чаще допускать ошибки, и ее было сложно поддерживать. Поэтому мы заменили эти процедуры извлечением информации из графа `Kythe`.

Вторая проблема — крупномасштабный рефакторинг, такой как в базовых библиотеках с открытым исходным кодом (<http://abseil.io>), не применяется автоматически в одном обновлении кода — его нужно развертывать в несколько этапов. Для этого обычно вводятся косвенные ссылки, скрывающие, например, перемещение файлов. Подобные косвенные ссылки снижают рейтинг перемещаемых файлов и затрудняют определение нового их местоположения. Кроме того, при перемещении обычно сбрасываются счетчики просмотров файлов, что еще больше усугубляет ситуацию. Поскольку такая глобальная реорганизация кодовой базы происходит сравнительно редко (большинство интерфейсов редко перемещается), самым простым решением является увеличение рейтинга файлов вручную на время переходных процессов (либо ожидание завершения миграции и увеличения рейтинга файла на новом месте).

Сигналы, зависящие от запроса

Сигналы, не зависящие от запросов, можно вычислить в автономном режиме, поэтому вычислительные затраты не являются серьезной проблемой, хотя и могут быть высокими. Например, «рейтинг страницы» зависит от всего корпуса поиска и требует пакетной обработки в стиле MapReduce. Сигналы, *зависящие* от запроса, вычисляются для каждого запроса отдельно и должны быть дешевыми с вычислительной точки зрения. Это означает, что они ограничены информацией, доступной в запросе и индексе.

В отличие от веб-поиска, Code Search ищет не только лексемы. Однако если есть точные совпадения с лексемами (то есть поисковый запрос совпадает с содержимым, обрамленным разрывами некоторого вида, например пробелами), то релевантность документа дополнительно увеличивается и учитывается регистр. Это означает, например, что поиск по запросу «Point» даст более высокую оценку релевантности файлу со строкой «*Point* *р», чем файлу с текстом «*appointed to the council*».

Для удобства при поиске кроме фактического содержимого сопоставляются также имена файлов и полные квалифицированные символы¹. Пользователь *может* задать

¹ В языках программирования символы, такие как имена функций, например `Alert`, часто определяются в некоторой области видимости, такой как класс (`Monitor`) или пространство имен (`abs1`). Полное квалифицированное имя такой функции может иметь вид `abs1::Monitor::Alert`, и его можно найти, даже если оно отсутствует в фактическом тексте.

конкретный тип сопоставления, но это необязательно. За совпадения с символами и именами файлов дается более высокая оценка релевантности, чем за совпадения в обычном содержимом, что точнее соответствует предполагаемым намерениям разработчиков. Так же как в случае с веб-поиском, разработчики могут добавлять дополнительные термины, чтобы сделать запросы более конкретными. Очень часто в запрос добавляются «подсказки» с именами файлов (например, «base» или «myproject»), а механизм ранжирования автоматически учитывает их, увеличивая рейтинг результатов за наличие совпадений с этими терминами и ставя такие результаты выше тех, которые содержат такие же отдельные слова в случайных местах.

Извлечение

Прежде чем оценить документ, выполняется писк кандидатов, которые могут соответствовать поисковому запросу. Этот этап называется извлечением. Поскольку нецелесообразно извлекать все документы, а оценивать можно только извлеченные, для поиска наиболее релевантных документов их извлечение и оценка должны производиться вместе. Например, в зависимости от популярности искомый класс может иметь тысячи применений, но только одно определение. Если поиск не был явно ограничен определениями классов, извлечение фиксированного числа результатов поиска имени класса может остановиться еще до достижения файла с единственным определением. Очевидно, что с ростом кодовой базы эта проблема только усложняется.

Основная задача на этапе извлечения — найти несколько наиболее релевантных файлов. Одно из решений, которое работает достаточно хорошо, называется *добавочным извлечением*. Оно преобразует исходный запрос в более специализированный. В нашем примере это означает, что добавочный запрос ограничит поиск только определениями и именами файлов и добавит вновь извлеченные документы в выходные данные этапа извлечения. При наивной реализации добавочного извлечения придется оценить больше документов, зато полученную дополнительную информацию можно будет использовать для более точной оценки только наиболее релевантных документов.

Разнообразие результатов

Другой важный аспект поиска — разнообразие результатов, то есть выбор лучших результатов из нескольких категорий. Простым примером может служить представление совпадений с именем функции в Java и Python вместо заполнения первой страницы поиска результатами для какого-то одного языка.

Это особенно важно, когда намерения пользователя неясны. Одна из проблем, связанных с разнообразием, заключается в существовании множества разных категорий, таких как функции, классы, файлы, локальные переменные, тесты, способы использования, примеры и т. д., в которые можно сгруппировать результаты. В пользовательском интерфейсе Code Search не так много свободного пространства, чтобы отобразить результаты из всех этих категорий или хотя бы их комбинаций, и это не всегда желательно. Code Search ищет не так хорошо, как веб-поиск, но его раскрывающийся список результатов (действующий подобно автодополнению в веб-поиске)

настроен для предоставления разнообразного множества имен файлов, определений и совпадений в текущем рабочем пространстве пользователя.

Некоторые компромиссы

Необходимость поддержки огромной кодовой базы и высокой скорости работы заставила разработчиков Code Search пойти на компромиссы, которые описаны в следующем разделе.

Полнота: главная ветвь репозитория

Мы уже видели, что увеличение кодовой базы отрицательно влияет на поиск: замедляет индексирование и увеличивает его стоимость, замедляет обработку запросов и искажает результаты. Можно ли уменьшить негативное влияние роста кодовой базы, если пожертвовать полнотой, то есть оставить часть содержимого кода вне индекса? Да, если действовать осторожно. Нетекстовые файлы (двоичные артефакты, изображения, видео, аудио и т. д.) обычно не предназначены для чтения, поэтому их содержимое отделяется от имени файла. А поскольку они часто имеют огромные размеры, такое решение экономит много ресурсов. Также из-за обфускации и потери структуры сгенерированные файлы JavaScript почти нечитаемы, поэтому исключение их из индексирования помогает сэкономить ресурсы и уменьшить шум. Исключение многомегабайтных файлов, которые редко содержат релевантную информацию, тоже бывает полезно.

Однако исключение файлов из индекса имеет один большой недостаток. Чтобы разработчики использовали Code Search, они должны доверять ему. К сожалению, невозможно сообщить о неполных результатах поиска по конкретному запросу, если файлы не были проиндексированы. Возникающая путаница и потеря продуктивности разработчиков — слишком высокая цена за экономию ресурсов. Даже если разработчики знают об ограничениях, но им все равно нужно что-то найти, они будут выполнять поиск необычным способом, подверженным ошибкам. Учитывая эти редкие, но потенциально высокие затраты, мы считаем, что лучше ошибиться, определив более высокие пределы для индекса, которые в основном выбираются для предотвращения злоупотреблений и обеспечения стабильности системы, чем сэкономить ресурсы.

С другой стороны, сгенерированные файлы не входят в кодовую базу, но их часто полезно индексировать. В настоящее время этого не делается, потому что для индексации потребовалась бы интеграция инструментов и конфигурации, что стало бы огромным источником сложностей, путаницы и задержек.

Полнота: все результаты, а не только наиболее релевантные

Обычный поиск жертвует полнотой ради скорости, делая ставку на то, что после ранжирования на первой странице окажутся все желаемые результаты. И действительно,

ранжированный поиск в Code Search является самым распространенным способом выявить среди миллионов совпадений что-то конкретное, например определение функции. Но иногда разработчикам нужны *все* результаты, например все вхождения определенного символа для рефакторинга. Требование всех результатов является обычным явлением для анализа, инструментов или рефакторинга, например с целью глобального поиска и замены. Необходимость предоставлять все результаты — фундаментальное отличие Code Search от веб-поиска, в котором допустимыми считаются многие сокращения, например по рейтингу.

Возможность возвращать очень большие наборы со *всеми* результатами обходится дорого, но мы понимаем, что это помогает инструментам и разработчикам доверять поиску. Однако поскольку для большинства запросов релевантны лишь некоторые результаты (когда имеется мало совпадений¹ или мало интересных результатов), мы не хотели жертвовать средней скоростью ради потенциальной полноты.

Для достижения обеих целей с помощью одной архитектуры мы разбили кодовую базу на сегменты с файлами, упорядоченными по их приоритету. Благодаря такой организации инженеру достаточно рассмотреть только совпадения с высокоприоритетными файлами из каждого фрагмента. Примерно так же работает веб-поиск. Но если это явно указано в запросе, Code Search может извлечь *все* результаты из каждого фрагмента, чтобы гарантировать поиск всех результатов². Это позволяет нам достигать обеих целей без замедления типичного поиска из-за реже используемой возможности возврата больших и полных наборов. Результаты также могут доставляться в алфавитном порядке, а не в порядке ранжирования, что удобно для некоторых инструментов.

В данном случае компромисс заключается в усложнении реализации и API за счет сужения возможностей, а не в уменьшении задержки за счет сужения полноты.

Полнота: главная ветвь, побочные ветви, вся история и рабочие пространства

С увеличением размера корпуса поиска встает вопрос: какие версии кода индексировать? В частности, следует ли индексировать что-то еще кроме текущего состояния главной ветви? Сложность системы, ресурсоемкость и общая стоимость резко возрастают при индексировании нескольких версий файла. Насколько нам известно, ни одна IDE не индексирует что-то еще кроме текущей версии кода. Многие DVCS, такие как Git или Mercurial, обеспечивают высочайшую эффективность в основном за счет сжатия хронологических данных. Но при построении обратных индексов компактность этих представлений теряется. Другая проблема заключается в сложности эффективного индексирования графовых структур, которые являются основой DVCS.

¹ Как показал анализ, около трети поисковых запросов дают менее 20 результатов.

² Мы делаем все возможное, чтобы ответы не стали слишком большими, а разработчики не нарушили работу всей системы, выполняя поисковые запросы с бесконечным количеством ответов (представьте поиск буквы «i» или пробела).

Индексировать несколько версий репозитория сложно, но такая индексация позволяет видеть, как менялся код, и находить удаленный код. Code Search индексирует (линейно) хронологию `Perf`. Благодаря этому можно производить поиск в кодовой базе в любом ее состоянии, находить удаленный код и авторов изменений.

Одним из самых больших преимуществ такого подхода является возможность просто удалить устаревший код. Раньше такой код перемещался в каталоги, отмеченные как устаревшие, чтобы потом его можно было найти. Индексирование всей хронологии также заложило основу для эффективного поиска в рабочих пространствах (с неотправленными изменениями), которые синхронизируются с конкретным состоянием кодовой базы. Хронологический индекс открывает возможность использования в будущем интересных сигналов для ранжирования, таких как авторство, активность кода и т. д.

Рабочие пространства имеют существенные отличия от глобального репозитория:

- каждый разработчик может иметь свои рабочие пространства;
- в рабочем пространстве обычно находится ограниченное количество измененных файлов;
- файлы в рабочем пространстве часто изменяются;
- рабочее пространство существует на протяжении относительно короткого времени.

Наиболее полезен индекс рабочего пространства, который точно отражает текущее состояние этого пространства.

Выразительность: лексемы, подстроки и регулярные выражения

Эффект масштаба в значительной степени зависит от поддерживаемого набора функций поиска. Code Search поддерживает поиск по регулярным выражениям, которые расширяют возможности языка запросов и позволяют указывать или исключать целые группы терминов. Их с успехом можно использовать для поиска в любом тексте, что особенно полезно для документов и языков, для которых отсутствуют инструменты с более глубокой семантикой.

Кроме того, всем известно, что в других инструментах (например, `grep`) и контекстах регулярные выражения повышают эффективность поиска без увеличения когнитивной нагрузки на разработчиков. Однако создание индекса для обработки запросов — сложная задача. Есть ли варианты проще?

Индекс на основе лексем (слов) хорошо масштабируется, потому что хранит только часть фактического исходного кода, и хорошо поддерживается стандартными поисковыми системами. С другой стороны, многие варианты использования сложно или даже невозможно эффективно реализовать с помощью индекса на основе лексем, потому что многие символы, которые обычно игнорируются при формировании лексем, в исходном коде имеют определенное значение. Например, поиск по `function()`

вместо `function(x)`, `(x ^ y)` или `=== myClass` сложно или невозможно реализовать на основе лексем.

Еще одна проблема, связанная с лексемами, состоит в неопределенности процедуры формирования лексем из идентификаторов в коде. Идентификаторы могут записываться разными способами, например в ВерблюжемРегистре, змеином_регистре или даже как простая смесь слов без разделителя. Поиск идентификатора по некоторым словам — сложная задача для индекса на основе лексем.

При формировании лексем также обычно не учитывается регистр букв (например, `r` и `R` считаются одинаковыми буквами) и значение слов оказывается размытым из-за сокращения до основы слова, например `searching` и `searched` сокращаются до основы `search`. Такая потеря точности — серьезная проблема при поиске кода. Наконец, использование лексем делает невозможным поиск по пробелам или другим разделителям слов (запятым, круглым скобкам), то есть по символам, имеющим значение в коде.

Следующий шаг¹ в развитии возможностей поиска — полный поиск подстроки, который позволяет отыскать любую последовательность символов. Одно из эффективных решений, обеспечивающих такую возможность, основано на использовании индекса триграмм². В простейшей форме получающийся индекс имеет размер намного меньше размера исходного кода и точность ниже, чем у индексов, полученных другими способами индексирования подстрок. Он замедляет обработку запросов, потому что несоответствия необходимо отфильтровать из набора результатов. Здесь важно найти хороший компромисс между размером индекса, задержкой поиска и потреблением ресурсов с учетом размера кодовой базы, доступности ресурсов и количества запросов в секунду.

Кстати, индекс подстрок легко расширить до поддержки поиска по регулярным выражениям. Для этого нужно преобразовать автомат регулярных выражений в набор поисков по подстрокам. Это преобразование несложно реализовать для индекса триграмм и обобщить для других индексов на основе подстрок. Поскольку нет идеального индекса для регулярных выражений, можно конструировать запросы, которые приводят к поиску методом полного перебора. Однако поскольку лишь небольшая часть запросов содержит сложные регулярные выражения, аппроксимация с использованием индексов на основе подстрок на практике должна работать очень хорошо.

¹ Есть и другие промежуточные варианты, такие как построение индекса префиксов и суффиксов, но обычно они имеют меньшую выразительность запросов и высокую сложность и стоимость индексации.

² Cox R. Regular Expression Matching with a Trigram Index or How Google Code Search Worked (<https://oreil.ly/VSZe7>).

Заключение

Code Search превратился из естественной замены `grep` в важнейший инструмент, повышающий продуктивность разработчиков и использующий технологии веб-поиска, разработанные в Google. Но что это значит для вас? Если вы работаете над небольшим проектом, который легко вписывается в вашу IDE, то Code Search вам вряд ли нужен. Если вы отвечаете за продуктивность инженеров, работающих с более крупной базой кода, то, вероятно, некоторые идеи, изложенные в этой главе, вам стоит взять на вооружение.

Самая важная из этих идей состоит в том, что понимание кода является ключом к его разработке и поддержке, а это означает, что инвестиции в инструменты исследования кода принесут реальные дивиденды, которые трудно переоценить. Каждая возможность Code Search ежедневно помогает разработчикам. Среди них: интеграция с Kythe (позволяет исследовать семантику кода) и поиск рабочих примеров (отличный от поиска или наблюдения за изменениями). Что касается влияния инструмента — о нем мы рассказываем сотрудникам на тренинге «Нуглер».

Вы можете настроить стандартный профиль индексирования для IDE, обмен знаниями о `grep`, запуск `ctags` или свои нестандартные инструменты индексирования, напоминающие Code Search. В любом случае они почти наверняка будут использоваться, причем часто совсем не так, как вы предполагали, и ваши разработчики от этого только выиграют.

Итоги

- Помощь разработчикам в исследовании кода может значительно повысить их продуктивность. Ключевым инструментом для этого в Google является Code Search.
- Инструмент поиска кода образует основу для других инструментов и играет роль стандартного инструмента, на который ссылаются вся документация и все остальные инструменты разработки.
- Огромный размер кодовой базы в Google предопределил создание специального инструмента, отличного, например, от `grep` или индексации в IDE.
- Как интерактивный инструмент, Code Search должен действовать быстро и обеспечивать бесперебойное течение процесса «вопрос-ответ». Также он должен иметь низкую задержку при выполнении всех своих функций: поиска, просмотра и индексирования.
- Инструмент поиска получит широкое использование, только если ему доверяют, а доверие к нему появится, только если он проиндексирует весь код, вернет все результаты, первыми отобразив наиболее релевантные. Более ранние и менее мощные версии Code Search тоже были полезны и использовались до тех пор, пока были понятны их ограничения.

Системы и философия сборки

Автор: Эрик Куффлер

Редактор: Лиза Кэри

Если спросить инженеров в Google, что больше всего им нравится в компании (помимо бесплатной еды и первоклассного оборудования), то можно услышать удивительный ответ: им нравится система сборки¹. В Google было потрачено много сил и времени на создание собственной системы сборки с нуля. Усилия оказались настолько успешными, что Blaze — основной компонент системы сборки — несколько раз был повторно реализован экс-гуглерами в других компаниях². В 2015 году Google наконец открыла исходный код Blaze под названием Bazel (<https://bazel.build>).

Назначение системы сборки

По сути, все системы сборки предназначены для преобразования исходного кода, написанного инженерами, в выполняемые двоичные файлы, понятные машинам. Хорошая система сборки обычно обладает двумя важными свойствами:

Скорость

Разработчик должен иметь возможность ввести одну команду, чтобы запустить сборку и получить двоичный файл в считанные секунды.

Безошибочность

Каждый раз, когда разработчик запускает сборку на любом компьютере, он должен получать один и тот же результат (при условии, что исходные файлы и другие входные данные совпадают).

Многие старые системы сборки пытаются найти компромисс между скоростью и безошибочностью, используя короткие пути, которые могут привести к получению несогласованных сборок. Bazel избавил разработчиков от необходимости выбирать между скоростью и безошибочностью.

¹ Согласно данным внутреннего опроса, 83 % гуглеров сообщили, что довольны системой сборки, что ставит ее на четвертое место среди 19 инструментов, предложенных в опросе. Средний показатель удовлетворенности инструментом составил 69 %.

² См. <https://buck.build/> и <https://www.pantsbuild.org/index.html>.

Системы сборки предназначены не только для людей; их также могут использовать машины, чтобы автоматически создавать сборки для тестирования кода или выпуска продукта в продакшен. Фактически большинство сборок в Google запускаются автоматически, а не вручную. Почти все наши инструменты разработки так или иначе связаны с системой сборки, что дает огромную выгоду всем, кто работает с нашей базой кода. Вот небольшой пример использования нашей автоматизированной системы сборки:

- Код автоматически собирается, тестируется и передается в продакшен без участия человека. Разные команды проводят сборки с разной скоростью: одни еженедельно, другие — ежедневно, а третьи — настолько быстро, насколько это возможно (глава 24).
- Изменения, внесенные разработчиком, автоматически тестируются, когда он отправляет их для обзора (глава 19), благодаря чему автор изменений и рецензент могут сразу увидеть любые проблемы, возникшие из-за изменений на этапе сборки или тестирования.
- Изменения повторно тестируются непосредственно перед включением их в главную ветвь репозитория, что значительно уменьшает вероятность сохранения неработоспособных изменений.
- Авторы низкоуровневых библиотек могут тестировать свои изменения во всей базе кода и гарантировать, что эти изменения безопасны для миллионов тестов и двоичных файлов.
- Инженеры могут вносить крупномасштабные изменения, затрагивающие десятки тысяч исходных файлов (например, переименовывать общий символ), и безопасно отправлять в репозиторий и тестировать эти изменения. Подробнее о крупномасштабных изменениях в главе 22.

Все это возможно только благодаря инвестициям Google в свою систему сборки. Конечно, Google уникальна с точки зрения масштаба, однако любая организация любого размера может получить аналогичные преимущества, включив в свой производственный процесс современную систему сборки. В этой главе мы выясним, какие системы сборки считаются в Google «современными» и как их использовать.

Так ли необходимы системы сборки?

Системы сборки позволяют масштабировать разработку. Как будет показано в следующем разделе, отсутствие надлежащей системы сборки порождает проблемы масштабирования.

Мне достаточно компилятора!

Потребность в системе сборки может быть неочевидна. В конце концов, едва ли кто-то из нас использовал систему сборки, когда учился программированию — почти

все мы начинали с таких инструментов командной строки, как `gcc` или `javac`, или их эквивалентов в IDE. Пока весь исходный код располагается в одном каталоге, вполне можно использовать такую команду:

```
javac *.java
```

Она предписывает компилятору Java взять каждый файл с исходным кодом на Java в текущем каталоге и преобразовать его в двоичный файл класса. В простейшем случае этого более чем достаточно.

Однако ситуация быстро усложняется с увеличением размеров проекта. Компилятор `javac` может найти импортируемый код в подкаталогах, находящихся в текущем каталоге, но он не умеет искать код, хранящийся в других местах файловой системы (например, в библиотеке, совместно используемой несколькими проектами). Кроме того, он может преобразовывать код только на Java. Поскольку разные части больших систем часто пишутся на разных языках программирования, которые имеют множество зависимостей между собой, ни один компилятор для единственного языка не сможет собрать всю систему.

Как только приходится иметь дело с кодом на нескольких языках или с несколькими единицами компиляции, сборка кода превращается в многоэтапный процесс. В такой ситуации приходится задумываться, от чего зависит тот или иной код, и выполнять сборку частей в правильном порядке, возможно, с помощью разных инструментов для каждой части. Если изменится любая из частей, этот процесс придется повторить, чтобы избежать зависимости от устаревших двоичных файлов. Для кодовой базы даже среднего размера этот процесс быстро становится утомительным и подверженным ошибкам.

Компилятор также не знает, как обрабатывать внешние зависимости, например сторонние файлы JAR в Java. Часто лучшее, что можно сделать в отсутствие системы сборки, — загрузить зависимость из интернета, поместить ее в папку `lib` на жестком диске и настроить компилятор для чтения библиотек из этой папки. Однако со временем легко забыть, какие библиотеки туда помещены, откуда они взялись и используются ли они до сих пор. И останется надеяться только на удачу при их обновлении по мере выхода новых версий.

Сценарии командной оболочки спасут ситуацию?

Предположим, что ваш любительский проект изначально достаточно прост, чтобы собрать его с помощью компилятора, но постепенно в нем появляются вышеописанные проблемы. Возможно, вы все еще думаете, что вам не нужна настоящая система сборки и достаточно автоматизировать наиболее утомительные этапы сборки, написав несколько простых сценариев командной оболочки, которые позаботятся о сборке компонентов в правильном порядке. Это принесет облегчение на какой-то период, но довольно скоро вы начнете сталкиваться с еще более сложными проблемами:

- Поддержка сборки становится все более утомительной. По мере усложнения системы работа со сценариями начинает отнимать столько же времени, сколько работа с фактическим кодом. Отладка сценариев командной оболочки — трудная задача, все больше и больше нестандартных случаев начинают наслаиваться друг на друга.
- Сборка происходит медленно. Чтобы гарантировать актуальность всех зависимостей, вы запускаете сценарий, который собирает их по порядку при каждом запуске. Вы задумываетесь о добавлении логики для определения частей, которые действительно необходимо собрать повторно, но эта задача кажется ужасно сложной. Или, может быть, вы думаете о том, чтобы каждый раз указывать, какие части следует пересобрать, но это опять приводит вас в исходную точку.
- Хорошая новость: пора выпускать новую версию! Вы вспоминаете, какие аргументы нужно передать команде `jar`, чтобы выполнить окончательную сборку (<https://xkcd.com/1168>), как выгрузить результат и отправить в центральный репозиторий, как создать и разместить обновления в документации, как отправить пользователям уведомления. Хм-м, пожалуй, для этого нужно написать еще один сценарий...
- Катастрофа! Ваш жесткий диск выходит из строя, и теперь вам нужно воссоздать всю систему. Вы были достаточно предусмотрительны и хранили все файлы с исходным кодом в VCS, но как быть с библиотеками, загруженными из интернета? Сможете ли вы найти их снова и гарантировать соответствие версий? Ваши сценарии, вероятно, зависели от определенных инструментов, установленных в определенных местах, — сможете ли вы воссоздать идентичную среду, чтобы сценарии работали как раньше? А как быть с переменными окружения, которые вы настроили давным-давно, чтобы добиться правильной работы компилятора, а потом забыли о них?
- Несмотря ни на что, ваш проект достаточно успешен, и вы можете нанять еще нескольких инженеров. Теперь вы понимаете, что предыдущие проблемы еще не были катастрофой — теперь вам нужно снова и снова проходить один и тот же болезненный процесс настройки окружения для каждого нового разработчика. И несмотря на прилагаемые усилия, все системы разработчиков будут немного отличаться друг от друга. Часто то, что работает на одном компьютере, не работает на другом, и каждый раз требуется несколько часов, чтобы правильно настроить пути к инструментам или проверить версии библиотек, чтобы выяснить, в чем разница.
- Вы решаете автоматизировать систему сборки. Теоретически для этого достаточно поставить новый компьютер и настроить на нем запуск сценария сборки по ночам с помощью `cron`. Вам по-прежнему нужно пройти болезненный процесс настройки, но теперь у вас нет преимуществ человеческого мозга, способного обнаруживать и решать мелкие проблемы. Теперь, входя по утрам в систему, вы видите, что ночная сборка потерпела неудачу, потому что вчера разработчик внес изменение, которое работало в его системе, но не работает в системе автоматической сборки.

Каждая такая проблема легко исправляется, но делать это приходится так часто, что каждый день вы тратите массу времени на поиск и применение простых исправлений.

- По мере развития проекта сборки выполняются все медленнее и медленнее. Однажды, ожидая завершения сборки, вы с грустью смотрите на пустующий рабочий стол коллеги, который находится в отпуске, и задумываетесь о возможности задействовать простаивающие вычислительные мощности.

Это проблема масштабирования. Для одного разработчика, работающего над парой сотен строк кода в течение одной-двух недель (типичный опыт младшего разработчика, только что окончившего университет), компилятора более чем достаточно. Сценарии помогут вам продвинуться немного дальше. Но как только вам потребуется координировать действия нескольких разработчиков и работу их компьютеров, даже идеального сценария сборки будет недостаточно, потому что очень трудно учесть незначительные различия между системами. На этом этапе нужно начинать инвестировать в систему сборки.

Современные системы сборки

К счастью, все описанные проблемы уже решены существующими универсальными системами сборки. По сути, они не сильно отличаются от вышеупомянутого подхода «сделай сам» на основе сценариев: они запускают те же самые компиляторы и требуют знания особенностей базовых инструментов. Но эти системы разрабатывались много лет, что сделало их гораздо более надежными и гибкими, чем сценарии, которые вы можете написать сами.

Все дело в зависимостях

Рассматривая проблемы, описанные выше, мы снова и снова наблюдали одну и ту же закономерность: намного легче управлять своим кодом, чем его зависимостями (глава 21). Зависимости могут быть самыми разнообразными: от задачи (например, «отправить документацию, прежде чем отметить выпуск завершенным») или от артефакта (например, «для сборки кода нужна последняя версия библиотеки компьютерного зрения»). Иногда ваш код может иметь внутренние зависимости от других частей кодовой базы или внешние зависимости от кода или данных, принадлежащих другим командам (в вашей или сторонней организации). Но в любом случае идея «мне нужно это, чтобы получить то» постоянно повторяется в системах сборки, и управление зависимостями, возможно, является их самой фундаментальной задачей.

Системы сборки на основе задач

Сценарии командной оболочки из предыдущего раздела были примером примитивной системы сборки на основе задач. Главной единицей работы такой системы

является задача. Каждая задача — это своего рода сценарий, который может выполнять любую логику и зависеть от других (заранее выполненных) задач. Большинство современных систем сборки, таких как Ant, Maven, Gradle, Grunt и Rake, основаны на задачах.

Большинство современных систем сборки требуют, чтобы вместо сценариев командной оболочки инженеры создавали *файлы сборки*, описывающие порядок сборки. Вот пример из руководства Ant (<https://oreil.ly/WL9ry>):

```
<project name="MyProject" default="dist" basedir=". ">
  <description>
    простой файл сборки
  </description>
  <!-- глобальные параметры для этой сборки -->
  <property name="src" location="src"/>
  <property name="build" location="build"/>
  <property name="dist" location="dist"/>

  <target name="init">
    <!-- Создать отметку времени -->
    <tstamp/>
    <!-- Создать структуру каталогов сборки для компилятора -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
    description="компиляция ресурсов">
    <!-- Скомпилировать код на Java из ${src} в ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
    description="создание дистрибутива">
    <!-- Создать каталог для дистрибутива -->
    <mkdir dir="${dist}/lib"/>

    <!-- Поместить все артефакты из ${build} в файл MyProject-${DSTAMP}.jar -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
  </target>

  <target name="clean"
    description="очистка">
    <!-- Удалить каталоги ${build} и ${dist} -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
  </target>
</project>
```

Файл сборки написан на XML и определяет простые метаданные о сборке и задаче (теги `<target>` в XML¹). Каждая задача выполняет список команд, поддерживаемых системой Ant, включая создание и удаление каталогов, запуск `javac` и создание файла JAR. Набор команд можно расширить с помощью подключаемых модулей (плагинов), чтобы охватить любую логику. Каждая задача может также определять задачи, от которых она зависит, перечислив их в атрибуте `depends`. Эти зависимости образуют ациклический граф (рис. 18.1).

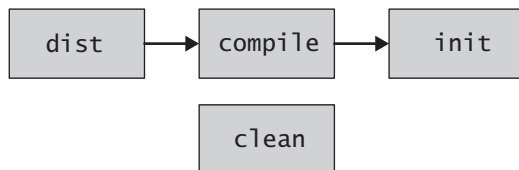


Рис. 18.1. Ациклический граф зависимостей

Пользователи выполняют сборку, вызывая инструмент командной строки Ant и передавая ему задачи в аргументах. Например, когда пользователь запускает команду `ant dist`, система сборки Ant выполняет следующие шаги:

1. Загружает файл `build.xml` в текущий каталог и анализирует его, чтобы создать граф (рис. 18.1).
2. Отыскивает задачу с именем `dist`, указанную в командной строке, и обнаруживает, что она зависит от задачи с именем `compile`.
3. Отыскивает задачу `compile` и обнаруживает, что она зависит от задачи `init`.
4. Отыскивает задачу `init` и обнаруживает, что она не имеет зависимостей.
5. Выполняет команды, перечисленные в задаче `init`.
6. Выполняет команды, перечисленные в задаче `compile`, после выполнения всех зависимостей этой задачи.
7. Выполняет команды, перечисленные в задаче `dist`, после выполнения всех зависимостей этой задачи.

В итоге код, выполняемый системой Ant при запуске задачи `dist`, эквивалентен следующему сценарию командной оболочки:

```

./createTimestamp.sh
mkdir build/
javac src/* -d build/
mkdir -p dist/lib/
jar cf dist/lib/MyProject-$(date --iso-8601).jar build/*
  
```

Если убрать синтаксис, то файл сборки мало чем отличается от сценария сборки. Тем не менее мы уже многого добились. Теперь мы можем создавать новые файлы

¹ В терминологии Ant задачи называются «целями» (target), а команды — «задачами» (task).

сборки в других каталогах, связывать их друг с другом и добавлять новые задачи, зависящие от имеющихся. Нам достаточно только передать имя одной задачи инструменту командной строки `ant`, и он позаботится обо всем остальном.

`Ant` — это очень старая система. Первая ее версия была выпущенная в 2000 году и была совсем не похожа на «современную» систему сборки! За прошедшие годы появились другие системы, такие как `Maven` и `Gradle`, по сути заменившие `Ant`. Они добавили такие возможности, как автоматическое управление внешними зависимостями и более ясный синтаксис без XML. Но природа этих новых систем осталась прежней: они позволяют инженерам писать сценарии сборки принципиальным и модульным способом в виде перечней задач и предоставляют инструменты для их выполнения и управления зависимостями.

Темная сторона систем сборки, основанных на задачах

По сути, эти системы позволяют инженерам определить любой сценарий как задачу и реализовать практически все, что только можно вообразить. Но работать с ними становится все труднее с увеличением сложности сценариев сборки. Проблема таких систем состоит в том, что они *многое перекладывают на плечи инженеров, почти ничего не делая самостоятельно*. Система не знает, что делают сценарии, и из-за этого страдает производительность, потому что система вынуждена действовать максимально консервативно, планируя и выполняя этапы сборки. Кроме того, система не может убедиться, что каждый сценарий выполняет то, что должен, поэтому сценарии усложняются и требуют отладки.

Сложность параллельного выполнения этапов сборки. Современные рабочие станции, используемые для разработки, обычно обладают большой вычислительной мощностью, имеют процессоры с несколькими ядрами и теоретически способны выполнять несколько этапов сборки параллельно. Но системы на основе задач часто не могут выполнять задачи параллельно, даже если это выглядит возможным. Предположим, что задача *A* зависит от задач *B* и *C*. Поскольку задачи *B* и *C* не зависят друг от друга, безопасно ли выполнять их одновременно, чтобы система могла быстрее перейти к задаче *A*? Возможно, если они не используют одни и те же ресурсы. Но если они используют один и тот же файл для хранения статуса выполнения, то их одновременный запуск вызовет конфликт. В общем случае система ничего не знает об этом, поэтому она должна рисковать вероятностью конфликтов (которые могут приводить к редким, но очень трудным для отладки проблемам сборки) или ограничиться выполнением сборки в одном потоке и в одном процессе. Из-за этого огромная вычислительная мощь машины разработчика может недоиспользоваться, возможность распределения сборки между несколькими машинами будет полностью исключена.

Сложности инкрементального выполнения сборки. Хорошая система сборки позволяет инженерам выполнять инкрементальные сборки, когда небольшое изменение не требует повторной сборки всей кодовой базы с нуля. Это особенно важно, если система сборки работает медленно и не может выполнять этапы сборки

параллельно по вышеупомянутым причинам. Но, к сожалению, и здесь системы сборки, основанные на задачах, показывают себя не с лучшей стороны. Поскольку задачи могут делать что угодно, невозможно проверить, были ли они уже выполнены. Многие задачи просто берут набор исходных файлов и запускают компилятор, чтобы создать набор двоичных файлов, поэтому их не нужно запускать повторно, если исходные файлы не изменились. Но, не имея дополнительной информации, система не может знать, загружает ли задача файл, который был изменен, или записывает ли она отметку времени, изменяющуюся при каждом запуске. Чтобы гарантировать безошибочность, система часто вынуждена повторно запускать каждую задачу во время каждой сборки.

Некоторые системы сборки пытаются разрешить инкрементальные сборки, позволяя инженерам указывать условия, при которых задача должна быть повторно запущена. Эта возможность реализуется гораздо сложнее, чем кажется. Например, в таких языках, как C++, которые позволяют напрямую подключать другие файлы, невозможно определить весь набор файлов, за изменениями в которых необходимо следить, без анализа исходного кода. Инженеры строят догадки и нередко ошибочно используют результат задачи повторно. Когда такое случается слишком часто, у инженеров вырабатывается привычка запускать задачу чистки (`clean`) перед каждой сборкой, чтобы обновить состояние, что полностью лишает смысла инкрементальную сборку. На удивление сложно выяснить, когда задача должна выполняться повторно, и с этим машины справляются лучше, чем люди.

Сложности сопровождения и отладки сценариев. Наконец, сами сценарии сборки, используемые системами сборки на основе задач, часто слишком сложны для сопровождения. Им часто уделяется меньше внимания, но, тем не менее, сценарии сборки — это точно такой же код, как и собираемая система, и в них тоже могут появляться ошибки. Вот несколько примеров ошибок, которые очень часто допускаются при работе с системой сборки на основе задач:

- Задача *A* зависит от задачи *B*, ожидая получить от нее определенный файл. Владелец задачи *B* не понимает, что от нее зависят другие задачи, поэтому он меняет ее, и в результате файл, генерируемый задачей *B*, оказывается в другом месте. Эта ошибка никак не проявляет себя, пока кто-то не попытается запустить задачу *A* и не обнаружит, что она терпит неудачу.
- Задача *A* зависит от задачи *B*, которая зависит от задачи *C*, которая создает определенный файл, необходимый задаче *A*. Владелец задачи *B* решает, что она больше не должна зависеть от задачи *C*, что заставляет задачу *A* терпеть неудачу, потому что задача *B* больше не вызывает задачу *C*!
- Разработчик новой задачи делает ошибочное предположение о машине, на которой выполняется задача, например о местоположении инструмента или значениях определенных переменных окружения. Задача работает на его компьютере, но терпит неудачу на компьютере другого разработчика.

- Задача выполняет недетерминированную операцию, например загружает файл из интернета или добавляет отметку времени в сборку. Из-за этого разработчики получают потенциально разные результаты при каждом запуске сборки, а значит, не всегда могут воспроизвести и исправить ошибки, возникающие друг у друга или в автоматизированной системе сборки.
- Задачи с множественными зависимостями могут оказываться в состоянии гонки. Если задача *A* зависит от задач *B* и *C*, а задачи *B* и *C* изменяют один и тот же файл, то задача *A* будет получать разный результат, в зависимости от того, какая из задач — *B* или *C* — завершится первой.

Нет универсального рецепта решения этих проблем производительности, безошибочности или удобства сопровождения в рамках описанной здесь структуры задач. Пока инженеры пишут произвольный код для выполнения во время сборки, у системы не будет достаточно информации, чтобы всегда быстро и безошибочно производить сборку. Чтобы решить эту проблему, нужно отобрать часть полномочий у инженеров и передать их системе, а также переосмыслить роль системы, но уже не в терминах выполняемых задач, а в терминах создаваемых артефактов. Этот подход используется в Blaze, и Bazel и описывается в следующем разделе.

Системы сборки на основе артефактов

Чтобы создать более удачную систему сборки, нужно отступить на шаг назад. Проблема ранних систем состоит в том, что они позволяли инженерам определять свои задачи. Возможно, лучше предложить инженерам небольшое количество задач, определяемых системой, которые они смогут настраивать в ограниченных пределах. И самой главной задачей выступит *сборка* кода. Инженеры по-прежнему будут сообщать системе, *что* собирать, но *как* выполнять сборку, будет решать система.

Именно этот подход использован в Blaze и в других системах сборки *на основе артефактов*, которые произошли от нее (включая Bazel, Pants и Buck). В них тоже есть файлы сборки, но вместо набора императивных команд на языке сценариев, полного по Тьюрингу, описывающих порядок создания выходных данных, они *декларативно описывают* набор артефактов, который нужно получить, их зависимости и ограниченный набор параметров, влияющих на процесс сборки. Когда инженеры выполняют команду `blaze` в командной строке, они сообщают ей набор целей для сборки («что»), а Blaze отвечает за настройку, выполнение и планирование этапов компиляции («как»). Поскольку такая система сборки полностью контролирует выбор и очередность выполнения инструментов, она имеет более надежные гарантии, которые позволяют ей действовать эффективно и без ошибок.

Функциональная перспектива

Легко провести аналогию между системами сборки на основе артефактов и функциональным программированием. Программы на традиционных императивных языках (таких как Java, C и Python) определяют списки операторов, которые должны

выполняться в определенной очередности, точно так же, как системы сборки на основе задач позволяют программистам определять последовательность выполняемых этапов. Программы на языках функционального программирования (например, Haskell и ML), напротив, имеют структуру, больше похожую на серию математических уравнений. В них программист описывает вычисления, но выбор, когда и как их выполнить, он оставляет компилятору. Это соответствует идее декларативного описания задач, получив которое система сборки на основе артефактов сама выбирает, как их выполнить.

Многие задачи сложно выразить на языке функционального программирования, но когда такое возможно, задачи получают дополнительную выгоду: компилятор с легкостью может организовать их параллельное выполнение и дать строгие гарантии их безошибочности, что невозможно в императивных языках. Функциональные языки позволяют выражать задачи, связанные с простым преобразованием одного фрагмента данных в другой, в виде набора правил или функций. И это именно то, что нужно системе сборки: вся система фактически является математической функцией, которая принимает исходные файлы (и инструменты, такие как компилятор) на входе и возвращает двоичные файлы на выходе.

Конкретный пример в Bazel. Bazel — это открытая версия внутреннего инструмента сборки Blaze, используемой в Google, и хороший пример системы сборки на основе артефактов. Вот как выглядит файл сборки в Bazel (которому обычно дается имя BUILD):

```
java_binary(  
  name = "MyBinary",  
  srcs = ["MyBinary.java"],  
  deps = [  
    ":mylib",  
  ],  
)  
  
java_library(  
  name = "mylib",  
  srcs = ["MyLibrary.java", "MyHelper.java"],  
  visibility = ["//java/com/example/myproduct:__subpackages__"],  
  deps = [  
    "//java/com/example/common",  
    "//java/com/example/myproduct/otherlib",  
    "@com_google_common_guava_guava//jar",  
  ],  
)
```

В системе Bazel файлы BUILD определяют *цели*. В этом примере определены два типа целей: `java_binary` и `java_library`. Каждая цель соответствует артефакту, который может быть создан системой: цели `binary` создают двоичные файлы, выполняемые непосредственно, а цели `library` создают библиотеки, которые могут использовать-

ся двоичными файлами или другими библиотеками. Каждая цель имеет атрибуты `name` (определяет имя для ссылки в командной строке и в других целях), `srcs` (определяет список файлов с исходным кодом, которые необходимо скомпилировать для создания артефакта цели) и `deps` (определяет другие цели, которые должны быть собраны перед этой целью и связаны с ней). Зависимости (`deps`) могут находиться в одном пакете с основной целью (как, например, зависимость `":mylib"` для `MyBinary`), в разных пакетах, но в одной иерархии исходного кода (как, например, зависимость `"//java/com/example/common"` для `mylib`) или в сторонних артефактах, находящихся за пределами иерархии исходного кода (как, например, зависимость `"@com_google_common_guava_guava//jar"` для `mylib`). Иерархии исходного кода называются *рабочими пространствами* и определяются наличием специального файла `WORKSPACE` в корневом каталоге.

Сборка в Bazel запускается с помощью инструмента командной строки, так же как при использовании Ant. Чтобы собрать цель `MyBinary`, пользователь должен выполнить команду `bazel build :MyBinary`. При первом вызове этой команды в чистом репозитории Bazel выполнит следующие действия:

1. Проанализирует все файлы `BUILD` в рабочем пространстве и создаст граф зависимостей между артефактами.
2. Использует граф, чтобы определить *транзитивные (промежуточные) зависимости* для `MyBinary`; то есть все цели, от которых зависит `MyBinary`, а также все цели, от которых зависят эти цели, и т. д.
3. По порядку соберет каждую из этих зависимостей (или загрузит, если зависимости внешние). Bazel начинает сборку с целей, не имеющих других зависимостей, и проверяет, какие еще зависимости необходимо собрать для каждой цели. После сборки зависимостей Bazel приступает к сборке самой цели. Этот процесс продолжается, пока не будут собраны все транзитивные зависимости для `MyBinary`.
4. Соберет `MyBinary` и создаст окончательный выполняемый двоичный файл, включающий все зависимости, собранные на шаге 3.

Может показаться, что этот процесс не сильно отличается от процесса в системе сборки, основанной на задачах. И действительно, конечный результат — все тот же двоичный файл, и процесс его создания основан на определении последовательности этапов для сборки зависимостей и последующего их выполнения по порядку. Но в этих подходах есть принципиальные отличия. Первое отличие — в шаге 3 Bazel знает, что в результате сборки каждой цели будут созданы библиотеки Java и для этого требуется запустить компилятор Java, а не произвольный пользовательский сценарий, поэтому она без опаски выполняет эти шаги параллельно. В результате производительность может увеличиться на порядок по сравнению с последовательной сборкой целей на машине с одноядерным процессором. Это возможно только потому, что подход на основе артефактов возлагает всю ответственность за выбор политики действий на систему сборки, чтобы та могла получить более надежные гарантии относительно параллелизма.

Однако параллелизм — не единственное преимущество. Другое преимущество, которое дает второй подход, становится очевидным, когда разработчик вводит команду `bazel build :MyBinary` во второй раз, не внося в исходный код никаких изменений. Bazel завершает работу менее чем через секунду и сообщает, что цель обновлена. Это возможно благодаря парадигме функционального программирования — Bazel знает, что каждая цель является результатом запуска компилятора Java. Если входные данные не менялись, можно повторно использовать результат компилятора, полученный ранее, и это верно на всех уровнях. Если `MyBinary.java` изменится, Bazel поймет, что нужно повторно собрать `MyBinary` и использовать при этом ранее собранную цель `mylib`. Если изменится исходный код `//java/com/example/common`, то Bazel поймет, что нужно повторно собрать эту библиотеку, а также цели `mylib` и `MyBinary` и использовать при этом ранее собранную библиотеку `//java/com/example/myproduct/otherlib`. Поскольку системе Bazel известны свойства инструментов, используемые на каждом этапе сборки, она может каждый раз повторно собирать минимальный набор артефактов, гарантируя, что цели, исходный код которых не изменялся, повторно собираться не будут.

Переосмысление процесса сборки в терминах артефактов, а не задач — дело тонкое, но стоящее. Ограничивая возможности программиста, система сборки может получить более точное представление о том, что делается на каждом этапе сборки, и использовать эти знания, чтобы повысить эффективность сборки за счет параллельного выполнения этапов и повторного использования результатов. Но на самом деле это лишь первый шаг на пути к распределенной и хорошо масштабируемой системе сборки.

Другие небольшие хитрости Bazel

Системы сборки на основе артефактов решают проблемы параллелизма и повторного использования, присущие системам сборки на основе задач. Но есть еще несколько проблем, которые мы не осветили. В Bazel были найдены для них интересные решения, и мы должны обсудить их, прежде чем двигаться дальше.

Инструменты как зависимости. Поскольку результаты сборки зависят от инструментов, установленных на компьютерах, в разных системах из-за использования разных версий инструментов или их установки в разных местах, не всегда можно воспроизвести одни и те же результаты. Проблема становится еще более сложной, когда в проекте используются языки, требующие разных инструментов, собранных или скомпилированных на разных платформах (например, в Windows или Linux), и для разных платформ требуются разные наборы инструментов для выполнения одной и той же работы.

Bazel решает первую часть этой проблемы, интерпретируя инструменты как зависимости для каждой цели. Каждая библиотека `java_library` в рабочем пространстве неявно зависит от хорошо известного компилятора Java, который может быть настроен глобально на уровне рабочего пространства. Когда Blaze выполняет сборку `java_library`, она проверяет доступность указанного компилятора в известном месте и загружает его, если проверка дала отрицательный результат. Компилятор Java

интерпретируется как зависимость: если он изменится, система повторно соберет все артефакты, зависящие от него. Для всех типов целей в Bazel используется одна и та же стратегия объявления инструментов, которые должны запускаться, и это гарантирует, что Bazel сможет загрузить их, в какой бы системе она ни работала.

Вторая часть проблемы — независимость от платформы — в Bazel решается с помощью наборов инструментов (<https://oreil.ly/ldiv8>). Вместо зависимости от конкретных инструментов цели фактически зависят от типов наборов инструментов. Набор инструментов содержит инструменты и обладает некоторыми свойствами, определяющими, как цель данного типа должна собираться на конкретной платформе. Рабочее пространство может определять конкретный набор инструментов для данного типа в зависимости от данного сетевого узла и целевой платформы (подробнее об этом в руководстве по Bazel).

Расширение системы сборки. В Bazel есть цели по умолчанию для нескольких популярных языков программирования, но инженеры всегда будут стремиться к большему — одним из преимуществ систем на основе задач является их гибкость в поддержке любого процесса сборки, и было бы странно отказываться от этого в системах сборки на основе артефактов. К счастью, Bazel позволяет расширять поддерживаемые типы целей и разрешает инженерам добавлять собственные правила (<https://oreil.ly/Vvg5D>).

Чтобы определить правило в Bazel, нужно объявить входные данные, необходимые для правила (в форме атрибутов в файле BUILD), и фиксированный набор выходных данных, которые создает правило. Также необходимо определить *действия*, которые будут выполняться правилом. Для каждого действия должны быть объявлены свои входные и выходные данные, конкретный выполняемый файл для запуска или определенная строка для записи в файл. Действия могут быть связаны с другими действиями через свои входные и выходные данные. То есть действия являются компонентными единицами самого нижнего уровня в системе сборки — действие может делать все, что потребуется, при условии, что оно использует только объявленные входные и выходные данные, а о планировании действий и кешировании их результатов позаботится Bazel.

Такая система не является абсолютно надежной, учитывая невозможность помешать разработчику внедрить в действие недетерминированный процесс. Но на практике такое случается нечасто, а перенос возможностей для злоупотреблений на уровень действий значительно снижает вероятность ошибок. Правила, поддерживающие распределенные языки и инструменты, широко доступны в интернете, и в большинстве проектов не требуется определять собственные правила. Но даже в случаях, когда это необходимо, правила должны быть определены в одном месте в репозитории, а это означает, что большинство инженеров смогут использовать эти правила, не заботясь об их реализации.

Изоляция окружения. На первый взгляд кажется, что действия могут столкнуться с теми же проблемами, что и задачи в других системах, потому что действия могут записывать результаты в один и тот же файл и конфликтовать друг с другом. В Bazel

такие конфликты невозможны, потому что она использует прием *изоляции* (<https://oreil.ly/IP5Y9>). В поддерживаемых системах все действия изолированы друг от друга посредством изоляции файловой системы. По сути, каждое действие видит только ограниченное представление файловой системы, которое включает объявленные входные данные и созданные выходные данные. Это обеспечивается такими системами, как LXC (Linux Containers) в Linux, реализующими ту же технологию, что и Docker. Это означает, что действия не могут конфликтовать друг с другом, потому что не могут читать файлы, которые они не объявляли, а любые выходные файлы, создаваемые ими, но не объявленные, будут уничтожены после завершения действия. Bazel также использует прием изоляции, чтобы ограничить сетевые взаимодействия между действиями.

Детерминизация внешних зависимостей. Остается еще одна проблема: системы сборки часто загружают зависимости (инструменты или библиотеки) из внешних источников, не создавая их напрямую. Примером может служить зависимость `@com_google_common_guava_guava//jar`, которая загружает файл JAR из Maven.

Использование файлов, находящихся за пределами текущего рабочего пространства, сопряжено с риском. Эти файлы могут измениться в любой момент, что требует от системы сборки постоянной проверки их актуальности. Если удаленный файл изменится, а в исходном коде в рабочем пространстве никаких изменений не будет, это тоже может привести к невоспроизводимым результатам сборки: сборка может безупречно выполняться в один день и завершиться ошибкой — в другой из-за незаметного изменения зависимости. Наконец, внешняя зависимость, принадлежащая третьей стороне, может представлять огромный риск для безопасности¹: если злоумышленнику удастся проникнуть на сторонний сервер, он сможет подменить файл зависимости чем-то своим и получить полный контроль над средой сборки и ее результатом.

Основная проблема заключается в том, что система сборки вынуждена получать внешние файлы не из VCS. Обновление зависимости должно производиться осознанно и централизованно, а не даваться на откуп отдельным инженерам или автоматической системе. Это связано с тем, что даже при использовании «главной ветви» мы по-прежнему хотим, чтобы сборки сохраняли детерминированность. То есть, извлекая исходный код, зафиксированный в репозитории на прошлой неделе, мы ждем, что и зависимости будут такими, какими они были тогда, а не сейчас.

Bazel и некоторые другие системы сборки решают эту проблему, требуя наличия файла объявления рабочей области, в котором определяется *криптографический хеш* для каждой внешней зависимости в рабочей области². Хеш — это краткий способ однозначно представить файл без его передачи в VCS. Чтобы сослаться на

¹ Такие атаки на «цепочки поставки ПО» (<https://oreil.ly/bfC05>) становятся все более распространенными.

² Недавно в Go была добавлена предварительная поддержка модулей, использующая ту же систему (<https://oreil.ly/IHGjt>).

новую внешнюю зависимость из рабочего пространства, в файл объявления необходимо вручную или автоматически добавить хеш этой зависимости. В момент запуска сборки Bazel проверит фактический хеш кешированной зависимости на соответствие ожидаемому, указанному в объявлении, и повторно загрузит файл, только если хеш отличается.

Если хеш загружаемого артефакта отличается от хеша, указанного в файле объявления, сборка завершится ошибкой. Изменить хеш в объявлении можно автоматически, но это изменение должно быть одобрено и зарегистрировано в VCS до того, как система сборки примет новую зависимость. Это означает, что всегда есть возможность узнать, когда зависимость была обновлена, а внешняя зависимость не может измениться без соответствующего изменения в объявлении рабочего пространства. Это также означает, что при извлечении из репозитория более старой версии исходного кода для сборки гарантированно будут использоваться те же зависимости, которые использовались в момент, когда эта версия была зафиксирована (или потерпит неудачу, если необходимые зависимости окажутся больше недоступными).

Конечно, может возникнуть проблема, если удаленный сервер станет недоступен или начнет посылать поврежденные данные — это может вызвать сбой всех ваших сборок, если у вас нет другой копии этой зависимости. Чтобы избежать этой проблемы, мы рекомендуем для любого нетривиального проекта зеркалировать все его зависимости на подконтрольных вам серверах или службах, которым вы доверяете. В противном случае вы рискуете оказаться во власти третьей стороны в смысле доступности для вашей системы сборки, даже несмотря на то, что зарегистрированные хеши гарантируют ее безопасность.

Распределенная сборка

База кода в Google огромна — в ней хранится более двух миллиардов строк кода, а цепочки зависимостей могут быть очень глубокими. Даже простые двоичные файлы в Google часто зависят от десятков тысяч целей сборки. В таком масштабе просто невозможно завершить сборку за разумный промежуток времени на одной машине: никакая система сборки не может обойти фундаментальные законы физики, которым подчиняется аппаратное обеспечение. Единственный способ выполнить эту работу — использовать систему сборки, которая поддерживает *распределенный* режим работы, в котором единицы работы распределяются между произвольным и масштабируемым количеством машин. Если мы разобьем работу на достаточно маленькие части (подробнее об этом позже), то сможем завершить любую сборку любого размера настолько быстро, насколько позволят вложенные средства. Масштабируемость — это чаша Грааля, которую мы стремились заполучить, создавая систему сборки на основе артефактов.

Удаленное кеширование

Самый простой тип распределенной сборки — с использованием только удаленного кеширования (рис. 18.2).

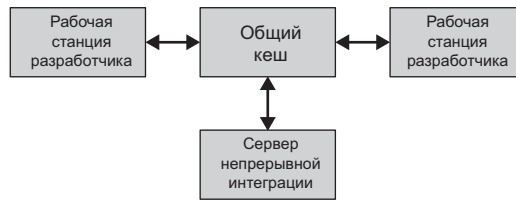


Рис. 18.2. Распределенная сборка с удаленным кешированием

Все системы, выполняющие сборку, включая рабочие станции разработчиков и системы непрерывной интеграции, пользуются общей службой удаленного кеширования. Роль этой службы может играть быстрое и локальное хранилище, такое как Redis, или облачная служба, такая как Google Cloud Storage. Всякий раз, когда пользователь хочет выполнить сборку артефакта, непосредственно или как зависимости, система сначала ищет этот артефакт в удаленном кеше и в случае успеха загружает его. В противном случае система сама производит сборку артефакта и выгружает результат в кеш. Благодаря этому сборка низкоуровневых зависимостей, которые меняются редко, может выполняться один раз, а ее результат — использоваться множеством пользователей. В Google многие артефакты извлекаются из кеша, что значительно увеличивает эффективность нашей системы сборки.

Чтобы система удаленного кеширования приносила пользу, система сборки должна гарантировать воспроизводимость ее результатов — они должны позволять определить набор входных данных для любой цели и получить один и тот же результат на любой машине. Это единственный способ гарантировать, что результаты загрузки артефакта совпадут с результатами сборки на локальной машине. К счастью, Bazel дает такую гарантию и поддерживает удаленное кеширование (<https://oreil.ly/D9doX>), которое привязывает каждый артефакт в кеше к цели и к хешу входных данных. Только так разные инженеры смогут создавать разные модификации одной и той же цели, а удаленный кеш будет хранить все полученные артефакты и обслуживать их без конфликтов.

Конечно, для получения выгоды от удаленного кеша загрузка артефакта должна происходить быстрее, чем его сборка. Однако это не всегда так, особенно если сервер кеша находится далеко от машины, выполняющей сборку. Компьютерная сеть и система сборки в Google настроены для быстрой загрузки результатов сборки. При настройке удаленного кеширования в своей организации учитывайте задержки в сети и опытным путем убедитесь, что кеширование действительно улучшает производительность.

Удаленное выполнение

Удаленное кеширование — это еще не настоящая распределенная сборка. Если данные в кеше потеряются или изменится некоторый низкоуровневый компонент, что потребует повторно собрать все зависимости, вам все равно придется выполнить сборку от начала до конца на локальной машине. Истинная цель распределенной

сборки — обеспечить возможность *удаленного выполнения*, когда фактическую работу по сборке можно распределить между любым числом рабочих машин (рис. 18.3). Инструмент сборки, выполняющийся на машине каждого пользователя (где пользователи — это инженеры или автоматизированные системы сборки), отправляет запросы главному серверу сборки. Главный сервер разбивает запросы на операции и распределяет их между рабочими машинами. Каждая рабочая машина выполняет требуемые от нее действия с входными данными, которые передал пользователь, и возвращает полученные артефакты. Затем эти артефакты совместно используются другими машинами, которым необходимы эти артефакты для выполнения своих действий. И так до тех пор, пока окончательный результат не будет получен и отправлен пользователю.

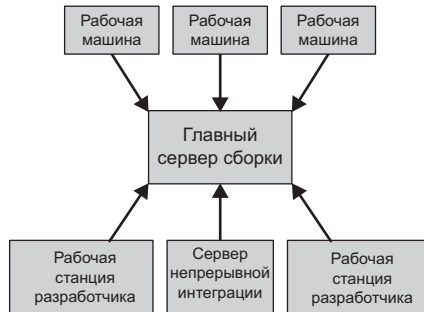


Рис. 18.3. Система с поддержкой удаленного выполнения

Самое сложное в реализации такой системы — управление взаимодействиями между рабочими машинами, главным сервером и локальной машиной пользователя. Одни рабочие машины могут ждать завершения сборки промежуточных артефактов другими рабочими машинами, чтобы окончательный результат можно было отправить обратно на локальный компьютер пользователя. Чтобы преодолеть эти сложности, в основу системы можно положить распределенный кеш, описанный выше, и заставить каждую рабочую машину выводить свои результаты в кеш и извлекать зависимости из кеша. Главный сервер может блокировать рабочие машины до завершения сборки необходимых им зависимостей, чтобы они могли извлечь входные данные из кеша. Конечный продукт тоже можно поместить в кеш, что позволит локальному компьютеру загрузить его оттуда. Обратите внимание, что также необходимо предусмотреть возможность экспорта локальных изменений из дерева исходных кодов пользователя, чтобы рабочие машины могли применить эти изменения перед сборкой.

Чтобы такая система на основе артефактов заработала, необходимо соединить все ее части, описанные выше. Среда сборки должна полностью описывать себя, чтобы рабочие машины могли действовать без участия человека. Сами процессы сборки должны быть полностью автономными, поскольку каждый шаг может выполняться

на разных машинах. Результаты должны быть полностью детерминированными, чтобы каждая рабочая машина могла доверять результатам, полученным от других. Такие гарантии чрезвычайно сложно обеспечить в системе, основанной на задачах, что практически не позволяет создать на ее основе надежную систему удаленного выполнения.

Распределенная система сборки в Google. С 2008 года в Google была применена распределенная система сборки, которая использовала и удаленное кеширование, и удаленное выполнение (рис. 18.4).

Удаленный кеш в Google называется ObjFS. Он состоит из серверной части, которая хранит результаты сборки в хранилище Bigtables (https://oreil.ly/S_N-D), распределенном по всему нашему парку производственных машин, и внешнего демона FUSE с именем `objfsd`, который выполняется на каждой машине разработчика. Демон FUSE позволяет просматривать результаты сборки, как если бы они были обычными файлами, хранящимися на рабочей станции, но загружает содержимое файлов только по требованию пользователя. Такое обслуживание содержимого файлов значительно сокращает использование сети и дисков, и система может выполнять сборку в два раза быстрее (<https://oreil.ly/NZxSp>) по сравнению с вариантом, когда все результаты сборки сохраняются на локальном диске разработчика.

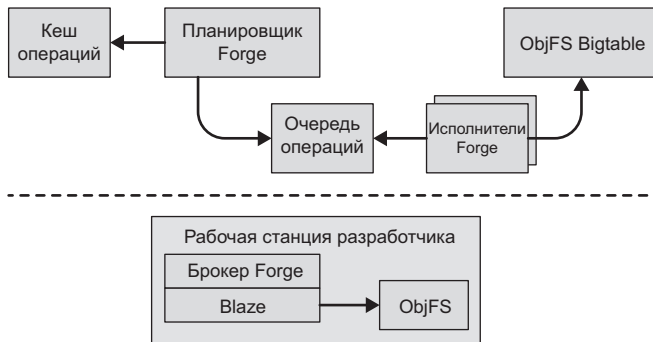


Рис. 18.4. Распределенная система сборки в Google

Система удаленного выполнения в Google называется Forge. Клиент Forge в Blaze, называемый брокером (distributor), посылает планировщику (scheduler) запросы на выполнение операции в наших центрах обработки данных. Планировщик поддерживает кеш результатов выполнения операций, что позволяет ему немедленно вернуть ответ, если операция уже была создана любым другим пользователем системы. В противном случае операция помещается в очередь. Большой пул исполнителей заданий постоянно извлекает операции из этой очереди, выполняет их и сохраняет результаты непосредственно в кеше ObjFS Bigtables. Эти результаты могут использоваться исполнителями при выполнении операций в будущем или загружаться конечным пользователем с помощью `objfsd`.

Такая система масштабируется для эффективной поддержки всех сборок, выполняемых в Google. *Каждый день* в Google выполняются миллионы сборок, запускаются миллионы тестов и производятся петабайты результатов сборки из миллиардов строк исходного кода. Такая система не только позволяет нашим инженерам быстро создавать сложные кодовые базы, но и внедрять огромное количество автоматизированных инструментов и систем, опирающихся на нашу систему сборки. На разработку этой системы мы потратили много лет, но в настоящее время доступны инструменты с открытым исходным кодом, используя которые любая организация сможет реализовать аналогичную систему. Для развертывания такой системы сборки нужны время и энергия, но конечный результат будет стоить затраченных усилий.

Время, масштаб и компромиссы

Главная задача системы сборки — упростить длительную работу с большими базами кода. Как и в любом другом аспекте программной инженерии, при выборе системы сборки необходимо учитывать возможные компромиссы. Подход «сделай сам» с использованием сценариев командной оболочки подходит только для самых маленьких проектов, в которых долго не изменяется код или используются такие языки, как Go, имеющие встроенную систему сборки.

Выбор системы сборки на основе задач вместо самодельных сценариев значительно улучшает масштабируемость проекта, позволяя автоматизировать сложные сборки и легче добиваться воспроизводимых результатов на разных машинах. Недостаток такого выбора заключается в том, что он заставляет инженера структурировать сборку и высчитывать затраты на написание файлов сборки (без автоматизированных инструментов). Для большинства проектов это вполне приемлемый компромисс, но в тривиальных проектах (например, весь код которых содержится в одном файле) затраты могут не окупиться.

С увеличением проекта системы сборки на основе задач начинают сталкиваться с некоторыми фундаментальными проблемами. Преодолеть эти проблемы можно с помощью системы сборки на основе артефактов. Такие системы сборки раздвигают горизонты масштабирования, позволяя распределить сборку между множеством машин и давая уверенность тысячам инженеров, что их сборки будут выполнены согласованным и воспроизводимым способом. Но эти системы недостаточно гибкие: они не позволяют описывать обобщенные задачи на настоящем языке программирования и проводят сборку в рамках своих ограничений. Обычно это не проблема для проектов, которые изначально создавались с прицелом на использование системы на основе артефактов. Но миграция системы, основанной на задачах, может породить значительные сложности и потребовать затрат сил и времени, которые могут не окупиться, особенно если используемая система сборки на основе задач продолжает удовлетворять требованиям к скорости или безошибочности.

Замена системы сборки проекта может стоить дорого, и эта стоимость увеличивается с увеличением проекта. Вот почему в Google считают, что почти каждый новый проект

выиграет, если изначально будет включен в систему сборки на основе артефактов, такую как Bazel. В Google практически весь код — от крошечных экспериментальных проектов до Google Search — собирается с помощью Blaze.

Модули и зависимости

Проекты, использующие системы сборки на основе артефактов, такие как Bazel, делятся на модули, причем зависимости между модулями выражаются через файлы BUILD. Правильная организация модулей и зависимостей может иметь огромное влияние не только на производительность системы сборки, но и на затраты, связанные с ее поддержкой.

Дробление на модули и правило 1 : 1 : 1

Первый вопрос, который возникает при структурировании сборки на основе артефактов: какой объем функциональности должен включать отдельный модуль? В Bazel модуль представлен целью, определяющей единицу сборки, такой как `java_library` или `go_binary`. С одной стороны, весь проект может содержаться в одном модуле, если поместить один файл BUILD в корневой каталог и рекурсивно объединить все исходные файлы этого проекта. С другой стороны, почти каждый исходный файл можно преобразовать в отдельный модуль, фактически потребовав, чтобы каждый файл перечислял в своем файле BUILD все остальные файлы, от которых он зависит.

Большинство проектов имеют структуру, занимающую промежуточное положение между этими крайностями, поскольку выбор количества модулей определяется компромиссом между производительностью и удобством сопровождения. Включив проект целиком в единственный модуль, вы избавите себя от необходимости прикасаться к файлу BUILD, кроме как для добавления внешних зависимостей, но при этом система сборки всегда будет вынуждена собирать весь проект целиком. То есть она не сможет выполнять сборку параллельно и кешировать уже собранные части. В противном случае, когда каждому файлу соответствует свой модуль, система сборки обладает максимальной гибкостью в кешировании и планировании этапов сборки, но инженерам приходится прилагать больше усилий для поддержания списков зависимостей при каждом их изменении.

Степень дробления на модули во многом зависит от языка, однако мы в Google склонны отдавать предпочтение модулям значительно меньшего размера, чем можно определить в системе сборки на основе задач. Типичный двоичный файл в Google зависит от десятков тысяч целей, и даже небольшая команда может владеть несколькими сотнями целей в своей кодовой базе. В таких языках, как Java, имеющих строгое деление на пакеты, каждый каталог обычно содержит один пакет, цель и файл BUILD (Pants, еще одна система сборки, основанная на Blaze, называет это правилом 1 : 1 : 1 (<https://oreil.ly/ISKbW>)). Языки с более слабыми соглашениями о пакетах часто определяют несколько целей для каждого файла BUILD.

Преимущества небольших целей сборки начинают по-настоящему проявляться в масштабе, потому что позволяют быстрее производить распределенную сборку и реже повторно собирать цели, и при тестировании, поскольку для мелких целей система сборки может запускать более ограниченное подмножество тестов. Мы в Google верим в преимущества систематического использования небольших целей, поэтому вкладываем силы и время в создание инструментов автоматического управления файлами BUILD, помогающих снять лишнее бремя с плеч разработчиков. Многие из этих инструментов (<https://oreil.ly/r0wO7>) теперь распространяются с открытым исходным кодом.

Уменьшение области видимости модуля

Vazel и другие системы сборки позволяют определять область видимости для каждой цели, то есть указывать, какие другие цели могут зависеть от нее. Цели могут быть общедоступными (`public`) — на них может ссылаться любая другая цель в рабочем пространстве; приватными (`private`) — на них могут ссылаться только цели, перечисленные в том же файле BUILD, или доступными только явно определенному списку других целей. Видимость — по сути это зеркальное отражение зависимости: если необходимо, чтобы цель *A* зависела от цели *B*, то цель *B* должна объявить себя видимой для цели *A*.

Так же как в большинстве языков программирования, обычно лучше минимизировать область видимости, насколько это возможно. Как правило, команды в Google объявляют свои цели общедоступными, только если они представляют широко используемые библиотеки, доступные любой команде в Google. Команды, которым требуется, чтобы другие команды координировали свои действия перед использованием их кода, ведут «белый список» клиентов своей цели, ограничивая ее видимость. Видимость внутренних целей, используемых только внутри команды, ограничена каталогами, принадлежащими этой команде, и в большинстве случаев ее файлы BUILD имеют только одну приватную цель.

Управление зависимостями

Модули должны иметь возможность ссылаться друг на друга. Но управление зависимостями между модулями требует времени (хотя существуют инструменты, которые могут помочь автоматизировать эту задачу). Для выражения этих зависимостей обычно требуется писать большие файлы BUILD.

Внутренние зависимости

В большом проекте, разбитом на множество мелких модулей, большинство зависимостей почти наверняка будут внутренними, то есть большинство модулей будут зависеть от других модулей, находящихся в том же репозитории. Основное отличие внутренних зависимостей от внешних состоит в том, что они собираются из исходного кода, а не загружаются в виде предварительно собранных артефактов.

Поэтому внутренние цели не поддерживают понятия «версии» — цель и все ее внутренние зависимости всегда собираются из одной и той же фиксации (исправления) в репозитории.

Единственная сложность внутренних зависимостей связана с обработкой *транзитивных (промежуточных) зависимостей* (рис. 18.5). Предположим, что цель *A* зависит от цели *B*, которая зависит от цели *C*, представляющей общую библиотеку. Должна ли цель *A* иметь возможность использовать классы, которые определены в цели *C*?



Рис. 18.5. Транзитивные зависимости

Что касается основных инструментов, здесь нет никаких сложностей; обе цели — *B* и *C* — будут связаны с целью *A* во время сборки, поэтому любые символы, объявленные в *C*, доступны *A*. Система Blaze позволяла это в течение многих лет, но по мере расширения Google мы стали замечать проблемы этого подхода. Предположим, что код цели *B* был реорганизован так, что ему стала не нужна зависимость от *C*. Если теперь удалить зависимость *B* от *C*, то работоспособность *A* и любой другой цели, использовавшей *C* через зависимость от *B*, нарушится. Фактически зависимости цели стали частью ее публичного контракта и появился риск нарушить работоспособность кода при их изменении. По этой причине зависимости стали накапливаться со временем и процессы сборки в Google стали замедляться.

Чтобы решить эту проблему, мы реализовали в Blaze «режим строгой транзитивной зависимости». В этом режиме Blaze определяет, пытается ли цель сослаться на символ, отсутствующий в ее непосредственных зависимостях, и если да, то завершается неудачей с сообщением об ошибке и примером команды, которую можно выполнить, чтобы автоматически добавить необходимые зависимости. Внедрение этого новшества в кодовую базу Google и рефакторинг каждой из миллионов наших целей сборки для явного перечисления их зависимостей потребовало многолетних усилий, но оно того стоило. Теперь сборка проектов у нас выполняется намного быстрее, потому что цели имеют меньше ненужных зависимостей¹, а инженеры могут удалять ненужные зависимости, не беспокоясь о нарушении работоспособности целей, которые от них зависят.

Как обычно, введение более строгого правила потребовало пойти на компромисс: файлы сборки стали более подробными, потому что часто используемые библиотеки теперь нужно указывать явно во многих местах, а инженеры должны прилагать больше усилий для добавления зависимостей в файлы BUILD. Мы разработали ин-

¹ На самом деле удаление этих зависимостей проводилось в рамках отдельного процесса. Но требование явного объявления всех необходимых зависимостей стало важным первым шагом на пути к новому подходу. Подробнее о проведении крупномасштабных изменений в Google в главе 22.

струменты, облегчающие этот труд. Они автоматически обнаруживают недостающие зависимости и добавляют их в файлы BUILD без участия разработчика. Но даже если бы у нас не было таких инструментов, этот шаг полностью оправдал себя с увеличением масштаба кодовой базы, поскольку явное добавление зависимости в файл BUILD — это одновременные затраты, а использование неявных транзитивных зависимостей может вызывать проблемы постоянно, пока существует цель сборки. Строгие правила в отношении транзитивных зависимостей (<https://oreil.ly/Z-CqD>) по умолчанию применяются системой Bazel к коду на Java.

Внешние зависимости

Внешние зависимости — это артефакты, которые создаются и хранятся за пределами системы сборки. Они импортируются непосредственно из *репозитория артефактов* (обычно доступного через интернет) и используются как есть, а не создаются из исходного кода. Одно из самых больших отличий внешних и внутренних зависимостей заключается в наличии версии у внешних зависимостей, которые существуют независимо от исходного кода проекта.

Автоматическое и ручное управление зависимостями. Системы сборки позволяют управлять версиями внешних зависимостей вручную или автоматически. При ручном управлении в файле сборки явно указывается номер версии, которая должна загружаться из репозитория артефактов, часто в виде семантической строки версии (<https://semver.org>), такой как «1.1.4». При автоматическом управлении в исходном файле указывается диапазон допустимых версий, а система сборки всегда будет пытаться загрузить самую последнюю версию. Например, Gradle позволяет объявить версию зависимости как «1.+», чтобы указать, что допускаются любые версия или исправление зависимости с основным номером версии 1.

Автоматическое управление зависимостями удобно использовать в небольших проектах, но обычно такой подход приводит к катастрофе в проектах большого размера, над которыми работают несколько инженеров. Проблема автоматического управления зависимостями заключается в невозможности управлять обновлением версии. Нет гарантий, что третья сторона не выпустит важное обновление (даже если она утверждает, что использует семантическое управление версиями), поэтому сборка, благополучно выполняющаяся сегодня, может начать терпеть неудачу завтра, и будет нелегко определить причину сбоя и устранить ее. Даже если сборка выполняется успешно, новая версия может иметь незначительные изменения в поведении или производительности, которые невозможно отследить.

Ручное управление зависимостями, напротив, требует передачи изменений в VCS, чтобы их можно было легко обнаружить и откатить, а также позволяет извлечь из репозитория старую версию и выполнить сборку со старыми зависимостями. Bazel требует, чтобы версии всех зависимостей указывались вручную. Даже в базах кода умеренного масштаба накладные расходы на ручное управление версиями окупаются благодаря той стабильности, которую оно обеспечивает.

Правило единственной версии. Разные версии библиотеки обычно представлены разными артефактами, поэтому теоретически ничто не мешает присвоить разные имена разным версиям одной и той же внешней зависимости в системе сборки. При таком подходе каждая цель сможет выбрать желаемую версию зависимости. Но мы в Google обнаружили, что на практике это вызывает множество проблем, поэтому мы строго следуем *правилу единственной версии* (<https://oreil.ly/OFa9V>) для всех сторонних зависимостей в нашей внутренней кодовой базе.

Самая большая проблема поддержки нескольких версий — образование *ромбовидных* (diamond) *зависимостей*. Представьте, что цель *A* зависит от цели *B* и от версии *v1* внешней библиотеки. Если позже будет выполнен рефакторинг цели *B* для добавления зависимости от версии *v2* той же внешней библиотеки, то цель *A* может потерять работоспособность, потому что теперь она неявно зависит от двух разных версий одной и той же библиотеки. В целом всегда небезопасно добавлять в цель новую зависимость от любой сторонней библиотеки с несколькими версиями, потому что любой из пользователей этой цели уже может зависеть от другой версии. Следование правилу единственной версии запрещает зависимость от двух версий — если цель добавляет зависимость от сторонней библиотеки, то любые существующие зависимости уже будут в этой же версии и смогут благополучно сосуществовать.

К этой проблеме в контексте большого монолитного репозитория мы еще вернемся в главе 21.

Транзитивные внешние зависимости. Многие репозитории артефактов, такие как Maven Central, позволяют артефактам определять зависимости от конкретных версий других артефактов в репозитории. Инструменты сборки, такие как Maven или Gradle, часто рекурсивно загружают все транзитивные зависимости по умолчанию, а это означает, что добавление одной зависимости в проект может потребовать загрузки десятков артефактов.

Это очень удобно: при добавлении зависимости от новой библиотеки было бы сложно вручную выявить и добавить все транзитивные зависимости этой библиотеки. Но у этого подхода есть серьезный недостаток: поскольку разные библиотеки могут зависеть от разных версий одной и той же сторонней библиотеки, между ними возникнет ромбовидная зависимость. Если цель зависит от двух внешних библиотек, которые используют разные версии одной и той же зависимости, то неизвестно, какую из них вы получите. Это также означает, что обновление внешней зависимости может вызвать кажущиеся несвязанными ошибки во всей кодовой базе, если для сборки новой версии будут извлекаться конфликтующие версии некоторых из ее зависимостей.

По этой причине Bazel никогда не загружает транзитивные зависимости автоматически. К сожалению, эта проблема не имеет универсального решения — альтернативный подход, поддерживаемый системой Bazel, заключается в создании глобального файла, в котором перечислены все внешние зависимости репозитория и явно указано, какая версия каждой зависимости должна использоваться во всем репозитории. К счастью, в Bazel имеются инструменты (<https://oreil.ly/kejfX>), которые могут автоматически

создать такой файл, перечисляющий транзитивные зависимости для набора артефактов Maven. Этот инструмент можно запустить один раз, чтобы получить начальный файл `WORKSPACE` для проекта, а затем обновлять этот файл вручную, корректируя версии каждой зависимости.

И снова приходится выбирать между удобством и масштабируемостью. Небольшие проекты могут добавлять внешние транзитивные зависимости автоматически. В более крупных масштабах стоимость ручного управления зависимостями намного ниже стоимости решения проблем, вызванных автоматическим управлением зависимостями.

Кеширование результатов сборки с использованием внешних зависимостей. Внешние зависимости чаще всего предоставляются третьими сторонами, которые выпускают стабильные версии библиотек, возможно, без предоставления исходного кода. Некоторые организации могут также сделать доступной часть своего кода в виде артефактов, позволяя другим частям кода зависеть от них. Теоретически это может ускорить сборку, если артефакты медленно создаются, но быстро загружаются.

Однако это также приводит к большим накладным расходам и увеличению сложности: кто-то должен нести ответственность за создание каждого из артефактов и их загрузку в репозиторий артефактов, а клиенты должны следить за тем, чтобы всегда оставаться в курсе последних версий. Отладка тоже становится намного сложнее, потому что разные части системы будут собираться из разных точек в репозитории и больше не будет существовать согласованного представления дерева исходных текстов.

Лучший способ решить проблему артефактов, сборка которых занимает много времени, — использовать систему сборки, поддерживающую удаленное кеширование, как было описано выше. Такая система сборки будет сохранять полученные артефакты в одном месте, совместно используемом инженерами, поэтому если разработчик зависит от артефакта, который недавно был создан кем-то другим, то система сборки автоматически загрузит его, минуя этап сборки из исходного кода. Этот подход обеспечивает все преимущества производительности, связанные с прямой зависимостью от артефактов, и гарантирует, что результаты сборки будут оставаться такими же согласованными, как если бы они всегда собирались из одного источника. Эта стратегия используется в Google, и система Bazel предоставляет возможность построить удаленный кеш.

Безопасность и надежность внешних зависимостей. Зависимость от артефактов, полученных из сторонних источников, по своей природе рискованна. Если сторонний источник (например, репозиторий артефактов) выйдет из строя, его артефакт окажется недоступным и сборка проекта может остановиться. Также, если сторонняя система будет скомпрометирована злоумышленником, этот злоумышленник сможет подменить артефакт своей версией и внедрить произвольный код в вашу сборку. Чтобы смягчить обе проблемы, создайте свое зеркало артефактов, от которых вы зависите, на контролируемых вами серверах и запретите своей системе сборки об-

ращаться к сторонним репозиториям артефактов, таким как Maven Central. Однако для обслуживания этих зеркал требуются силы и ресурсы, поэтому выбор такого подхода часто зависит от масштаба проекта. Проблему безопасности также можно полностью устранить, потребовав указать хеш для каждого стороннего артефакта в исходном репозитории, что вызовет остановку сборки, если кто-то попытается подделать артефакт.

Другая альтернатива, полностью устраняющая эти проблемы, — *официальная поставка* зависимостей для вашего проекта. Когда сторонний проект официально предоставляет свои зависимости, он извлекает их из VCS вместе с исходным кодом проекта либо в виде исходного кода, либо в виде двоичных файлов. Фактически это означает преобразование всех внешних зависимостей проекта во внутренние. Google использует этот подход для внутренних нужд, сохраняя каждую стороннюю библиотеку, на которую есть ссылки в Google, в каталог `third_party` в корне дерева исходных кодов. Однако в Google этот подход работает только потому, что VCS Google специально создана для работы с чрезвычайно большим монолитным репозиторием, поэтому официальная поставка может быть недоступна для других организаций.

Заключение

Система сборки — одна из самых важных частей инженерной организации. Каждый разработчик взаимодействует с ней десятки и сотни раз в день, и часто это может оказаться ограничивающим фактором для его продуктивности. Поэтому качество работы системы сборки заслуживает внимания.

Один из самых удивительных уроков, извлеченных в Google, заключается в том, что *ограничение возможностей и гибкости может повысить производительность инженеров*. Мы смогли создать систему сборки, которая отвечает нашим потребностям. Она не дает инженерам свободу определять, как выполнять сборку, но предлагает четко структурированную среду, которая выполняет наиболее интересные решения с помощью автоматизированных инструментов. И как ни странно, инженеров это не возмущает: гуглерам нравится, что эта система работает почти сама по себе и позволяет им сосредоточиться на более интересных аспектах разработки приложений. Доверие к системе сборки позволяет легко выполнять инкрементальные сборки без очистки кешей сборки или применения `clean`.

Опираясь на полученный опыт, мы создали систему сборки совершенно нового типа — *на основе артефактов*, — существенно отличающуюся от традиционных систем сборки *на основе задач*. Переосмысление подходов к сборке позволило масштабировать ее процессы до размеров Google. Новые подходы позволяют создать *распределенную систему сборки*, которая может использовать ресурсы всего вычислительного кластера для повышения производительности инженеров. Конечно, ваша организация может быть не настолько большой, чтобы извлечь выгоду из подобных инвестиций, но мы считаем, что системы сборки на основе артефактов могут масштабироваться не только вверх, но и вниз: даже небольшие проекты системы сборки, такие как

Vazel, могут дать значительные преимущества с точки зрения скорости выполнения и безошибочности.

В этой главе мы также рассмотрели особенности управления зависимостями в мире, основанном на артефактах, и пришли к выводу, что *дробление кода на мелкие модули масштабируется лучше*. Мы также обсудили трудности управления версиями зависимостей и правило единственной версии и отметили, что для всех зависимостей следует *явно и вручную определять их версии*. Эти приемы помогают миновать известные ловушки, такие как проблема ромбовидной зависимости, и работать с базами кода масштаба Google, насчитывающими миллиарды строк кода в едином репозитории с единой системой сборки.

Итоги

- Для продуктивной работы разработчиков с ростом организации необходима полноценная система сборки.
- Мощность и гибкость системы сборки имеют свою цену. Выбор правильных ограничений для системы сборки упрощает ее использование.
- Системы сборки, организованные вокруг артефактов, обычно лучше масштабируются и более надежны, чем системы сборки, организованные вокруг задач.
- При определении артефактов и зависимостей лучше стремиться разбивать код на более мелкие модули, потому что при этом полнее используются преимущества параллельной и инкрементальной сборки.
- Версии внешних зависимостей должны явно указываться в VCS. Использование «последних» версий приводит к катастрофическим и невозпроизводимым сборкам.

Critique: инструмент обзора кода в Google

*Авторы: Кейтлин Садовски, Ильхам Курния и Бен Рольфс
Редактор: Лиза Кэри*

Как мы уже писали в главе 9, обзор кода является жизненно важной частью разработки ПО, особенно в больших масштабах. Основная цель обзора кода — улучшить удобочитаемость и простоту поддержки кодовой базы, и достижение этой цели обеспечивается процессом обзора и инструментами, поддерживающими этот процесс.

В этой главе мы посмотрим, как в Google проводится обзор кода с использованием собственного инструмента *Critique*. Он явно поддерживает основные цели рецензирования, давая рецензентам и авторам возможность исследовать и комментировать изменения. Также *Critique* дает возможность фильтровать код, поступающий в кодовую базу (см. раздел, посвященный «оценке» изменений). Информация из *Critique* используется при поиске причин технических решений по диалогам в обзорах кода (например, когда отсутствуют встроенные комментарии). *Critique* — не единственный, но самый популярный инструмент обзора кода в Google.

Принципы оснащения обзора кода инструментами

Выше упоминалось, что *Critique* предоставляет необходимые функции поддержки целей обзора кода (мы рассмотрим их далее в этой главе), но почему этот инструмент пользуется таким успехом? *Critique* сформирован культурой разработки в Google, в которой обзор кода является неотъемлемой частью рабочего процесса. Это культурное влияние выражается в наборе руководящих принципов, которые *Critique* подчеркивает:

Простота

Пользовательский интерфейс *Critique* создавался с целью упростить обзоры кода. Он не имеет лишних элементов управления, работает очень гладко, быстро загружается, имеет простую навигацию, поддерживает горячие клавиши, а также четкие визуальные индикаторы, сообщающие, было ли проверено то или иное изменение.

Доверие

Обзоры кода должны не замедлять работу инженеров, а расширять их возможности. Они основываются на максимальной доверии к коллегам, например к авторам изменений. Расширению доверия способствует также общедоступность (для просмотра и проверки) изменений в Google.

Универсальная коммуникация

Сложности с коммуникацией редко решаются с помощью инструментов. Critique отдает приоритет универсальным способам комментирования изменений в коде, а не сложным протоколам. Вместо того чтобы усложнять модель данных и процесс, Critique поощряет пользователей пояснять в своих комментариях, чего они хотят, и предлагает правки к комментариям. Но даже с использованием лучшего инструмента обзора кода коммуникация может не получаться, потому что пользователи — это люди.

Интеграция в рабочий процесс

Critique имеет ряд точек интеграции с другими инструментами разработки ПО. Разработчики с легкостью могут переходить от просмотра проверяемого кода в Code Search к правке в веб-инструменте редактирования кода или к просмотру результатов тестирования изменений.

Из всех этих руководящих принципов наибольшее влияние на инструмент оказала простота. Изначально мы думали о добавлении множества интересных возможностей, но потом решили не усложнять модель, чтобы обеспечить поддержку максимально широкого круга пользователей.

Простота также имеет интересное противоречие с интеграцией Critique в рабочий процесс. Мы рассматривали возможность создания единого инструмента Code Central для правки, обзора и поиска кода, но потом отказались от этой идеи. Critique имеет множество точек соприкосновения с другими инструментами, но мы сознательно решили нацелить его на обзор кода. Многие возможности, доступные в Critique, реализованы в разных подсистемах.

Процесс обзора кода

Обзоры кода могут выполняться на разных этапах разработки ПО (рис. 19.1). Обычно они проводятся до передачи изменений в кодовую базу, поэтому их называют *обзорами перед передачей в репозиторий*. Краткое описание процесса проверки кода уже приводилось в главе 9, но мы повторим его здесь, дополнив некоторыми ключевыми аспектами Critique. В следующих разделах мы подробно рассмотрим каждый этап.

Вот типичные этапы обзора кода:

1. **Добавление изменений.** Пользователь вносит изменение в код в своем рабочем пространстве. Затем *автор* выгружает *снимок* (отражающий изменения в определенный момент) в Critique, который запускает инструменты автоматического анализа кода (глава 20).

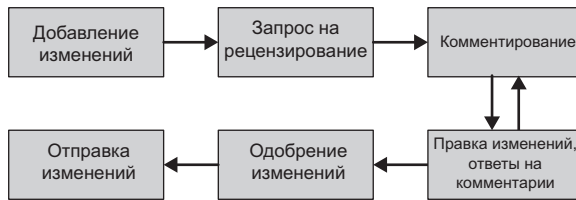


Рис. 19.1. Процесс обзора кода

2. **Запрос на рецензирование.** Закончив вносить изменения и удовлетворившись результатами автоматического анализа в Critique, автор отправляет изменение одному или нескольким рецензентам по электронной почте.
3. **Комментирование.** Рецензенты открывают изменение в Critique и добавляют свои комментарии. По умолчанию комментарии отмечаются как *нерешенные*, то есть требующие реакции автора. Также рецензенты могут добавлять *решенные* комментарии, которые не требуют реакции автора и имеют исключительно информационный характер. Результаты автоматического анализа кода, если они есть, также доступны рецензентам. После того как рецензент подготовит набор комментариев, он *публикует* их для автора, выражая свою оценку изменения после его обзора. Прокomentировать изменение может любой желающий в «попутном обзоре».
4. **Правка изменений, ответы на комментарии.** Основываясь на комментариях, автор вносит новые правки в изменение, выгружает новые снимки и передает обновленный вариант изменения рецензентам. Автор реагирует (по меньшей мере) на все нерешенные комментарии, либо изменяя код, либо просто отвечая на комментарий и отмечая комментарий как *решенный*. Автор и рецензенты могут посмотреть различия между любыми парами снимков, чтобы увидеть, что изменилось. Шаги 3 и 4 могут повторяться несколько раз.
5. **Одобрение изменений.** Удовлетворившись последним состоянием изменения, рецензенты одобряют его и отмечают как «мне нравится» (LGTM). При желании они могут добавить комментарии, требующие решения. После признания изменения пригодным для отправки в репозиторий оно будет отмечено зеленым цветом в пользовательском интерфейсе.
6. **Отправка изменений.** Если изменение одобрено (как обсуждается ниже), автор может запустить процесс его фиксации. Если инструменты автоматического анализа и другие средства оценки перед отправкой не обнаружили никаких проблем, изменение фиксируется в кодовой базе.

Даже после запуска процесса система позволяет отклониться от типичной процедуры обзора. Например, рецензенты могут отказаться от оценки изменений или явно передать эту роль кому-то другому, а автор может вообще приостановить процесс обзора. В экстренных случаях автор может принудительно зафиксировать свое изменение в репозитории и запросить его обзор после фиксации.

Уведомления

По мере продвижения изменения через этапы, описанные выше, Critique публикует уведомления о событиях, которые могут использоваться другими вспомогательными инструментами. Модель уведомлений позволяет инструменту Critique оставаться основным средством обзора кода, интегрированным в рабочий процесс разработчика, и не превращаться в многоцелевой инструмент. Уведомления позволяют разделить задачи так, чтобы Critique мог просто посылать события, а другие системы обслуживали их.

Например, пользователи могут установить расширение для Chrome, чтобы подписаться на эти уведомления. Когда появляется изменение, требующее внимания пользователя, например если настала очередь пользователя оценить изменение или предварительная проверка перед отправкой потерпела неудачу, расширение выводит на экран уведомление с кнопкой для перехода непосредственно к изменению или для отключения уведомления. Мы заметили, что одним разработчикам нравятся немедленные уведомления об обновлении изменений, а другие не пользуются этим расширением, потому что оно мешает работе.

Critique может рассылать уведомления по электронной почте. Например, таким способом рассылаются уведомления о наиболее важных событиях. Некоторые инструменты автоматического анализа тоже имеют настройки для пересылки результатов по электронной почте дополнительно к их отображению в пользовательском интерфейсе Critique. Кроме того, Critique может обрабатывать ответы по электронной почте и переводить их в комментарии для тех пользователей, которые предпочитают электронную почту. Обратите внимание, что для многих пользователей электронные письма не являются ключевой функцией обзора кода и для управления отзывами они используют панель инструментов Critique (подробнее ниже).

Этап 1: добавление изменений

Инструмент обзора кода должен обеспечивать поддержку на всех этапах процесса рецензирования и не задерживать фиксацию изменений. На этапе предварительного обзора авторам проще отшлифовать свое изменение перед отправкой на проверку, что помогает сократить время рецензирования. Critique позволяет отображать только значимые различия и игнорировать изменение количества пробелов. Также Critique отображает результаты сборки, тестирования и статического анализа, включая проверку стиля (глава 9).

Critique позволяет автору видеть свои изменения так, как их видят рецензент и автоматический анализ, поддерживает возможность корректировки изменения прямо в инструменте обзора и предлагает подходящих рецензентов. Отправляя запрос, автор может добавить предварительные комментарии к изменению, что дает возможность напрямую задать рецензентам любые вопросы. Все это предотвращает недопонимание.

Чтобы предоставить рецензентам дополнительную информацию, автор может связать изменение с конкретной ошибкой. Critique использует службу автодополнения, чтобы подсказать соответствующие ошибки, отдавая приоритет ошибкам, устранение которых поручено автору.

Представление различий

Главная задача обзора кода — упростить понимание изменения. Крупные изменения обычно труднее понять, чем мелкие. Поэтому представление различий в коде, обусловленных изменением, в наиболее простом и понятном виде является основным требованием для хорошего инструмента обзора кода.

В Critique этот принцип распространяется на несколько уровней (рис. 19.2). Компонент представления различий, начиная с алгоритма оптимизации самой длинной общей подпоследовательности, дополнен следующими возможностями:

- подсветка синтаксиса;
- поддержка перекрестных ссылок (с помощью Kythe, глава 17);
- отображение различий внутри строк, показывающее разницу на уровне символов (рис. 19.2);
- настройки, позволяющие игнорировать различия в количестве пробелов;
- фрагменты кода, перемещенные из одного места в другое, отмечаются как перемещенные, а не удаленные в одном месте и добавленные в другом, как это делают многие алгоритмы сравнения.

```

22 @NgModule({
23   imports: [
24     AnalysisModule,
25     CommaSeparatedModule,
26     CommonModule,
27     DateModule,
28     LinkifyModule,
29     LinkifiedListModule,
30     MatButtonModule,
31     MatCheckboxModule,
32     MatDialogModule,
33     MatDividerModule,
34     MatIconModule,
35     MatInputModule,
36     MatMenuModule,
37     PopupsModule,
38     ScorePanelModule,
39     UtilModule,
40     UserModule,
41 ],
42 declarations: [
43   AnalysisChip,

```

```

26 @NgModule({
27   imports: [
28     AnalysisModule, CommaSeparatedModule, CommonModule, DateModule,
29     MatTabsModule, LinkifyModule, LinkifiedListModule, MatButtonModule,
30     MatCheckboxModule, MatDialogModule, MatDividerModule, MatIconModule,
31     MatInputModule, MatTabsModule, MatMenuModule, PopupsModule,
32     RouterModule, ScorePanelModule, UtilModule, UserModule,
33 ],
34 declarations: [
35   AnalysisChip,

```

Рис. 19.2. Отображение различий внутри строк показывает разницу на уровне символов

Пользователи также могут просматривать различия в разных режимах, например в режиме наложения или рядом друг с другом. Создавая Critique, мы решили, что важно показать различия рядом друг с другом, чтобы упростить процесс обзора. Такое расположение различий занимает много места, и мы упростили структуру представления различий, отказавшись от границ и отступов и оставив только сами различия и номера строк. Нам пришлось поэкспериментировать с разными шрифтами и кеглями, пока мы не добились представления различий, которое учитывает ограничение размеров строк в 100 символов в Java и типичную ширину экрана в Critique — 1440 пикселей.

Critique также поддерживает дополнительные инструменты, представляющие различия в артефактах, созданных в результате изменения, например снимки пользовательского интерфейса или файлы конфигурации, полученные в результате изменения.

Чтобы упростить процесс навигации по различиям, мы постарались максимально сэкономить пространство и обеспечить быструю загрузку различий, даже таких, как изображения и большие файлы. Мы также определили комбинации клавиш для быстрой навигации по файлам при обзоре измененных разделов.

Когда пользователи переходят на уровень файлов, Critique предоставляет виджет с компактным отображением цепочки версий снимков, с помощью которого пользователи могут выбирать интересующие их версии для сравнения. Этот виджет автоматически сворачивает похожие версии, помогая сосредоточить внимание на наиболее важных из них. Благодаря этому пользователь может видеть, как развивался файл в ходе изменения, например какие части изменения охвачены тестами, а какие части уже проверены или содержат комментарии. Чтобы решить проблему масштаба, Critique заранее выбирает все необходимое, поэтому загрузка различных снимков выполняется очень быстро.

Анализ результатов

Выгрузка снимка приводит к запуску автоматических инструментов анализа кода (глава 20). Результаты и обобщенные итоги анализа отображаются в Critique на странице изменения (рис. 19.3). Более подробные детали отображаются во вкладке Analysis (Анализ) (рис. 19.4).

The screenshot displays the Critique interface for a change titled "Change 243497582 by illiam". The change is in a "Pending" state. The main content area shows a description: "Implement pizza supplier (1/6). Add a skeleton for the pizza supplier system. We follow the organic framework for establishing the connection between the basic ingredients to the supplier." Below this, there is a "Score" section indicating "LGTM - Missing Approvals coverage - No approvals necessary" and an "Analysis" section with an actionable item "PresubmitCheckProtoSyntax".

A table lists the files added in the change:

File	Comments	Inline	Modified	Delta
pizza/BUILD Added		Diff	3:05 PM	7
pizza/PizzaSupplierApp.java Added		Diff	3:06 PM	22
pizza/pizza.proto Added		Hide	3:05 PM	28

Below the table, a summary box highlights a "PresubmitCheckProtoSyntax" error at 3:06 PM, stating: "Your change contains one or more new .proto files without a syntax statement. Each .proto file must have a syntax statement in order to submit." The error is marked as "Not useful". A code snippet is shown below the error:

```
package google.pizza;
import google.Timestamp;
message Ingredient {
  required int64 id = 1;
  required string name = 2;
  required int64 unit = 3;
  optional Timestamp season = 4;
```

Рис. 19.3. Обобщенные итоги анализа и представление различий

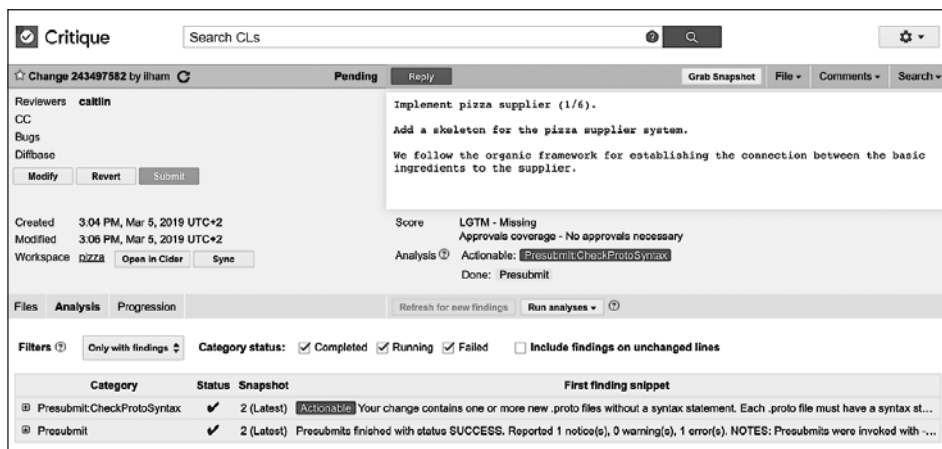


Рис. 19.4. Результаты анализа

Инструменты анализа могут подсвечивать конкретные результаты красным цветом. Еще не завершенные виды анализа выделяются желтыми значками, а остальные — серыми. Для простоты Critique не предлагает других вариантов выделения. Если тот или иной инструмент возвращает какие-то результаты, их можно открыть щелчком мыши на значке. Подобно комментариям, результаты анализа могут отображаться внутри различий, но оформляются иначе, чтобы их было проще отличить. Иногда результаты анализа включают предложения по исправлению замечаний, которые автор может просмотреть и применить прямо в Critique.

Предположим, что инструмент статического анализа обнаруживает нарушение стиля — лишние пробелы в конце строки. На странице изменения появится значок этого инструмента. Щелкнув на нем, автор может быстро перейти к различиям, показывающим код, и парой щелчков мыши исправить нарушение. Большинство замечаний, возвращаемых инструментом статического анализа, также включают предложения по исправлению. Одним щелчком автор может просмотреть предлагаемое исправление (например, предложение удалить лишние пробелы), а вторым — применить его.

Тесная интеграция инструментов

В Google есть инструменты, созданные на основе Piper — монолитного репозитория исходного кода (глава 16), например:

- Cider — онлайн-IDE для редактирования исходного кода, хранящегося в облаке;
- Code Search — инструмент для поиска в кодовой базе;
- Tricorder — инструмент для отображения результатов статического анализа (упоминался выше);

- Rapid — инструмент для выпуска новых версий, который упаковывает и развертывает двоичные файлы, содержащие серии изменений;
- Zapfahn — инструмент вычисления доли кода, охваченной тестами.

Кроме того, существуют службы, возвращающие информацию об изменениях (например, имена пользователей — авторов изменений или ошибки, для устранения которых создавались изменения). Critique идеально подходит для быстрого доступа к этим системам и организации пользовательского интерфейса к ним, однако мы не используем его для этого, чтобы не жертвовать простотой. Например, на странице изменения в Critique автору достаточно щелкнуть только один раз, чтобы начать редактирование изменения в Cider. Существует также поддержка навигации по перекрестным ссылкам Kythe и просмотра кода в главной ветви с помощью Code Search (глава 17). Critique поддерживает связь с инструментом, показывающим, относится ли отправленное изменение к конкретной версии. В Critique отдается предпочтение ссылкам, а не встроенным операциям, чтобы не отвлекать пользователей от основного процесса обзора кода. Единственным исключением является оценка охвата тестами: информация об охвате строк кода тестами отображается разными цветами фона на полях в представлении различий (не все проекты используют отдельный инструмент для оценки охвата тестирования).

Обратите внимание, что тесная интеграция Critique с рабочим пространством разработчика во многом возможна благодаря хранению рабочих пространств в распределенной файловой системе FUSE, расположенной за пределами компьютера конкретного разработчика. Источник истины находится в облаке и доступен для всех инструментов.

Этап 2: запрос на рецензирование

После того как автор доведет изменение до желаемого состояния, он может передать его для обзора (рис. 19.5). Найти рецензента в небольшой команде может показаться простой задачей, но даже в таком случае желательно равномерно распределить рецензирование между членами команды и учитывать такие ситуации, когда кто-то находится в отпуске. Для этого можно создать отдельный электронный адрес, куда будут направляться все запросы на рецензирование, после чего инструмент *GwsQ* (названный в честь команды Google Web Server, первой предложившей этот прием) на основе своих настроек выберет рецензентов из списка для выполнения обзора.

Учитывая размер кодовой базы в Google и количество инженеров, постоянно изменяющих ее, иногда трудно определить, кто вне вашего проекта лучше подходит для обзора изменений. Поиск рецензентов — сложная задача, которую придется решать, достигнув определенного масштаба кодовой базы. Critique должен поддерживать работу в масштабе и обладает функциональными возможностями для выбора рецензентов, обладающих достаточными полномочиями для одобрения изменения. Функция выбора рецензентов учитывает следующие факторы:

- кому принадлежит изменяемый код (см. следующий раздел);
- кто лучше всего знаком с кодом (то есть кто недавно его изменял);
- кто может заняться обзором (то есть находится на рабочем месте и желательно в том же часовом поясе);
- настройки GwsQ в команде.

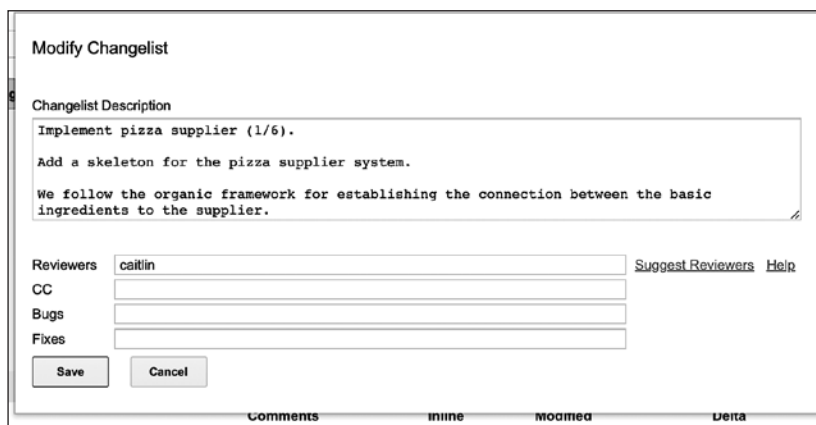


Рис. 19.5. Отправка запроса на рецензирование

Выбор рецензента для обзора изменения инициирует отправку запроса на рецензирование. Этот запрос попадает в обработчик «предварительной проверки», подходящий для изменения: команды могут по-разному настраивать обработчики, связанные с их проектами. Чаще всего используются обработчики:

- автоматически добавляющие изменение в списки рассылки для повышения осведомленности и прозрачности;
- выполняющие автоматизированные наборы тестов для проекта;
- добавляющие постоянные элементы в код (для соблюдения локальных ограничений оформления кода) и в описание изменения (для создания примечаний к версии и добавления другой информации).

Поскольку для тестирования требуются значительные ресурсы, в Google тесты выполняются в ходе предварительной проверки (запускаются при отправке запроса на рецензирование и при фиксации изменений в репозитории), а не для каждого снимка, как, например, анализ с помощью Tricorder. Critique отображает результаты выполнения обработчиков, подобно результатам инструментов анализа, но с важным отличием — неудачный результат блокирует отправку изменения на обзор или его фиксацию. Critique уведомляет автора по электронной почте, если предварительная отправка изменения не удалась.

Этапы 3 и 4: исследование и комментирование изменения

После начала процесса обзора автор и рецензенты вместе работают над достижением общей цели — повысить качество кода изменения и зафиксировать его в репозитории.

Комментирование

Комментирование — второе по распространенности действие, которое пользователи выполняют в Critique после просмотра изменений (рис. 19.6). Комментирование в Critique открыто для всех. Оставить свой комментарий может кто угодно, а не только автор изменения и назначенные рецензенты.

Также Critique дает возможность следить за тем, как протекает процесс обзора. Рецензенты могут устанавливать флажки, отмечая отдельные файлы в последнем снимке как проверенные. Когда автор изменяет файл, флажки «проверен», установленные всеми рецензентами, снимаются с этого файла.

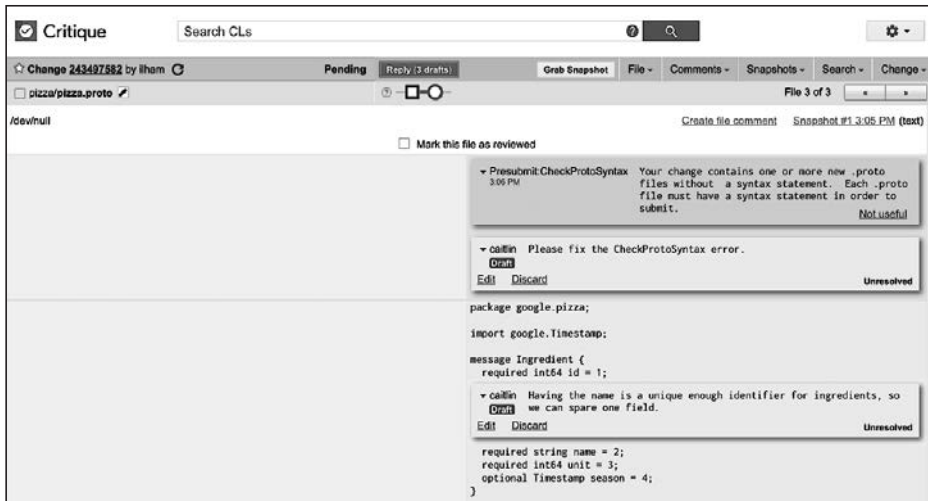


Рис. 19.6. Комментирование в представлении различий

Увидев результат анализа, рецензент может щелкнуть на кнопке **Please fix** (Исправьте), чтобы создать нерешенный комментарий с просьбой к автору отреагировать на замечание. Рецензенты также могут предложить свое исправление, напрямую отредактировав последнюю версию файла. Critique превратит это действие в комментарий с исправлением, которое автор сможет применить к коду.

Critique не накладывает ограничений на комментарии, создаваемые пользователями, но предоставляет ярлыки для быстрого ответа на некоторые общие комментарии. Автор изменения может щелкнуть на кнопке **Done** (Готово) в панели комментариев, чтобы сообщить о своей реакции на комментарий рецензента, или на кнопке **Ack** (Подтвердить), чтобы подтвердить, что комментарий был прочитан, — так обычно поступают с информационными комментариями или комментариями, не требующими выполнения каких-либо действий. В обоих случаях цепочки комментариев отмечаются как решенные. Эти кнопки упрощают рабочий процесс и сокращают время, необходимое для ответа на комментарии рецензентов.

Как упоминалось выше, комментарии создаются по мере исследования изменения, а затем «публикуются» вместе (рис. 19.7). Это позволяет авторам и рецензентам проверить правильность своих комментариев перед их отправкой.



Рис. 19.7. Подготовка комментариев для автора

Выяснение состояния изменения

В Critique есть ряд механизмов, позволяющих понять, на каком этапе в процессе «комментировать и повторить» находится изменение в данный момент. К ним относится возможность определения, кто должен сделать следующий шаг, а также панель мониторинга для всех изменений, в рецензировании или разработке которых принимает участие конкретный разработчик.

Возможность определения «чей ход»

Одним из факторов ускорения процесса обзора является информирование о том, кому принадлежит следующий ход, особенно когда обзор изменения назначен нескольким рецензентам. Такое случается, когда автор желает, чтобы его изменение было рассмотрено инженером-программистом и лицом, отвечающим за эргономику пользовательского интерфейса или надежность службы. Critique помогает определить, кто следующим должен рассмотреть изменение, поддерживая *набор внимания* (attention set) для каждого изменения.

Набор внимания определяет группу людей, которые занимаются исследованием изменения в данный момент. Когда рецензент или автор находится в наборе внимания, ожидается, что он ответит в ближайшее время. Critique старается своевременно обновлять набор внимания, когда пользователь публикует свои комментарии, но пользователи тоже могут управлять набором внимания. Его полезность возрастает с увеличением количества рецензентов изменения. Набор внимания отображается в Critique с помощью выделения имен пользователей жирным шрифтом.

После реализации этой возможности наши пользователи недоумевали: как они обходились без нее раньше? До внедрения этой функции авторам и рецензентам приходилось напрямую общаться друг с другом, чтобы выяснить, кто следующий работает с изменением. Эта функция также подчеркивает поэтапный характер обзора кода, в котором право действовать всегда принадлежит хотя бы одному человеку.

Панель мониторинга и поисковая система

На главной странице пользователя Critique отображается панель мониторинга (рис. 19.8). Она состоит из разделов, настраиваемых пользователем, каждый из которых содержит сводную информацию об изменениях.

Страница с панелью мониторинга использует поисковую систему *Changelist Search*, которая индексирует последнее состояние всех доступных изменений (до и после отправки в репозиторий) для всех пользователей в Google и позволяет искать изменения с помощью запросов на основе регулярных выражений. Каждый раздел в панели мониторинга определяется запросом к системе Changelist Search. Мы потратили немало времени, чтобы сделать поиск в Changelist Search достаточно быстрым для интерактивного использования. Теперь индексирование в ней выполняется очень быстро, чтобы авторы и рецензенты не тратили время на ожидание, несмотря на то что в Google одновременно создается огромное количество изменений.

Для удобства в первом разделе панели мониторинга Critique по умолчанию отображаются изменения, требующие внимания пользователя, но это условие можно настроить. Также в ней есть панель для поиска по всем изменениям и просмотра результатов. Рецензентам обычно нужен только набор внимания, а авторам — список их изменений, ожидающих обзора, чтобы узнать, не пора ли проверить комментарии. Мы старались не давать инженерам слишком широких возможностей в настройке пользовательского интерфейса Critique, но обнаружили, что пользователям так же

нравится настраивать информационные панели по-своему, как и организовывать свою электронную почту¹.

Critique							
Search CLs							
Dashboard Weekly snippets Starred CLs Refresh Switch user Plain text							
Needs attention 4 Changes							
Change	Author	Status	Last Action	Reviewers	Size	Description	
42972248	ilham	Pending	Apr 11 by gwsq	caitlin	XS	Implement pizza supplier (6/8).	
42974683	ilham	Pending	Apr 11 by tap	caitlin	S	Implement pizza supplier (5/8).	
37099895	ilham	Pending	Apr 11 by ilham	caitlin	M	Implement pizza supplier (4/8).	
27761071	caitlin	Pending	Jan 8 by ilham	ilham	XS	Implement pizza maker (3/3).	
Incoming reviews 6 Changes							
Change	Author	Status	Last Action	Reviewers	Size	Description	
42972248	ilham	Pending	Apr 11 by gwsq	caitlin	XS	Implement pizza supplier (6/8).	
42974683	ilham	Pending	Apr 11 by tap	caitlin	S	Implement pizza supplier (5/8).	
37099895	ilham	Pending	Apr 11 by ilham	caitlin	M	Implement pizza supplier (4/8).	
42161351	ilham	LGTM	Apr 9 by caitlin	caitlin	XS	Implement pizza supplier (3/8).	
40374250	ilham	Unresolved	Apr 4 by caitlin	caitlin	XS	Implement pizza supplier (2/6).	
36387832	ilham	Unresolved	Mar 5 by caitlin	caitlin	L	Implement pizza supplier (1/6).	
Outgoing reviews 3 Changes							
Change	Author	Status	Last Action	Reviewers	Size	Description	
27761071	caitlin	Pending	Jan 8 by caitlin	ilham	XS	Implement pizza maker (3/3).	
15068925	caitlin	Pending	Jan 6 by caitlin	ilham	S	Implement pizza maker (2/3).	
15416497	caitlin	Pending	Jan 2 by caitlin	ilham	M	Implement pizza maker (1/3).	

Рис. 19.8. Панель мониторинга

Этап 5: одобрение изменений (оценка изменений)

Чтобы выразить свое мнение об изменении, рецензент должен оставить свои комментарии и предложения. Также должен быть какой-то механизм для выражения общего одобрения изменения. В Google он складывается из трех слагаемых:

- отметка LGTM;
- одобрение;
- отсутствие нерешенных комментариев.

Отметка LGTM, выставленная рецензентом, означает: «Я осмотрел это изменение и считаю, что оно соответствует нашим стандартам и его можно зафиксировать после разрешения нерешенных комментариев». Одобрение означает: «Как цензор я решаю зафиксировать это изменение в кодовой базе». Рецензент может добавить комментарии, отмеченные как нерешенные, чтобы потребовать от автора предпри-

¹ Централизованные «глобальные» рецензенты крупномасштабных изменений склонны настраивать эту информационную панель, чтобы избежать ее переполнения во время обзора таких изменений (глава 22).

нять какие-то действия. Если у изменения имеется хотя бы одна отметка LGTM, достаточное количество одобрений и отсутствуют нерешенные комментарии, автор сможет зафиксировать это изменение в репозитории. Обратите внимание, что для каждого изменения требуется хотя бы одна отметка LGTM, независимо от наличия одобрений, которая гарантирует, что как минимум две пары глаз рассмотрели изменение. Это простое правило оценки позволяет Critique уведомить автора, когда изменение будет готово к фиксации (отображается на видном месте в виде зеленого заголовка страницы).

В процессе создания Critique мы сознательно решили упростить систему отметок. Изначально Critique содержал оценку «Needs more work» («Требуется доработка»), а также «LGTM++». Впоследствии мы остановились на модели, требующей только комбинации отметки LGTM и одобрения. Если изменение нуждается в повторном обзоре, основные рецензенты могут добавлять комментарии, но без отметки LGTM или одобрения. После того как изменение достигнет состояния «почти готово», рецензенты обычно доверяют авторам внести небольшие правки без необходимости повторно обращаться за получением отметки LGTM независимо от размера изменения.

Эта система оценок оказала положительное влияние на культуру обзора кода. Рецензенты не могут просто отвергнуть изменение, не добавив полезный комментарий. Все отрицательные отзывы от рецензентов должны быть связаны с чем-то конкретным, что необходимо исправить (например, с нерешенным комментарием). Фраза «нерешенный комментарий» тоже была специально выбрана как наиболее точная.

Critique отображает панель оценки рядом со значками инструментов анализа и выводит в ней следующую информацию:

- кто поставил оценку LGTM;
- чье одобрение еще необходимо получить и почему;
- сколько нерешенных комментариев осталось.

Такое представление информации об оценках помогает автору быстро понять, что еще он должен сделать, чтобы зафиксировать изменение.

Отметка LGTM и одобрение *строго* необходимы и ставятся только рецензентами. Также рецензенты в любой момент могут отозвать свои отметки LGTM и одобрения, пока изменения не зафиксированы. Нерешенные комментарии — это *мягкие* требования. Отвечая на комментарий, автор может отметить его как решенный. Это способствует воспитанию доверия и общению между автором и рецензентами. Например, рецензент может поставить отметку LGTM и добавить нерешенные комментарии, не проверяя впоследствии, были ли они решены, что подчеркивает доверие рецензента к автору. Это доверие особенно важно для экономии времени, когда автор и рецензент находятся в разных часовых поясах. Выражение доверия — хороший способ укрепить отношения и сплотить команду.

Этап 6: фиксация изменения

И последний важный этап: в Critique есть кнопка для фиксации изменения после обзора, которая запрещает возможность переключения в интерфейс командной строки.

После фиксации: история изменений

Кроме основной роли инструмента для обзора кода перед фиксацией изменений в репозитории Critique выступает в роли инструмента для археологических изысканий среди изменений. Для большинства файлов разработчики могут видеть список изменений, которые привели файл к текущему состоянию в системе Code Search (глава 17), или перейти непосредственно к изменению. Любой инженер в Google может просмотреть историю изменений общедоступных файлов, включая комментарии и последовательность развития изменения. Эта информация может использоваться для аудита и изучения причин, почему были внесены изменения или как появились ошибки. Разработчики могут использовать Critique, чтобы узнать, как были спроектированы изменения, а результаты обзора кода применить для обучения.

Critique также поддерживает возможность добавления комментариев после фиксации изменения, например когда обнаружена проблема или требуется внести дополнительные сведения для будущих читателей. Critique также позволяет отменить изменение и проверить, было ли конкретное изменение отменено.

КЕЙС: GERRIT

Critique недоступен извне из-за его тесной связи с нашим большим монолитным репозиторием и другими внутренними инструментами. По этой причине некоторые наши команды, работающие над проектами с открытым исходным кодом (включая Chrome и Android) или внутренними проектами, которые не могут размещаться в монолитном репозитории, используют другой инструмент обзора кода — Gerrit.

Gerrit — это самостоятельный инструмент с открытым исходным кодом, предназначенный для рецензирования и тесно интегрированный с Git. Он предлагает веб-интерфейс с поддержкой многих функций Git, включая просмотр кода, слияние ветвей, выбор версий и, конечно, обзор кода. Кроме того, Gerrit имеет обширную модель разрешений, которую можно использовать для ограничения доступа к репозиториям и ветвям.

Оба инструмента, Critique и Gerrit, используют одну и ту же модель обзора кода, в которой каждая фиксация проверяется отдельно. Gerrit поддерживает объединение нескольких изменений и их выгрузку для отдельного обзора, а также позволяет атомарно фиксировать цепочки изменений после обзора.

Как инструмент с открытым исходным кодом, Gerrit предлагает больше вариантов использования: богатая система плагинов Gerrit обеспечивает возможность тесной интеграции в пользовательские среды. Для поддержки разных вариантов использования Gerrit также имеет сложную систему оценки. Рецензент может наложить veto на изменение, поставив оценку -2, а сама система оценок может настраиваться в весьма широких пределах.



Получить дополнительную информацию о Gerrit и увидеть его в действии можно по адресу: <https://www.gerritcodereview.com>.

Заключение

Использование инструмента обзора кода требует неявных компромиссов. Время, потраченное на обзор кода, не связано с созданием нового кода, поэтому любая оптимизация процесса обзора может повысить продуктивность компании. Если для фиксации изменения (в большинстве случаев) достаточно согласия только двух человек (автора и рецензента), то скорость обзора будет высокой. В Google ценят образовательные аспекты обзора кода, несмотря на то что их трудно измерить.

Чтобы процесс обзора изменения выполнялся быстро, он должен протекать беспрепятственно, своевременно информировать пользователя об аспектах, требующих внимания, и выявлять потенциальные проблемы до этапа рецензирования (проблемы выявляются инструментами анализа и системой непрерывной интеграции). По возможности результаты проверки более быстрыми инструментами представляются до завершения проверки более медленными инструментами.

В инструменте Critique есть несколько путей поддержки масштаба. Он обрабатывает большое количество запросов на рецензирование без снижения производительности. Поскольку Critique находится на пути к фиксации изменений, он быстро загружается и всегда готов для использования в особых ситуациях, таких как необычно большие изменения¹. Интерфейс Critique оказывает поддержку пользователям (например, в поиске релевантных изменений) и помогает рецензентам и авторам перемещаться по кодовой базе. Например, Critique сам подбирает подходящих рецензентов для обзора конкретного изменения, не требуя от инженеров выяснять, кто владеет кодом или сопровождает его (это особенно важно для крупномасштабных изменений, таких как миграция на другой API, которая может затронуть многие файлы).

Critique упорядочивает процесс обзора кода и предоставляет удобный интерфейс. При этом он допускает настройки, позволяя применить в обзоре инструменты анализа и предварительные обработчики, формирующие контекст для изменений, а также правила, характерные для команды (например, требование отметки LGTM от нескольких рецензентов).

Доверие и общение — вот основа процесса обзора кода. Инструмент может дополнить опыт инженера, но не может заменить его. Также важным фактором успеха Critique является его тесная интеграция с другими инструментами.

¹ Большинство изменений имеет небольшой размер (менее 100 строк), однако иногда Critique используется для обзора крупных изменений после рефакторинга, которые могут затрагивать сотни и тысячи файлов и должны выполняться атомарно (глава 22).

Итоги

- Доверие и общение — основа процесса обзора кода. Инструмент может дополнить опыт инженера, но не может заменить его.
- Важным фактором успеха Critique является тесная интеграция с другими инструментами.
- Небольшие оптимизации рабочего процесса, такие как добавление явного «набора внимания», могут повысить ясность и существенно снизить задержки выполнения обзора кода.

Статический анализ

Автор: Кейтлин Садовски

Редактор: Лиза Кэри

Целью статического анализа кода является поиск потенциальных проблем, таких как ошибки, антипаттерны и другие недостатки, которые можно выявить *без выполнения программы*. Слово «статический» подразумевает анализ именно кода, а не выполняющейся программы (которую исследует «динамический» анализ). Статический анализ позволяет обнаружить ошибки на самых ранних этапах разработки, до того как они попадут в продакшен. Например, он может идентифицировать константные выражения, вызывающие переполнение, тесты, которые никогда не запускаются, или строки недопустимого формата в операторах журналирования, которые могут вызвать аварийное завершение при выполнении¹. Однако статический анализ можно использовать не только для поиска ошибок. Мы в Google с помощью статического анализа внедряем передовые приемы, помогаем поддерживать актуальность кода на уровне современных API и устранить или сократить технический долг. В качестве примеров такого применения статического анализа можно назвать проверку соблюдения соглашений об именах, выявление попыток использования устаревших API или предложение более простых, но эквивалентных выражений, которые упрощают чтение кода. Кроме того, статический анализ является неотъемлемой частью процесса устаревания API (глава 22). Мы также обнаружили, что проверки кода средствами статического анализа способствуют обучению разработчиков и действительно предотвращают попадание антипаттернов в кодовую базу².

В этой главе мы обсудим причины эффективности статического анализа и наш опыт его внедрения и реализации³.

Характеристики эффективного статического анализа

Исследования статического анализа, направленные на разработку новых и узкоспециализированных видов анализа, ведутся уже не одно десятилетие, но внимание

¹ <http://errorprone.info/bugpatterns>.

² *Sadowski C. et al.* Tricorder: Building a Program Analysis Ecosystem (<https://oreil.ly/9Y-tP>), International Conference on Software Engineering (ICSE), May 2015.

³ Хороший академический справочник по теории статического анализа: Flemming Nielson et al. «*Principles of Program Analysis*» (Gernamy: Springer, 2004).

масштабируемости и *удобству использования* инструментов статического анализа стало уделяться относительно недавно.

Масштабируемость

Поскольку размеры ПО постоянно растут, инструменты анализа должны поддерживать возможность масштабирования, чтобы разработчик мог быстро получать результаты. Инструменты статического анализа в Google должны масштабироваться до размеров кодовой базы, насчитывающей миллиарды строк. Для этого они должны поддерживать возможность сегментирования и инкрементального поиска. Вместо анализа больших проектов целиком наши инструменты концентрируются на файлах, вовлеченных в изменение кода, и обычно показывают результаты анализа только для отредактированных файлов или строк. Масштабирование имеет также свои преимущества: поскольку наша база кода очень велика, в ней действительно много мелких ошибок, которые приходится искать. Но кроме возможности работы с большой базой кода мы также должны предоставлять разнообразие видов анализа, которое расширяется благодаря инициативам наших сотрудников. Еще один аспект масштабируемости статического анализа — масштабируемость *процесса*: инфраструктура статического анализа в Google показывает результаты анализа только тем инженерам, которые имеют непосредственное отношение к этим результатам.

Удобство использования

Размышляя об удобстве использования инструментов статического анализа, важно учитывать соотношение их цены и выгоды для пользователей. Цена может выражаться во времени, которое потратил разработчик, или качестве кода. Исправление предупреждения, полученного от инструмента статического анализа, может привести к ошибке. Зачем «исправлять» код, который редко изменяется и нормально работает в продакшене? Например, исправление предупреждения о «мертвом» коде путем добавления его вызова может привести к внезапному запуску непроверенного (и, возможно, ошибочного) кода. Выгоды от этих действий неочевидны, а их цена порой слишком высока. По этой причине мы обычно концентрируемся только на вновь появляющихся предупреждениях. Существующие проблемы в работающем коде обычно стоит выделять (и исправлять), только если они действительно важны (проблемы безопасности, исправления серьезных ошибок и т. д.). Благодаря концентрации внимания на новых предупреждениях (или предупреждениях в измененных строках) разработчики, просматривающие предупреждения, получают более релевантный контекст проблем.

Время разработчика — ценный ресурс! Затраты времени на сортировку отчетов с результатами анализа или исправление выделенных проблем должны компенсироваться преимуществами конкретного анализа. Если автор анализа может сэкономить время (например, предложив автоматическое исправление), то он должен сделать это, чтобы уменьшить затраты. Все, что можно исправить автоматически,

должно исправляться автоматически. Мы также стараемся особо отметить отчеты о проблемах, которые могут оказать отрицательное влияние на качество кода, чтобы разработчики не тратили время на просмотр нерелевантных результатов анализа.

Чтобы еще больше уменьшить затраты на обзор результатов статического анализа, мы делаем упор на бесшовную интеграцию инструментов в рабочий процесс разработчика. Еще одно преимущество интеграции всех инструментов в один рабочий процесс состоит том, что команда, разрабатывающая инструменты, может обновлять их одновременно с рабочим процессом и кодом, что позволяет развивать инструменты анализа вместе с исходным кодом.

Мы считаем, что наши пути и решения, направленные на улучшение масштабируемости и удобства использования средств статического анализа, естественным образом вытекают из нашего внимания к трем основным принципам, которые сформулированы в виде извлеченных уроков в следующем разделе.

Ключевые уроки внедрения статического анализа

В ходе внедрения статического анализа мы в Google извлекли три ключевых урока. Рассмотрим их в следующих подразделах.

Внимание к удовлетворенности разработчика

Мы упомянули некоторые способы, помогающие сэкономить время разработчика и снизить стоимость взаимодействия с инструментами статического анализа. Но также мы следим за тем, насколько хорошо работают наши инструменты. Без этого невозможно решать проблемы. Мы внедряем инструменты анализа только с низким уровнем ложных срабатываний (подробнее об этом ниже), активно собираем отзывы разработчиков, использующих результаты статического анализа, и оперативно реагируем на них. Поддержание обратной связи с пользователями инструментов статического анализа и их разработчиками создает эффективный цикл, укрепляющий доверие пользователей друг к другу и к самому инструменту.

«Ложноотрицательным» результатом в статическом анализе называют ситуацию, когда фрагмент кода содержит проблему, которую инструмент не замечает. «Ложноположительный» результат имеет место, когда инструмент ошибочно отмечает код как имеющий проблему. Исследования в области статического анализа традиционно сосредоточены на уменьшении количества ложноотрицательных результатов, но на практике низкий уровень ложноположительных результатов часто имеет не менее решающее значение для разработчиков, которые действительно хотят использовать инструмент — кому захочется проридаться через сотни ложных сообщений о проблемах, чтобы выбрать из них действительно заслуживающие внимания¹?

¹ Обратите внимание, что для некоторых видов анализа рецензенты готовы мириться с гораздо более высоким уровнем ложноположительных результатов: одним из примеров является анализ безопасности, выявляющий критические проблемы.

Кроме того, ключевым аспектом в определении доли ложноположительных результатов является *восприятие*. Если инструмент статического анализа генерирует технически обоснованные предупреждения, но ошибочно интерпретируемые пользователями как ложноположительные (например, из-за туманных формулировок сообщений), то пользователи будут реагировать на них, как если бы эти предупреждения действительно были ложноположительными. Аналогичную реакцию вызывают технически верные, но по большому счету не важные предупреждения. Частоту результатов, ложноположительных с точки зрения пользователя, мы называем «эффективной ложноположительной» частотой. Проблема считается «эффективно ложноположительной», если разработчики не предприняли никаких положительных действий после ее обнаружения. Это означает, что если анализ неверно сообщает о проблеме, но разработчик с готовностью вносит исправление, чтобы улучшить читаемость кода или повысить удобство его сопровождения, то эта проблема не считается эффективно ложноположительной. Например, у нас есть анализ для Java, который отмечает случаи вызова метода `contains` хеш-таблицы (эквивалентного вызову `containsValue`), когда на самом деле предполагалось вызвать `containsKey`, — даже если разработчик действительно намеревался проверить значение, для удобочитаемости предпочтительнее использовать `containsValue`. Точно так же если анализ сообщает о действительной ошибке, но разработчик не понял смысла сообщения и не предпринял никаких действий, то такой результат считается эффективно ложноположительным.

Интеграция статического анализа в рабочий процесс разработчика

Мы в Google сделали статический анализ составной частью рабочего процесса, интегрировав его в инструменты обзора кода. В Google весь код, отправляемый в репозиторий, подвергается обзору. На момент отправки кода на проверку разработчики уже готовы к внесению изменений, поэтому исправления, предлагаемые инструментами статического анализа, добавляются без особых проблем. Есть и другие преимущества интеграции статического анализа в обзор кода. После отправки кода на проверку и до получения отзывов рецензентов у разработчиков появляется время для запуска анализа. Рецензенты со своей стороны заставляют инженеров обратить внимание на предупреждения статического анализа. Кроме того, статический анализ может экономить время рецензента, автоматически выделяя распространенные проблемы. Трудно найти лучшее время для статического анализа, чем обзор кода¹.

Предоставление возможности внести свой вклад

В Google работает много экспертов в предметной области, чьи знания могут улучшить создаваемый код. Статический анализ дает этим экспертам возможность распростра-

¹ Далее в этой главе мы поговорим о дополнительных точках интеграции с процессами редактирования и просмотра кода.

нения своего опыта через создание новых инструментов анализа или разработку отдельных проверок в рамках выбранного инструмента.

Например, эксперты, хорошо знающие конкретный тип конфигурационных файлов, могут написать инструмент для проверки определений свойств в этих файлах. В свою очередь, разработчики, обнаружив ошибку, могут предотвратить ее повторное появление где-либо еще в кодовой базе. Для всех специалистов мы создаем легко подключаемую экосистему статического анализа. Мы стремимся разрабатывать простые API, которые могут использоваться всеми инженерами в Google, а не только специалистами по анализу или языку. Например, Refaster¹ позволяет написать анализатор, представив фрагменты кода, которые демонстрируют, как код выглядит до преобразования этим анализатором и как он должен выглядеть после преобразования.

Tricorder: платформа статического анализа в Google

Фундаментом статического анализа в Google служит платформа Tricorder². Она появилась в результате нескольких неудачных попыток интегрировать статический анализ в рабочий процесс разработчика³. Ключевым отличием Tricorder от предыдущих попыток является наше неуклонное стремление дать своим пользователям только ценные результаты. Tricorder тесно интегрирована с Critique — основным инструментом обзора кода в Google. Предупреждения, генерируемые Tricorder, отображаются в средстве просмотра различий в Critique в виде серых панелей с комментариями (рис. 20.1).

Для масштабирования Tricorder использует архитектуру микросервисов. Она отправляет серверам анализа запросы вместе с метаданными об изменениях в коде, а серверы используют эти метаданные для получения версий файлов с изменениями из файловой системы FUSE и входных и выходных данных из кешей системы сборки. Затем серверы запускают каждый отдельный инструмент анализа, записывают результаты в хранилище, а самые последние результаты для каждой категории затем отображаются в Critique. Поскольку иногда для выполнения анализа требуется несколько минут, серверы анализа также сообщают информацию о его состоянии, чтобы авторы изменений и рецензенты могли видеть, что инструменты запущены

¹ *Wasserman L.* Scalable, Example-Based Refactorings with Refaster (<https://oreil.ly/XUKFp>). Workshop on Refactoring Tools, 2013.

² *Sadowski C., Gogh J. van, Jaspan C., Söderberg E., Winter C.* Tricorder: Building a Program Analysis Ecosystem (<https://oreil.ly/mJXTD>), International Conference on Software Engineering (ICSE), May 2015.

³ *Sadowski C., Aftandilian E., Eagle A., Miller-Cushon L., Jaspan C.* Lessons from Building Static Analysis Tools at Google. Communications of the ACM, 61 No. 4 (April 2018): 58–66, <https://cacm.acm.org/magazines/2018/4/226371-lessons-from-building-static-analysis-tools-at-google/fulltext>.

или уже завершили анализ. Tricorder анализирует более 50 000 изменений в день и часто выполняет несколько анализов в секунду.



Рис. 20.1. Средство просмотра различий в Critique, отображающее панели с предупреждениями, сгенерированными платформой статического анализа Tricorder

Разработчики в Google пишут анализы для Tricorder (так называемые «анализаторы») или добавляют отдельные «проверки» в существующие анализы. Есть четыре критерия, которыми должны руководствоваться разработчики при внедрении новых проверок в Tricorder:

Понятность

Текст сообщения должен быть понятен инженеру.

Практичность и простота исправления

Исправление проблемы может потребовать много времени, размышлений или усилий, поэтому результат анализа должен включать описание, поясняющее, как можно исправить проблему.

Не более 10 % эффективно ложноположительных результатов

Разработчики должны быть уверены, что сообщение указывает на реальную проблему как минимум в 90 % случаев (<https://oreil.ly/ARSzt>).

Существенное влияние на качество кода

Отмеченные проблемы могут не влиять на правильность кода, но разработчики должны серьезно отнестись к ним и со всей ответственностью подойти к их исправлению.

Анализаторы Tricorder поддерживают более 30 языков и различные виды анализа. В Tricorder есть более 100 анализаторов, большинство из которых создано за пределами команды Tricorder. Семь из этих анализаторов сами имеют расширяемую архитектуру и включают сотни дополнительных проверок, созданных разработчиками Google. Общая частота эффективно ложноположительных результатов находится на уровне чуть ниже 5%.

Интегрированные инструменты

В Tricorder интегрировано множество инструментов статического анализа.

Error Prone (<http://errorprone.info>) и clang-tidy (<https://oreil.ly/DAMiv>) расширяют компилятор, добавляя проверки антипаттернов AST для Java и C++ соответственно. Эти антипаттерны могут представлять настоящие ошибки. Например, взгляните на следующий фрагмент кода, выполняющий хеширование поля `f` типа `long`:

```
result = 31 * result + (int) (f ^ (f >>> 32));
```

Теперь представьте, что `f` имеет тип `int`. Код все равно будет компилироваться, но сдвиг вправо на 32 позиции является пустой операцией для этого типа, и в результате будет выполнена операция исключающего ИЛИ (XOR) поля `f` с самим собой, никак не влияющая на получаемое значение. Мы исправили 31 появление этой ошибки в кодовой базе Google, включив проверку как ошибку компилятора в Error Prone. И таких примеров очень много (<https://errorprone.info/bugpatterns>). Устранение антипаттернов AST также может способствовать улучшению удобочитаемости кода, например за счет удаления избыточных вызовов метода `.get()` интеллектуального указателя.

Другие анализаторы отражают взаимосвязи между разрозненными файлами в корпусе анализа. Анализатор удаленных артефактов Deleted Artifact Analyzer предупреждает об удалении исходного файла, на который есть ссылки в других местах базы кода, не относящихся к анализируемому коду (например, в документации, хранящейся в репозитории). IfThisThenThat позволяет разработчикам указать, что два разных файла должны изменяться вместе (и предупреждает, обнаружив изменение только в одном из них). Анализатор Chrome Finch работает с файлами конфигурации, предназначенными для А/В-экспериментов в Chrome, выявляя распространенные проблемы, включая отсутствие необходимых разрешений на запуск эксперимента или перекрестные конфликты с другими текущими экспериментами, затрагивающими ту же популяцию. Анализатор Finch выполняет RPC в других службах, чтобы предоставить информацию об удаленных процедурах.

Некоторые анализаторы работают не только с исходным кодом, но и с другими артефактами, созданными этим исходным кодом. Во многих проектах включена функция проверки размера двоичного файла, которая предупреждает, когда изменение существенно влияет на этот размер.

Почти все анализаторы являются внутрипроцедурными, то есть их результаты основаны на анализе кода внутри процедуры (функции). Композиционные и инкрементальные методы межпроцедурного анализа технически осуществимы, но требуют дополнительных вложений в инфраструктуру (например, в анализ и хранение сводных данных о методах во время работы анализаторов).

Интегрированные каналы обратной связи

Как упоминалось выше, обратная связь между пользователями и разработчиками анализа чрезвычайно важна для удовлетворенности разработчиков. В Tricorder мы даем

пользователям возможность щелкнуть на кнопке **Not useful** (Бесполезно) в результатах анализа, чтобы сформировать сообщение об ошибке анализа с описанием, почему анализ был расценен как бесполезный, и отправить это сообщение *непосредственно автору анализатора*. Рецензенты кода также могут попросить авторов изменений исправить выявленные проблемы, щелкнув на кнопке **Please fix** (Исправьте) в результатах анализа. Команда Tricorder внимательно наблюдает, как часто рецензенты просят исправить проблемы, выявленные в ходе анализа, и отключает анализаторы, если они не помогают в решении проблем или имеют высокий показатель «бесполезности». Организация и настройка обратной связи потребовали много усилий, но эти трудозатраты не раз приносили дивиденды в виде улучшения результатов анализа и взаимодействия с пользователями — до того как мы установили четкие каналы обратной связи, многие разработчики просто игнорировали результаты анализа, которые казались им непонятными.

Иногда исправить недостатки в анализаторе было довольно просто — нужно было всего лишь изменить текст сообщения! Например, однажды мы развернули в Error Prone проверку, которая отмечала вызовы printf-подобной функции в Guava со слишком большим количеством аргументов. Эта функция принимала только %s и никакие другие спецификаторы printf. Команда Error Prone еженедельно получала сообщения «Бесполезно», в которых утверждалось, что анализ дает неправильные результаты, потому что количество спецификаторов формата соответствовало количеству аргументов, но пользователи пытались передавать спецификаторы, отличные от %s. После того как группа изменила диагностический текст, прямо указав, что функция принимает только спецификатор %s, приток отчетов об ошибках прекратился. Уточняющее сообщение, генерируемое инструментом анализа, пояснило, что именно неправильно, почему и как исправить проблему. Такие уточнения особенно важны для разработчиков, которые узнают что-то новое, читая сообщения.

Предлагаемые исправления

Проверки в Tricorder также могут *предлагать варианты исправления* проблемы, когда это возможно (рис. 20.2).

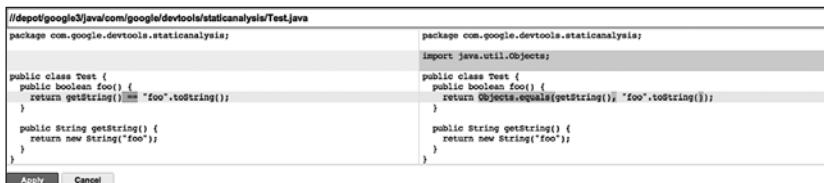


Рис. 20.2. Пример результатов анализа с предлагаемым исправлением в Critique

Автоматически предлагаемые исправления служат дополнительным источником информации, когда сообщение оказывается недостаточно ясным, и, как упоминалось

выше, сокращают затраты на устранение проблем, выявленных статическим анализом. Исправления можно применять непосредственно из Critique или с помощью инструмента командной строки. Но не все анализаторы предлагают исправления. Мы придерживаемся мнения, что проблемы *стиля* должны устраняться инструментами, которые автоматически форматируют файлы с исходным кодом. В Google есть свои руководства по стилю для каждого языка, где перечислены правила форматирования, поэтому сообщения с описанием ошибок форматирования не стоят внимания рецензента. Рецензенты щелкают на кнопке Please Fix (Исправьте) несколько тысяч раз в день, авторы применяют автоматически предлагаемые исправления примерно 3000 раз в день, и анализаторы Tricorder получают около 250 щелчков Not useful (Бесполезно) в день.

Настройки для конкретных проектов

Создав платформу, заслужившую доверие пользователей отображением только максимально достоверных результатов, мы добавили в нее возможность запуска дополнительных «необязательных» анализаторов в конкретных проектах. Примером такого дополнительного анализатора может служить *Proto Best Practices*. Он выявляет потенциально опасные изменения в формате данных в Protocol Buffers (<https://developers.google.com/protocol-buffers>) — не зависящем от языка формате сериализации данных, разработанном в Google. Эти изменения могут быть опасными, только когда сериализованные данные хранятся, например, в журналах сервера. Проверка Protocol Buffers не требуется в проектах, не хранящих сериализованные данные. Мы также добавили возможность настройки существующих анализаторов, хотя и ограниченную, и многие проверки применяются по умолчанию единообразно для всей кодовой базы.

Некоторые анализаторы начинали свой путь как дополнительные, улучшались на основе отзывов, расширяли круг своих пользователей, а затем переходили в категорию анализаторов, включенных по умолчанию, как только заслуживали доверие пользователей. Например, у нас есть анализатор, который предлагает исправления, улучшающие удобочитаемость кода на Java, но обычно не влияющие на его поведение. Пользователи Tricorder сначала волновались, что этот анализатор будет генерировать слишком много сообщений, но в итоге в них проснулось желание получать больше результатов от такого анализа.

Ключ к успешной настройке платформы статического анализа — сосредоточиться на *настройке на уровне проекта, а не на уровне пользователя*. Настройка на уровне проекта гарантирует, что все члены команды получают единообразное представление результатов анализа для своего проекта, и предотвратит ситуации, когда один разработчик пытается исправить проблему, а другой порождает ее.

Изначально платформа Tricorder поддерживала набор относительно простых средств проверки стиля и отображала результаты в Critique, а Critique предоставлял возможность настройки уровня достоверности отображаемых результатов и подавления

результатов конкретных анализаторов. Потом мы удалили из Critique эти настройки и сразу начали получать от пользователей жалобы на раздражающие результаты анализа. Вместо того чтобы вернуть возможность настройки, мы предложили пользователям рассказать, что их раздражает, и обнаружили в анализаторах стиля множество ошибок. Например, анализатор стиля для C++ при использовании для Objective-C давал неверные и бесполезные результаты. Мы исправили эту проблему, изменив инфраструктуру анализа стиля. Анализатор HTML имел чрезвычайно высокую ложноположительную частоту и очень низкий уровень полезного сигнала, из-за чего обычно отключался разработчиками, пишущими разметку HTML. Поскольку этот инструмент оказался практически бесполезным, мы просто отключили его. Так возможность настройки на уровне пользователя привела к появлению множества скрытых ошибок и подавлению обратной связи.

Предварительные проверки

Кроме обзора кода в Google есть другие точки интеграции статического анализа в рабочий процесс. Поскольку разработчики могут игнорировать предупреждения статического анализа, отображаемые при обзоре кода, в Google была реализована дополнительная возможность добавить анализ, блокирующий попытку фиксации изменения в репозитории, который мы называем *предварительной проверкой*, или *проверкой перед отправкой*. К числу предварительных проверок относятся очень простые настраиваемые внутренние проверки содержимого или метаданных изменения, такие как проверка присутствия в сообщении фиксации текста «DO NOT SUBMIT» (НЕ ОТПРАВЛЯТЬ) или подключения тестовых файлов в соответствующих файлах с кодом. Команды также могут указать, какие наборы тестов должны выполняться или какие категории проблем не должны выявляться системой Tricorder. В процессе предварительной проверки также оценивается правильность формата кода. Предварительные проверки обычно выполняются при отправке изменения для обзора, а затем снова во время фиксации, но также могут запускаться по запросу между этими двумя точками. Подробнее о предварительных проверках в главе 23.

Некоторые команды написали свои предварительные проверки. Они дополняют базовый набор проверок, позволяют реализовать передовые практики и добавляют новые виды анализа для конкретных проектов. Благодаря им новые проекты могут руководствоваться более строгими принципами, чем, например, проекты с большим количеством устаревшего кода. Но предварительные проверки для конкретных команд могут усложнить процесс крупномасштабных изменений (см. главу 22), поэтому некоторые из них пропусаются для изменений с текстом «CLEANUP =>» в описании.

Интеграция с компилятором

Блокировка попыток фиксации с помощью статического анализа — это, конечно, хорошо, но лучше было бы уведомлять разработчиков о проблемах еще раньше.

Когда это возможно, мы стараемся добавить статический анализ в компилятор. Прерывание сборки — это предупреждение, которое невозможно игнорировать, но во многих случаях его нельзя организовать. Однако некоторые анализы имеют механический характер и не дают эффективно ложноположительных результатов. Примером могут служить проверки «ERROR» в Error Prone (<https://errorprone.info/bugpatterns>). Они включены в компилятор Java, который используется в Google, и предотвращают повторное появление ошибок в нашей кодовой базе. Проверки в компиляторе должны выполняться быстро, чтобы не замедлять сборку, и отвечать трем критериям (аналогичные критерии существуют для компилятора C++):

- практичность и простота исправления (по возможности сообщение об ошибке должно включать предложение по исправлению, которое можно применить автоматически);
- отсутствие эффективно ложноположительных результатов (анализ никогда не должен останавливать сборку правильного кода);
- сообщение только о проблемах, влияющих на правильность кода, но не на стиль или использование передовых практик.

Включать новую проверку нужно после удаления из базы кода всех проблем, выявляемых ею, чтобы сборка существующих проектов не нарушилась из-за эволюции компилятора. Это также означает, что новая проверка, интегрируемая в компилятор, должна быть достаточно ценной, чтобы взяться за работу по исправлению всех выявляемых проблем. В Google имеется своя кластерная инфраструктура для запуска различных компиляторов (таких, как clang и javac) для всей кодовой базы, действующая как MapReduce. Когда компиляторы запускаются в режиме MapReduce, проверки статического анализа должны сами исправлять обнаруженные проблемы, чтобы автоматизировать чистку. После подготовки и тестирования изменения с исправлениями ко всей кодовой базе мы фиксируем его и устраним выявленные проблемы. Затем мы включаем проверку в компиляторе, чтобы никакие новые проявления проблемы не смогли проникнуть в кодовую базу и прерывать процедуру сборки. Прерывания сборки обнаруживаются после фиксации изменений нашей системой непрерывной интеграции или перед их фиксацией с помощью предварительных проверок (как описывалось выше).

Мы также стараемся никогда не генерировать сообщения в форме предупреждений компилятора, потому что, как показал опыт, разработчики часто игнорируют предупреждения. Поэтому мы включаем проверку в компилятор как ошибку (и прерываем сборку) или не показываем ее в выводе компилятора. Поскольку во всей кодовой базе используются одни и те же флаги компилятора, это решение принято глобально. Проверки, которые не должны прерывать сборку, либо подавляются, либо отображаются при проверке кода (например, в Tricorder). Эта политика применяется не ко всем языкам в Google, но для наиболее часто используемых языков она реализована. Компиляторы Java и C++ не отображают предупреждения компилятора. А в Go являются ошибками определенные сообщения, которые в других языках считаются

предупреждениями (например, неиспользуемые переменные или импортирование нетребуемых пакетов).

Анализ в процессе редактирования и просмотра кода

Еще одна потенциальная точка интеграции статического анализа — IDE. Однако для использования в IDE анализ должен выполняться очень быстро (не более одной секунды, желательно — менее 100 мс), поэтому некоторые инструменты не подходят для интеграции с IDE. Кроме того, существует проблема обеспечения идентичности результатов одного и того же анализа в нескольких IDE. Мы также заметили, что популярность IDE может расти и падать (мы не требуем использовать какую-то единую IDE), поэтому интеграция с IDE имеет более беспорядочный характер, чем интеграция с процессом обзора кода. Процесс обзора — самое подходящее место для отображения результатов анализа. Анализ может учитывать весь контекст изменения, но некоторые анализы могут ошибаться при работе с неполным кодом (например, при анализе «мертвого» кода функция реализуется до добавления ее вызовов). Отображение результатов анализа в обзоре кода также вынуждает авторов кода убедить рецензентов игнорировать результаты анализа, если это необходимо. Тем не менее в IDE тоже можно отобразить результаты статического анализа.

Мы стараемся отображать только самые последние предупреждения статического анализа, но иногда разработчики хотят видеть результаты анализа для всей кодовой базы при ее просмотре (например, при анализе безопасности). Специализированные группы безопасности в Google хотят иметь целостное представление обо всех проявлениях проблем. Кроме того, разработчики любят просматривать результаты анализа кодовой базы при планировании чистки. Таким образом, в некоторых случаях инструмент просмотра кода является более чем подходящим местом для отображения результатов анализа.

Заключение

Статический анализ — отличный инструмент для улучшения кодовой базы и раннего обнаружения ошибок, который позволяет в более дорогостоящих процессах (таких, как проверка и тестирование кода вручную) сосредоточиться на проблемах, которые невозможно выявить автоматически. Улучшив масштабируемость и удобство использования инфраструктуры статического анализа, мы превратили статический анализ в эффективный компонент разработки ПО в Google.

Итоги

- *Внимание к удовлетворенности разработчика.* Мы много трудились над созданием каналов обратной связи между пользователями и авторами анализа и продолжаем активно развивать процесс анализа, чтобы уменьшить количество ложных результатов.

-
- *Интеграция статического анализа в рабочий процесс разработчика.* Главной точкой интеграции статического анализа в Google является процесс обзора кода, в котором инструменты анализа предлагают возможные исправления и распределяют внимание рецензентов. Также мы интегрировали анализ в дополнительных точках (компиляторе, системе фиксации кода в репозитории, IDE и средствах просмотра кода).
 - *Предоставление возможности внести свой вклад.* Мы масштабируем свою работу, используя опыт экспертов для создания и обслуживания инструментов и платформы анализа. Разработчики постоянно добавляют новые виды анализа и проверки, которые делают их работу проще, а нашу кодовую базу — лучше.

Управление зависимостями

Автор: Титус Винтер

Редактор: Лиза Кэри

Управление зависимостями — коллекциями сторонних библиотек, пакетов и зависимостей — является одной из наиболее сложных задач в программной инженерии. Управление зависимостями фокусируется на вопросах: «Как обновлять версии внешних зависимостей?», «Как правильно их описывать?», «Какие типы изменений допустимы или ожидаемы в зависимостях?», «Как не ошибиться, принимая решение о создании зависимости от кода, разрабатываемого другими организациями?»

Наиболее близкой к задаче управления зависимостями является задача управления версиями исходного кода. Обе связаны с управлением исходным кодом. Управление версиями отвечает на простые вопросы: «Откуда брать исходный код?», «Как добавить исходный код в сборку?» После признания ценности разработки в главной ветви большинство повседневных вопросов управления версиями сводится к одному: «У меня есть новый код, в какой каталог его следует поместить?»

Управление зависимостями добавляет к этим вопросам дополнительные сложности, имеющие отношение к масштабированию. В задаче управления исходным кодом на основе главной ветви вы знаете, что после внесения изменений запустите тесты, чтобы убедиться, что не повредили существующий код. Суть этой идеи проста: вы работаете с общей базой кода, имеете представление о том, как используется тот или иной код, и можете запустить сборку и тесты. Управление зависимостями фокусируется на проблемах, которые возникают, когда изменения вносятся вне вашей организации и вам неизвестно, что именно и как изменилось. Поскольку внешние проекты часто не согласовывают с вами свои изменения, эти изменения могут нарушить ваш процесс сборки и привести к сбою ваши тесты. Как справиться с этим? Отказаться от внешних зависимостей? Потребовать от внешних проектов согласовывать изменения с вами? Когда следует обновить зависимость до новой версии?

Еще сложнее импортировать зависимости от целой сети внешних проектов. Как только появляется несколько зависимостей, легко оказаться в ситуации, когда какие-то две зависимости начинают конфликтовать. Обычно это происходит из-за того, что одна зависимость предъявляет некоторое требование¹, а другая с ним несовместима.

¹ Это может быть любое требование: версия языка, версия низкоуровневой библиотеки, версия оборудования, операционная система, флаги компилятора, версия компилятора и т. д.

Простые решения, ориентированные на управление единственной внешней зависимостью, обычно не учитывают реалии управления обширной сетью зависимостей. Большую часть этой главы мы посвятим обсуждению конфликтующих требований.

Управление исходным кодом и управление зависимостями — это родственные задачи, отличающиеся только ответом на вопрос: «Контролирует ли организация разработку/обновление/управление в используемом проекте?» Например, если каждая команда в компании имеет свои репозитории, преследует свои цели, использует свои методы разработки и организации взаимодействий и по-своему управляет исходным кодом, то такой компании придется решать скорее задачу управления зависимостями, чем управления исходным кодом. В свою очередь, большая организация с единым репозиторием (монолитным и, возможно, виртуальным) может значительно расширить свои масштабы с помощью политик управления исходным кодом — именно этот подход используется в Google. Проекты с открытым исходным кодом, безусловно, считаются отдельными организациями, в которых управление взаимозависимостями между несвязанными и не обязательно сотрудничающими проектами относится к задаче управления зависимостями. Наш самый действенный совет по этой теме: *если возможно, отдавайте предпочтение управлению исходным кодом, а не управлению зависимостями*. Старайтесь использовать более широкое определение термина «организация» (например, вся компания, а не одна команда). Задача управления исходным кодом решается намного проще, чем задача управления зависимостями.

Модель ПО с открытым исходным кодом продолжает развиваться и проникать в новые области, и графы зависимостей во многих популярных проектах продолжают расширяться со временем, поэтому управление зависимостями становится, пожалуй, самой важной задачей в программной инженерии. Софтверные компании больше не острова в безбрежном океане, связанные одним или двумя уровнями API. Современное ПО опирается на возвышающиеся столпы зависимостей, но, создавая такие столпы, мы еще не научились сохранять их устойчивость и надежность.

В этой главе мы рассмотрим конкретные сложности управления зависимостями, исследуем решения (общие и новые) и их ограничения, а также посмотрим на реалии работы с зависимостями, в том числе в Google. Мы много *думали* над этой проблемой и имеем большой опыт в вопросах рефакторинга и сопровождения, который показывает практические недостатки существующих подходов. И должны признаться, что у нас нет свидетельств существования решений, хорошо работающих в больших организациях. В какой-то степени эта глава представляет собой краткое описание подходов, которые не работают (или, по крайней мере, могут не работать в больших компаниях), и подходов, которые, по нашему мнению, имеют определенный потенциал. Если бы у нас было больше ответов, мы не назвали бы эту проблему одной из самых важных в программной инженерии.

Почему управлять зависимостями так сложно?

Даже определение задачи управления зависимостями представляет сложности. Многие полусырые решения в этой области вытекают из слишком узкой формулировки

проблемы: «Как импортировать пакет, от которого зависит наш локальный код?» Это реальный вопрос, но не полный. Сложность не в том, чтобы выяснить, как управлять одной зависимостью, а в том, чтобы понять, как управлять *сетью* зависимостей и их будущими изменениями. Часть зависимостей из этой сети прямо необходимы вашему коду, другая часть — подтягивается промежуточными зависимостями. В течение достаточно длительного периода все узлы этой сети зависимостей обновятся, и некоторые из этих обновлений будут играть важную роль¹. Как управлять получившимся каскадом обновлений? Как упростить поиск совместимых версий зависимостей, которые мы не контролируем? Как проанализировать сеть зависимостей? Как управлять этой сетью, особенно если граф зависимостей постоянно расширяется?

Конфликтующие требования и ромбовидные зависимости

Центральный вопрос в управлении зависимостями: «Что случится, если два узла в сети зависимостей предъявят конфликтующие требования и ваша организация зависит от них обоих?» Эта ситуация может возникнуть по многим причинам, начиная от ограничений платформы (ОС, версии языка, версии компилятора и т. д.) и заканчивая утилитарной проблемой несовместимости версий. Каноническим примером несовместимости версий из-за невыполнимого требования может служить проблема *ромбовидной зависимости*. Мы обычно не включаем в граф зависимостей такие требования, как «выбор версии компилятора», но большинство проблем с конфликтующими требованиями сродни «добавлению в граф зависимостей (скрытого) узла, представляющего это требование». Поэтому в первую очередь мы обсудим конфликтующие требования с точки зрения ромбовидных зависимостей, но имейте в виду, что `libbase` на самом деле может быть абсолютно любым ПО, участвующим в создании двух или более узлов в сети зависимостей.

Проблема ромбовидной зависимости и другие формы конфликтующих требований имеют как минимум три уровня (рис. 21.1).

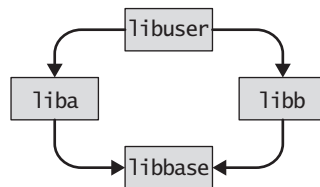


Рис. 21.1. Проблема ромбовидной зависимости

В этой упрощенной модели `libbase` используется двумя компонентами, `liba` и `libb`, которые, в свою очередь, используются компонентом более высокого уровня `libuser`. Если когда-нибудь в `libbase` будет внесено несовместимое изменение, есть шанс, что

¹ Например, ошибки безопасности, устаревание, присутствие в наборе зависимостей от зависимостей более высокого уровня, в которых есть ошибки безопасности, и т. д.

`liba` и `libb`, как продукты разных организаций, обновятся не одновременно. Если `liba` зависит от новой версии `libbase`, а `libb` — от старой, у `libuser` (то есть вашего кода) не будет никакой возможности собрать все вместе. Этот ромб может сформироваться в любом масштабе: если в сети зависимостей существует низкоуровневый узел, который может присутствовать в виде двух несовместимых версий (в силу наличия двух путей к этим версиям от одного узла более высокого уровня), то возникнет проблема.

Проблема ромбовидной зависимости с разной степенью проявляется в разных языках программирования. В некоторых языках допускается встраивать в сборку несколько (изолированных) версий зависимости: обращения к `libbase` из `liba` и `libb` могут вызывать разные версии одного API. Например, в Java есть хорошо отлаженные механизмы переименования символов в подобных зависимостях¹, тогда как C++ практически не допускает ромбовидных зависимостей в обычной сборке и они с большой вероятностью будут вызывать произвольные ошибки и неопределенное поведение вследствие явного нарушения правила определения в C++ (<https://oreil.ly/VTZe5>). В лучшем случае можно использовать идею затенения, заимствованную из Java, чтобы скрыть некоторые символы в динамически подключаемой библиотеке (DLL, dynamic-link library), или применять отдельную сборку и компоновку. Однако во всех языках программирования, известных нам, эти обходные решения являются неполными: можно построить несколько версий, настроив имена *функций*, но если между зависимостями передаются типы, возникнут новые проблемы. Например, невозможно семантически согласованным способом передать экземпляр типа `map`, определенного в `libbase v1`, через какие-либо библиотеки в API, предоставляемый `libbase v2`. Характерные для языка приемы сокрытия или переименования сущностей в отдельно скомпилированных библиотеках могут смягчить проблему ромбовидной зависимости, но в общем случае не являются решением конфликта.

Если вы столкнулись с проблемой конфликтующих требований, лучшее решение — взять следующие или предыдущие версии зависимостей, чтобы найти среди них совместимые между собой. Если это невозможно, то необходимо прибегнуть к локальному исправлению конфликтующих зависимостей, что является особенно сложной задачей, потому что причина несовместимости в них, скорее всего, неизвестна инженеру, который первым обнаружил несовместимость. Это неизбежно: разработчики `liba` по-прежнему работают в режиме совместимости с `libbase v1`, а разработчики `libb` уже обновились до `v2`. И только разработчик, участвующий в обоих этих проектах, имеет шанс обнаружить проблему, и, конечно же, нет никаких гарантий, что он окажется достаточно близко знакомым с `libbase` и `liba`, чтобы пройти через обновление. Проще всего понизить версию `libbase` и `libb`, хотя это не лучший вариант, если обновление вызвано проблемами с безопасностью.

Политики и технологии управления зависимостями в значительной степени сводятся к вопросу: «Как избежать конфликтов в требованиях, но при этом разрешить изменения в несогласованных проектах?» Если вам удастся найти общее решение

¹ Их называют *затенением*, или *версионированием*.

проблемы ромбовидной зависимости, которое учитывает реалии постоянно меняющихся требований (как самих зависимостей, так и требований к платформе) на всех уровнях сети, то вы опишете самое интересное решение в управлении зависимостями.

Импортирование зависимостей

С точки зрения программирования лучше повторно использовать существующую инфраструктуру, чем строить новую. Если каждый новичок будет заново писать синтаксический анализатор JSON и механизм регулярных выражений, технологический прогресс остановится. Повторное использование выгодно, особенно по сравнению с затратами на переделку качественного ПО с нуля. Если имеющаяся внешняя зависимость удовлетворяет требованиям вашей задачи и у вас нет ничего лучше, то используйте ее.

Обещание совместимости

Когда в задаче появляется фактор времени, приходится принимать непростые решения. Возможность избежать лишних затрат на *разработку* не означает, что импорт зависимости является правильным выбором. В софтверной организации ведется учет текущих расходов на обслуживание. Даже если зависимость импортируется без намерения обновлять ее в будущем, обнаруженные уязвимости в системе безопасности, смена платформы и развивающиеся сети зависимостей могут вынудить провести обновление зависимости. Насколько дорого оно обойдется? Некоторые зависимости позволяют довольно точно определить ожидаемые затраты на их обслуживание, но как оценить совместимость версий? Насколько кардинальными могут быть изменения, обусловленные развитием технологий? Как учитывать такие изменения? Как долго поддерживаются версии?

Мы хотели бы, чтобы разработчики зависимостей давали четкие ответы на эти вопросы. Возьмем для примера несколько крупных инфраструктурных проектов, считающих миллионы пользователей, и рассмотрим их обещания в отношении совместимости версий.

C++

Для стандартной библиотеки C++ используется модель с почти неограниченной обратной совместимостью. Ожидается, что двоичные файлы, скомпилированные с использованием более старой версии стандартной библиотеки, будут компоноваться с новым стандартом: стандарт обеспечивает не только совместимость с API, но и постоянную обратную совместимость двоичных артефактов, известную как *ABI-совместимость* (application binary interface — двоичный интерфейс приложения). Степень поддержки такой совместимости для разных платформ разная. Код, скомпилированный с помощью gcc в Linux, должен нормально работать в течение примерно десяти лет. Стандарт не заявляет о своей приверженности к ABI-совместимости — по этому поводу не было опубликовано официальных документов. Однако был опу-

бликован документ «Standing Document 8» (SD-8, <https://oreil.ly/LoJq8>), в котором перечисляется набор типов изменений, которые могут вноситься в стандартную библиотеку между версиями. Этот документ неявно определяет, к каким видам изменений нужно готовиться. Аналогичный подход используется для Java: исходный код совместим между версиями языка, а файлы JAR из более старых версий должны работать с более новыми версиями.

Go

Для разных языков совместимость имеет разный приоритет. Язык Go явно обещает совместимость на уровне исходного кода, но не обещает двоичной совместимости. У вас не получится собрать библиотеку на Go с одной версией языка и связать ее с программой на Go, собранной с другой версией языка.

Abseil

Проект Google библиотека Abseil очень похож на Go, но с одной важной оговоркой относительно времени. Мы не стремимся обеспечивать совместимость *бесконечно долго*: Abseil лежит в основе многих наших внутренних вычислительных сервисов, которые, как предполагается, будут долго использоваться. Это означает, что мы оставляем за собой полное право вносить изменения в Abseil, особенно в детали ее реализации и ABI, чтобы обеспечить максимальную производительность. API не раз превращался в источник проблем и ошибок постфактум, и мы не хотим вынуждать десятки тысяч разработчиков мириться с подобными проблемами в течение неопределенного времени. У нас уже есть около 250 миллионов строк кода на C++, которые зависят от Abseil, и мы не будем вносить критические изменения в API, не предоставив инструмента для автоматического рефакторинга, который преобразует код со старым API в новый. Мы считаем, что это значительно уменьшит риск непредвиденных расходов для пользователей: для какой бы версии Abseil не была написана зависимость, пользователи этой зависимости и Abseil должны иметь возможность применить самую последнюю версию Abseil. Вся сложность должна быть заключена в том, чтобы запустить этот инструмент и, возможно, отправить полученное исправление на проверку в зависимости среднего уровня (*liba* или *libb* из предыдущего примера). Однако пока проект достаточно новый, поэтому мы еще не вносили существенных изменений в API и не можем сказать, насколько хорошо будет работать этот подход для экосистемы в целом, но теоретически баланс между стабильностью и простотой обновления выглядит неплохо.

Boost

Библиотека Boost на C++ не обещает совместимости между версиями (<https://www.boost.org/users/faq.html>). Большая часть кода, конечно, не меняется, но многие библиотеки Boost активно развиваются и улучшаются, поэтому обратная совместимость с предыдущей версией не всегда возможна. Пользователям рекомендуется выполнять обновление только в те периоды жизненного цикла проекта, когда изменения не вызывают проблем. Цель Boost принципиально отличается от стандартной библиотеки

или Abseil: Boost — это экспериментальный полигон. Отдельные версии Boost стабильны и подходят для использования во многих проектах, но совместимость между версиями не является главной целью проекта Boost, поэтому другим долгосрочным проектам, зависящим от Boost, бывает сложно поддерживать актуальность. Разработчики Boost столь же опытные, как и разработчики стандартной библиотеки¹, но речь идет не о технических знаниях: это исключительно вопрос, что проект делает, а не что обещает и как расставляет приоритеты.

Рассматривая библиотеки в этом обсуждении, важно понимать, что перечисленные проблемы совместимости связаны с *программной инженерией*, а не с *программированием*. Вы можете загрузить библиотеку, такую как Boost, которая не обещает обратной совместимости, глубоко встроить ее в наиболее важные и долгоживущие системы организации, и они будут *работать* прекрасно. Основные проблемы связаны не с созданием реализации, а с изменением зависимостей, необходимостью менять свой код в соответствии с обновлениями и принуждением разработчиков беспокоиться об обслуживании. В Google существуют рекомендации, помогающие инженерам понять разницу между утверждениями: «Я заставил работать эту функцию» и «Эта функция работает и поддерживается» (вспомните закона Хайрама).

В целом важно понимать, что управление зависимостями имеет совершенно разную природу в программировании и в программной инженерии. В предметной области, где сопровождение играет важную роль, управление зависимостями затруднено. В сиюминутных решениях, где не потребуется ничего обновлять, напротив, вполне разумно использовать столько зависимостей, сколько понадобится, не задумываясь об их ответственном использовании или планировании обновлений. Заставить программу работать сегодня, нарушив все положения в SD-8, и положиться на двоичную совместимость с Boost и Abseil можно, если в будущем вы не собираетесь работать с обновленными версиями стандартной библиотеки, Boost, Abseil и всего остального, что не зависит от вас.

Рекомендации при импорте

Импорт зависимости в проекте программирования почти не влечет затрат, кроме затрат времени на то, чтобы убедиться, что она делает то, что нужно, не подрывает безопасность и обходится дешевле, чем ее альтернатива. Даже если зависимость дает некоторые обещания в отношении совместимости, то, пока не предполагается обновлять ее в будущем, можно полагаться на выбранную версию, сколько бы правил ни было нарушено при использовании API. Но в проектах программной инженерии зависимости становятся немного дороже и возникает множество скрытых затрат и вопросов, на которые необходимо ответить. Надеюсь, вы учитываете эти расходы перед импортированием и умеете отличать проекты программирования от проектов программной инженерии.

¹ Многие разработчики работают сразу в обоих этих проектах.

Мы в Google рекомендуем сначала ответить на следующий (неполный) список вопросов, прежде чем импортировать зависимости:

- Есть ли в проекте тесты, которые можно запустить?
- Эти тесты выполняются успешно?
- Кто разрабатывает зависимость? Даже проекты с открытым исходным кодом, не дающие никаких гарантий, создаются опытными разработчиками. Но зависимость от совместимости со стандартной библиотекой C++ или Java-библиотекой Guava нельзя сравнить с выбором случайного проекта в GitHub или npm. Репутация зависимости — это не главный критерий ее выбора, но ее стоит учитывать.
- К какой совместимости стремится проект?
- Фокусируется ли проект на определенном сценарии использования?
- Насколько популярен проект?
- Как долго организация будет зависеть от этого проекта?
- Как часто в проект вносятся изменения, ломающие совместимость?

Добавьте к ним еще несколько наших внутренних вопросов:

- Насколько сложно реализовать зависимость в Google?
- Какие стимулы будут побуждать вас поддерживать зависимость в актуальном состоянии?
- Кто будет выполнять обновление?
- Насколько сложно будет выполнить обновление?

Расс Кокс писал об этом более подробно (<https://research.swtch.com/deps>). Мы не можем дать идеальные критерии, которые помогут понять, что лучше — импортировать зависимость или реализовать ее аналог у себя, — потому что мы сами часто ошибаемся в этом вопросе.

Как в Google импортируются зависимости

Если говорить кратко, то далеко не идеально.

В Google подавляющее большинство зависимостей разрабатывается внутри компании. То есть основная часть нашей внутренней истории управления зависимостями на самом деле не имеет отношения к управлению зависимостями — она связана с управлением версиями исходного кода. Как уже упоминалось, управлять сложностями и рисками, связанными с добавлением зависимостей, гораздо проще, когда эти зависимости производятся и потребляются внутри одной организации, имеют надлежащую видимость и участвуют в процессе непрерывной интеграции (глава 23). Большинство проблем в управлении зависимостями исчезает, когда есть возможность видеть, как используется код, и знать точно, как повлияет то или иное изменение. Управлять исходным кодом (когда есть возможность контролировать проекты) намного проще, чем управлять зависимостями (когда такая возможность отсутствует).

Эта простота управления теряется, когда мы начинаем использовать внешние проекты. Проекты, импортируемые из экосистемы ПО с открытым исходным кодом или от коммерческих партнеров, мы добавляем в отдельный каталог нашего монолитного репозитория с отметкой `third_party`. Давайте посмотрим, как это происходит.

Алиса работает инженером-программистом в Google и занимается некоторым проектом. В какой-то момент она замечает, что для ее проекта существует решение с открытым исходным кодом. Ей очень хотелось бы завершить и продемонстрировать свой проект в ближайшее время, чтобы потом спокойно поехать в отпуск. Она оказывается перед выбором: реализовать все необходимые функции самой или загрузить пакет с открытым исходным кодом и добавить его в каталог `third_party`. Скорее всего, Алиса посчитает, что для ускорения разработки есть смысл использовать открытое решение. Она загрузит открытый пакет и выполнит несколько шагов, которые требуют правила включения стороннего ПО: убедится, что пакет собирается в нашей системе сборки и его текущая версия ранее не загружена в `third_party`, и зарегистрирует не меньше двух инженеров в роли ВЛАДЕЛЬЦЕВ пакета для его обслуживания. Алиса уговаривает Боба — своего товарища по команде — ей помочь. Никому из них не требуется опыт поддержки сторонних пакетов с отметкой `third_party`, и оба благополучно избегают необходимости что-либо понимать в *реализации* этого пакета. Они приобретают небольшой опыт работы с интерфейсом пакета, используя его для демонстрации проекта перед отпуском.

После этого пакет становится доступным для других команд в Google. Алиса и Боб могут не подозревать, что пакет, который они загрузили и обещали поддерживать, стал популярным. И даже следя за новым прямыми попытками использовать пакет, они могут не заметить расширение его *транзитивного* использования. Если, в отличие от Алисы и Боба, которые использовали внешнюю зависимость только для демонстрации, Чарли добавит эту же зависимость в нашу инфраструктуру поиска, то пакет внезапно превратится из безобидной реализации в очень важную инфраструктуру. Но у нас нет рычагов, которые заставят Чарли насторожиться перед добавлением этой зависимости.

Возможно, это идеальный сценарий: зависимость хорошо написана, не содержит ошибок безопасности и не зависит от других проектов с открытым исходным кодом. *Возможно*, пакет просуществует несколько лет, не обновляясь, хотя это не будет *разумно*: внешние изменения могли бы оптимизировать его, добавить новые особенности или устранить дыры в безопасности до того, как они будут обнаружены CVE¹. Чем дольше существует пакет, тем выше вероятность появления зависимостей (прямых и косвенных). Чем дольше пакет остается стабильным, тем выше вероятность, что мы, согласно закону Хайрама, будем полагаться на его версию, помещенную в каталог `third_party`.

Однажды Алиса и Боб узнают, что пакет нужно обновить. Причиной может быть обнаружение уязвимости в самом пакете или в проекте, который зависит от него. Но Боб перешел на руководящую должность и давно не касался кодовой базы, а Алиса

¹ Common vulnerabilities and exposures — общий перечень уязвимостей и рисков.

перешла в другую команду после демонстрации пакета и больше его не использовала. Однако никто не менял файл OWNERS. К настоящему моменту тысячи проектов зависят от этого пакета, и мы не можем просто удалить его, не сломав сборку Search и других крупных проектов. Никто не имеет опыта работы с деталями реализации этого пакета. Команда, в которой теперь работает Алиса, возможно, не имеет опыта в устранении тонкостей, обусловленных действием закона Хайрама.

Все это говорит о том, что Алисе и другим пользователям пакета предстоит сложное и дорогостоящее обновление под давлением группы безопасности, требующей решить проблему незамедлительно. В этом сценарии обновление охватывает множество выпущенных версий пакета за весь период между первоначальным включением пакета в `third_party` и обнаружением уязвимости.

Наши правила в отношении `third_party` не работают в таких распространенных, к сожалению, сценариях. Мы понимаем, что нам нужен более высокий уровень владения, чтобы одновременно упростить (и сделать более выгодным) регулярное обновление и усложнить потерю владения для пакетов в `third_party`. Сложность заключается в том, что очень трудно объяснить разработчикам, что они не должны использовать пакет, который идеально решает их задачу, потому что у нас нет ресурсов для постоянного обновления его версии во всех наших продуктах. Импортирование проектов, пользующихся большой популярностью и не обещающих поддерживать совместимость (как, например, Boost), сопряжено с большим риском: наши разработчики могут быть хорошо знакомы с использованием такой зависимости для решения задач программирования за пределами Google, но не должны позволить ей укорениться в структуре нашей кодовой базы. Ожидаемая продолжительность жизни нашей кодовой базы составляет несколько десятилетий, и внешние проекты, в которых стабильность не является приоритетом, представляют для нее риск.

Теория управления зависимостями

Теперь, оценив сложность управления зависимостями, обсудим подробнее стоящие перед нами задачи и способы их решения. На протяжении всей этой главы мы снова и снова возвращаемся к формулировке: «Как управлять кодом, получаемым из-за пределов нашей организации (или кодом, который мы не контролируем полностью): как обновлять его, как управлять тем, от чего он зависит?» Мы должны четко понимать, что хорошее решение должно исключать любые противоречивые требования к разным версиям, в том числе конфликты версий в ромбовидных зависимостях, даже в динамической экосистеме, в которую могут добавляться новые зависимости или другие требования (в любой точке сети). Мы также должны учитывать влияние времени: любое ПО содержит ошибки, в частности ошибки безопасности, поэтому будет *критически важно* обновлять некоторые зависимости в течение длительного времени.

Стабильная схема управления зависимостями должна быть гибкой в отношении времени и масштаба: мы не можем предполагать бессрочную стабильность узла в графе

зависимостей и не можем гарантировать, что в код (который мы контролируем или от которого мы зависим) не будут добавляться новые зависимости. Если решение для управления зависимостями предотвращает конфликты между зависимостями, это хорошее решение. Если оно не требует стабильности версий зависимостей или разветвления зависимостей, координации или информирования между организациями и значительных вычислительных ресурсов, то это отличное решение.

Можно выделить четыре основных решения: ничего не менять, семантическое версионирование (SemVer), объединение всего необходимого (координация не для каждого проекта, но для каждого дистрибутива) и «жизнь в главной ветви».

Ничего не менять (модель статической зависимости)

Самый простой способ получить стабильные зависимости — никогда не менять их ни в API, ни в поведении, ни в чем-то еще. Исправления ошибок допускаются, только если от этого не пострадает код, использующий зависимость. Этот подход ориентирован на совместимость и стабильность. Очевидно, что он не идеален из-за предположения бессрочной стабильности. В мире, где не существует проблем безопасности и исправления ошибок, а зависимости никогда не меняются, этот подход выглядит привлекательно: выполнив все требования версии один раз, мы сможем сохранять ее совместимость с кодом бесконечно долго.

Хотя эта модель не является устойчивой в долгосрочной перспективе. Именно ее применяет каждая организация, пока не выяснит, что ожидаемая продолжительность жизни проекта растет. Эта модель хорошо подходит для большинства новых организаций. Сравнительно редко бывает заранее известно, что новый проект просуществует десятилетия и *потребуется* возможность планомерно обновлять зависимости. Гораздо выгоднее надеяться, что стабильность реальна, и сделать вид, что зависимости будут оставаться стабильными в течение первых нескольких лет развития проекта.

Но эта модель *оказывается* ложной по истечении длительного времени, и момент, когда она станет непригодной, наступит внезапно. У нас нет систем заблаговременного предупреждения ошибок безопасности или других критических проблем, которые требуют обновления зависимости, а из-за цепочек зависимостей одно обновление теоретически может заставить организацию обновить всю сеть зависимостей.

Выбор версии в этой модели очень прост: организация выбирает любую подходящую версию.

SemVer

Де-факто современным стандартом управления сетью зависимостей является SemVer (семантическое версионирование)¹. Это почти повсеместная практика

¹ Строго говоря, SemVer не связано с определением требований к совместимости пронумерованных версий. Существует множество мелких вариаций этих требований в разных эко-

представления номеров версий для некоторых зависимостей (особенно библиотек) с использованием трех целых чисел, разделенных точками, например 2.4.72 или 1.1.4. Согласно общепринятым соглашениям, эти три числа представляют старший номер версии, младший номер версии и номер исправления. Изменение старшего номера версии указывает на существенное изменение API, которое может нарушать совместимость с предыдущей версией. Изменение младшего номера указывает на расширение функциональных возможностей без ущерба для совместимости. А изменение номера исправления указывает на изменение деталей реализации (исправление ошибок), не влияющее на API и совместимость.

Семантическое разделение номеров версий предполагает, что требование к версии можно выразить как «нечто более новое, чем», за исключением несовместимых изменений (изменения старшего номера версии). Обычно мы видим требования вида $\text{libbase} \geq 1.5$. Такое требование означает совместимость с любой версией libbase с номером 1.5, включая 1.5.1, и, возможно, с версией 1.6 и выше, но не с libbase 1.4.9 (из-за отсутствия функций, появившихся в версии 1.5) или 2.x (некоторые API в libbase могли претерпеть несовместимые изменения). Изменение старшего номера версии сообщает о существенной несовместимости: поскольку существующие возможности изменились (или были удалены), есть вероятность несовместимости для всех зависимостей. Требования к версии существуют (явно или неявно) всегда, когда одна зависимость использует другую, например « liba требует $\text{libbase} \geq 1.5$ » и « libb требует $\text{libbase} \geq 1.4.7$ ».

Формализовав эти требования, мы можем определить сеть зависимостей как набор программных компонентов (узлов) и требований между ними (ребер). Метки ребер в этой сети изменяются в зависимости от версии исходного узла либо при добавлении (или удалении) зависимостей, либо при обновлении требования из-за изменения исходного узла (которому, например, может потребоваться новая функция, добавленная в зависимость). Поскольку со временем вся эта сеть меняется асинхронно, поиск взаимно совместимого набора зависимостей, удовлетворяющего все транзитивные требования вашего приложения, может оказаться сложной задачей¹. Метод проверки соответствия версий в SemVer действует как решатель задач выполнимости булевых формул (SAT-solvers): он сосредоточен на поиске набора версий для рассматриваемых узлов, удовлетворяющего всем ограничениям, которые задаются требованиями к версиям в ребрах, связывающих зависимости. Большинство экосистем управления пакетами построено на основе таких графов и управляется соответствующими SAT-решателями.

SemVer и SAT-решатели не гарантируют существования решения для данного набора ограничений. Ситуации, когда ограничения зависимостей не могут быть удовлетворены, возникают постоянно, как мы уже видели: если компонент нижнего

системах, но в целом система «номер версии + ограничения», описанная здесь как SemVer, является репрезентативной практикой.

¹ Фактически доказано, что задача применения ограничений SemVer к управлению сетью зависимостей, является NP-полной (<https://research.swtch.com/version-sat>).

уровня (*libbase*) увеличивает старший номер версии, и при этом обновились некоторые (но не все) библиотеки, которые зависят от него (например, *libb*, но не *liba*), появляется ромбовидная зависимость.

Выбор версии при использовании SemVer сводится к запуску алгоритма поиска версий для зависимостей в сети, которые удовлетворяют всем ограничениям. Если версии невозможно распределить, мы называем эту ситуацию «адом зависимостей».

Далее в этой главе мы подробнее рассмотрим некоторые ограничения SemVer.

Модели создания комплексных дистрибутивов

Мы уже несколько десятилетий наблюдаем применение в нашей отрасли следующей мощной модели управления зависимостями: организация формирует набор зависимостей, находит взаимно совместимый набор и выпускает его как коллекцию. Эта модель применяется, например, при создании дистрибутивов Linux — не все компоненты, включенные в дистрибутив, выпускаются одновременно. Фактически многие зависимости нижнего уровня несколько старше зависимостей более высокого уровня из-за разного времени, которое требуется на их интеграцию.

Эта модель «формирования и выпуска коллекции» вводит в управление сетью зависимостей новых участников — дистрибьюторов. Те, кто создает и сопровождает отдельные зависимости, могут ничего или почти ничего не знать о других зависимостях, но *дистрибьюторы*, находящиеся уровнем выше, участвуют в процессе поиска, исправления и тестирования совместимого набора версий, которые необходимо включить в коллекцию. Дистрибьюторами являются инженеры, ответственные за формирование набора версий для объединения, тестирование и решение любых проблем в этом дереве зависимостей.

С точки зрения внешнего пользователя это прекрасная модель при условии, что он полагается только на один дистрибутив. По сути, это модель преобразования сети зависимостей в единую комплексную зависимость с присвоением ей номера версии. Вместо «Я полагаюсь на эти 72 библиотеки с этими версиями» можно сказать: «Я полагаюсь на RedHat версии N» или «Я полагаюсь на граф NPM в момент времени T».

В модели создания комплексных дистрибутивов версии выбирают дистрибьюторы.

Жизнь в главной ветви

Модель, продвигаемая некоторыми гуглерами¹, теоретически разумна, но предъявляет ряд новых требований к компонентам сети зависимостей. Она совершенно не похожа на модели, существующие в современной экосистеме открытого ПО, и пока не совсем ясно, как отрасль могла бы перейти к ней. В такой большой организации, как Google, использование этой модели станет дорогим, но эффективным решением,

¹ В частности, автор и другие члены сообщества C++ в Google.

и мы чувствуем, что оно более или менее верно распределяет большую часть затрат и стимулов. Мы называем эту модель «жизнь в главной ветви» («Live at head»). Ее можно рассматривать как расширение управления зависимостями в модели разработки в главной ветви. Мы расширяем эту модель в той ее части, где говорится о политиках управления версиями применительно к внешним зависимостям.

Жизнь в главной ветви предполагает возможность открепить зависимости, отказаться от SemVer и положиться на поставщиков зависимостей, которые будут тестировать изменения во всей экосистеме перед их фиксацией. Жизнь в главной ветви — это явная попытка избежать затрат на решение проблемы управления зависимостями: всегда зависеть только от текущей версии и никогда не вносить никаких изменений, усложняющих адаптацию потребителей. Изменение, которое существенно (и непреднамеренно) меняет API или поведение, будет обнаружено системой непрерывной интеграцией в зависимостях, расположенных ниже в потоке, и, как результат, отвергнуто при попытке зафиксировать его. В случаях, когда такое изменение действительно необходимо (например, по соображениям безопасности), оно *должно* быть выполнено только после обновления нижестоящих зависимостей или создания автоматизированного инструмента для обновления на месте. (Этот инструмент особенно важен для компаний, выпускающих ПО с закрытым исходным кодом: его цель в том, чтобы дать возможность перейти на использование нового API любому пользователю, не обладающему экспертными знаниями об этом API. Это значительно снижает затраты «сторонних наблюдателей» на внесение существенных изменений.) Такой философский сдвиг в ответственности в экосистеме с открытым исходным кодом трудно провести одновременно: возложение бремени на поставщика API по тестированию и изменению всех его клиентов — это значительное расширение круга обязанностей поставщиков API.

Для подтверждения безопасности изменения в модели «жизнь в главной ветви» используются тесты и системы непрерывной интеграции. Если изменение затрагивает только эффективность или детали реализации, то все тесты будут выполняться успешно, что докажет безопасность изменения и отсутствие разрушительного влияния на пользователей. Изменение, которое меняет синтаксис или семантику экспортируемого API, будет приводить к сотням или даже тысячам ошибок во время тестирования. В этом случае автор изменения должен определить, стоит ли тратить силы и время на устранение этих сбоев ради фиксации изменения. Автор будет взаимодействовать со всеми своими потребителями, чтобы заранее устранить возможные ошибки в тестах (то есть избавиться от хрупких предположений в тестах), и, возможно, создаст инструмент для автоматического выполнения необходимого рефакторинга в как можно большем объеме.

Структура стимулов и технологические допущения в этой модели необычные: мы предполагаем, что существуют юнит-тесты и система непрерывной интеграции, что поставщики API связаны требованием не нарушать работоспособность подчиненных зависимостей и что потребители API обеспечивают успешное выполнение своих тестов, полагаются на свои зависимости поддерживаемыми способами. Такой подход

значительно лучше будет работать в экосистеме с открытым исходным кодом (где исправления могут распространяться заранее), чем в условиях скрытых или закрытых зависимостей. Поставщики API будут стремиться вносить изменения так, чтобы пользователи легко могли мигрировать. Потребители API будут заинтересованы в том, чтобы поддерживать успешное выполнение и полезность тестов.

В подходе «жизнь в главной ветви» выбирается последняя стабильная версия зависимости. Если поставщик будет ответственно подходить к внесению изменений, то зависимость не вызовет сбоев.

Ограничения SemVer

Подход «жизнь в главной ветви» основан на признанных методах управления версиями (разработка в главной ветви), но его практическая ценность в большом масштабе пока не подтверждена. В настоящее время SemVer является фактическим стандартом управления зависимостями, но этот метод имеет свои ограничения, которые мы обсудим ниже.

В определении значений чисел, разделенных точками, на самом деле заложен дополнительный смысл. Они что-то обещают? Или номер версии носит лишь оценочный характер? Что хотят сообщить разработчики `libbase`, когда выпускают новую версию и выбирают для нее старший номер, младший номер и номер исправления? Можно ли утверждать, что переход с версии 1.1.4 на версию 1.2.0 безопасен, потому что новая версия отличается от предыдущей только дополнительными возможностями в API и исправлениями ошибок? Конечно, нет. Пользователи могут применить `libbase` необычными, непредусмотренными способами и в результате столкнуться с проблемами при сборке или с изменением поведения после «простого» расширения API¹. По сути, в отношении совместимости ничего нельзя *утверждать*, рассматривая только исходный код, — нужно еще знать, о совместимости с чем идет речь.

Однако идея «оценки» совместимости начинает ослабевать в контексте сетей зависимостей и SAT-решателей, применяемых к этим сетям. Существует разница между значениями узлов в традиционных SAT и значениями версий в графе зависимостей при использовании SemVer. Узел в графе SAT имеет значение либо `True`, либо `False`. Значение версии (1.1.14) в графе зависимостей является *оценкой* совместимости новой версии для кода, использовавшего предыдущую версию. Логiku совместимости версий мы строим на шатком фундаменте, рассматривая оценки и собственное мнение как абсолютные величины. Как мы увидим далее, даже если эта логика нормально работает в ограниченных случаях, этого недостаточно для поддержания экосистемы в здоровом состоянии.

¹ Например, неудачно реализованное полизаполнение, заранее добавляющее новый API в `libbase`, может создать противоречие. Или использование API отражения языка в зависимости от точного числа API в `libbase` может приводить к сбою при изменении этого числа. Этого не должно происходить, но если это происходит по ошибке, разработчики `libbase` не могут *доказать* совместимость.

Признав, что SemVer является неполной оценкой и представляет лишь часть возможного объема изменений, мы можем начать рассматривать его как грубый инструмент. Теоретически его можно использовать для приблизительной оценки. Но на практике, особенно когда мы строим на его основе SAT-решатели, SemVer может подвести нас, излишне ограничивая и недостаточно защищая от ошибок.

SemVer может чрезмерно ограничивать

Рассмотрим ситуацию, когда `libbase` не является единым монолитом и почти во всех ее библиотеках есть независимые интерфейсы. Даже если в ней всего две функции, легко представить случаи, в которых SemVer будет накладывать на нее чрезмерные ограничения. Допустим, что `libbase` действительно состоит всего из двух функций: `Foo` и `Bar`. Зависимости среднего уровня `liba` и `libb` используют только `Foo`. Если разработчик `libbase` внесет важное изменение в `Bar`, то, следуя правилам SemVer, он должен будет увеличить старший номер версии библиотеки. Известно, что `liba` и `libb` зависят от `libbase 1.x`, и решатели зависимостей на основе SemVer не примут версию `2.x` этой зависимости, хотя в действительности эти библиотеки прекрасно работали бы вместе: изменилась только функция `Bar`, которую `liba` и `libb` не используют. Требование «после добавления несовместимого изменения следует увеличить старший номер версии» влечет потери, когда применяется не на уровне детализации отдельной атомарной единицы API. Некоторые зависимости могут быть достаточно дробными, чтобы изменение номера версии точно отражало ситуацию¹, но в целом это не является нормой для экосистемы SemVer.

Чрезмерное ограничение, накладываемое SemVer и обусловленное неоправданно большим скачком или недостаточно высокой точностью номеров версий, вынуждает диспетчеров пакетов и SAT-решателей сообщать, что зависимости нельзя обновить или установить, даже если они будут точно совместимы. Любой сталкивавшийся с адом зависимостей во время обновления скажет, насколько сильно его раздражает, что большая часть усилий тратится впустую.

SemVer может слишком много обещать

С другой стороны, SemVer заставляет предположить, что оценка совместимости API, сделанная его разработчиком, достаточно точная и что изменения делятся на три группы: нарушающие совместимость (путем изменения или удаления компонентов), добавляющие новые возможности и не влияющие на API. Если считать, что SemVer точно отражает риск изменения путем классификации синтаксических и семантических изменений, то как охарактеризовать изменение, которое добавляет задержку в одну миллисекунду в API, чувствительный ко времени? Как охарактеризовать изменение, меняющее формат записей, выводимых в журнал, порядок импорта внешних зависимостей или порядок результатов, возвращаемых в «неупорядоченном» потоке?

¹ В экосистеме Node есть достойные внимания примеры зависимостей, которые предоставляют ровно один API.

Разумно ли предполагать, что изменения «безопасны» только потому, что они не являются частью синтаксиса или контракта API? А если в документации сказано: «API может измениться в будущем»? Или API имеет имя «ForInternalUseByLibBaseOnlyDoNotTouchThisIReallyMeanIt» (ТолькоДляВнутреннегоИспользованияVLibBaseНеМенятьЯСерьезно)¹?

Идея о том, что исправления в SemVer, в которых изменены только детали реализации, являются «безопасными», абсолютно противоречит опыту компании Google с ее законом Хайрама: «При достаточно большом количестве пользователей каждое наблюдаемое поведение системы будет зависеть от кого-то из них». Изменение порядка импортирования зависимостей или порядка результатов в «неупорядоченном» потоке в большом масштабе неизбежно нарушит предположения, на которые (возможно, ошибочно) опирался пользователь API. Сам термин «критическое изменение» вводит в заблуждение: есть изменения, которые теоретически несовместимы, но безопасны на практике (например, удаление неиспользуемого API). Существуют также изменения, которые теоретически безопасны, но на практике нарушают работоспособность клиентского кода (любой из предыдущих примеров действия закона Хайрама). Это можно увидеть в любой системе, использующей SemVer для управления зависимостями, в которой система требований к номеру версии допускает ограничения на номер исправления: если можно сказать «liba требует libbase > 1.1.14» вместо «liba требует libbase 1.1», это явно указывает на то, что имеются заметные различия в версиях, отличающихся только номером исправления.

Изменение само по себе не является нарушающим или не нарушающим совместимость. Фраза: «Это изменение нарушает совместимость» требует уточнения, для какого (известного или неизвестного) круга пользователей и для каких вариантов использования. Но реальная оценка изменения фактически зависит от ответа на вопрос, которого нет в формулировке SemVer: как пользователи используют зависимость?

Поэтому решатель ограничений SemVer может сообщить, что зависимости совместимы, когда в действительности они не совместимы, потому что увеличение версии было произведено неправильно или что-то в сети зависимостей, согласно закону Хайрама, зависит от чего-то, что не является частью наблюдаемой поверхности API. В этих случаях могут возникать ошибки во время сборки или выполнения с неопределенной верхней границей ущерба.

Мотивация

Есть еще один аргумент, согласно которому SemVer может мешать разработчикам создавать стабильный код. На разработчика произвольной зависимости действует, в той или иной степени, системный стимул — *не* добавлять критических изменений

¹ Как показывает наш опыт, такое именование не решает проблему доступности закрытых API для пользователей. Для этой цели лучше использовать языки, предлагающие более надежные средства управления доступностью публичных и частных API.

и *не* увеличивать старший номер версии. Некоторые проекты уделяют большое внимание совместимости и стараются сделать все возможное, чтобы избежать проблем с изменением старшего номера версии. Другие, напротив, действуют агрессивно и по расписанию намеренно выпускают версии с увеличенным старшим номером. Проблема в том, что у большинства пользователей зависимости нет веских причин интересоваться предстоящими изменениями, они не подписываются на рассылки или другие уведомления о выпуске новых версий.

Таким образом, независимо от количества пользователей, которые столкнутся с неудобствами из-за несовместимого изменения в популярном API, на долю его разработчика приходится лишь крошечная часть затрат, вызванных повышением версии. Разработчики, одновременно являющиеся пользователями, могут иметь стимул к нарушению совместимости: улучшенный интерфейс проще разрабатывать, когда отсутствуют устаревшие ограничения. Это одна из причин, почему мы считаем, что проекты должны публиковать свои намерения в отношении совместимости, порядка использования и критических изменений. Даже если изменения реализованы с максимальной эффективностью, не являются обязательными или игнорируются многими пользователями, сообщения о них все равно помогают оценить, является ли это изменение «стоящим», без привлечения конфликтующих стимулов.

Go (<https://research.swtch.com/vgo-import>) и Clojure (https://oreil.ly/Iq9f_) прекрасно справляются с информированием: в их стандартных экосистемах управления пакетами с увеличением старшего номера версии создается совершенно новый пакет. Согласитесь, если вы решили нарушить обратную совместимость для своего пакета, зачем делать вид, что это тот же набор API? Переупаковка и переименование явно сообщают, что поставщик решил отказаться от обратной совместимости.

И наконец, свою лепту в управление зависимостями вносит человеческий фактор. В общем случае, если следовать правилам SemVer, *семантические* изменения, наравне с синтаксическими, тоже должны приводить к увеличению версии, поскольку изменение поведения API имеет такое же значение, как и изменение его структуры. Теоретически можно разработать инструменты для оценки наличия синтаксических изменений в конкретной версии набора общедоступных API, но оценить наличие значимых и преднамеренных семантических изменений вычислительными средствами не представляется возможным¹. С практической точки зрения даже потенциально возможные инструменты для выявления синтаксических изменений имеют серьезные ограничения. Почти во всех случаях решение об увеличении старшего или младшего номера версии или номера исправления зависит от разработчика API. Если вы используете только несколько зависимостей, поддерживаемых на высоком профессиональном уровне, то вероятность столкнуться с ошибкой инженера,

¹ В мире вездесущих юнит-тестов мы могли бы идентифицировать изменения, требующие изменить поведение теста, но все равно было бы сложно алгоритмически отделить «изменение поведения» от «исправления ошибки в поведении, которое не было задумано или обещано».

скорее всего, будет невысока¹. Но если вы используете сеть из тысяч зависимостей, то будьте готовы встретиться с некоторым беспорядком, обусловленным простыми ошибками разработчика.

Выбор минимальной версии

В 2018 году в серии статей о создании системы управления пакетами для языка программирования Go Расс Кокс описал интересный вариант семантического управления зависимостями — выбор минимальной версии (MVS, minimum version selection) (<https://research.swtch.com/vgo-mvs>). После обновления версии узла в сети зависимостей может потребоваться обновить его зависимости до более новых версий, чтобы удовлетворить изменившееся требование. Это, в свою очередь, может вызвать дальнейшие изменения. В большинстве формулировок удовлетворения ограничений и выбора версии предлагается выбирать самые свежие версии нижестоящих зависимостей. В конце концов, рано или поздно все равно придется обновить зависимости до этих версий, верно?

Метод MVS предлагает делать противоположное: если спецификация `liba` требует `libbase ≥ 1.7`, то нужно попробовать ограничиться обновлением `libbase` до версии 1.7, даже если доступна версия 1.8. Это «создает высокоточные сборки, в которых зависимости, создаваемые пользователем, максимально приближены к использовавшимся автором»². Это утверждение раскрывает критически важную истину: когда `liba` требует `libbase ≥ 1.7`, это почти всегда означает, что разработчик `liba` использовал `libbase 1.7`. Если предположить, что перед публикацией новой версии разработчик провел хотя бы минимальное тестирование³, то у нас есть по меньшей мере отдельные свидетельства проверки совместимости этой версии `liba` с версией `libbase 1.7`. Это не доказательство, полученное системой непрерывной интеграции или полноценным юнит-тестированием, но уже кое-что.

При отсутствии точных входных ограничений, полученных из 100%-но точного предсказания будущего, двигаться вперед лучше небольшими шагами. Мы знаем, что безопаснее отправить в репозиторий результаты работы за час, чем за год, и точно так же безопаснее двигаться небольшими шагами при обновлении зависимостей. Метод MVS предлагает двигаться вперед по каждой зависимости ровно настолько, насколько это необходимо, и говорит: «Итак, я продвинулся достаточно далеко, чтобы получить то, о чем вы просили (и не дальше). Почему бы вам не провести несколько тестов и не проверить, все ли в порядке?»

В основе идеи MVS лежит понимание, что более новая версия может вызвать несовместимость, даже если *в теории* номера версий говорят об обратном. Это признание главной проблемы SemVer: выражение изменений в ПО в виде номеров версий

¹ Поэтому, когда это важно в долгосрочной перспективе, выбирайте хорошо поддерживаемые зависимости.

² Cox R. Minimal Version Selection. February 21, 2018, <https://research.swtch.com/vgo-mvs>.

³ Если это предположение не выполняется, то вам стоит перестать зависеть от `liba`.

неизбежно влечет некоторую потерю точности. Метод MVS дает дополнительную практическую точность, ограничивая выбор версий наиболее близкими к тем, совместимость с которыми предположительно была проверена. Этого может быть достаточно, чтобы обширная сеть зависимостей функционировала должным образом. К сожалению, мы не нашли хорошего способа эмпирически проверить эту идею. Пока еще нет доказательств, что MVS делает SemVer «достаточно хорошим» без решения основных теоретических проблем и проблем со стимулами, но мы по-прежнему считаем, что MVS — это явное улучшение в применении сегодняшних ограничений SemVer.

То есть SemVer работает?

SemVer хорошо работает в ограниченных масштабах. Однако важно понимать, что утверждает этот подход, а что — нет. SemVer будет работать нормально при условии, что:

- поставщики ваших зависимостей точны и ответственны (и разработчики не допускают ошибок при назначении версий);
- ваши зависимости достаточно дробные (чтобы не опасаться чрезмерных ограничений при обновлении неиспользуемых API в них и связанного с этим риска невыполнения требований SemVer);
- все случаи использования API не выходят за пределы ожидаемого использования (чтобы избежать неожиданного нарушения работоспособности из-за предположительно совместимого изменения напрямую или в коде, от которого вы зависите транзитивно).

Когда в графе зависимостей есть лишь несколько тщательно отобранных и хорошо поддерживаемых зависимостей, SemVer может быть идеальным решением.

Однако наш опыт показывает, что *ни одно* из этих трех свойств невозможно масштабировать и поддерживать в течение долгого времени. По мере разрастания сети зависимостей, как по размеру каждой зависимости, так и по их количеству (а также по любым эффектам, обусловленным использованием монолитного репозитория, в котором присутствует нескольких проектов, зависящих от одной сети внешних зависимостей), совокупная потеря точности SemVer будет только увеличиваться. Эти проблемы проявляются как в ложноположительных сбоях (когда теоретически совместимые версии оказываются несовместимыми на практике), так и в ложноотрицательных (когда фактически совместимые версии отвергаются SAT-решателями, что приводит к аду зависимостей).

Управление зависимостями с бесконечными ресурсами

Проведем мысленный эксперимент и посмотрим, как выглядело бы управление зависимостями, если бы все мы имели доступ к неограниченным вычислительным ресурсам? На что можно было бы надеяться, если бы мы были ограничены только

видимостью и слабой координацией между организациями? В настоящее время SemVer популярен в отрасли программной инженерии по трем причинам:

- дает только локальную информация (разработчику API *не нужно* знать детали проектов их пользователей);
- не предполагает наличия тестов (пока тестирование в отрасли выполняется не повсеместно, но мы определенно придем к этому в следующем десятилетии), вычислительных ресурсов для выполнения тестов или систем непрерывной интеграции с целью мониторинга результатов тестирования;
- является существующей практикой.

«Предоставление» локальной информации на самом деле не обязательное, в частности потому, что сети зависимостей, как правило, формируются только в двух средах:

- внутри отдельных организаций;
- внутри экосистемы открытого ПО, где исходный код доступен всем, даже если проекты не сотрудничают явно.

В любом из этих случаев *доступна* важная информация об использовании зависимостей, даже если в настоящий момент она не раскрывается или не используется. То есть доминирование SemVer отчасти обусловлено тем, что мы предпочитаем игнорировать информацию, которая теоретически доступна. Если бы у нас был доступ к большому количеству вычислительных ресурсов и информация о зависимостях была бы легкодоступна, то сообщество наверняка нашло бы ей применение.

От пакета открытого ПО может зависеть бесчисленное множество компонентов с закрытым исходным кодом, однако обычно популярные пакеты открытого ПО популярны среди как открытого, так и закрытого ПО. Сети зависимостей не смешивают (и не могут) бесконтрольно открытые и закрытые зависимости: как правило, есть подмножество открытых зависимостей и отдельный закрытый подграф¹.

Теперь вспомним *намерение* SemVer: «По моей оценке, это изменение будет легко (или сложно) принять». Есть ли лучший способ передать эту информацию? Да, в виде практического опыта, демонстрирующего, что изменение совместимо. Как получить такой опыт? Если большая часть (или, по крайней мере, репрезентативная выборка) наших зависимостей общедоступна, каждое предлагаемое изменение будет неизменно тестироваться. При достаточно большом количестве таких тестов у нас будет, по крайней мере, статистический аргумент в пользу безопасности изменения (по закону Хайрама). Если тесты по-прежнему выполняются успешно, то изменение совместимое и не имеет значения, изменило ли оно структуру API, исправило ли ошибки или сделало и то и другое — нам не нужно ни классифицировать, ни оценивать это изменение.

Теперь представьте, что экосистема открытого ПО оказалась в мире, где изменения сопровождаются доказательствами их безопасности. Если не учитывать затраты на

¹ Потому что сеть зависимостей в открытом ПО обычно не может зависеть от группы закрытых узлов, разве только от встроенного закрытого ПО, управляющего графикой.

вычисления, *истинная*¹ оценка «насколько это безопасно» будет зависеть только от выполнения соответствующих тестов в зависимых проектах.

Даже без формальной непрерывной интеграции, применяемой ко всей экосистеме открытого ПО, мы можем использовать такой граф зависимостей и другие вторичные сигналы для более целенаправленного анализа изменений. Первостепенное внимание в таком случае должно уделяться тестам в зависимостях, которые широко используются, имеют хорошую профессиональную поддержку, систематически предоставляют хороший сигнал и высококачественные результаты тестирования. Помимо особого внимания к тестам, которые могут дать больше всего экспериментальной информации о качестве изменений, можно использовать информацию от авторов изменений, чтобы оценить риск и выбрать подходящую стратегию тестирования. Выполнение «всех соответствующих» тестов теоретически необходимо, если цель состоит в том, чтобы убедиться, что «ни одна чья-то зависимость не изменяется с нарушением совместимости». Если цель состоит в том, чтобы «уменьшить риск», статистический аргумент становится более привлекательным (и наименее затратным).

В главе 12 мы определили четыре разновидности изменений: от простого рефакторинга до изменения существующей функциональности. Учитывая модель на основе непрерывной интеграции для обновления зависимостей, мы можем начать отображать эти разновидности изменений в модели, подобной SemVer, используя которую автор изменения оценивает риск и применяет соответствующий уровень тестирования. Например, простой рефакторинг, изменяющий только внутренние API, можно отнести к категории с самым низким риском и для его проверки достаточно выполнить тесты только в нашем проекте и, возможно, в некоторых других важных проектах, напрямую зависящих от изменения. С другой стороны, изменение, которое удаляет устаревший интерфейс или изменяет наблюдаемое поведение, может потребовать столько тестирования, сколько вообще мы сможем себе позволить.

Что необходимо изменить в экосистеме открытого ПО, чтобы применить такую модель? К сожалению, довольно много:

- Все зависимости должны содержать юнит-тесты. Мы неумолимо движемся к миру, в котором юнит-тестирование будет использоваться повсеместно.
- Сеть зависимостей для большей части экосистемы открытого ПО понятна. Но неясно, есть ли механизм для выполнения графовых алгоритмов в этой сети — информация *открыта* и *доступна*, но фактически никак не индексируется или не используется. Многие системы управления пакетами или зависимостями позволяют видеть зависимости проекта, но не отображают, какие проекты зависят от тех или иных зависимостей.
- Объем вычислительных ресурсов, доступных для непрерывной интеграции, все еще очень ограничен. Большинство разработчиков не имеют вычислительных кластеров для сборки и тестирования.

¹ Или близкая к истинной.

- Зависимости часто выражаются как фиксированные. Разработчик `libbase` не может экспериментально проверить совместимость своих изменений с использованием тестов для `liba` и `libb`, если они явно зависят от конкретной фиксированной версии `libbase`.
- Мы могли бы явно включить историю и рейтинг в результаты непрерывной интеграции. Потеря работоспособности из-за предложенного изменения в проекте с длинной историей успешного тестирования выглядит намного более весомым аргументом для поиска ошибки, чем сбой в проекте, появившемся недавно и имеющем историю сбоев тестирования по несвязанным причинам.

В связи с этим возникает вопрос: с какими версиями каждой из зависимостей в сети вы тестируете изменения перед отправкой в репозиторий? Для тестирования всех комбинаций всех хронологических версий потребуются поистине ошеломляющий объем вычислительных ресурсов, даже по меркам Google. Наиболее очевидным упрощением политики выбора версии могло бы быть «тестирование текущей стабильной версии» (в конце концов, целью является разработка в главной ветви). Таким образом, модель управления зависимостями при неограниченных ресурсах фактически является моделью «жизни в главной ветви». Остается только определить, сможет ли эта модель эффективно применяться при использовании более реалистичного, ограниченного объема ресурсов и готовы ли разработчики API взять на себя больше ответственности за тестирование практической безопасности своих изменений. Выявление мест, где недостаточность ресурсов приводит к чрезмерному упрощению трудно вычислимой истины, продолжает оставаться полезным упражнением.

Экспорт зависимостей

До сих пор мы говорили только о входящих зависимостях, то есть о зависимостях от ПО, написанного другими людьми. Но стоит подумать также о том, как мы создаем ПО, которое можно *использовать* в качестве зависимости. Это выходит за рамки простой механики упаковки ПО и его выгрузки в репозиторий: мы должны подумать о выгодах, затратах и рисках предоставления ПО как для нас самих, так и для его потенциальных пользователей.

Есть два основных пути, по которым безобидный и, надеюсь, благотворительный акт, такой как «открытие исходного кода библиотеки», может стать возможной потерей для организации. Во-первых, это может стать тормозом для репутации вашей организации, если реализация не отличается высоким качеством или не поддерживается должным образом. Как говорится в сообществе Apache, на первое место мы должны ставить «сообщество, а не код». Даже если вы выпускаете отличный код, но избегаете участия в жизни сообщества, это все равно может нанести вред вашей организации и более широкому сообществу. Во-вторых, открытие исходного кода из благих побуждений может ухудшить эффективность разработки, если вы не сможете удержать баланс. Со временем поддержка ответвлений станет слишком дорогостоящей.

Пример: открытие исходного кода gflags

В качестве примера ущерба репутации рассмотрим случай из истории Google, произошедший примерно в 2006 году, когда мы открыли исходный код наших библиотек управления флагами командной строки на C++. Конечно, вклад в сообщество открытого исходного кода — это исключительно хороший поступок, который ничем плохим нам не грозит, верно? К сожалению, нет. Множество причин как будто сговорились превратить этот хороший поступок во что-то, что точно навредило нашей репутации и, возможно, навредило сообществу открытого ПО:

- В то время у нас не было возможности выполнять крупномасштабный рефакторинг, поэтому весь код, использовавший эту библиотеку, должен был оставаться неизменным — мы не могли переместить код в новое место в кодовой базе.
- Мы разделили наш репозиторий на «код, разработанный внутри организации» (который можно свободно копировать для создания производных при правильном подходе к переименованию) и «код, использование которого может иметь юридические/лицензионные последствия» (имеющий некоторые ограничения на использование).
- Если проект открытого ПО принимает код от внешних разработчиков, это, как правило, влечет юридические проблемы: автор проекта *не владеет* этим кодом, он лишь имеет право использовать его.

В результате проект gflags был обречен превратиться либо «в отрезанный ломоть», либо в отдельную производную. Исправления, предлагаемые проекту, невозможно было включить в исходный код внутри Google, мы не могли переместить проект в пределах нашего монолитного репозитория, потому что еще не освоили эту форму рефакторинга, и мы не могли использовать открытую версию как зависимость в своем внутреннем коде.

Кроме того, как и в большинстве организаций, наши приоритеты менялись со временем. Примерно в то же время мы заинтересовались продуктами, выходящими за рамки нашего традиционного пространства (веб-приложения, поиск), такими как Google Earth, который имел более традиционный механизм распространения в виде предварительно скомпилированных двоичных файлов для различных платформ. В конце 2000-х было необычным, хотя и не единичным случаем, когда библиотека из нашего монолитного репозитория, особенно такая низкоуровневая, как gflags, использовалась на различных платформах. С течением времени и ростом Google наше внимание сузилось до такой степени, что стало крайне редким явлением создание какой-либо библиотеки с использованием чего-либо, кроме нашей собственной настроенной инструментальной цепочки, с последующим развертыванием в нашем производственном парке. Решение проблем «переносимости» для правильной поддержки открытого проекта, такого как gflags, стало почти невозможным: наши внутренние инструменты просто не поддерживали эти платформы, и нашему среднему разработчику не приходилось взаимодействовать с внешними инструментами. Это была постоянная борьба за сохранение переносимости.

Первоначальные авторы и сторонники открытого ПО постепенно переходили в другие компании или в другие команды, и настал момент, когда никто изнутри больше не поддерживал открытый проект `gflags`. Учитывая, что проект не являлся основной работой конкретной команды и никто не мог сказать, почему важно продолжать развивать его, мы фактически позволили ему начать гнить снаружи¹. Внутренняя и внешняя версии со временем расходились все дальше друг от друга, и, в конце концов, некоторые внешние разработчики создали свой проект на основе внешней версии и уделили ему должное внимание.

Кроме: «О, смотрите, Google внесла свой вклад в мир открытого ПО», ничто из перечисленного не укрепляло нашу репутацию, и, тем не менее, все это имело смысл, если учесть приоритеты нашей организации. Те, кто был близок к этому, надежно усвоили урок: «Не выпускайте ничего, если не планируете поддерживать это в долгосрочной перспективе». Усвоили ли этот урок все инженеры Google — еще предстоит выяснить. Мы слишком большая организация.

Помимо туманного «мы плохо выглядим» эта история также показывает, как легко столкнуться с техническими проблемами из-за некачественно подготовленных и плохо поддерживаемых внешних зависимостей. Библиотека `gflags` была общедоступной, хотя и игнорировалась, но в Google все еще оставались проекты с открытым исходным кодом или проекты, которые должны были оставаться доступными за пределами нашей экосистемы монолитного репозитория. Неудивительно, что авторы этих других проектов смогли выявить² общее подмножество API для внутренней и внешней версий библиотеки. Поскольку это общее подмножество почти не менялось в течение длительного времени, оно постепенно превратилось в «лучший фундамент» для тех редких команд, которые имели необычные требования к переносимости примерно в период с 2008 по 2017 год. Их код мог встраиваться как во внутреннюю, так и во внешнюю экосистему, отключая производные версии библиотеки `gflags`, в зависимости от среды.

Затем по не связанным с этим причинам команды разработчиков библиотек на C++ начали изменять наблюдаемые, но не документированные части внутренней реализации `gflags`. В этот момент все, кто зависел от стабильности и эквивалентности неподдерживаемой внешней версии, начали кричать, что их сборки и версии внезапно сломались. Возможность оптимизации ценой нескольких тысяч совокупных процессоров во всем парке Google значительно задержалась, но не потому, что было сложно обновить API, от которого зависело 250 миллионов строк кода, а потому, что небольшая горстка проектов полагалась на непредсказуемое и неожиданное поведение. И снова проявилось действие закона Хайрама, в данном случае даже в отношении разветвленных API, поддерживаемых разными организациями.

¹ Нельзя сказать, что это правильно или мудро, просто как организация мы позволяем чему-то ускользать от нашего внимания.

² Часто методом проб и ошибок.

КЕЙС: APPENGINE

Еще более наглядный пример, как мы подвергаемся риску неожиданной технической зависимости, — публикация службы Google AppEngine. Эта служба дает пользователям возможность писать свои приложения поверх существующей инфраструктуры на одном из нескольких популярных языков программирования. Если приложение придерживается правильной модели управления хранилищем и состоянием, служба AppEngine способна масштабировать его до огромной величины: внутреннее хранилище и внешний интерфейс управляются и клонируются по мере необходимости производственной инфраструктурой Google.

Первоначально поддержка Python в AppEngine представляла собой 32-битную сборку, работающую со старой версией интерпретатора. Сама система AppEngine была (конечно) реализована в нашем монолитном репозитории и собиралась вместе с нашими общими инструментами на Python и C++. В 2014 году мы начали процесс существенного обновления среды выполнения Python вместе с нашим компилятором C++ и стандартной библиотекой, в результате чего мы эффективно связали «код, который создается с помощью текущего компилятора C++», с «кодом, использующим обновленную версию Python». Этот проект обновил одновременно две зависимости. Для большинства проектов это не было проблемой. А для нескольких проектов, из-за пограничных ситуаций и действия закона Хайрама, наши эксперты по языковым платформам провели исследование и отладку, чтобы облегчить переход. Однако разработчики AppEngine обнаружили ужасающее проявление закона Хайрама в практической плоскости бизнеса. Многие пользователи — наши клиенты, которые платят деньги, — не смогли (или не захотели) обновиться. Кто-то не хотел переходить на более новую версию Python, кто-то не мог позволить себе изменить потребление ресурсов, связанное с переходом с 32-битной версии Python на 64-битную. Поскольку некоторые наши клиенты платили весьма значительные суммы за услуги AppEngine, команда AppEngine смогла убедительно обосновать необходимость отсрочки принудительного перехода на новые версии языка и компилятора. По сути это означало, что каждый фрагмент кода на C++ в транзитивном замыкании зависимостей от AppEngine должен быть совместим со старыми версиями компилятора и стандартной библиотеки: любые исправления ошибок или оптимизации производительности, которые могли быть внесены в эту инфраструктуру, должны были сохранять совместимость между версиями. Такая ситуация сохранялась почти три года.

При наличии достаточного количества пользователей кто-то обязательно станет полагаться на некое «наблюдаемое» проявление вашей системы. Мы в Google ограничиваем своих внутренних пользователей рамками нашего технологического стека и обеспечиваем прозрачность с помощью монолитного репозитория и систем индексации кода, поэтому нам гораздо проще гарантировать возможность добавления полезных изменений. Когда мы переходим от управления исходным кодом к управлению зависимостями и теряем представление о том, как используется код, или подчиняемся конкурирующим приоритетам внешних групп (особенно тех, которые нам платят), становится гораздо труднее идти на чисто инженерные компромиссы. Выпуск API любого рода порождает возможность появления конкурирующих приоритетов и непредвиденных ограничений с внешней стороны. Это не означает, что вам не следует выпускать API, а лишь предупреждает, что обслуживание внешних пользователей API обходится намного дороже, чем внутренних.

Совместное использование кода с внешним миром, будь то версия с открытым или с закрытым исходным кодом, — это не просто вопрос благотворительности (в случае открытого ПО) или бизнес-возможностей (в случае закрытого ПО). Зависящие пользователи в разных организациях с разными приоритетами, которых вы не можете контролировать, в итоге начнут действовать по закону Хайрама в отношении этого кода. Если вы работаете с большими временными рамками, невозможно точно предсказать набор необходимых или полезных изменений, которые могут стать ценными. Принимая решение о выпуске чего бы то ни было, помните о долгосрочных рисках: изменение внешних общих зависимостей часто обходится намного дороже с течением времени.

Заключение

Управление зависимостями — сложная задача, и мы продолжаем искать решения для управления сложными поверхностями API и сетями зависимостей, где разработчики зависимостей обычно не предполагают координации. Фактическим стандартом управления сетью зависимостей является SemVer, предоставляющее сжатую (с потерями) информацию о возможном риске принятия какого-либо изменения. SemVer предполагает возможность предсказать серьезность изменения без учета информации о том, как используется рассматриваемый API. Однако SemVer достаточно хорошо работает в малых масштабах и еще лучше при использовании подхода MVS. По мере расширения сети зависимостей проблемы с законом Хайрама и потеря точности в SemVer делают управление выбором новых версий все более сложным.

Однако мы постепенно движемся к миру, в котором оценки совместимости, предоставляемые разработчиками (семантические номера версий), будут заменены на доказательства, основанные на опыте, — результаты тестирования зависимых пакетов. Если разработчики API возьмут на себя больше ответственности за тестирование своих изменений совместно с пользователями и четко объявят, какие типы изменений ожидаются, у нас появится возможность создавать сети зависимостей с более высокой точностью в еще большем масштабе.

Итоги

- Старайтесь заменить задачи управления зависимостями задачами управления версиями: если есть возможность создать больше кода внутри своей организации, то это существенно упростит прозрачность и координацию зависимостей.
- Добавление зависимости в проект разработки ПО обходится дорого, и установление постоянных доверительных отношений является сложной задачей. Импорт зависимостей в вашу организацию должен выполняться осторожно, с учетом текущих затрат на поддержку.
- Зависимость — это контракт, компромисс, согласно которому производители и потребители имеют определенные права и обязанности. Производители должны

четко понимать, что они дают обещание, которое придется выполнять с течением времени.

- SemVer — это сжатая (с потерями) оценка, выражающая, «насколько рискованно изменение». SemVer с SAT-решателем в диспетчере пакетов интерпретирует эту оценку как абсолютную. Это может приводить к ограничению, чрезмерному (ад зависимостей) или недостаточному (когда версии, которые должны работать вместе, оказываются несовместимыми).
- Тестирование и непрерывная интеграция предоставляют фактическое свидетельство совместимости и работоспособности нового набора зависимостей.
- Политики обновления до минимальной версии в SemVer и управления пакетами наиболее точны. Они все еще зависят от способности людей точно оценивать риск увеличения версии, но значительно повышают вероятность, что эксперт верно оценит связи между производителем и потребителем API.
- Юнит-тестирование, непрерывная интеграция и (недорогие) вычислительные ресурсы могут изменить подход к управлению зависимостями. Для такого фазового перехода требуется фундаментально изменить взгляд на проблему управления зависимостями в отрасли, а также повысить ответственность производителей и потребителей.
- Предоставление зависимости не проходит бесследно: простой подход «передать и забыть» может стоить вам репутации и стать проблемой для совместимости. Необходимость поддерживать стабильность может ограничить вам широту выбора дальнейших действий и сузить круг внутреннего использования. Поддержка без стабильности может подорвать ваш престиж или подвергнуть вас риску со стороны важных потребителей, зависящих от чего-то в соответствии с законом Хайрама и нарушающих ваши «нестабильные» планы.

Крупномасштабные изменения

Автор: Хайрам Райт

Редактор: Лиза Кэри

Подумайте немного о своей кодовой базе. Сколько файлов вы сможете надежно обновить за одну фиксацию? Какие факторы ограничивают это число? Вы когда-нибудь пробовали выполнить настолько крупное изменение? Сможете ли вы провести его в разумные сроки в чрезвычайной ситуации? Как соотносится размер самой большой выполненной вами фиксации с фактическим размером вашей кодовой базы? Как бы вы протестировали такое изменение? Скольким людям потребуется оценить это изменение, прежде чем оно будет принято? Сможете ли вы отменить это изменение после фиксации? Ответы на эти вопросы могут вас удивить (и то, что вы *думаете* о них, и то, что они на самом деле значат для вашей организации).

Мы в Google давно отказались от идеи внесения в кодовую базу атомарных изменений большого объема. Как показывает наш опыт, с ростом кодовой базы и числа работающих с ней инженеров максимально возможное атомарное изменение, как ни странно, *уменьшается*, потому что затрудняется выполнение предварительных проверок и тестов для него и теряется возможность гарантировать перед его отправкой актуальность каждого его файла. Поскольку вносить радикальные изменения в нашу кодовую базу становится все труднее, но мы стремимся постоянно улучшать базовую инфраструктуру, нам пришлось разработать новые способы обоснования крупномасштабных изменений и способов их реализации.

В этой главе мы поговорим о методах, как социальных, так и технических, которые позволяют поддерживать гибкость большой кодовой базы в Google и реагировать на изменения в базовой инфраструктуре. Мы рассмотрим несколько реальных примеров использования этих методов. Ваша база кода может отличаться от кодовой базы в Google, но понимание описываемых здесь принципов и их адаптация помогут вашей софтверной организации расширяться и иметь возможность вносить существенные изменения в кодовую базу.

Что такое крупномасштабное изменение?

Сначала определим, какие изменения считаются крупномасштабными. По нашему опыту, крупномасштабное изменение — это любой набор изменений, которые логически связаны, но практически не могут быть представлены в виде атомарной

единицы. Это может быть обусловлено вовлечением в изменение настолько большого количества файлов, что базовый инструмент не может зафиксировать их все сразу, или настолько большим объемом изменений, что попытка их слияния вызывает конфликты. Часто принадлежность изменений к категории крупномасштабных определяется топологией репозитория: если организация использует коллекцию распределенных или объединенных репозиториях¹, то внесение атомарных изменений в них может оказаться технически невозможным². Подробнее о препятствиях для атомарных изменений далее в этой главе.

В Google крупномасштабные изменения почти всегда создаются с использованием автоматизированных инструментов. Такие изменения делятся на несколько основных категорий:

- устранение типичных антипаттернов с помощью инструментов анализа всей кодовой базы;
- замена вызовов устаревших библиотечных функций;
- добавление улучшений низкоуровневой инфраструктуры, например обновление компилятора;
- перевод пользователей со старой системы на новую³.

Количество инженеров, работающих над этими задачами, может быть небольшим, но их клиентам полезно иметь представление об инструментах и процессе крупномасштабных изменений. По своей природе крупномасштабные изменения затрагивают большое количество клиентов, а инструменты, используемые для этого, легко масштабируются вниз и могут использоваться командами, вносящими всего несколько десятков связанных изменений.

За конкретными крупномасштабными изменениями могут стоять более широкие изменения. Например, новый стандарт языка ввел более эффективную идиому для выполнения некоторой задачи, или интерфейс внутренней библиотеки изменился, или новая версия компилятора потребовала исправления существующих проблем, которые она интерпретирует как ошибки. В Google большинство крупномасштабных изменений на самом деле практически не влияют на функциональные возможности. Как правило, это обширные изменения в тексте, цель которых — увеличение ясности, оптимизация или улучшение совместимости в будущем. Но не все крупномасштабные изменения сохраняют поведение кодовой базы.

В базах кода, по размерам сопоставимых с Google, командам разработчиков инфраструктуры может потребоваться регулярно менять сотни тысяч отдельных ссылок на

¹ Некоторые идеи, помогающие понять причины, описаны в главе 16.

² В таком объединенном мире можно сказать: «Мы просто будем выполнять фиксации в каждый репозиторий как можно быстрее, чтобы продолжительность перерывов в сборке была небольшой!» Но этот подход действительно не масштабируется с ростом числа объединенных репозиториях.

³ Подробнее об этой практике в главе 15.

устаревший паттерн или символ. Пока в самых масштабных случаях мы затрагивали миллионы ссылок и надеемся, что процесс будет и дальше хорошо масштабироваться. Мы давно поняли, что выгодно как можно раньше вкладывать средства в инструменты поддержки крупномасштабных изменений. Мы также поняли, что эффективные инструменты помогают инженерам вносить небольшие изменения. Инструменты, позволяющие эффективно изменять тысячи файлов, достаточно хорошо масштабируются вниз до уровня десятков файлов.

Кто занимается крупномасштабными изменениями?

В основном с крупномасштабными изменениями работают команды, занимающиеся инфраструктурой, которые создают наши системы и управляют ими. Но инструменты и ресурсы доступны всем инженерам в компании. Если вы пропустили главу 1, то могли бы спросить, почему эту работу выполняют инфраструктурные команды. Почему нельзя просто добавить новый класс, функцию или систему и потребовать от всех, кто пользуется старой версией, перейти на обновленный аналог? С практической точки зрения это может выглядеть проще, но, как оказывается, этот подход плохо масштабируется по нескольким причинам.

Во-первых, инфраструктурные команды, которые создают базовые системы и управляют ими, обладают знаниями в предметной области, необходимыми для исправления сотен тысяч ссылок. Команды, использующие инфраструктуру, вряд ли будут обладать достаточным уровнем компетенции для выполнения множества миграций, и нет смысла полагаться на повторное изучение опыта, который уже имеется у инфраструктурных команд. Централизация внесения изменений также позволяет быстрее восстанавливать работоспособность после ошибок, потому что ошибки обычно делятся на известные категории, и команда, выполняющая миграцию, может иметь сценарий — формальный или неформальный — их устранения.

Представьте, сколько времени потребуется на выполнение первого из серии полумеханических изменений, которые вы не понимаете. Вероятно, вы потратите время на знакомство с обоснованием и характером изменения, найдете простой пример, попытаетесь понять, как в нем применяются предоставленные предложения, а затем применить их к своему коду. Повторение этого процесса в каждой команде в организации значительно увеличит общие затраты. Организовав небольшое число централизованных команд, ответственных за крупномасштабные изменения, мы в Google одновременно снизили эти затраты и обеспечили возможность проводить такие изменения более эффективно.

Во-вторых, никому не нравится выполнять неоплачиваемую работу¹. Новая система может быть многократно лучше той, которую она заменяет, но эти преимущества ча-

¹ Под «неоплачиваемой работой» мы подразумеваем «выполнение дополнительных внешних требований без компенсации трудозатрат». К ней относится, например, требование носить вечерние платья по пятницам без прибавки к зарплате для покупки этих платьев.

сто распространяются по всей организации и потому вряд ли будут иметь достаточно большое значение для отдельных команд, которые пожелают выполнить обновление самостоятельно. Если новая система достаточно важна, затраты на миграцию будут нести разные подразделения. Централизованная миграция почти всегда проходит быстрее и обходится значительно дешевле, чем естественная миграция отдельных команд.

Кроме того, наличие команд; владелец систем помогает согласовать стимулы для осуществления крупномасштабных изменений в этих системах. Как показывает наш опыт, естественные миграции редко оказываются полностью успешными, отчасти потому, что при разработке нового кода инженеры склонны опираться на примеры в существующем коде. Наличие команды, заинтересованной в удалении старой системы и ответственной за миграцию, помогает гарантировать успех миграции. Комплектование и финансирование команды для такого рода миграций может выглядеть как дополнительные расходы, но на самом деле этот шаг сужает влияние внешних эффектов, которые создает неоплачиваемая работа, и дает дополнительные преимущества масштабирования.

КЕЙС: ЗАПОЛНЕНИЕ ДОРОЖНЫХ ВЫБОИН

Системы поддержки крупномасштабных изменений используются в Google в основном для высокоприоритетных миграций, но мы также обнаружили, что их доступность открывает широкие возможности для различных мелких исправлений в нашей кодовой базе. Подобно дорожным организациям, которые строят новые дороги и ремонтируют старые, инфраструктурные команды в Google занимаются не только разработкой новых систем и миграцией пользователей на них, но также много времени тратят на исправление существующего кода.

Например, в начале нашей истории появилась библиотека шаблонов, дополняющая стандартную библиотеку шаблонов C++. Эта библиотека, которую мы назвали Google Template Library, состояла из нескольких заголовочных файлов, требующих реализации. По причинам, затерянным в тумане времени, один из этих файлов заголовков назывался `stl_util.h`, а другой — `map-util.h` (обратите внимание на разные разделители в именах файлов). Эта разница не только волновала сторонников последовательности, но и влекла снижение продуктивности: инженерам приходилось запоминать, в каком файле какой разделитель использовался, и обнаруживать ошибку после потенциально длительного цикла компиляции.

Исправление единственного символа может показаться сложной задачей, особенно в такой кодовой базе, как в Google, но зрелость наших инструментов и процессов крупномасштабных изменений позволили нам сделать это всего за пару недель в фоновом режиме. Авторы библиотеки могли находить и применять это изменение в массовом порядке, не беспокоя конечных пользователей. В результате мы сумели уменьшить количество сбоев во время сборки, вызванных этой проблемой. Повышение продуктивности (и удовлетворенности) с лихвой окупил время, потраченное на внесение изменений.

Одновременно с совершенствованием возможности внесения изменений во всю нашу кодовую базу росло разнообразие изменений, и теперь мы можем принимать некоторые инженерные решения, будучи уверенными, что нам удастся изменить их в будущем. Иногда стоит приложить усилия, чтобы заполнить несколько выбоин.

Препятствия к атомарным изменениям

Прежде чем приступить к обсуждению фактического процесса крупномасштабных изменений, используемого в Google, мы должны поговорить о том, почему многие виды изменений нельзя зафиксировать атомарно. В идеальном мире все логические изменения можно упаковать в одну атомарную фиксацию, которую можно протестировать, оценить и зафиксировать независимо от других изменений. К сожалению, с ростом репозитория и числа работающих с ним инженеров этот идеал становится все менее достижимым даже в небольшом масштабе, в котором используется набор распределенных или объединенных репозиторияев.

Технические ограничения

Начнем с того, что операции большинства VCS линейно масштабируются с размером изменения. Система может нормально обрабатывать небольшие фиксации (например, затрагивающие около десятков файлов), но ей может не хватить памяти или вычислительной мощности для атомарной фиксации тысяч файлов. В централизованных VCS фиксации могут блокировать другие операции, выполняющие запись (а в старых системах — и чтение), то есть большие фиксации задерживают других пользователей.

Таким образом, иногда невозможно вносить большие изменения атомарно в конкретной инфраструктуре. Деление большого изменения на более мелкие независимые фрагменты позволяет обойти эти ограничения, но усложняет процесс изменения¹.

Конфликты слияния

По мере увеличения размера изменения растет вероятность конфликтов слияния. Все известные нам VCS требуют обновления и слияния, возможно, проводимых вручную, если в центральной репозитории хранится более новая версия файла. По мере увеличения количества файлов в изменении (и инженеров, работающих с репозиторияем) вероятность столкнуться с конфликтом слияния возрастает.

В небольшой компании можно незаметно (в выходные) внести изменение, которое затронет каждый файл в репозитории. В некоторых компаниях даже может иметься неформальная система захвата глобальной блокировки репозитория путем передачи виртуального (или даже физического!) ключа команде разработчиков. Но в такой большой компании, как Google, подобные решения просто невозможны: кто-то всегда вносит изменения в репозиторий.

При небольшом количестве изменяемых файлов вероятность конфликтов слияния уменьшается, поэтому эти файлы с большей вероятностью будут зафиксированы без проблем. Это свойство также справедливо для следующих областей.

¹ См. <https://ieeexplore.ieee.org/abstract/document/8443579>.

Никаких кладбищ с привидениями

В командах, отвечающих за надежность производственных служб Google, есть мантра: «Никаких кладбищ с привидениями». В данном случае под кладбищем с привидениями подразумевается система настолько старая и сложная, что никто не осмеливается в нее войти. Это может быть критически важная для бизнеса система, остановленная в развитии, потому что любая попытка ее изменения может привести к сбою, который будет стоить бизнесу реальных денег. Часто она представляет риск для существования бизнеса и может потреблять чрезмерный объем ресурсов.

Однако кладбища с привидениями существуют не только в производственных системах — их можно найти в базах кода. У многих организаций есть устаревшие и не поддерживаемые программные компоненты, написанные кем-то давно ушедшим из команды, и лежащие на пути к некоторым важным функциям, приносящим доход. Эти системы также остановлены в развитии и похоронены под пластами бюрократических запретов, чтобы предотвратить изменения, которые могут вызвать нестабильность. Никто не хочет быть сапером, который перережет не тот проводок!

Эти части кодовой базы исключены из процесса крупномасштабных изменений, потому что препятствуют массовым миграциям, выводу из эксплуатации других систем, на которые они полагаются, или обновлению используемых компиляторов и библиотек.

Мы в Google обнаружили, что появлению таких кладбищ прекрасно противодействует наше любимое тестирование. Имея всеобъемлющие и исчерпывающие тесты, мы можем вносить в ПО любые изменения, пребывая в уверенности, что ничего не нарушим независимо от старости или сложности системы. Написание тестов требует больших усилий, но позволяет кодовой базе, такой масштабной, как в Google, развиваться в течение длительного времени, не превращаясь в дом с призраками.

Неоднородность

Крупномасштабные изменения действительно возможны, только когда большую часть работы по их применению выполняют компьютеры, а не люди. Насколько хорошо люди способны принимать неоднозначные решения, настолько же хорошо компьютеры справляются с правильным применением преобразований в нужных местах в согласованной среде. Если в вашей организации используется множество разных VCS, систем непрерывной интеграции и инструментов для конкретного проекта или руководств по форматированию, то вам будет сложно внести радикальные изменения во всю кодовую базу. Упрощение окружения для большей согласованности поможет не только инженерам, но и роботам выполнять преобразования.

Например, во многих проектах в Google настроено выполнение тестов перед отправкой изменений в кодовую базу. Эти тесты могут быть очень сложными и проверять самые разные аспекты — от присутствия новых зависимостей в белом списке

до проверки связи изменения с ошибкой. Многие из этих проверок актуальны для команд, пишущих новые функции, но для крупномасштабных изменений они лишь добавляют ненужные сложности.

Мы решили смириться со сложностями, такими как выполнение тестов перед отправкой, сделав сложные процедуры стандартными для всей кодовой базы, но рекомендуем командам опускать специальные проверки, когда крупномасштабные изменения затрагивают какие-то части их проектов. Большинство команд рады помочь в проведении крупномасштабных изменений, которые принесут пользу для их проектов.



К автоматизированным инструментам также применимы многие преимущества согласованности для инженеров (глава 8).

Тестирование

Всякое изменение должно тестироваться (об этом мы поговорим чуть ниже), но чем крупнее изменение, тем сложнее протестировать его надлежащим образом. Система непрерывной интеграции в Google будет запускать не только тесты, напрямую связанные с измененными файлами, но и любые тесты, транзитивно зависящие от них¹. То есть изменение будет охвачено тестами максимально широко, но чем дальше в графе зависимостей находится тест от файлов, затронутых изменением, тем ниже вероятность, что неудача при тестировании будет обусловлена самим изменением.

Небольшие независимые изменения легче проверить, потому что каждое из них влияет на меньший набор тестов, а также потому, что обнаруженные в них ошибки легче диагностировать и исправлять. Найти основную причину неудачи тестирования при изменении 25 файлов довольно просто, но поиск одного файла среди 10 000 сродни поиску иголки в стоге сена.

Однако небольшие изменения ведут к тому, что одни и те же тесты будут выполняться несколько раз, особенно тесты, которые зависят от больших разделов в кодовой базе. Поскольку время, которое инженеры тратят на поиск ошибок, выявленных при тестировании, намного дороже процессорного времени, необходимого для выполнения дополнительных тестов, мы сознательно пошли на этот компромисс. Однако он не всегда приемлем, поэтому вам стоит поискать правильный баланс для своей организации.

¹ Это может показаться излишним, и, вероятно, так оно и есть. Мы ведем активные исследования, стараясь определить наилучший способ выбора «правильного» набора тестов для конкретного изменения и найти баланс между временем выполнения тестов и трудозатратами в случае неправильного выбора.

КЕЙС: ТЕСТИРОВАНИЕ КРУПНОМАСШТАБНЫХ ИЗМЕНЕНИЙ

Адам Бендер

В настоящее время стало обычным делом, когда двузначный процент (от 10 до 20 %) изменений в проекте является результатом крупномасштабных изменений, то есть значительный объем кода в проекте изменяется людьми, чья постоянная работа не связана с этим проектом. Без хороших тестов это было бы невозможно, и база кода в Google быстро атрофировалась бы под собственным весом. Крупномасштабные изменения позволяют нам систематически переносить всю кодовую базу на новые API, выводить из эксплуатации старые API, обновлять версии языков и избавляться от популярных, но опасных практик.

Даже простое изменение сигнатуры функции выливается в сложный процесс, если затрагивает тысячи вызовов этой функции в сотнях разных продуктов и служб¹. Написав изменение, вы должны организовать обзор кода в десятках команд. Наконец, после одобрения вам потребуется провести как можно больше тестов, чтобы убедиться, что изменение безопасно². Мы говорим «как можно больше», потому что крупномасштабное изменение может привести к запуску каждого отдельного теста в Google, что может занять некоторое время. Фактически для многих крупномасштабных изменений приходится выделять время, чтобы выявить клиентов, чей код ломается в процессе крупномасштабного изменения.

Тестирование крупномасштабного изменения — медленный и утомительный процесс. Когда изменение достаточно велико, локальная среда почти гарантированно рассинхронизируется с главной ветвью, потому что окружающая база кода пересыпается как песок. В таких обстоятельствах вы заметите, что снова и снова запускаете тесты, только чтобы убедиться, что ваши изменения продолжают действовать. Когда в проекте присутствуют нестабильные тесты или код, не охваченный юнит-тестами, может потребоваться тестирование вручную, замедляющее весь процесс. Для ускорения мы используем стратегию под названием «поезд TAP» (test automation platform — автоматизированная платформа тестирования).

Поездка на поезде TAP

Крупномасштабные изменения редко связаны друг с другом, и большинство затронутых тестов успешно выполняются для большинства крупномасштабных изменений. Поэтому мы можем тестировать сразу несколько изменений и уменьшить общее количество выполняемых тестов. Модель поезда оказалась очень эффективной для тестирования крупномасштабных изменений.

Поезд TAP использует в своих интересах два обстоятельства:

- Крупномасштабные изменения обычно являются результатом рефакторинга и поэтому имеют очень узкий охват, сохраняя локальную семантику.
- Отдельные изменения часто проверяются проще и тщательнее, поэтому чаще всего они правильные.

¹ Самая длинная серия из когда-либо выполнявшихся крупномасштабных изменений в течение трех дней удалила из репозитория больше миллиарда строк кода. Главной целью этого изменения было удаление устаревшего кода из репозитория, который переселился в новый дом. Только подумайте: насколько вы должны быть уверены в себе, чтобы удалить миллиард строк кода?

² Крупномасштабные изменения обычно поддерживаются инструментами, позволяющими относительно легко находить, вносить и анализировать изменения.

Модель поезда также имеет еще одно преимущество: она работает сразу с несколькими изменениями и не требует, чтобы каждое отдельное изменение выполнялось изолированно¹.

Поезд выполняет пять остановок и отправляется в путь каждые три часа:

1. Для каждого изменения в поезде запускается выборка из 1000 случайных тестов.
 2. Отбираются все изменения, успешно прошедшие 1000 тестов, и из них создается «поезд».
 3. Для изменений в поезде выполняются все тесты, непосредственно затронутые этими изменениями. При достаточно объемном (или низкоуровневом) крупномасштабном изменении это может означать выполнение всех тестов, имеющихся в репозитории Google. Этот процесс может занять больше шести часов.
 4. Каждый тест, потерпевший неудачу, повторно запускается для каждого отдельного изменения в поезде, чтобы определить, какие из них вызвали сбой.
 5. Платформа TAP генерирует отчет для каждого изменения в поезде. В отчете описаны все выполненные и невыполненные цели, и он может использоваться как доказательство, что крупномасштабное изменение можно без опаски отправить в репозиторий.
-

Код-ревью

Наконец, все изменения должны проходить процесс обзора (глава 9), и это правило действует также в отношении крупномасштабных изменений. Обзор масштабных фиксаций может быть утомительным, обременительным и даже приводить к ошибкам, особенно если изменения вносятся вручную (процесс, которого следует избегать, о чем мы поговорим далее в этой главе). Чуть ниже мы расскажем, как в обзоре могут помочь автоматизированные инструменты, но для некоторых классов изменений мы предпочитаем, чтобы их правильность проверялась людьми. Разбивка крупномасштабных изменений на части значительно упрощает задачу обзора.

КЕЙС: ПЕРЕИМЕНОВАНИЕ `SCOPED_PTR` В `STD::UNIQUE_PTR`

С самого начала в кодовой базе на C++ в Google существовал интеллектуальный указатель для упаковки объектов C++, размещаемых в куче, который обеспечивал их автоматическое уничтожение и освобождение памяти, а затем покидал область видимости. Этот указатель назывался `scoped_ptr` и широко использовался в кодовой базе в Google для гарантии надлежащего управления временем жизни объектов. Он не был идеальным, но с учетом ограничений действующего на тот момент стандарта C++98 этот тип делал программы безопаснее.

В C++11 появился новый стандартный тип: `std::unique_ptr`. Он выполнял ту же функцию, что и `scoped_ptr`, но дополнительно предотвращал ряд других классов ошибок. Тип `std::unique_ptr` был намного лучше, чем `scoped_ptr`, но в кодовой базе Google оставалось еще более 500 000 ссылок на `scoped_ptr`, разбросанных среди миллионов файлов с исходным кодом. Для перехода на более современный тип потребовалось самое большое на тот момент крупномасштабное изменение.

¹ Можно запросить у платформы автоматизированного тестирования выполнить «изолированный» прогон для единственного изменения, но это очень дорого и такие запросы выполняются только в часы затишья.

В течение нескольких месяцев несколько инженеров параллельно занимались этим вопросом. Используя крупномасштабную инфраструктуру миграции Google, они смогли заменить ссылки `scoped_ptr` на `std::unique_ptr`, а также постепенно придать типу `scoped_ptr` поведение, более близкое к поведению `std::unique_ptr`. На пике процесса миграции они ежедневно генерировали, тестировали и фиксировали более 700 независимых изменений, затрагивавших до 15 000 файлов. В настоящее время, усовершенствовав методы и улучшив инструменты, мы иногда достигаем пропускную способность, в десять раз превышающую тот результат.

Как и многие другие, это крупномасштабное изменение имело длинный хвост нюансов поведения (еще одно проявление закона Хайрама), состояний гонки с другими инженерами и содержало использования в сгенерированном коде, которые не обнаруживались нашими автоматизированными инструментами, но выявлялись инфраструктурой тестирования, и мы продолжали работать над ними вручную.

`scoped_ptr` также использовался как параметр типа в некоторых широко используемых API, что затрудняло выполнение небольших и независимых изменений. Мы хотели написать систему анализа графа вызовов, которая могла бы изменить API и вызывающие его стороны транзитивно, за одну фиксацию, но были обеспокоены тем, что получающиеся изменения сами по себе будут слишком большими для атомарной фиксации.

В конце концов, мы смогли удалить все ссылки на `scoped_ptr`, сначала сделав его псевдонимом типа `std::unique_ptr`, а затем выполнив текстовую замену старого псевдонима на новый и, наконец, полностью удалить старый псевдоним `scoped_ptr`. В настоящее время база кода в Google пользуется всеми преимуществами от использования того же стандартного указателя, что и остальная экосистема C++, и это стало возможным только благодаря нашим технологиям и инструментам для крупномасштабных изменений.

Инфраструктура для крупномасштабных изменений

Компания Google вложила значительные средства в инфраструктуру поддержки крупномасштабных изменений, которая включает инструменты для их создания, контроля, анализа и тестирования. Однако наиболее важной поддержкой стало развитие культурных норм вокруг крупномасштабных изменений и их соблюдение. Наборы технических и социальных инструментов в разных организациях могут отличаться, но общие принципы инфраструктуры должны быть одинаковыми.

Политики и культура

Большая часть исходного кода в Google хранится в едином монолитном репозитории (глава 16), и каждый инженер может видеть почти весь этот код. Такая степень открытости означает, что любой инженер может отредактировать любой файл и отправить свои изменения для обзора тем, кто может их одобрить. Однако каждая из таких правок требует затрат как на создание, так и на проверку¹.

¹ Здесь становятся очевидными технические затраты с точки зрения вычислений и хранения, но трудозатраты на своевременный анализ изменения намного перевешивают технические.

Исторически эти затраты были в некоторой степени симметричными, что ограничивало объем изменений, которые могли произвести один инженер или команда. По мере совершенствования инструментария поддержки крупномасштабных изменений в Google стало проще создавать большое количество изменений с очень небольшими затратами, и один инженер мог с легкостью загрузить работой большое количество рецензентов. Мы, конечно, поощряем повсеместное улучшение кодовой базы, но хотим быть уверенными, что они хорошо продуманы и не выполняются хаотично и бесконтрольно¹.

И мы упростили процесс одобрения для команд и отдельных лиц, стремящихся выполнять крупномасштабные изменения. За этим процессом наблюдает группа опытных инженеров, знакомых с нюансами разных языков, а для оценки конкретных изменений приглашаются эксперты в предметной области. Цель процесса не в запрете крупномасштабных изменений, а в желании помочь их авторам создать самые лучшие изменения, максимально использующие технический и кадровый капитал Google. Иногда эта группа может решить, что предложенная чистка кода, например устранение типичной опечатки без какого-либо способа предотвратить ее повторение, не стоит внимания и затрат.

С этой политикой связано изменение культурных норм, касающихся крупномасштабных изменений. Для владельцев кода важно чувствовать ответственность за свое ПО, но им также важно понимать, что крупномасштабные изменения являются важной составляющей усилий Google по расширению наших практик разработки ПО. Так же как команды программных продуктов лучше всего знакомы со своим ПО, команды библиотечной инфраструктуры хорошо знают нюансы своей инфраструктуры, и заставить команды программных продуктов доверять опыту экспертов в предметной области является важным шагом на пути к всеобщему признанию крупномасштабных изменений. В результате этого культурного сдвига команды программных продуктов набрались опыта и стали доверять авторам крупномасштабных изменений в их предметной области.

Иногда владельцы кода ставят под сомнение цель конкретной фиксации, сделанной в рамках более широкого крупномасштабного изменения, и авторы изменений отвечают на их комментарии так же, как и на комментарии рецензентов. В социальном плане важно, чтобы владельцы кода понимали цель изменений в их ПО, но также осознавали, что не имеют права вето в отношении более широкого крупномасштабного изменения. Со временем мы обнаружили, что подробные ответы на часто задаваемые вопросы и солидный послужной список улучшений вызывают широкую поддержку крупномасштабных изменений в Google.

Понимание кодовой базы

Для проведения крупномасштабных изменений бесценной оказалась возможность крупномасштабного анализа нашей кодовой базы как на уровне текста, с использованием традиционных инструментов, так и на семантическом уровне. Например,

¹ Например, мы не хотим, чтобы полученные инструменты использовались в качестве механизма для борьбы за правильную орфографию в комментариях.

инструмент семантического индексирования Kythe (<https://kythe.io>) позволяет получить полную карту связей между сегментами нашей кодовой базы, что дает возможность отвечать на вопросы: «Откуда вызывается эта функция?» или «Какие классы наследуют этот класс?» Kythe и другие подобные инструменты также предоставляют программный доступ к своим данным, позволяя включить их в инструменты рефакторинга. (Дополнительные примеры в главах 17 и 20.)

Мы также используем индексы, создаваемые компилятором, для анализа и преобразования нашей кодовой базы на основе абстрактного синтаксического дерева. Такие инструменты, как ClangMR (<https://oreil.ly/c6xvO>), JavacFlume и Refaster (<https://oreil.ly/Er03J>), которые могут выполнять преобразования параллельно, используют эти данные. Авторы небольших изменений могут использовать специализированные настраиваемые инструменты, perl или sed, сопоставление с регулярными выражениями или даже простые сценарии командной оболочки.

Какой бы инструмент ни использовала организация для создания изменений, важно, чтобы человеческие трудозатраты линейно масштабировались вместе с базой кода. Время на создание коллекции всех необходимых изменений не должно зависеть от размера репозитория. Инструменты для создания изменений тоже должны иметь возможность охватывать всю кодовую базу, чтобы автор мог убедиться, что его изменение охватывает все случаи, которые он пытается исправить.

Как и в других областях, затрагиваемых в этой книге, вложение средств, сил и времени в инструменты обычно с лихвой окупается в краткосрочной или среднесрочной перспективе. Как показывает наш опыт, если для изменения требуется более 500 правок, то будет намного эффективнее, если инженер научится пользоваться инструментами для создания изменений, вместо внесения этих правок вручную. Опытные «чистильщики кода» делают намного меньше правок.

Управление кодом

Наиболее важной частью инфраструктуры крупномасштабных изменений является набор инструментов, которые делят основное изменение на более мелкие части и управляют процессом тестирования, отправки запросов по почте, обзором и фиксацией изменений. В Google этот инструмент называется Rosie, и мы обсудим его более подробно чуть ниже, когда поближе познакомимся с нашим процессом крупномасштабных изменений. Во многих отношениях Rosie — это не просто инструмент, а целая платформа для создания крупномасштабных изменений в масштабе Google. Он дает возможность разбивать большие наборы обширных изменений, производимых с помощью инструментов, на более мелкие, которые можно тестировать, рецензировать и отправлять в репозиторий независимо друг от друга.

Тестирование

Тестирование — еще одна часть инфраструктуры, обеспечивающей крупномасштабные изменения. Тесты — это один из важнейших способов проверки работо-

способности ПО (глава 11). Они особенно необходимы при применении изменений, созданных автоматически. Надежная культура и инфраструктура тестирования дают уверенность другим инструментам, что эти изменения не приведут к непредвиденным последствиям.

Стратегия тестирования крупномасштабных изменений в Google немного отличается от политики тестирования обычных изменений, но обе политики используют одну базовую инфраструктуру непрерывной интеграции. Тестирование крупномасштабных изменений гарантирует не только то, что большое изменение не вызовет сбоев, но и что каждый его сегмент можно безопасно и независимо отправить в репозиторий. Поскольку каждый сегмент может содержать произвольные файлы, мы не используем стандартные предварительные тесты, как для обычных проектов. Вместо этого прогоняем для каждого сегмента все тесты, которые он может затронуть, как обсуждалось выше.

Поддержка языка

Крупномасштабные изменения в Google обычно выполняются отдельно для каждого языка, и некоторые языки позволяют производить такие изменения проще, чем другие. Мы обнаружили, что некоторые особенности языка, такие как поддержка псевдонимов типов и возможность передачи функций, неоценимы, поскольку позволяют пользователям продолжать работу, пока мы вводим новые системы и переводим на них пользователей. Для языков, в которых отсутствуют эти возможности, часто бывает сложно осуществлять пошаговый переход¹.

Мы также обнаружили, что в языках со статической типизацией намного проще выполнять большие автоматические изменения, чем в языках с динамической типизацией. Инструменты на основе компилятора вместе со строгим статическим анализом дают значительный объем информации, которую можно использовать в инструментах для анализа крупномасштабных изменений и отклонения недопустимых преобразований еще до того, как они дойдут до этапа тестирования. Досадным следствием этой особенности является большая сложность сопровождения языков с динамической типизацией, таких как Python, Ruby и JavaScript. Выбор языка часто тесно связан с продолжительностью жизни кода: языки, которые, как правило, более ориентированы на продуктивность разработчиков, нередко труднее поддерживать.

Наконец, стоит отметить, что средства автоматического форматирования кода тоже являются важной частью инфраструктуры крупномасштабных изменений. Мы много внимания уделяем удобочитаемости исходного кода и хотим быть уверены, что любые изменения, производимые автоматизированными инструментами, будут понятны не только рецензентам, но и будущим читателям кода. Все инструменты, генерирующие крупномасштабные изменения, запускают автоматическое форматирование,

¹ Язык Go недавно представил подобные возможности специально для поддержки крупномасштабных изменений (см. <https://talks.golang.org/2016/refactor.article>).

соответствующее языку, отдельно, поэтому инструментам, главной целью которых является собственно изменение, не нужно заботиться о правилах форматирования. Применение автоматического форматирования, например `google-java-format` (<https://github.com/google/google-java-format>) или `clang-format` (<https://clang.llvm.org/docs/ClangFormat.html>), к нашей кодовой базе обеспечивает единообразное оформление, которое упрощает разработку в будущем. Без автоматического форматирования крупномасштабные автоматизированные изменения ни за что не заслужили бы такое высокое доверие в Google.

КЕЙС: ОПЕРАЦИЯ ROSEHUB

Крупномасштабные изменения стали важной частью внутренней культуры Google, но они начинают обретать популярность и за его пределами. Самым известным случаем стала «Операция RoseHub» (<https://oreil.ly/txtDj>).

В начале 2017 года в библиотеке Apache Commons была обнаружена уязвимость, позволявшая скомпрометировать любое Java-приложение, использующее уязвимую версию библиотеки прямо или опосредованно. Эта уязвимость стала известна как «Mad Gadget» (безумный гаджет). Среди прочего она позволила алчному хакеру зашифровать системы Управления общественным городским транспортом Сан-Франциско и остановить его деятельность. Поскольку для уязвимости достаточно, чтобы библиотека находилась где-то на пути к классам, то все, что зависело хотя бы от одного из многих проектов с открытым исходным кодом на GitHub, оказалось уязвимо.

Чтобы решить эту проблему, некоторые инициативные гуглеры запустили свою версию процесса крупномасштабного изменения. Используя такие инструменты, как `BigQuery` (<https://cloud.google.com/bigquery>), они определили проекты, подверженные уязвимости, и отправили более 2600 исправлений для обновления их версий библиотеки Commons до версии, исправляющей уязвимость Mad Gadget. Вместо автоматизированных инструментов процессом управляли более 50 человек.

Процесс крупномасштабных изменений

Теперь мы поговорим о процессе создания крупномасштабных изменений. Он делится примерно на четыре этапа (с очень размытыми границами между ними):

1. Авторизация.
2. Создание изменения.
3. Управление частями изменения.
4. Очистка.

Обычно эти этапы выполняются после того, как были написаны новые система, класс или функция, но о них важно помнить еще на этапе проектирования. Мы в Google стремимся разрабатывать системы-преемники с учетом путей миграции со старых систем, чтобы специалисты, сопровождающие систему, могли автоматически переводить своих пользователей на новую систему.

Авторизация

Мы просим потенциальных авторов представить короткий документ, объясняющий причину предлагаемого изменения, его предполагаемое влияние на кодовую базу (на сколько сегментов можно разделить большое изменение) и содержащий ответы на любые вопросы, которые могут возникнуть у потенциальных рецензентов. Этот процесс заставляет авторов подумать о том, как они будут описывать изменение другому инженеру в форме сборника ответов на часто задаваемые вопросы. Авторы также получают «предметный обзор» от владельцев реорганизуемого API.

Этот документ-предложение затем пересылается по электронной почте комитету специалистов, в которую входит около десятка человек. После обсуждения комитет дает рекомендации о том, как двигаться дальше. Например, одним из наиболее распространенных изменений, вносимых комитетом, является направление всех обзоров кода для крупномасштабного изменения одному «глобальному утверждающему лицу». Многие авторы, впервые предпринимающие попытку внести крупномасштабные изменения, склонны думать, что владельцы проектов должны все проверить, но для большинства механических изменений дешевле иметь одного эксперта, который понимает природу изменения и строит автоматизацию для его анализа.

После утверждения изменения автор может продолжить отправку своего изменения. По традиции комитет очень либерально подходит к одобрению¹ и часто утверждает не только конкретное изменение, но и широкий набор связанных изменений. Члены комитета могут по своему усмотрению ускорить очевидные изменения без полного обсуждения.

Цель этого процесса — обеспечить надзор и управление и освободить от этой ответственности авторов крупномасштабных изменений. Комитет также наделен полномочиями органа по рассмотрению проблем или конфликтов в крупномасштабном изменении: владельцы проектов, не согласные с изменением, могут обратиться к этой группе, чтобы уладить конфликт. Но на практике такое случается редко.

Создание изменения

Получив одобрение, автор крупномасштабного изменения приступает к редактированию кода. Иногда правки могут быть сгенерированы в виде одного большого и глобального изменения, которое впоследствии будет разделено на множество более мелких независимых частей. Обычно размер изменения слишком велик, чтобы уместиться в одно глобальное изменение из-за технических ограничений базовой VCS.

¹ Комитет категорически отклоняет только изменения, признанные опасными, например преобразование всех экземпляров `NULL` в `nullptr`, или слишком малозначимые, например изменение орфографии с британского английского на американский английский или наоборот. С ростом нашего опыта в отношении таких изменений стоимость крупномасштабных изменений снизилась и снизился порог утверждения.

Процесс генерирования изменений должен быть максимально автоматизирован, чтобы родительское изменение можно было повторить, если пользователи вернутся к старому варианту использования¹ или возникнут конфликты при слиянии измененного кода. В редких случаях, когда технические инструменты не способны сгенерировать глобальное изменение, мы распределяем генерацию изменений между инженерами (см. врезку «Пример: операция RoseHub» выше). Это требует намного больше трудозатрат, но позволяет вносить глобальные изменения гораздо быстрее, что особенно важно для приложений, чувствительных ко времени.

Напомним, что мы оптимизируем нашу кодовую базу для удобства чтения, поэтому независимо от инструмента, генерирующего изменения, мы хотим, чтобы они получились максимально похожими на изменения, созданные человеком. Это требование приводит к необходимости следования руководствам по стилю и использования средств автоматического форматирования (глава 8)².

Деление на части и отправка

Сгенерировав глобальное изменение, автор запускает Rosie. Этот инструмент делит большое изменение на части по границам проектов с учетом правил владения, которые *можно* отправлять атомарно. Затем каждая часть изменения независимо проходит через конвейер тестирования, обзора и отправки. Система Rosie может активно пользоваться другими частями инфраструктуры разработки в Google, поэтому она ограничивает количество одновременно обрабатываемых частей любого заданного крупномасштабного изменения, выполняется с низким приоритетом и взаимодействует с остальной инфраструктурой, определяя, какую нагрузку она может создать для общей инфраструктуры тестирования.

Ниже мы подробнее поговорим о конкретном процессе тестирования, обзора и отправки для каждой части.

Тестирование

Каждая независимая часть изменения тестируется с помощью TAP — платформы непрерывной интеграции в Google. Мы запускаем все тесты, зависящие от файлов в данном изменении прямо или опосредованно, что часто создает высокую нагрузку на систему непрерывной интеграции.

Этот процесс может показаться слишком дорогостоящим с точки зрения вычислительных ресурсов, но на практике подавляющее большинство частей затрагивает менее тысячи тестов из миллионов в нашей кодовой базе. Те части, которые затрагивают большее количество тестов, мы можем сгруппировать: сначала объединить все затронутые тесты для всех частей, а затем для каждой отдельной части выполнить

¹ Это может случиться по разным причинам: копирование кода из существующих примеров, фиксация изменений, находившихся в разработке в течение некоторого времени, или просто зависимость от старых привычек.

² Фактически именно по этой причине были начаты работы над clang-format для C++.

только пересечение множества затронутых ею тестов с множеством тестов, потерпевших неудачу в первом прогоне. Большинство таких объединений вызывают выполнение почти всех тестов в кодовой базе, поэтому добавление дополнительных изменений в этот пакет сегментов практически на приводит к увеличению затрат.

СКОТ И ДОМАШНИЕ ЛЮБИМЦЫ

Мы часто используем аналогию «скот и домашние любимцы», ссылаясь на отдельные машины в распределенной вычислительной среде, но эту же аналогию можно применить к изменениям в кодовой базе.

В Google, как и в большинстве организаций, большинство изменений в кодовую базу вручную вносят отдельные инженеры, работающие над конкретными функциями или исправлениями ошибок. Инженеры могут потратить дни или недели на создание, тестирование и анализ одного изменения. Они близко знакомы с изменением и гордятся тем, что оно наконец попадет в основной репозиторий. Создание такого изменения сродни владению и воспитанию домашнего любимца.

Для эффективного обращения с крупномасштабными изменениями, напротив, требуются высокая степень автоматизации и генерирование огромного количества отдельных изменений. В этих условиях мы сочли полезным рассматривать определенные изменения как скот: безмянные и безликие фиксации, которые можно отменить или иным образом отклонить в любой момент и с небольшими затратами, если это не затронет все стадо. Часто такое происходит из-за непредвиденной проблемы, не выявленной тестами, или даже из-за простого конфликта слияния.

В отношении фиксации-любимца часто сложно не принять отказ на свой счет, но при работе с большим количеством изменений в рамках крупномасштабного изменения отказ — это просто особенности работы. Наличие автоматизации означает, что инструменты можно обновлять и вносить новые изменения с очень низкими затратами, поэтому периодическая потеря нескольких голов скота не воспринимается как проблема.

Один из недостатков выполнения такого большого количества тестов состоит в том, что независимые и маловероятные события почти гарантированы в достаточно большом масштабе. Нестандартные и нестабильные тесты (глава 11), не наносящие вреда командам, которые их пишут и поддерживают, особенно трудны для авторов крупномасштабных изменений. Нестабильные тесты незначительно влияют на отдельные команды, но могут серьезно повлиять на пропускную способность системы крупномасштабных изменений. Автоматические системы обнаружения и устранения нестабильных тестов помогают решить эту проблему, но могут потребоваться постоянные усилия, чтобы гарантировать, что команды, создающие нестабильные тесты, сами будут нести свои расходы.

Получив опыт применения крупномасштабных изменений, сохраняющих семантику и генерируемых машиной, мы обрели уверенность в правильности отдельного изменения. Тесты, проявившие нестабильность в недавнем прошлом, теперь игнорируются нашими автоматизированными инструментами при отправке. Теоретически это означает, что отдельная часть изменения может вызвать регрессию, которая

обнаруживается только с помощью нестабильного теста. Но на практике это случается настолько редко, что проще решить проблему в ходе человеческого общения, чем с помощью автоматизации.

В процессе любого крупномасштабного изменения отдельные его части должны фиксироваться независимо. Это означает, что механизм деления может сгруппировать зависимые изменения (например, файл заголовка и файл реализации) вместе. Как и любые другие изменения, части крупномасштабных изменений также должны проходить проверки в конкретных проектах перед обзором и фиксацией.

Отправка изменений для обзора

После проверки безопасности изменения Rosie отправляет это изменение подходящему рецензенту. В такой большой компании, как Google, насчитывающей тысячи инженеров, поиск рецензентов сам по себе является сложной задачей. Код в репозитории организован с помощью файлов OWNERS (главе 9) со списками пользователей, обладающих правом одобрения для определенного поддерева в репозитории. Rosie использует службу определения владельцев, которая читает файлы OWNERS и оценивает каждого владельца на предмет возможности его привлечения к обзору рассматриваемой части изменения. Если выбранный владелец не ответит в установленный срок, Rosie автоматически добавит дополнительных рецензентов, чтобы обеспечить своевременный обзор изменения.

В процессе рассылки изменений для обзора Rosie также запускает инструменты тестирования перед фиксацией для каждого проекта, которые могут выполнять дополнительные проверки. Для крупномасштабных изменений мы выборочно отключаем некоторые проверки, например проверки нестандартного форматирования описания изменений. Такие проверки полезны для отдельных изменений в конкретных проектах, но они являются источником неоднородности в кодовой базе и могут значительно усложнить процесс крупномасштабных изменений. Эта неоднородность препятствует масштабированию наших процессов и систем, и нельзя требовать, чтобы инструменты и авторы крупномасштабных изменений понимали особые правила каждой команды.

Мы также стараемся игнорировать ошибки, выявленные в ходе предварительной проверки и существовавшие до внесения рассматриваемого изменения. Работая над отдельным проектом, инженер может с легкостью исправить их и продолжить свою первоначальную работу, но этот метод не подходит для крупномасштабных изменений в кодовой базе Google. Владельцы проектов отвечают за отсутствие ранее существовавших ошибок в своей кодовой базе в рамках социального контракта между ними и инфраструктурными командами.

Обзор

Изменения, сгенерированные системой Rosie, должны пройти стандартный процесс обзора кода. Как показывает наш опыт, владельцы проектов не часто относятся к крупномасштабным изменениям так же строго, как к обычным изменениям, — они

слишком доверяют инженерам, создающим крупномасштабные изменения. На практике владельцы проектов только бегло просматривают эти изменения. Поэтому мы передаем владельцам только определенные изменения, чтобы они оценили их в контексте своих проектов. Все остальные изменения могут передаваться «глобальному утверждающему лицу», имеющему право одобрять любые изменения во всем репозитории.

Когда в процесс обзора вовлекается глобальный утверждающий, все отдельные части изменения передаются ему, а не владельцам разных проектов. Глобальные утверждающие обычно обладают экспертными знаниями языка и библиотек, которые они проверяют. Они взаимодействуют с авторами крупномасштабных изменений, стараясь узнать детали изменения и возможные отказы, и на основе полученных сведений выстраивают свой рабочий процесс.

Вместо обзора каждого изменения по отдельности глобальные рецензенты используют отдельный набор инструментов на основе паттернов для проверки каждого изменения и автоматического одобрения тех, что соответствуют их ожиданиям. Как результат, им приходится исследовать вручную только небольшую подгруппу изменений, в которой выявились аномалии из-за конфликтов слияния или неправильной работы инструментов, что обеспечивает хорошее масштабирование процесса.

Отправка в репозиторий

Наконец, отдельные изменения фиксируются в репозитории. Так же как на этапе обзора, перед фактической фиксацией в репозитории изменение проходит различные проверки.

Благодаря Rosie мы можем эффективно создавать, тестировать, рецензировать и фиксировать тысячи изменений в день во всей кодовой базе в Google и даем нашим командам возможность эффективно переводить своих пользователей на новые системы. Технические решения, которые раньше имели бесповоротный характер, такие как выбор имени для широко используемого символа или местоположения популярного класса в кодовой базе, теперь стало возможным менять.

Очистка

Разные крупномасштабные изменения завершаются по-разному, от полного удаления старой системы до переноса только важных ссылок и оставления старых для исчезновения естественным путем¹. Почти во всех случаях важно иметь систему, которая предотвращает повторное появление символа или системы, которые удаляются крупномасштабным изменением. Мы в Google используем для этого платформу Tricorder (главы 19 и 20). Она отмечает новые попытки использовать устаревший объект и является эффективным средством для предотвращения отката назад. Подробнее о процессе устаревания и прекращения поддержки в главе 15.

¹ К сожалению, системы, которые хотелось бы убрать естественным путем, оказываются наиболее устойчивыми. Они — пластик в экосистеме кода.

Заключение

Крупномасштабные изменения являются важной частью экосистемы разработки ПО в Google. Они открывают широкие возможности для проектирования, давая уверенность, что некоторые решения можно будет изменить потом. Процесс крупномасштабных изменений также позволяет разработчикам базовой инфраструктуры удалять из кодовой базы в Google старые системы, версии языков и библиотечные идиомы, заменяя их новыми, и сохранять при этом целостность кодовой базы как в пространстве, так и во времени. И для всего этого достаточно нескольких десятков инженеров, поддерживающих десятки тысяч других сотрудников.

Независимо от размера вашей организации подумайте, как вы будете вносить радикальные изменения в свою коллекцию исходного кода. Наличие такой возможности улучшит масштабирование вашей организации и поможет сохранить исходный код гибким долгое время.

Итоги

- Процесс крупномасштабных изменений позволяет иначе взглянуть на бесповоротность некоторых технических решений.
- Традиционные модели рефакторинга не подходят для масштабных изменений.
- Реализация поддержки крупномасштабных изменений помогает выработать привычку проводить такие изменения.

Непрерывная интеграция

Автор: Рейчел Танненбаум

Редактор: Лиза Кэри

Непрерывная интеграция обычно определяется как «практика разработки ПО, когда участники команды часто интегрируют результаты своей работы <...> Каждая интеграция проверяется автоматической сборкой (и тестированием) для максимально быстрого выявления ошибок»¹. Проще говоря, главная цель непрерывной интеграции — как можно раньше и автоматически обнаружить проблемные изменения.

Но что означает «частая интеграция» для современного распределенного приложения? Современные системы имеют множество движущихся частей, помимо кода последней версии в репозитории. Фактически, если следовать последней тенденции к использованию микросервисов, маловероятно, что изменения, нарушающие работу приложения, будут находиться непосредственно в кодовой базе проекта, скорее они будут находиться в слабосвязанных микросервисах на другой стороне сетевого соединения. Традиционная непрерывная сборка тестирует изменения в двоичном файле, а ее расширенная версия может тестировать изменения в вышестоящих микросервисах. Зависимость просто переносится из стека вызовов функции в HTTP-запрос или RPC.

Даже вдали от своих зависимостей приложение может периодически получать данные или обновлять модели машинного обучения. Оно может работать в меняющихся операционных системах, средах выполнения, службах облачного хостинга и на устройствах. Может быть функцией, находящейся на вершине растущей платформы, или платформой, вмещающей растущую базу функций. Все это следует рассматривать как зависимости, и мы должны стремиться «непрерывно интегрировать» изменения в них. Ситуация осложняется еще больше тем, что эти изменяющиеся компоненты часто принадлежат сторонним разработчикам, которые развертывают изменения по собственному графику.

С учетом вышесказанного более точное определение непрерывной интеграции в современном мире, особенно при масштабной разработке, выглядит так:

Непрерывная интеграция — это непрерывные сборка и тестирование всей сложной и быстро развивающейся экосистемы.

¹ См. <https://www.martinfowler.com/articles/continuousIntegration.html>

Тестирование тесно связано с непрерывной интеграцией, и мы будем подчеркивать это на протяжении всей главы. В предыдущих главах мы обсудили широкий спектр видов тестирования, от юнит-тестирования до интеграционного и системного.

С точки зрения тестирования непрерывная интеграция — это парадигма, помогающая понять:

- *какие* тесты и *когда* должны запускаться в рабочем процессе разработки при непрерывной интеграции изменений в коде (и в других компонентах);
- как сформировать SUT с учетом таких аспектов, как точность и затраты.

Например, какие тесты запускать перед отправкой в репозиторий, какие — после отправки, а какие еще позже, например на этапе развертывания в промежуточной среде? Соответственно как представить SUT в каждой из этих точек? Как вы уже наверняка догадались, требования к формированию SUT для выполнения проверок перед отправкой изменений в репозиторий могут значительно отличаться от требований к той же системе в промежуточной среде. Например, для приложения, собранного из кода, ожидающего рецензирования на этапе предварительной проверки, может быть опасно взаимодействовать с реальными производственными службами (подумайте о безопасности и лимите на уязвимости), тогда как часто это вполне приемлемо для тестирования в промежуточной среде.

И почему мы должны пытаться оптимизировать этот нередко тонкий баланс «правильных решений» в «правильное время» с помощью непрерывной интеграции? Богатый опыт прошлого показал выгоды непрерывной интеграции для инженерных организаций в частности и для бизнеса в целом¹. Эти выгоды обусловлены мощной гарантией: наличием неоспоримых и своевременных доказательств, поддающихся проверке, готовности приложения к переходу на следующий этап. Нам не нужно просто надеяться, что все участники интеграции осторожны, ответственны и внимательны, — мы можем гарантировать работоспособность нашего приложения на разных этапах от сборки до выпуска, повышая тем самым качество наших продуктов и уверенность и продуктивность наших команд.

В оставшейся части этой главы мы познакомимся с ключевыми идеями, передовыми практиками и проблемами непрерывной интеграции, а затем рассмотрим, как она реализуется в Google, какие инструменты непрерывной сборки используются, немного поговорим о TAP и в заключение подробно изучим, как развивалась непрерывная интеграция одного из приложений.

¹ Форсгрэн Н., Хамбл Дж., Ким Дж. Ускоряйся! Наука DevOps. Как создавать и масштабировать высокопроизводительные цифровые организации. Интеллектуальная литература, 2020. — Примеч. пер.

Идеи непрерывной интеграции

Начнем с обзора некоторых основных идей непрерывной интеграции.

Короткий цикл обратной связи

Стоимость ошибки растет почти экспоненциально в зависимости от времени ее обнаружения (глава 11). На рис. 23.1 показаны все места в жизненном цикле кода, где может быть обнаружено проблемное изменение.



Рис. 23.1. Жизненный цикл изменения в коде

В общем, чем позднее обнаруживаются проблемы, тем дороже они обходятся, потому что:

- их должен решить инженер, который, скорее всего, не знаком с проблемным изменением;
- автору изменения требуется дополнительное время, чтобы вспомнить и изучить это изменение;
- они негативно влияют как на инженеров, так и на конечных пользователей.

Чтобы уменьшить стоимость ошибок, непрерывная интеграция рекомендует использовать как можно более *короткие циклы обратной связи*¹. Каждый раз, когда мы интегрируем изменение в сценарий тестирования и наблюдаем за результатами, мы создаем новый *цикл обратной связи*. Обратная связь может иметь разные формы (некоторые из них приведены ниже в порядке от самой быстрой к самой долгой):

- результаты, получаемые на этапе правки-компиляции-отладки в локальной среде разработки;
- результаты автоматического тестирования на этапе предварительной проверки;
- результаты интеграционного тестирования изменений в двух проектах после тестирования и фиксации изменений в этих проектах по отдельности;
- несовместимость проекта с вышестоящим микросервисом, обнаруженная тестировщиком из подразделения контроля качества в промежуточной среде после развертывания последних изменений в вышестоящей службе;

¹ Иногда это называют «сдвигом влево при тестировании».

- отчет об ошибке, созданный внутренним пользователем, опробовавшим изменение раньше сторонних пользователей;
- отчет об ошибке, созданный сторонним пользователем или опубликованный в прессе.

Канареечное развертывание, при котором измененная версия получает ограниченное использование, может помочь минимизировать последствия проблем в продакшене, если таковые появятся, за счет включения в цикл обратной связи только части продакшена. Однако канареечное развертывание тоже может вызвать проблемы, особенно в отношении совместимости между развертываниями, происходящими одновременно. Иногда это состояние распределенной системы, в которой содержится несколько несовместимых версий кода, данных и конфигурации, называют *перекосом версий*. Как и многие другие проблемы, которые рассматриваются в этой книге, перекося версий является еще одним примером сложной проблемы, которая может возникнуть при разработке ПО и управлении им с течением времени.

Эксперименты и флаги переключения функций — это чрезвычайно мощные циклы обратной связи. Они снижают риски, обусловленные изоляцией изменений в модульных компонентах, которые можно динамически переключать в продакшене. Применение флагов переключения функций — это распространенная парадигма непрерывной поставки (глава 24).

Доступная и полезная обратная связь

Также очень важно, чтобы обратная связь от непрерывной интеграции была широко доступна. В дополнение к нашей культуре открытости кода мы также культивируем широкую доступность результатов тестирования. У нас есть унифицированная система отчетов о тестировании, в которой любой может легко найти сборку или результаты тестирования, включая все журналы (кроме личной информации пользователя), будь то локальный прогон, выполненный отдельным инженером, или результаты автоматической или промежуточной сборки.

Наряду с журналами наша система отчетов о тестировании предоставляет подробную историю, когда цели сборки или тестирования начали терпеть неудачу, включая сведения о том, где сборка прерывалась в каждой попытке, где она запускалась и кем. Также у нас есть система нестабильных тестов, которая использует статистику для классификации нестабильных тестов на уровне всей компании Google, поэтому инженерам не нужно выяснять, привело ли их изменение к неудачному завершению теста в другом проекте (если тест нестабильный, то, возможно, неудача не связана с изменением).

Доступность истории тестирования позволяет инженерам обмениваться полученными результатами, что очень важно для разрозненных групп, пытающихся выяснить причины сбоев интеграции между своими системами и извлечь уроки. Точно так же сообщения об ошибках доступны всем в Google с полной историей комментариев (кроме личных данных клиента).

Наконец, любая обратная связь, касающаяся тестов, выполняемых системой непрерывной интеграции, должна быть не только доступна, но и полезна — быть простой

в использовании и способной находить и устранять проблемы. Далее в этой главе мы рассмотрим пример улучшения недружественной обратной связи. Повышая удобочитаемость результатов тестирования, вы автоматически упрощаете понимание обратной связи.

Автоматизация

Хорошо известно, что автоматизация задач, связанных с разработкой, экономит инженерные ресурсы (<https://oreil.ly/UafCh>). Интуитивно понятно, что автоматизация проверки измененного кода перед отправкой снижает вероятность ошибки. Конечно, в автоматизированных процессах, как и в любом другом ПО, возможны свои ошибки; но, реализованные достаточно эффективно, автоматические проверки все равно будут быстрее, проще и надежнее, чем проверки, выполняемые вручную.

Непрерывная интеграция, в частности, автоматизирует процессы *сборки* и *выпуска*, осуществляя непрерывные сборку и поставку. Непрерывное тестирование мы рассмотрим в следующем разделе.

Непрерывная сборка

Непрерывная сборка (continuous build) интегрирует последние изменения в главную ветвь¹ и выполняет автоматическую сборку и тестирование. Поскольку непрерывная сборка запускает тесты, а также код сборки, к нарушениям сборки или сбоям в сборке относятся ошибки тестирования, а также ошибки компиляции.

После отправки изменения непрерывная сборка должна запустить все необходимые тесты. Если изменение успешно пройдет все тесты, оно отмечается зеленым цветом, как это часто имеет место в пользовательских интерфейсах. Этот процесс эффективно вводит в репозиторий две разные версии главной ветви: *истинную главную ветвь*, содержащую последнее зафиксированное изменение, и *зеленую главную ветвь*, содержащую последнее изменение, проверенное системой непрерывной сборки. Инженеры могут синхронизировать свою локальную среду разработки с любой из этих версий. Обычно для работы со стабильной версией, проверенной системой непрерывной сборки, и создания новых изменений синхронизация выполняется с зеленой главной ветвью, но некоторые процессы требуют синхронизации с истинной главной ветвью.

Непрерывная поставка

Первый шаг в непрерывной поставке (continuous delivery, глава 24) — *автоматизация выпуска*, которая непрерывно собирает последний код и конфигурацию из главной ветви и создает предварительную версию, рассматриваемую как кандидат на выпуск. В Google большинство команд используют для этого зеленую, а не истинную главную ветвь.

¹ *Главная ветвь* — это последняя версия кода в нашем монолитном репозитории. В других рабочих процессах ее также называют *основной* (master) ветвью или *магистралью* (trunk). Соответственно интеграция с главной ветвью также известна как *разработка в главной ветви*.

Предварительная версия (RC, release candidate) — подготовленный к развертыванию модуль, созданный с помощью автоматизированного процесса¹ из кода, конфигурации и других зависимостей, прошедших этап непрерывной сборки.

Обратите внимание, что в предварительную версию включается также конфигурация — это чрезвычайно важно, несмотря на то что она может меняться в зависимости от среды по мере продвижения предварительной версии. Мы не призываем вас компилировать конфигурацию в свои двоичные файлы — на самом деле рекомендуем использовать динамическую конфигурацию, позволяющую проводить эксперименты и переключать флаги управления функциями².

Мы лишь говорим, что любая *имеющаяся* статическая конфигурация должна продвигаться в составе предварительной версии, чтобы ее можно было протестировать вместе с кодом. Как показывает практика, значительная доля ошибок, появляющихся в продакшене, вызвана «глупыми» ошибками в конфигурации, поэтому тестировать конфигурацию так же важно, как тестировать код (и нужно тестировать ее *вместе* с кодом, который будет ее использовать). Часто в процессе продвижения предварительной версии обнаруживается перекосяк версий. Это, конечно, предполагает, что статическая конфигурация должна храниться в VCS — в Google статическая конфигурация хранится в VCS вместе с кодом и, соответственно, проходит тот же процесс обзора.

Теперь можно дать определение непрерывной поставки:

Непрерывная поставка — непрерывная сборка предварительных версий с последующим их продвижением и тестированием в нескольких средах, иногда включающих продакшен.

Процесс продвижения и развертывания во многом зависит от команды (см. пример далее в этой главе).

Для команд в Google, которым нужна постоянная обратная связь от новых изменений в продакшене (создаваемых, например, непрерывным развертыванием), обычно невозможно постоянно отправлять целые двоичные файлы, как правило, очень большие, по зеленой ветви. По этой причине часто используется *выборочное* непрерывное развертывание с помощью экспериментов или флагов переключения функций.

По мере продвижения предварительной версии через среду ее артефакты (например, двоичные файлы, контейнеры) не должны повторно компилироваться или собираться. Использование контейнеров, таких как Docker, помогает обеспечить согласованность

¹ В Google автоматизация выпуска реализуется отдельной от TAP системой. Мы не будем акцентировать внимание на том, *как* автоматизация выпуска собирает предварительную версию, но если вам это интересно, то обращайтесь к книге «Site Reliability Engineering. Надежность и безотказность как в Google» (<https://landing.google.com/sre/books>), в которой подробно рассматривается наша технология автоматизации выпуска (система под названием Rapid).

² Подробнее о непрерывной поставке с экспериментами и флагами переключения функций в главе 24.

предварительной версии между средами, начиная с локальной среды разработки. Точно так же использование инструментов оркестрации, таких как Kubernetes (у нас чаще используется Borg (<https://oreil.ly/89yPv>)), помогает обеспечить согласованность между развертываниями. Обеспечивая согласованность выпуска и развертывания между средами, мы достигаем более высокой достоверности результатов тестирования на ранних этапах и меньшего количества сюрпризов в продакшене.

Непрерывное тестирование

Давайте посмотрим, как непрерывные сборка и поставка сочетаются с непрерывным тестированием (continuous testing) изменений кода на протяжении всего его жизненного цикла (рис. 23.2).

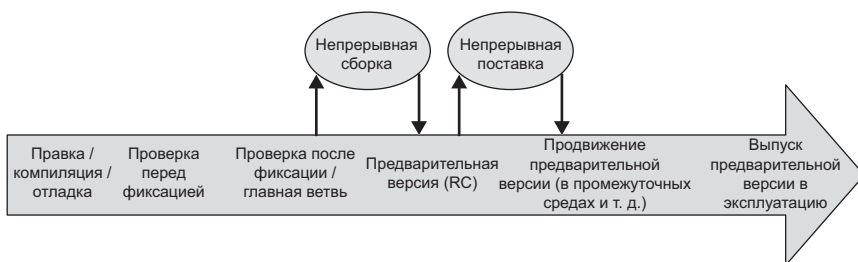


Рис. 23.2. Жизненный цикл изменения кода с непрерывными сборкой и поставкой

Стрелка вправо показывает направление движения одного изменения кода от локальной среды разработки до продакшена. Как вы помните, одна из ключевых задач непрерывной интеграции — определить, *что* и *когда* тестировать. Далее в этой главе мы познакомимся с разными этапами тестирования и рассмотрим некоторые способы определения, что тестировать до и после фиксации, а также на этапе создания предварительной версии и следующих за ним этапах. Мы покажем, что при движении вправо изменение подвергается все более масштабному автоматизированному тестированию.

Почему тестирования перед фиксацией недостаточно

Стараясь как можно быстрее выявлять проблемные изменения и выполнять автоматическое тестирование перед фиксацией, вы можете задаться вопросом: почему нельзя просто запускать все тесты перед фиксацией?

Во-первых, это слишком дорого. Продуктивность инженера имеет высокую цену, и долгое ожидание выполнения всех тестов во время фиксации может серьезно снизить ее. Кроме того, устраняя ограничение на полноту предварительных проверок, можно добиться значительного увеличения эффективности, если тесты будут завершаться успехом гораздо чаще, чем неудачей. Например, тесты могут ограничиваться определенными областями или выбираться на основе модели, прогнозирующей вероятность обнаружения сбоя.

Точно так же слишком дорого будут обходиться простые инженеров из-за сбоев нестабильных тестов, не имеющих ничего общего с фиксируемым изменением.

Во-вторых, пока мы выполняем тесты перед фиксацией, чтобы убедиться в безопасности изменения, база кода в репозитории может измениться и стать несовместимой с тестируемыми изменениями. То есть два изменения, затрагивающие совершенно разные файлы, могут вызвать сбой теста. Мы называем эту ситуацию столкновением в воздухе, и в масштабе нашей компании оно очень редко случается. Системы непрерывной интеграции для небольших репозиториях или проектов могут выстраивать фиксации в очередь, чтобы не было разницы между тем, что предполагалось добавить, и тем, что в итоге добавлено.

Тестирование до и после фиксации

Итак, какие тесты *следует* выполнять перед фиксацией? Мы следуем общему правилу: только быстрые и надежные. На этапе тестирования перед фиксацией можно пожертвовать некоторой полнотой охвата, но это означает, что вы должны выявить любые оставшиеся проблемы на следующем этапе тестирования после фиксации и быть готовыми выполнить некоторое количество откатов. После фиксации можно позволить себе потратить больше времени на борьбу с нестабильностью, если для этого есть надлежащие механизмы.



В разделе «Непрерывная интеграция в Google» мы увидим, как TAP осуществляет управление сбоями.

Мы не хотим жертвовать продуктивностью инженеров, поэтому на этапе предварительной проверки выполняем только тесты для проекта, в котором происходят изменения. Мы также выполняем тесты параллельно, поэтому учитываем затраты вычислительных ресурсов. Наконец, мы не хотим выполнять ненадежные тесты на этапе предварительной проверки, потому что слишком высока стоимость привлечения инженеров к устранению проблемы, которая не связана с изменениями.

На этапе предварительной проверки большинство команд в Google выполняют свои маленькие тесты (например, юнит-тесты)¹, которые, как правило, самые быстрые и надежные. Но есть ли смысл выполнять тесты с широким охватом перед фиксацией? Ответ на этот вопрос зависит от команды. Команды, считающие нужным их выполнять, используют проверенный подход герметичного тестирования, помогающий уменьшить нестабильность, свойственную таким тестам. Другой возможный подход: выполнять ненадежные тесты с широким охватом при предварительной проверке, но отключать их, когда они начинают терпеть неудачу.

¹ Каждая команда в Google настраивает подмножество тестов для своего проекта, которые должны выполняться до (а не после) фиксации. Наша система непрерывной сборки оптимизирует некоторые такие тесты и переносит их выполнение в этап тестирования после фиксации. Мы поговорим об этом далее в этой главе.

Тестирование предварительной версии

После того как изменение прошло непрерывную сборку (на это может потребоваться несколько циклов, если имели место сбои при тестировании), его вскоре обнаружит система непрерывной поставки и включит в предварительную версию.

Во время сборки предварительной версии система непрерывной поставки будет выполнять более крупные тесты. Предварительная версия неоднократно тестируется по мере продвижения через серию тестовых сред и при каждом развертывании. Эта серия может включать комбинацию временных и изолированных или общих тестовых сред, таких как среда разработки и промежуточные испытательные среды. Также в общих тестовых средах обычно выполняется тестирование вручную для проверки качества предварительной версии.

Есть несколько причин, почему важно выполнять комплексный набор тестов для предварительной версии, даже если он в точности совпадает с набором, который был выполнен системой непрерывной сборки после фиксации (при условии, что непрерывная поставка завершилась успехом):

Для проверки работоспособности

Мы дважды проверяем, что ничего необычного не произошло, пока код тестировался и собирался в предварительную версию.

Для контроля

Если инженер захочет изучить результаты тестирования предварительной версии, он с легкостью сможет получить их и ему не придется для этого искать информацию в журналах системы непрерывной сборки.

Чтобы позволить выборочное применение исправлений

Если к предварительной версии применялось некоторое исправление, то ее исходный код будет отличаться от последней версии, протестированной системой непрерывной сборки.

Для экстренных случаев

Система непрерывной поставки может извлечь код из истинной главной ветви и запустить минимальный набор тестов, не дожидаясь окончания полного цикла непрерывной сборки.

Тестирование в продакшене

Наш автоматизированный процесс непрерывного тестирования распространяется и на конечную точку развертывания — продакшен. Мы должны выполнить в продакшене тот же набор тестов (которые иногда называют *зондами*), что и для предварительной версии, чтобы проверить: 1) работоспособность продакшена и 2) актуальность наших тестов для продакшена.

Непрерывное тестирование на каждом этапе развития приложения служит напоминанием о ценности подхода «глубокоэшелонированной защиты» с точки зрения обнаружения ошибок — это не просто часть технологии или политики, которые обеспечивают качество и стабильность, это сочетание множества подходов к тестированию.

НЕПРЕРЫВНАЯ ИНТЕГРАЦИЯ — ЭТО СИСТЕМА ОПОВЕЩЕНИЯ

Титус Винтерс

Как и в случае ответственных систем, для надежного обслуживания программных систем необходим постоянный автоматический мониторинг. По аналогии с системами мониторинга и оповещения в продакшене, которые помогают понять, как эти среды реагируют на изменения, система непрерывной интеграции показывает, как наше ПО реагирует на изменения в своей среде. Подобно мониторингу, использующему пассивные предупреждения и активные средства контроля, непрерывная интеграция использует юнит- и интеграционные тесты для обнаружения изменений в ПО до его развертывания. Сравнение этих двух областей позволяет нам переносить опыт и знания из одной области в другую.

И система непрерывной интеграции, и система оповещения в продакшене служат одной цели: максимально быстро выявлять проблемы. Непрерывная интеграция ориентирована на раннее выявление проблем в процессе разработки и обнаруживает ошибки с помощью тестирования. Система оповещения следит за противоположным концом того же рабочего процесса и выявляет проблемы, наблюдая за показателями и сообщая, когда они превышают некоторые пороговые значения. Обе системы являются формами «автоматического выявления проблем как можно скорее».

Надежная система оповещения помогает гарантировать выполнение целей уровня обслуживания (SLO, service-level objectives). Хорошая система непрерывной интеграции помогает гарантировать, что процесс сборки выполняется корректно — код компилируется, тесты завершаются успехом, и вы можете развернуть новую версию, если это потребуется. Передовой опыт в обеих областях в значительной степени сосредоточен на идеях точности и полезности предупреждений: тесты должны терпеть неудачу, только когда нарушается какое-то важное базовое условие, а не потому, что тест хрупкий или нестабильный. Нестабильный тест, терпящий неудачу через каждые несколько циклов непрерывной интеграции, является такой же проблемой, как и ложное предупреждение, срабатывающее каждые несколько минут. Если такое предупреждение невозможно исправить, оно не должно быть предупреждением. Если базовое условие не нарушается, тест не должен терпеть неудачу.

Системы непрерывной интеграции и оповещения имеют общую концептуальную основу. Например, существует определенная связь между локальными сигналами (юнит-тесты, мониторинг изолированной статистики или оповещение на основе причин) и перекрестно-зависимыми сигналами (интеграционные тесты и тесты предварительной версии, зондирование методом черного ящика). Самыми точными индикаторами работоспособности комплексной системы являются сквозные сигналы, но за их точность приходится платить нестабильностью, увеличенной ресурсоемкостью и трудностями в устранении основных причин.

Также есть связь между отказами в обеих областях. Неустойчивые оповещения срабатывают при пересечении произвольного порогового значения (например, числа повторных попыток за последний час), без наличия фундаментальной связи между этим порогом и работоспособностью системы с точки зрения конечного пользователя. Хрупкие тесты терпят неудачу, когда нарушается произвольное условие, и при этом не обязательно существует фундаментальная связь между этим условием и корректностью тестируемого ПО. Чаще всего такие тесты легко пишутся и могут пригодиться при отладке более серьезной проблемы. В обоих случаях они являются приблизительными показателями общего состояния работоспособности, не отражая целостного поведения. Если нет простого сквозного сигнала,

но вы легко можете получить некоторую совокупную статистику, то команды напишут пороговые предупреждения на основе произвольной статистики. Если нет высокоуровневого способа сказать: «Тест должен терпеть неудачу, если декодированное изображение не похоже хотя бы примерно на это декодированное изображение», то команды создадут тесты, проверяющие идентичность потоков байтов.

Предупреждения, основанные на причинах, и хрупкие тесты все еще могут иметь значение. Они просто не являются идеальным способом выявления потенциальных проблем в сценарии оповещения. В случае фактического сбоя для отладки полезно иметь как можно больше сведений. Когда специалисты по надежности отлаживают собой, их интересует информация в форме: «Час назад пользователи начали чаще сталкиваться с неудачами при выполнении запросов. Примерно в то же время стало расти количество повторных попыток. Начнем расследования отсюда». Точно так же хрупкие тесты могут дать дополнительную отладочную информацию: «Конвейер отображения изображений начал выдавать мусор. Один из юнит-тестов предполагает, что мы получаем разные байты от компрессора JPEG. Начнем расследования отсюда».

Мониторинг и оповещения относятся к области управления надежностью, где хорошо известно понятие «бюджет ошибок»¹, тогда как непрерывная интеграция все еще опирается на абсолютные значения. Отношение к непрерывной интеграции как к «сдвигу оповещений влево» позволяет шире рассуждать об этих политиках и находить более удачные практики:

- Обеспечение 100 % зеленых тестов на этапе непрерывной интеграции, как и 100 % безотказности службы в продакшене, стоит ужасно дорого. Если вы *действительно* ставите перед собой такую цель, то одной из самых больших проблем для вас станет состояние гонки между тестированием и фиксациями.
- Рассматривать все оповещения как равноценные причины для тревоги обычно неправильно. Если оповещение появляется в продакшене, но в действительности не влияет на работу службы, то правильное будет отключить такое оповещение. То же относится к ошибкам в тестах: пока наши системы непрерывной интеграции не научатся говорить: «Известно, что этот тест терпит неудачу по несущественным причинам», — нам, вероятно, следует более либерально принимать изменения, которые отключают тест, терпящий неудачу. Не все неудачи при тестировании могут служить признаками предстоящих проблем в продакшене.
- Политики, гласящие: «Никто не должен выполнять фиксацию в репозиторий, пока цикл непрерывной интеграции не завершится успехом», — скорее всего, ошибочны. Если непрерывная интеграция сообщает о проблеме, ее обязательно нужно *исследовать*, прежде чем позволить инженерам зафиксировать новые изменения или усугубить проблему. Но если причина хорошо изучена и не влияет на работоспособность в продакшене, то блокировать фиксации будет неразумно.

Такое отношение к системе непрерывной интеграции, как к «системе оповещения», пока не получило широкого распространения, и мы все еще пытаемся понять, какие параллели более уместны. Принимая во внимание высокие ставки, неудивительно, что специалисты по надежности тщательно проанализировали опыт, связанный с мониторингом и опове-

¹ Стремление к 100%-ной работоспособности — ошибочная цель. Выберите другой порог, например 99,9 или 99,999%, в качестве компромисса, определите и отслеживайте фактическое время безотказной работы и используйте этот «бюджет» как исходную информацию о том, насколько агрессивны вы готовы продвигать рискованные версии.

нениями, но все еще считают непрерывную интеграцию роскошью¹. В ближайшие несколько лет задача программной инженерии будет состоять в том, чтобы переосмыслить существующую практику поддержания высокой надежности и безотказности в контексте непрерывной интеграции, чтобы помочь реформировать ландшафт тестирования и непрерывной интеграции — и, возможно, передовой опыт тестирования сможет помочь прояснить цели и политики мониторинга и оповещения.

Сложности непрерывной интеграции

Мы обсудили некоторые из устоявшихся практик непрерывной интеграции и познакомились с некоторыми связанными с этим проблемами, такими как потенциальное нарушение продуктивности инженеров из-за медленных и конфликтующих тестов или просто из-за слишком большого их количества. Ниже перечислено еще несколько дополнительных проблем, сопутствующих внедрению непрерывной интеграции:

- *Оптимизация предварительной проверки*, в том числе определение, *какие* тесты запускать во время проверки перед фиксацией с учетом потенциальных проблем, описанных выше, и *как* их запускать.
- *Выявление причин и изоляция сбоев*, в том числе определение, какой код или изменение вызывает проблему и в какой системе произошел сбой. «Интеграция с вышестоящими микросервисами» — это один из подходов к изоляции сбоев в распределенной архитектуре, когда требуется выяснить, что является источником ошибки — собственные службы или внешние. Этот подход предполагает опробование в промежуточной среде комбинаций ваших стабильных служб с новыми внешними микросервисами (то есть интеграцию последних версий внешних микросервисов в свои тесты). Этот подход может осложняться явлением перекоса версий: эти среды не только часто оказываются несовместимыми, но также содержат ложноположительные срабатывания при комбинациях, которые не будут наблюдаться в продакшене.
- *Ограниченный объем ресурсов*. Для выполнения тестов требуются ресурсы, и большие тесты могут быть очень дорогостоящими. Кроме того, затраты на поддержку инфраструктуры для автоматизированного тестирования могут оказаться весьма значительными.

Существует также проблема *управления сбоями* — что делать, если тесты завершаются неудачей. Обычно мелкие проблемы можно быстро исправить, но многие наши команды считают, что очень трудно обеспечивать постоянный успех тестового набора, когда в нем присутствуют большие сквозные тесты. Они по своей природе склонны к нестабильности и сложны в отладке. Должен быть механизм для их временного отключения и отслеживания без остановки процесса выпуска. В Google широко используется метод «горячих списков» ошибок, которые заполняются дежурным

¹ Мы считаем, что на самом деле непрерывная интеграция играет важнейшую роль в экосистеме программной инженерии: это не роскошь, а обязательный элемент. Но пока не все это поняли.

инженером или специалистом по выпуску и передаются соответствующей команде. Еще лучше, когда эти ошибки могут автоматически генерироваться и регистрироваться. Такая возможность поддерживается в некоторых наших крупных продуктах, таких как Google Web Server (GWS) и Google Assistant. Горячие списки должны тщательно проверяться, чтобы гарантировать немедленное исправление любых ошибок, блокирующих выпуск. Также следует исправлять другие, менее срочные ошибки, чтобы набор тестов продолжал приносить пользу, а не превратился в грудку старых отключенных тестов. Часто проблемы, обнаруживаемые при анализе сбоев сквозных тестов, на самом деле связаны с тестами, а не с кодом.

Узкая специализация тестов подрывает уверенность в них, поскольку сбой в них происходит непостоянно и найти в таких тестах изменение для отката труднее, чем в тестах, терпящих неудачу. Некоторые команды полагаются на инструменты временного исключения узкоспециализированных тестов из предварительной проверки, пока их поведение исследуется и исправляется. Это сохраняет высокую степень уверенности в тестах и дает больше времени на устранение проблемы.

Нестабильность тестов порождает еще одну серьезную проблему, которую мы рассматривали в контексте предварительной проверки. Один из способов решения этой проблемы — выполнять такие тесты несколько раз. Это достигается с помощью обычного параметра конфигурации тестов, который широко используют наши команды. Кроме того, в различных точках в коде теста тоже могут выполняться повторные попытки.

Другой подход, помогающий справиться с нестабильностью тестов (и с другими проблемами непрерывной интеграции), — герметичное тестирование, которое мы рассмотрим в следующем разделе.

Герметичное тестирование

Поскольку взаимодействиям с живым сервером присуща некоторая ненадежность, для выполнения тестов с широким охватом мы часто используем герметичные серверы (<https://oreil.ly/-PbRM>). Это особенно полезно, когда тесты должны выполняться на этапе предварительной проверки перед фиксацией и стабильность имеет первостепенное значение. Идею герметичного тестирования мы упоминали в главе 11:

Герметичные тесты выполняются в тестовой среде (то есть используют серверы приложений и их ресурсы), которая полностью автономна (то есть не имеет внешних зависимостей, таких как серверы).

Герметичные тесты обладают двумя важными свойствами: высоким детерминизмом (то есть стабильностью) и изолированностью. Герметичные серверы зависят от системного времени, генерации случайных чисел и состояний гонки, но герметичные тесты не зависят от внешних зависимостей, поэтому при каждом выполнении с одним и тем же приложением код теста возвращает одни и те же результаты. Если герметичный тест терпит неудачу, высока вероятность, что это связано с изменением кода приложения или тестов (они могут завершаться неудачей из-за редчайшей

реорганизации герметичной тестовой среды). По этой причине, когда системы непрерывной интеграции повторно запускают тесты через несколько часов или дней, чтобы получить дополнительные сигналы, герметичность упрощает локализацию ошибок тестирования.

Другое важное свойство — изолированность — означает, что проблемы продакшена не должны влиять на тесты. Обычно мы выполняем такие тесты на одном компьютере, поэтому нам не нужно беспокоиться о проблемах с подключением к сети. Верно и обратное: проблемы, выявленные герметичными тестами, не должны влиять на продакшен.

Успех герметичного теста не должен зависеть от пользователя, выполняющего тест. Это позволяет инженерам воспроизводить тесты, выполняемые системой непрерывной интеграции, и запускать тесты (например, разработчикам библиотек), принадлежащие другим командам.

Один из видов герметичного тестирования основан на использовании фиктивного сервера. Этот подход может обойтись дешевле, чем запуск реального сервера, но требует усилий для поддержки и имеет ограниченную точность (глава 13).

Самый чистый вариант интеграционного тестирования перед фиксацией — создать полностью герметичную конфигурацию, то есть запустить весь стек в изолированной программной среде¹, и в Google есть готовые конфигурации таких тестовых сред для популярных компонентов, таких как базы данных. Этот вариант проще реализовать для небольших приложений с незначительным количеством компонентов, но в Google есть исключение: DisplayAds запускает около четырехсот серверов на этапе предварительной проверки перед каждой фиксацией, а также после фиксации. Однако уже после создания этой системы большую популярность приобрела парадигма записи и воспроизведения для тестирования крупных систем, которая обходится дешевле, чем запуск большого стека в изолированной среде.

Системы записи и воспроизведения (глава 14) записывают ответы действующих серверов, кешируют их и воспроизводят в герметичной тестовой среде. Запись и воспроизведение — мощный инструмент, помогающий уменьшить нестабильность во время тестирования, но увеличивающий хрупкость тестов. Трудно найти баланс между:

Ложноположительными результатами

Тест выполняется успешно из-за слишком частого попадания в кеш, хотя так не должно быть, потому что есть вероятность упустить проблемы, которые могут возникнуть при получении нового ответа.

Ложноотрицательными результатами

Тест терпит неудачу из-за слишком редкого попадания в кеш, хотя так не должно быть. Для решения этой проблемы требуется обновить кеш с ответами, что может занять много времени и привести к сбоям тестов, требующим исправления, многие

¹ На практике часто сложно создать *полностью* изолированную тестовую среду, тем не менее стабильность можно повысить, уменьшив до минимума количество внешних зависимостей.

из которых могут не быть реальными проблемами. Этот процесс часто блокирует фиксацию изменений и далек от идеала.

В идеале система записи и воспроизведения должна обнаруживать только проблемные изменения и реагировать на промах кеша, только если запрос существенно изменился. Если такое изменение вызывает проблему, автор изменения должен повторить тестирование с обновленным ответом, убедиться, что тест по-прежнему терпит неудачу, и только тогда интерпретировать неудачу как предупреждение о проблеме. На практике узнать, когда в большой и постоянно меняющейся системе запрос изменился значимым образом, может быть невероятно сложно.

ГЕРМЕТИЧНЫЙ GOOGLE ASSISTANT

Google Assistant предлагает инженерам платформу для выполнения сквозных тестов, в том числе средства тестирования с возможностью настройки запросов, позволяющие указать, следует ли смоделировать процессы на телефоне или на устройстве умного дома, и проверить ответы во время обмена с Google Assistant.

В качестве примера можно вспомнить одну из самых известных историй успеха полной герметизации набора тестов на этапе предварительной проверки. На начальном этапе команда выполняла негерметичные тесты, которые часто терпели неудачу. Через несколько дней члены команды заметили, что более 50 изменений приняты, невзирая на результаты тестирования. После перевода предварительной проверки в герметичный режим время на выполнение тестов сократилось в 14 раз и они стали завершаться практически без сбоев. Сбои по-прежнему были, но их было легко найти и откатить.

После того как негерметичные тесты были перенесены на этап проверки после фиксации, сбои стали накапливаться там. Отладка сквозных тестов, терпящих неудачу, все еще остается сложной задачей, и у некоторых команд нет времени даже попытаться их исправить, поэтому они просто отключают такие тесты. Это лучше, чем остановить разработку для всех, но может привести к сбоям в продакшене.

Одна из текущих задач команды — продолжить настройку механизмов кеширования, чтобы предварительная проверка могла выявлять больше типов проблем, которые в прошлом обнаруживались только после фиксации, и не была излишне хрупкой.

Другой вопрос — как выполнить предварительную проверку для децентрализованного Assistant, учитывая, что отдельные компоненты переносятся в свои микросервисы. Поскольку Assistant имеет большой и сложный стек, затраты на запуск герметичного стека на этапе предварительной проверки будут очень высокими.

Наконец, команда воспользовалась преимуществами этой децентрализации в новой политике изоляции ошибок на этапе проверки после фиксации. Для каждого из N микросервисов в Assistant команда запускала тестовую среду для проверки после фиксации, содержащую микросервис, собранный из главной ветви, вместе с продакшен- (или близкими к ним) версиями $N - 1$ микросервисов, чтобы изолировать проблемы, порождаемые новыми версиями. Если считать упрощенно, стоимость такой конфигурации равна $O(N^2)$, но команда использовала интересную возможность под названием *hotswapping* (горячая замена), чтобы сократить эту стоимость до $O(N)$. Это позволило давать серверу команду «поменять местами» микросервисы для вызова. В результате требовалось запустить только N микросервисов, по одному экземпляру каждого, собранных из главной ветви, и можно было повторно использовать один и тот же набор служб, подключенных к каждому из этих N «сред».

Итак, герметичное тестирование может уменьшить нестабильность в тестах с большим охватом и помочь изолировать сбои, решая две важные проблемы непрерывной интеграции, которые были определены в предыдущем разделе. Однако герметичные среды также могут быть очень дорогостоящими, потому что требуют больше ресурсов и времени для развертывания. Многие команды используют комбинации герметичных и действующих служб в своих тестовых средах.

Непрерывная интеграция в Google

Теперь перейдем к реализации непрерывной интеграции в Google. Сначала мы исследуем нашу глобальную систему непрерывной сборки — платформу тестирования TAP, используемую подавляющим большинством команд в Google, и рассмотрим, как она позволяет использовать некоторые приемы и решать некоторые проблемы, перечисленные в предыдущем разделе. Вы узнаете, как преобразование непрерывной интеграции помогло масштабировать приложение Google Takeout как платформу и как услугу.

TAP: ГЛОБАЛЬНАЯ СИСТЕМА НЕПРЕРЫВНОЙ СБОРКИ В GOOGLE

Адам Бендер

Мы в Google используем систему массовой непрерывной сборки для всей нашей кодовой базы, которая называется TAP. Она отвечает за выполнение большинства наших автоматических тестов. Как прямое следствие использования монолитного репозитория, TAP является шлюзом для почти всех изменений в Google. Каждый день она обрабатывает более 50 000 уникальных изменений и выполняет более четырех миллиардов тестов.

TAP — это сердце инфраструктуры разработки в Google. Концептуально ее рабочий процесс очень прост. Когда инженер пытается отправить код в репозиторий, TAP выполняет соответствующие тесты и сообщает об успехе или неудаче. Если тесты выполняются успешно, изменения включаются в кодовую базу.

Оптимизация предварительной проверки

Чтобы быстро и последовательно выявлять проблемы, тесты должны выполняться для каждого изменения. Но выполнение тестов обычно остается на усмотрение отдельного инженера, и это часто приводит к тому, что несколько мотивированных инженеров пытаются выполнить все тесты и выявить все сбои.

Как обсуждалось выше, долгое ожидание выполнения всех тестов на этапе предварительной проверки, для чего порой требуется несколько часов, может действовать разрушительно. Чтобы минимизировать время ожидания, подход к непрерывной сборке в Google позволяет вносить критические изменения в репозиторий (как вы помните, они сразу становятся видимыми для всей компании!). Все, что мы просим у инженера, — это создать быстрое подмножество тестов, часто из юнит-тестов проекта, которые можно запустить до отправки изменения (обычно до обзора кода) в репозиторий. Опытным путем установлено, что изменение, успешно прошедшее предварительную проверку, имеет очень высокую вероятность (95 % и выше) успешно пройти остальные тесты, и мы оптимистично позволяем интегрировать его, чтобы другие инженеры могли начать его использовать.

После отправки изменения мы используем TAP для асинхронного выполнения всех тестов, так или иначе затронутых изменением, включая большие и медленные тесты.

Когда изменение вызывает сбой теста в ТАР, оно должно быть исправлено максимально быстро, чтобы не препятствовать работе других инженеров. Мы установили культурную норму, не рекомендуемую приниматься за любую новую работу, которая может быть затронута неудачными тестами, однако специализированные тесты затрудняют следование ей. Поэтому, когда предпринимается попытка зафиксировать изменение, нарушающее нормальную сборку проекта в ТАР, такое изменение может помешать команде двигаться вперед или выпустить новую версию. Поэтому незамедлительно устранять ошибки жизненно необходимо.

Чтобы справиться с такими проблемами, в каждой команде есть «наблюдающий за сборкой». Перед ним стоит задача — обеспечить успешное выполнение всех тестов в проекте, независимо от принадлежности ошибки. Когда наблюдающий получает уведомление о неудаче теста в своем проекте, он откладывает свою работу и исправляет ошибку. Обычно для этого он выявляет проблемное изменение и определяет, нужно ли его откатить (предпочтительное решение) или исправить в будущем (более рискованное решение).

На практике возможность зафиксировать изменения до проверки всеми тестами действительно окупилась: среднее время ожидания для отправки изменения составляет около 11 минут, и часто отправка выполняется в фоновом режиме. Введя неформальную должность наблюдающего за сборкой, мы получили возможность устранять ошибки, обнаруженные продолжительными тестами, с минимальными задержками.

Выявление причин

Одна из проблем, с которыми мы сталкиваемся в Google при работе с большими наборами тестов, — поиск конкретного изменения, вызвавшего сбой во время тестирования. По идее, в этом не должно быть ничего сложного: получить изменение, выполнить тесты и, если какие-то тесты завершатся неудачей, отметить изменение как плохое. К сожалению, из-за того, что проблемы возникают часто и иногда причина кроется в самой тестовой инфраструктуре, не всегда есть полная уверенность в том, что сбой вызван изменением. Хуже того, ТАР должна оценивать очень много изменений в день (более одного в секунду) и не запускать каждый тест при каждом изменении. Для этого она использует пакетную обработку связанных изменений, что сокращает общее количество выполняемых тестов. Этот подход может ускорить выполнение тестов, но точно так же он может затруднить выявление в пакете изменения, которое привело к сбою.

Для ускорения выявления причин сбоев мы используем два подхода. Во-первых, ТАР автоматически разбивает пакет с ошибкой на отдельные изменения и повторно выполняет тесты для каждого компонента. Иногда этот процесс может занять некоторое время, прежде чем будет обнаружен сбой, поэтому мы также создали инструменты для поиска причин, которые разработчики могут использовать для бинарного поиска изменений в пакете.

Управление сбоями

После выявления «виновника» сбоя важно исправить ошибку как можно быстрее. Наличие неудачных тестов может быстро подорвать доверие к набору тестов. Как упоминалось выше, за исправление ошибок, выявленных в ходе сборки, отвечает наблюдающий за сборкой. Самый эффективный инструмент наблюдающего — это откат.

Откат изменения часто является самым быстрым и безопасным способом восстановить сборку, потому что он восстанавливает систему до заведомо исправного состояния¹.

¹ В Google любое изменение в кодовой базе можно отменить двумя щелчками мышью!

Фактически в TAP недавно была добавлена возможность автоматического отката изменений при высокой уверенности, какое именно изменение стало причиной сбоя.

Быстрые откаты работают рука об руку с наборами тестов и обеспечивают высокую продуктивность. Тесты дают нам уверенность в необходимости изменений, откаты — в необходимости отмены этих изменений. Без тестов откат не будет безопасным. Без откатов неработающие тесты будет невозможно исправить быстро, что снизит доверие к системе.

Ограничения на ресурсы

Инженеры могут запускать тесты локально, и все же большинство тестов выполняется в распределенной системе сборки и тестирования под названием *Forge*. *Forge* позволяет инженерам запускать свои сборки и тесты в наших центрах обработки данных параллельно. В нашем масштабе для выполнения всех тестов, запускаемых инженерами вручную, и всех тестов, запускаемых в процессе непрерывной сборки, требуются огромные ресурсы. Даже с учетом имеющихся у нас вычислительных ресурсов такие системы, как *Forge* и TAP, ограничены в ресурсах. Чтобы обойти эти ограничения, инженеры, управляющие платформой TAP, придумали несколько способов, помогающих определить, какие тесты следует запускать и когда, чтобы гарантировать, что на проверку изменения будет потрачен минимальный объем ресурсов.

Основным механизмом выбора тестов для запуска является анализ графа зависимостей для каждого изменения. Распределенные инструменты сборки в Google, такие как *Forge* и *Blaze*, поддерживают версию глобального графа зависимостей почти в масштабе реального времени, и этот граф доступен платформе TAP. Благодаря этому TAP может быстро определить, какие тесты следует выполнить для любого изменения, и запустить минимальный набор, чтобы проверить безопасность изменения.

Еще один фактор, влияющий на работу TAP, — скорость выполнения тестов. TAP часто использует для проверки изменения сокращенный набор тестов. Это побуждает инженеров писать узконаправленные изменения. Разница во времени между ожиданием выполнения 100 тестов и 1000 тестов может составлять десятки минут в напряженный день. Инженеры, стремящиеся тратить меньше времени на ожидание, в конечном итоге вносят более мелкие и узконаправленные изменения, что выгодно для всех.

Пример непрерывной интеграции: Google Takeout

Google Takeout начал создаваться в 2011 году как продукт для резервного копирования и загрузки данных. Основатели этого проекта впервые предложили идею «освобождения данных», согласно которой пользователи должны иметь постоянный доступ к своим данным в удобном формате, где бы они ни находились. Сначала Takeout был интегрирован в несколько продуктов Google в виде архивов фотографий пользователей, списков контактов и т. д., доступных для загрузки в любой момент. Однако Takeout быстро развивался и превратился в платформу и сервис для самых разных продуктов Google. Как мы увидим далее, эффективная непрерывная интеграция играет ключевую роль в поддержании работоспособности любого крупного проекта, но особенно важна для быстро растущих приложений.

Сценарий 1: постоянно неработающее развертывание в среде разработки

Проблема: как только Takeout приобрел репутацию мощного инструмента для извлечения, архивирования и загрузки данных в масштабах всей компании Google, другие команды начали обращаться с просьбой открыть его API, чтобы их собственные приложения тоже могли пользоваться функциями резервного копирования и загрузки, в том числе Google Drive (загрузка папок осуществляется с помощью Takeout) и Gmail (для предварительного просмотра содержимого файлов ZIP). В итоге Takeout вырос из службы, предназначенной только для оригинального продукта Google Takeout, в API для как минимум десяти других продуктов Google, предлагающий широкий спектр возможностей.

Команда решила развернуть каждый из новых API в виде отдельного экземпляра, используя имеющиеся двоичные файлы Takeout, но настроив их для работы в несколько ином контексте. Например, среда для Google Drive имела самый большой парк, наибольшую часть, зарезервированную для извлечения файлов из Drive API, и некоторую логику аутентификации, позволяющую пользователям, не выполнившим вход, загружать общедоступные папки.

Вскоре Takeout столкнулся с «проблемой флагов». Флаги, добавленные для одного из экземпляров, нарушали работоспособность других, и попытки их развертывания прерывались, когда серверы не могли запуститься из-за несовместимой конфигурации. Помимо функциональных настроек имелись также настройки безопасности и ACL. Например, клиентская служба загрузки Google Drive не должна была иметь доступа к ключам, которые шифруют корпоративный экспорт Gmail. Конфигурация стала быстро усложняться и приводила к почти постоянным поломкам.

Были предприняты некоторые попытки распутать и разделить конфигурацию на модули, но возникла более серьезная проблема: когда инженер Takeout хотел внести изменения, он не смог вручную проверить запуск каждого сервера с каждой конфигурацией. Инженеры узнавали об ошибках в конфигурации только после развертывания на следующий день. Конечно, существовали юнит-тесты, выполнявшиеся в ходе проверки до и после фиксации (в TAP), но их было недостаточно для выявления подобных проблем.

Что предприняла команда. Команда создала для каждого из экземпляров временные изолированные мини-среды, которые запускались на этапе предварительной проверки и оценивали возможность запуска всех серверов. Использование временных сред на этапе предварительной проверки помогло предотвратить 95 % ошибок, обусловленных неправильной конфигурацией, и снизило количество сбоев ночного развертывания на 50 %.

Но новые тесты не устранили сбои развертывания полностью. В частности, сквозные тесты Takeout по-прежнему часто прерывали развертывание и их было трудно запустить на этапе предварительной проверки (из-за использования тестовых учетных записей, похожих на настоящие учетные записи и требующих того же уровня

гарантий безопасности и конфиденциальности). Сделать их более дружелюбными для тестирования было бы слишком сложной задачей.

Если команда не может выполнять сквозные тесты на этапе предварительной проверки, то когда их запускать? Разработчики хотели иметь возможность получать результаты сквозного тестирования раньше, чем будет выполнено развертывание в среде разработки, и решили, что запуск тестов каждые два часа — это хорошая отправная точка. Но команда не хотела так часто проводить полное развертывание в среде разработки из-за слишком больших накладных расходов и нарушения продолжительных процессов, которые инженеры тестировали в той же среде. Создание новой совместно используемой тестовой среды также было сопряжено со слишком большими накладными расходами, причем поиск ошибок (то есть обнаружение причин прерывания развертывания) мог потребовать нежелательной ручной работы.

В итоге команда решила повторно использовать изолированные среды из предварительной проверки, расширив их для тестирования после фиксации. В отличие от предварительной проверки, проверка после фиксации соответствовала нормам безопасности для использования тестовых учетных записей (в том числе и потому, что код был одобрен), поэтому появилась возможность запускать сквозные тесты. Непрерывная интеграция запускалась каждые два часа, брала последний код и конфигурацию из зеленой главной ветви, создавала предварительную версию и выполняла для нее тот же набор сквозных тестов, который использовался в среде разработки.

Извлеченный урок. Короткие циклы обратной связи предотвращают проблемы при развертывании в среде разработки:

- Перенос тестов для разных продуктов Takeout из этапа «после ночного развертывания» в этап предварительной проверки помог предотвратить 95 % ошибок, обусловленных неправильной конфигурацией, и снизил количество сбоев ночного развертывания на 50 %
- Перенести все сквозные тесты в этап предварительной проверки было невозможно, но их удалось перенести из этапа «после ночного развертывания» в этап «после фиксации в течение двух часов». Это помогло сократить количество ошибок в 12 раз.

Сценарий 2: нечитаемые журналы результатов тестирования

Проблема: по мере увеличения числа поддерживаемых продуктов Google Takeout превратился в зрелую платформу, позволяющую командам разработчиков добавлять плагины для выборки конкретных данных непосредственно в двоичный файл Takeout. Например, плагин Google Photos извлекал фотографии, метаданные альбомов и т. п. В результате Takeout превратился из первоначальной «горстки» продуктов в платформу, интегрированную более чем в 90 продуктов.

Сквозные тесты Takeout записывали ошибки в журнал, но этот подход не масштабировался до уровня 90 плагинов. По мере роста интеграции возникало все больше отказов. Несмотря на то что с добавлением непрерывной интеграции после фиксации

команда проводила тестирование раньше и чаще, многочисленные сбои продолжали накапливаться, и их было легко пропустить. Просмотр журналов превратился в утомительную трату времени, и тесты почти всегда заканчивались неудачей.

Что предприняла команда. Команда реорганизовала тесты в динамический набор на основе конфигурации (с использованием параметризованного инструмента запуска тестов (<https://oreil.ly/UxkHk>)), который сообщал результаты, четко отмечая результаты отдельных тестов зеленым или красным цветом, благодаря чему отпала необходимость копаться в журналах. Они также значительно упростили отладку сбоев, в частности отображая информацию о сбоях со ссылками на сообщения об ошибках в журналах. Например, если Takeout не удалось получить файл из Gmail, тест динамически создавал ссылку для поиска идентификатора этого файла в журналах Takeout и включал ее в сообщение об ошибке теста. Это помогло автоматизировать большую часть процесса отладки для инженеров плагинов и уменьшило потребность в помощи со стороны команды Takeout для отправки журналов (рис. 23.3).

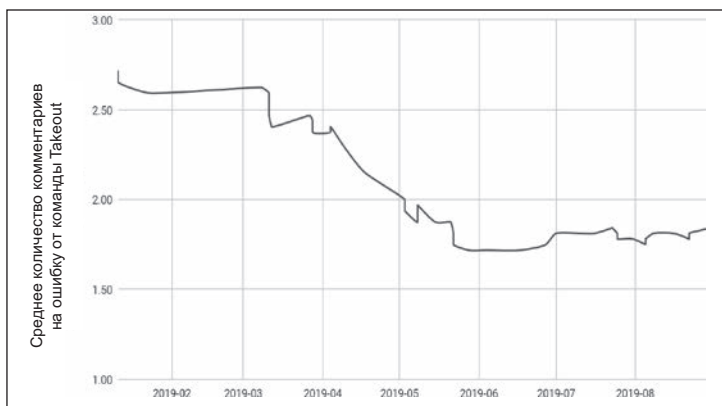


Рис. 23.3. Вовлеченность команды в отладку ошибок на стороне клиентов

Извлеченный урок. Доступная и полезная обратная связь от непрерывной интеграции уменьшает количество сбоев во время тестирования и увеличивает продуктивность инженеров. Эти инициативы уменьшили вовлеченность команды Takeout в устранение ошибок на стороне клиента (плагинов) на 35 %.

Сценарий 3: отладка «всего кода в Google»

Проблема: интересный побочный эффект от внедрения непрерывной интеграции, которого команда Takeout не ожидала, заключался в том, что из-за проверки 90 сторонних продуктов, ориентированных на конечных пользователей, в виде архива, они фактически тестировали «весь код в Google» и выявляли проблемы, никак не связанные с Takeout. С одной стороны, это хорошо — команда Takeout смогла внести свой вклад в повышение качества продуктов Google в целом. Но с другой стороны,

это породило проблему для процессов непрерывной интеграции: команде требовалась более полная изоляция сбоев, чтобы иметь возможность определить, какие проблемы находятся в их сборке (а их было меньшинство), а какие в микросервисах, находящихся за вызываемыми ими API продуктов.

Что предприняла команда. Команда решила непрерывно запускать в продакшене тот же набор тестов, что и на этапе проверки после фиксации. Это было легко реализовать и позволяло команде определять, какие сбои появились в их сборке, а какие в продакшене, например в результате выпуска новой версии микросервиса где-то еще в Google.

Извлеченный урок. Запуск одного и того же набора тестов в продакшене и в ходе непрерывной интеграции на этапе после фиксации (с недавно созданными двоичными файлами, но с теми же действующими службами) — это очень недорогой способ изолировать сбои.

Остающиеся сложности. В дальнейшем нагрузка, связанная с тестированием «всего кода в Google» (конечно, это преувеличение, потому что большинство проблем с продуктами решаются соответствующими командами), будет расти по мере интеграции Takeout с большим количеством продуктов и усложнения этих продуктов. Сравнение вручную результатов тестирования на этапе непрерывной интеграции и в продакшене будет требовать все больше затрат времени наблюдающего за сборкой.

Улучшения в будущем. Открывается интересная возможность попробовать герметичное тестирование с записью и воспроизведением в непрерывной интеграции после фиксации. Теоретически это может помочь устранить сбои, вызванные API сторонних служб, что сделало бы набор тестов более стабильным и эффективным для обнаружения сбоев в течение двух часов после добавления изменений в Takeout, что является его предполагаемой целью.

Сценарий 4: сохранение набора тестов зеленым

Проблема: платформа поддерживала все больше плагинов сторонних продуктов, каждый из которых включает свои сквозные тесты, завершающиеся неудачей, поэтому весь комплект сквозных тестов почти всегда терпел неудачу. Не все сбои можно быстро исправить. Многие из них были обусловлены ошибками в двоичных файлах плагинов, которые команда Takeout не контролирует. Кроме того, некоторые ошибки важнее других: малозначительные ошибки и ошибки в коде тестов не должны блокировать выпуск новых версий, тогда как более значимые ошибки должны останавливать этот процесс. Команда может отключать тесты, закоментировав их, но тогда ошибки будет слишком легко забыть.

Один из типичных источников ошибок — развертывание новых функций в плагинах сторонних продуктов. Например, функция загрузки списка воспроизведения для плагина YouTube может быть включена для тестирования в среде разработки в течение нескольких месяцев, прежде чем будет добавлена в продакшен-версию. Тесты Takeout знали только об одном результате для проверки, поэтому часто приходилось

отключать их в некоторых средах и настраивать вручную по мере развертывания новых функций.

Что предприняла команда. Команда придумала способ отключения неудачных тестов, отмечая их соответствующей ошибкой и отправляя уведомление соответствующей команде (обычно команде, занимающейся разработкой плагина). Когда тест, отмеченный ошибкой, терпел неудачу, среда тестирования Takeout подавляла его. Это позволило набору тестов оставаться зеленым и давать уверенность, что все остальное, кроме известных проблем, работает как ожидается (рис. 23.4).

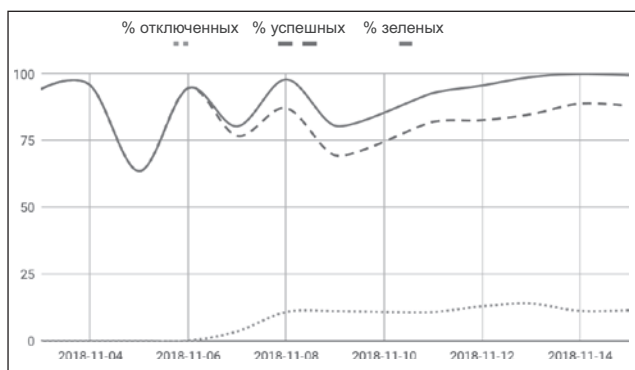


Рис. 23.4. Сохранение набора тестов зеленым за счет (ответственного) отключения тестов

Для решения проблемы команда добавила для инженеров плагинов возможность указывать имя флага новой функции или идентификатор изменения в коде, который активировал конкретную функцию вместе с ожидаемым результатом, как с функцией, так и без нее. В тесты была добавлена возможность отправки запросов в тестовую среду, чтобы определить, включена ли в ней данная функция, и соответствующим образом проверить ожидаемый результат.

По мере накопления тестов, отключенных из-за ошибок и давно не обновлявшихся, команда выполняла их автоматическую очистку. После очистки тесты вновь проверяли, была ли ошибка закрыта, запрашивая нашу систему ошибок. Если тест с отметкой об ошибке выполнялся успешно дольше некоторого установленного предела времени, то предлагалось сбросить отметку (и пометить ошибку как исправленную, если это еще не сделано). В этой политике было одно исключение: нестабильные тесты. Такие тесты команда могла отметить как нестабильные, чтобы система не предлагала сбросить отметку «нестабильный» на этапе очистки, если тест выполняется успешно.

Эти изменения сделали набор тестов практически самоподдерживаемым (рис. 23.5).

Извлеченный урок. Отключение тестов, сообщающих об ошибках, которые нельзя исправить немедленно, — это практический подход к сохранению набора тестов

зеленым и дающим уверенность, что никакие сбои не будут забыты. Кроме того, автоматизация обслуживания набора тестов, включая управление оповещением и отслеживанием исправленных ошибок, поддерживает чистоту набора и предотвращает технический долг. На языке DevOps мы могли бы назвать метрику (рис. 23.5) MTTCU (mean time to clean up – среднее время на очистку).

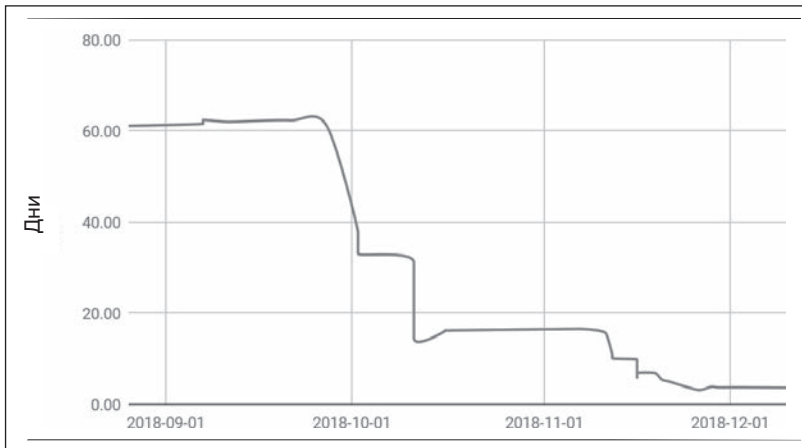


Рис. 23.5. Среднее время устранения ошибки после фиксации

Улучшения в будущем. Следующим полезным шагом могла бы стать автоматизация регистрации ошибок. В настоящее время это все еще ручной и обременительный процесс. Как упоминалось выше, некоторые из наших крупных команд уже ее реализовали.

Сложности в будущем. Описанные сценарии — далеко не единственные проблемы непрерывной интеграции, с которыми столкнулся проект Takeout. Есть и другие проблемы, требующие решения. Например, мы упоминали трудность изоляции отказов, обусловленных вышестоящими службами, в разделе «Сложности непрерывной интеграции» выше. Takeout продолжает сталкиваться с этой проблемой из-за редких сбоев, возникающих в вышестоящих службах, например когда обновление безопасности в потоковой инфраструктуре, используемой Takeout API «Загрузка папок на диске», нарушило расшифровку архива после развертывания в продакшене. Вышестоящие службы проходят свой этап опробования и тестирования, но нет простого способа с помощью непрерывной интеграции автоматически проверить их совместимость с Takeout после развертывания в продакшене. Первоначальное решение включало создание промежуточной среды непрерывной интеграции для тестирования двоичных файлов Takeout с предварительными версиями зависимостей. Однако этот подход оказался трудным в поддержке из-за дополнительных проблем совместимости между промежуточной и продакшен-версиями.

Но я не могу позволить себе организовать непрерывную интеграцию

Возможно, вы думаете, что все, о чем рассказывалось, это хорошо, но у вас нет ни времени, ни денег, чтобы внедрить хоть что-то из этого. Мы готовы признать, что у Google больше ресурсов и возможностей для внедрения непрерывной интеграции, чем у типичного стартапа. Тем не менее многие из наших продуктов росли так быстро, что у них тоже не было времени на разработку системы непрерывной интеграции (по крайней мере, адекватной).

Попробуйте подумать о цене, которую вы заплатили в своих проектах и организациях за проблемы, обнаруженные и решенные в ходе эксплуатации. Эти проблемы отрицательно сказываются не только на конечном пользователе или клиенте, но и на команде. Частое тушение пожаров в продакшене вызывает стресс и деморализует. Да, создание систем непрерывной интеграции обходится дорого, но это не всегда дополнительная цена, потому что непрерывная интеграция помогает сместить затраты влево, на более ранний и более предпочтительный этап, уменьшить частоту и, следовательно, цену проблем, возникающих справа. Непрерывная интеграция дает более стабильный продукт и более благоприятную культуру разработки, в которой инженеры чувствуют себя увереннее и могут положиться на то, что «система» сама обнаружит проблемы, а они смогут сосредоточиться на развитии продукта, а не на исправлении ошибок.

Заключение

В этой главе мы описали наши процессы непрерывной интеграции и некоторые способы их автоматизации, но это не означает, что они идеальны. В конце концов, сама по себе система непрерывной интеграции — это всего лишь ПО, которое никогда не бывает достаточно полным, и его приходится развивать в соответствии с растущими требованиями применения и инженеров, для которых она предназначена. Мы попытались проиллюстрировать это развитие на примере проекта Takeout и показали возможные улучшения в будущем.

Итоги

- Система непрерывной интеграции решает, какие тесты использовать и когда.
- Системы непрерывной интеграции становятся все более необходимыми по мере роста и развития кодовой базы.
- Система непрерывной интеграции должна использовать более быстрые и надежные тесты на этапе предварительной проверки и более медленные и менее детерминированные тесты на этапе проверки после фиксации.
- Доступная и полезная обратная связь позволяет системе непрерывной интеграции стать более эффективной.

Непрерывная поставка

*Авторы: Радха Нараян, Бобби Джонс,
Шери Шите и Дэвид Оуэнс
Редактор: Лиза Кэри*

Учитывая, насколько быстро и непредсказуемо меняется технологический ландшафт, конкурентное преимущество любого продукта заключается в его способности быстро выйти на рынок. Скорость выпуска продукта является критическим фактором конкурентной способности организации, качества ее продуктов и услуг и ее способности адаптироваться к новым правилам. Но эта скорость ограничивается временем развертывания. Развертывание редко завершается успехом при первом запуске. Среди преподавателей есть поговорка, что ни один план урока не переживает первого контакта со студентами. Точно так же ни одно ПО не работает идеально при первом запуске, и единственная гарантия дальнейшего успеха — возможность быстро обновить ПО. Очень быстро.

Работа над ПО с долгим сроком службы требует быстрого изучения новых идей, быстрого реагирования на изменение ландшафта или пользовательские проблемы, а также обеспечения высокой скорости разработки в масштабе. Как рассказывает Эрик Раймонд (Eric Raymond) в эссе «The Cathedral and the Bazaar» и Эрик Рис в книге «Бизнес с нуля»¹, ключ к долгосрочному успеху любой организации всегда заключался в ее способности быстро воплощать идеи на практике и быстро реагировать на отзывы пользователей. Мартин Фаулер (Martin Fowler) в статье «Continuous Delivery» (<https://oreil.ly/B3WFD>) указывает, что «самый большой риск для любого ПО заключается в создании чего-то бесполезного. Чем раньше и чаще вы будете передавать действующее ПО реальным пользователям, тем быстрее вы получите обратную связь и узнаете, насколько оно действительно ценно».

Работа, которая продолжается долгое время, прежде чем начинает приносить пользу пользователям, сопряжена с высоким риском и высокой стоимостью и даже может подорвать моральный дух команды. Мы в Google стремимся выпускать продукты и их новые версии как можно раньше и чаще, чтобы команды могли быстро увидеть результаты своей работы и быстрее адаптироваться к меняющимся условиям рынка. Код приобретает ценность не во время разработки, а когда становится доступным

¹ Рис Э. Бизнес с нуля. Метод Lean Startup для быстрого тестирования идей и выбора бизнес-модели. М.: Альпина Паблишер, 2018. — Примеч. пер.

пользователям. Сокращение времени между завершением разработки кода и появлением обратной связи от пользователей минимизирует стоимость продолжающейся работы.

Вы добиваетесь выдающихся результатов, осознав, что запущенный проект *никогда не завершится*, и с его запуском вы начинаете цикл обучения, в ходе которого исправляете что-то очень важное, оцениваете результат, исправляете что-то следующее и т. д. — этот процесс *не имеет конца*.

Дэвид Уикли, бывший менеджер по продуктам в Google

Методы, которые мы описываем в этой книге, позволяют сотням (а в некоторых случаях тысячам) инженеров Google быстро устранять проблемы, параллельно работать над новыми функциями, не заботясь о выпуске новой версии, и исследовать эффективность новых функций с помощью А/В-тестирования. В этой главе основное внимание уделено ключевым рычагам, способствующим быстрому внедрению инноваций, а именно управлению рисками, масштабированию скорости разработки и оценке компромиссов между стоимостью и ценностью каждой созданной функции.

Идиомы непрерывной поставки в Google

Основной принцип непрерывной поставки, а также методологии гибкой разработки заключается в том, что со временем меньшие пакеты изменений будут давать более высокое качество. Другими словами, *чем быстрее, тем безопаснее*. На первый взгляд это утверждение может показаться противоречивым, особенно если механизмы и условия, необходимые для внедрения непрерывной поставки, — например, непрерывная интеграция и тестирование — еще не созданы. Для реализации идеальной системы непрерывной поставки может потребоваться время, поэтому мы сначала разрабатываем ее аспекты, которые независимо друг от друга приносят пользу на пути к конечной цели. Вот некоторые из них:

Гибкость

Частый выпуск изменений небольшими порциями.

Автоматизация

Уменьшение или устранение повторяющихся накладных расходов из-за частых выпусков.

Изоляция

Стремление к модульной архитектуре, чтобы изолировать изменения и упростить устранение неполадок.

Надежность

Измерение ключевых показателей работоспособности, таких как сбои или задержки, и их улучшение.

Принятие решений на основе данных

Использование А/В-тестирования показателей работоспособности.

Поэтапное внедрение

Развертывание изменений для нескольких пользователей перед открытием полного доступа к этим изменениям.

На первый взгляд частый выпуск новых версий ПО может показаться рискованным делом. С увеличением числа пользователей могут возникнуть опасения их негативной реакции, если они обнаружат какие-то ошибки, оставшиеся незамеченными при тестировании, например из-за слишком большого объема нового кода в продукте, чтобы его можно было полностью протестировать. Но именно здесь может помочь непрерывная поставка. В идеале между выпусками должно вноситься достаточно мало изменений, чтобы потом легко можно было устранить неполадки. При использовании непрерывной поставки каждое изменение проходит через конвейер контроля качества и автоматически внедряется в производство. Во многих командах это не является практической реальностью, поэтому в качестве промежуточного шага часто проводится работа по изменению культуры разработки с целью внедрения непрерывной поставки, в ходе которой команды могут повысить свою готовность к развертыванию в любое время и укрепить свою уверенность в возможности в будущем выпускать новые версии чаще.

Скорость — это командная победа: как разделить процесс развертывания на управляемые этапы

В небольших командах изменения в кодовую базу вносятся с определенной скоростью. Мы наблюдали появление антипаттерна, сопровождающего укрупнение с последующим разделением команды на подгруппы: подгруппа отвечает свой код, чтобы никому не наступать на ноги, а затем борется с интеграцией и ищет источники проблем. Мы в Google предпочитаем, чтобы команды продолжали развивать общую базу кода и автоматизировали тестирование, непрерывную интеграцию, развертывание и поиск источников проблем для быстрого их выявления (глава 23).

Одна из наших самых больших баз кода — YouTube — представляет собой большое монолитное приложение на Python. Процесс выпуска новых версий этого приложения очень трудоемок, и в нем участвуют специалисты по сборке, менеджеры по выпуску и другие добровольцы. Почти в каждую версию этого приложения включается несколько тщательно отобранных изменений или исправлений. Существует также 50-часовой цикл регрессионного тестирования вручную, выполняемый удаленной командой контроля качества для каждого выпуска. Когда издержки на выпуск так высоки, начинает развиваться цикл ожидания выхода новой версии, чтобы ее можно было еще немного потестировать. Между тем кто-то хочет добавить еще одну функцию, которая почти готова, и вскоре вы получаете трудоемкий, медленный и подверженный ошибкам процесс выпуска версии. Хуже всего, если эксперты, готовившие

версии в последнее время, выгорят и покинут команду и никто не будет знать, как устранить странные сбои, возникающие при попытке выпустить обновление. Это может заставить вас паниковать только от одной мысли, что придется нажать на кнопку выпуска.

Когда выпуск новых версий обходится слишком дорого и сопряжен с риском, *инстинкт* подсказывает, что нужно уменьшить частоту выпусков и увеличить период стабильности. Однако это дает лишь краткосрочное улучшение и со временем снижает скорость разработки и расстраивает команды и пользователей. *Решение* заключается в снижении затрат и повышении дисциплины, при этом очень важно сопротивляться очевидным операционным исправлениям и инвестировать в долгосрочные архитектурные изменения. Вот самые очевидные решения этой проблемы: возвращение к традиционной модели планирования, которая оставляет мало места для обучения или итераций, добавление более жесткого управления и надзора за процессом разработки, а также внедрение обзоров рисков или вознаграждение за добавление функций, часто малоценных, зато не несущих больших рисков.

Однако наиболее выгодным вложением является переход на микросервисную архитектуру, которая даст большой группе разработчиков возможность делиться и оставаться инновационной, одновременно снижая риски. В некоторых случаях мы в Google переписывали приложение с нуля, а не просто переносили его в новую архитектуру. Любой из этих вариантов может занять месяцы и, вероятно, будет очень болезненным в краткосрочной перспективе, но эти затраты окупятся в течение нескольких лет жизни приложения.

Оценка изменений в изоляции: флаги управления функциями

Ключ к надежным непрерывным выпускам версий — добавление «охранных флагов» ко всем изменениям. По мере развития продукта на разных этапах его разработки будет иметься несколько функций, сосуществующих в двоичном файле. Защитные флаги могут использоваться для управления этими функциями в продукте и устанавливаться по-разному в сборках для выпуска и для разработки. Флаги, отключающие функции, должны позволять инструментам сборки исключать такие функции из выпуска, если это позволяет язык. Например, стабильная функция, которая уже доставлена клиентам, может быть включена как для разработки, так и для выпуска. Разрабатываемая функция может быть включена только для разработки, чтобы защитить пользователей от не готовой к выпуску функции. Код новой функции находится в двоичном файле вместе со старым кодом, и оба могут работать, но доступ к новому коду охраняется флагом. Если новый код работоспособен, вы сможете удалить старый код и открыть доступ к функции в следующем выпуске. Если есть проблема, значение флага можно обновить независимо от двоичной версии с помощью динамического обновления конфигурации.

Раньше нам приходилось тщательно приурочивать пресс-релизы к двоичным выпускам. Мы должны были сначала провести успешное развертывание и только потом публиковать пресс-релиз о новых возможностях или функциях. То есть новые функции становились доступны до того, как о них было объявлено, и имелся реальный риск, что их обнаружат раньше времени.

И в этой ситуации проявляется вся ценность защитных флагов. Если доступ к новому коду защищен флагом, его можно изменить, чтобы включить функцию непосредственно перед выпуском пресс-релиза, что минимизирует риск утечки информации о ней. Обратите внимание, что защита кода флагом не является идеальным решением для действительно важных функций. Код все еще можно обнаружить и проанализировать, если он плохо защищен, и не все функции можно скрыть за флагами, не увеличив при этом сложность. Более того, даже изменения флагов должны выполняться с осторожностью. Включение флага сразу для 100 % пользователей — не лучшая идея, поэтому хорошим вложением может стать организация службы, управляющей безопасным развертыванием конфигураций. Тем не менее возможность отделить конкретную функцию от общего выпуска продукта является мощным рычагом для продвижения долгосрочной устойчивости приложения.

Стремление к гибкости: создание серии выпусков

Двоичный файл Google Search является первым и самым старым. С его помощью можно воссоздать летопись происхождения Google — поиск в нашей кодовой базе все еще позволяет находить код, написанный по меньшей мере в 2003 году, а часто и раньше. Когда смартфоны начали набирать популярность, в код, написанный в первую очередь для развертывания на серверах, стали добавляться функции поддержки мобильных устройств. По мере того как приложение Search становилось все более интерактивным, развертывание сборок вызывало все больше сложностей. В какой-то момент мы выпускали двоичный файл Search только раз в неделю, и даже этот срок удавалось соблюсти не всегда.

Когда Шери Шипе (Sheri Shipe), один из авторов этой главы, взялся за работу по увеличению частоты выпусков Search, на подготовку каждого выпуска у группы инженеров уходило несколько дней. Они собирали двоичный файл, интегрировали данные, а затем начинали тестирование. Каждую обнаружившуюся ошибку приходилось исследовать вручную, чтобы убедиться, что она не повлияет на качество поиска, пользовательский опыт и/или доход. Это был изнурительный процесс, требовавший много времени и не соответствовавший ни объему, ни частоте изменений. В результате разработчик не имел возможности узнать, когда его функция будет выпущена в производство. Это затрудняло выбор времени для пресс-релизов и публичных заявлений.

Новые выпуски формируются не на пустом месте, и наличие надежного процесса подготовки выпусков упрощает синхронизацию зависимых факторов. В течение нескольких лет специальная группа инженеров внедряла непрерывный процесс

выпуска, который упростил все, что касалось передачи двоичного файла Search во внешний мир. Мы автоматизировали все, что могли, установили сроки внедрения новых функций и упростили интеграцию плагинов и данных в двоичный файл. Теперь мы можем выпускать новые двоичные файлы Search через день.

На какие компромиссы нам пришлось пойти, чтобы добиться предсказуемости цикла выпуска? Они сводятся к двум основным идеям, положенным в основу системы.

Идеальных двоичных файлов не бывает

Первая идея — *идеальных двоичных файлов не бывает*. Это особенно верно для сборок, включающих результаты труда десятков или даже сотен разработчиков, независимо разрабатывающих десятки важнейших функций. Исправить каждую ошибку невозможно, поэтому нам постоянно приходилось взвешивать такие вопросы, как: «Если переместить линию на два пиксела влево, повлияет ли это на отображение рекламы и потенциальный доход?», «Если немного изменить оттенок этого прямоугольника, пользователи с ослабленным зрением увидят в нем текст?» Остальная часть этой книги небезосновательно посвящена минимизации непредвиденных последствий для выпуска, но, в конце концов, мы должны признать, что ПО — в принципе сложная штука. Не существует идеального двоичного кода — каждый раз, когда в производство внедряется новое изменение, приходится принимать решения и идти на компромиссы. Ключевые показатели эффективности метрик с четкими пороговыми значениями позволяют запускать в эксплуатацию функции, даже если они не идеальны¹, и добавляют ясности при решении спорных вопросов.

В связи с этим вспоминается одна ошибка, связанная с редким диалектом, на котором говорят только на одном острове на Филиппинах. Когда пользователь формулировал поисковый вопрос на этом диалекте, он в ответ получал пустую веб-страницу. Нам нужно было определить, стоит ли потратить силы и время на исправление этой ошибки и отложить выпуск новой важной функции.

Мы бегали из офиса в офис, пытаясь выяснить, сколько людей говорит на этом языке, возникает ли ошибка каждый раз, когда пользователь вводил запрос на этом языке, и использовали ли эти люди Google на регулярной основе. Каждый инженер по качеству, с которым мы говорили, отсылал нас к вышестоящему специалисту. Наконец, собрав данные, мы задали вопрос старшему вице-президенту Search. Должны ли мы отложить выпуск важной функции, чтобы исправить ошибку, которая затронула лишь маленький остров на Филиппинах? Оказалось, каким бы маленьким ни был ваш остров, вы должны получать надежные и точные результаты поиска, поэтому мы отложили выпуск и исправили ошибку.

¹ Как говорят наши специалисты по надежности относительно «бюджета ошибок»: совершенство редко бывает лучшей целью — вы должны выяснить, каков допустимый бюджет ошибок и какая часть этого бюджета была потрачена в последнее время, а потом использовать эти цифры, чтобы отрегулировать компромисс между скоростью и стабильностью.

Соблюдение сроков выпусков версий

Вторая идея: *если вы опоздали на поезд, он уйдет без вас*. Хочется вспомнить словицу: «Сроки определены, а жизнь — нет». В какой-то момент, чтобы соблюсти график выпуска, вам придется принять жесткое решение и отказать разработчикам во включении в выпуск их новых функций. Честно говоря, никакие просьбы или мольбы не заставят нас включить функцию в последний выпуск после истечения крайнего срока.

Впрочем, есть одно *редкое* исключение. Представьте: поздний вечер пятницы и шесть инженеров-программистов в панике врываются в кабинет менеджера по выпуску. У них есть контракт с NBA, и они закончили работу над его реализацией минуту назад. Но изменения должны быть выпущены до завтрашней игры. Выпуск необходимо приостановить и добавить эту функцию в двоичный файл, иначе мы нарушим контракт! Инженер по выпуску с затуманенными глазами качает головой и говорит, что на сборку и тестирование нового двоичного файла уйдет четыре часа, а у его сына сегодня день рождения и ему еще нужно забрать воздушные шары.

В мире регулярных выпусков, если разработчик опоздал на поезд, он сможет сесть на следующий, отправляющийся через несколько часов, а не дней. Это ограничивает панику разработчиков и значительно улучшает баланс между работой и личной жизнью инженеров по выпуску.

Качество и ориентация на пользователя: поставляйте только то, что используется

Раздувание — неприятный побочный эффект жизненного цикла разработки любого ПО, и чем большим успехом пользуется продукт, тем больше раздувается его база кода. Один из недостатков быстрой и эффективной серии выпусков — раздувание часто ускоряется и может приводить к проблемам для команды, развивающей продукт, и даже для пользователей. В частности, если ПО поставляется непосредственно клиенту, как в случае с мобильными приложениями, это может означать, что на устройстве пользователя будет расходоваться больше пространства и ему придется платить за загрузку продукта, данные и даже функции, которые он никогда не использовал и не будет использовать, а разработчикам придется мириться с уменьшением скорости сборки, сложностью развертывания и редкими ошибками. В этом разделе мы поговорим о том, как динамическое развертывание позволяет поставлять только то, что используется, и о компромиссах между полезностью и стоимостью функций. В Google часто создаются специальные команды, занимающиеся увеличением эффективности продукта на постоянной основе.

Веб-приложения выполняются в облаке, а клиентские приложения используют ресурсы устройства пользователя — телефона или планшета. Выбор из двух типов приложений сам по себе демонстрирует компромисс: клиентские приложения часто обладают большей производительностью и устойчивостью к нестабильным соеди-

нениям, но они более трудные для обновления и более восприимчивы к проблемам на уровне платформы. Распространенный аргумент против частого развертывания клиентских приложений: пользователи не любят частые обновления и вынуждены платить за данные и нарушения работоспособности. Могут существовать и другие факторы, ограничивающие этот выбор, такие как доступность сети или лимит на количество перезагрузок устройства, необходимых для получения обновления.

Иногда приходится идти на компромисс и искусственно уменьшать частоту обновления продукта, но *этот выбор должен быть осознанным*. В хорошо отлаженном процессе непрерывной поставки частота *создания* жизнеспособных версий может не совпадать с частотой их *получения* пользователем. Вы можете развертывать новые версии еженедельно, ежедневно или ежечасно, но вам следует осознанно настроить процесс выпуска с учетом конкретных потребностей ваших пользователей и целей вашей организации, а также определить оптимальную модель поддержки устойчивости продукта в долгосрочной перспективе.

Выше в этой главе мы говорили о необходимости модульной организации кода. Она позволяет организовать динамическое и настраиваемое развертывание и эффективнее использовать ограниченные ресурсы, например пространство памяти на устройстве пользователя. Без нее каждый пользователь будет вынужден получать код, который он никогда не станет использовать, например для поддержки не нужных ему переводов или архитектур других типов устройств. Динамическое развертывание позволяет поддерживать небольшие размеры приложений и доставлять на устройство только код, который будет полезен пользователям, а A/B-тестирование помогает найти компромисс между затратами на создание функции и ее ценностью для пользователей и вашего бизнеса.

Настройка этих процессов требует затрат, а выявление и устранение трений, удерживающих частоту выпусков ниже желаемой, — кропотливый труд. Но долгосрочные выгоды с точки зрения управления рисками, скорости разработки и обеспечения быстрого внедрения инноваций настолько высоки, что эти затраты быстро окупятся.

Сдвиг влево: раннее принятие решений на основе данных

Если вы ориентируетесь на самый широкий круг пользователей, у вас могут быть клиенты с интеллектуальными экранами, акустическими системами или телефонами и планшетами на Android и iOS, а ваше ПО может быть достаточно гибким, чтобы пользователи могли настраивать его под свои нужды. Даже если вы разрабатываете только для устройств на Android, огромное разнообразие более чем двух миллиардов устройств может сделать перспективу специализации выпусков невыполнимой задачей. А с учетом скорости развития науки и техники, когда вы будете читать эту главу, могут появиться совершенно новые категории устройств.

Один из наших менеджеров по выпуску версий поделился мудростью, изменившей ситуацию. Он сказал, что разнообразие клиентского рынка — это не *проблема*, а *факт*. Приняв эту мудрость, мы можем изменить модель специализации версий следующим образом:

- Если *всеобъемлющее* тестирование невозможно, нужно использовать *репрезентативное* тестирование.
- Поэтапное развертывание с постепенным увеличением доли охватываемых пользователей позволяет быстро вносить исправления.
- Автоматические А/В-версии позволяют получать статистически значимые результаты, подтверждающие качество версии, без утомительной нагрузки на пользователей, которые должны смотреть на информационные панели и принимать решения.

В разработке ПО для Android Google использует специализированные пути тестирования и поэтапное развертывание с постепенно увеличивающейся долей пользователей, и на каждом этапе тщательно анализируются все возникающие проблемы. Поскольку Play Store предлагает неограниченное количество путей для тестирования, мы также можем создавать команды контроля качества в любой стране, где планируем выпуск, что позволяет в мгновение ока провести глобальное тестирование ключевых функций.

Одна из проблем, которые мы заметили при развертывании версий на Android, заключалась в том, что *простая отправка обновления* могла вызвать статистически значимые изменения в оценке продукта пользователями. То есть, даже если мы не вносим никаких изменений в продукт, выпуск обновления может оказать непредсказуемое влияние на поведение устройств и пользователей. В результате, несмотря на то что ограничение распространения обновления узким кругом пользователей может дать ценную информацию о сбоях или проблемах со стабильностью, оно не позволяло нам понять, действительно ли новая версия приложения лучше старой.

Дэн Сирокер и Пит Кумен уже обсудили в своей книге ценность А/В-тестирования¹. В Google некоторые из наших крупных приложений тоже проводят А/В-тестирование своих *развертываний*. Они отправляют две версии продукта, одна из которых имитирует желаемое обновление (то есть это старая версия приложения). Поскольку обе версии развертываются одновременно для достаточно широкого круга похожих пользователей, появляется возможность сравнить одну версию с другой и увидеть, действительно ли последняя версия ПО лучше предыдущей. При достаточно большой базе пользователей можно получить статистически значимые результаты в течение нескольких дней или даже часов. Автоматизированный конвейерный обработчик метрик может обеспечить максимальную скорость выпуска версии, распространив

¹ Siroker D., Koomen P. A/B Testing: The Most Powerful Way to Turn Clicks Into Customers. Hoboken: Wiley, 2013.

новую версию среди более широкого круга пользователей, как только будет накоплено достаточно данных, подтверждающих, что предельные метрики не затронуты.

Очевидно, что этот метод применим не ко всем приложениям и может привести к значительным накладным расходам при недостаточно большой базе пользователей. В этих случаях рекомендуется стремиться выпускать версии, нейтральные к изменениям. Все новые функции в таких версиях защищены флагами, поэтому единственное, что проверяется во время развертывания, — это стабильность самого развертывания.

Изменение культуры команды: дисциплина развертывания

Принцип «постоянной готовности к развертыванию» помогает решить некоторые проблемы, влияющие на скорость разработки, но кроме него существуют также определенные методы, решающие проблемы масштабирования. Команда, выпустившая продукт, может насчитывать менее десяти человек, каждый из которых по очереди выполняет развертывание и осуществляет мониторинг. Со временем команда может вырасти до сотен человек с подгруппами, отвечающими за определенные функции. По мере расширения организации количество изменений в каждом развертывании и количество рисков при каждой попытке выпуска новой версии увеличиваются в геометрической прогрессии. Каждый выпуск версии — это месяцы пота и слез. Успешный выпуск требует все больше усилий. Часто разработчик оказывается перед выбором, что хуже: отказаться от выпуска версии, которая на четверть состоит из новых функций и исправлений ошибок, или выпустить новую версию без уверенности в ее качестве.

Повышенная сложность обычно проявляется в увеличении задержки между выпусками. Даже если вы выпускаете новую версию каждый день, для подготовки полностью безопасного выпуска к развертыванию может потребоваться неделя или больше, и вам останется неделя для отладки любых проблем. Вот где принцип «постоянной готовности к развертыванию» может вернуть проект в эффективную форму. Частые выпуски позволяют минимизировать отклонения от заведомо хорошей версии, а новизна изменений помогает решать проблемы. Но как команда сможет гарантировать поступательное движение вперед при всей сложности, присущей большой и быстро расширяющейся кодовой базе?

В Google Maps мы считаем, что функции важны, но лишь немногие функции важны настолько, чтобы из-за них стоило отложить выпуск. Если выпуски происходят часто, задержка внедрения одной функций оказывает намного менее отрицательное влияние, чем задержка внедрения всех новых функций, особенно если пользователи могут столкнуться со спешно разработанной и не совсем готовой функцией.

Одна из обязанностей менеджера по выпуску версий — защитить продукт от разработчиков.

Чрезмерный энтузиазм, который разработчик испытывает при выпуске новой функции, никогда не должен мешать ему ориентироваться на пользовательский опыт. Это означает, что новые функции должны изолироваться от других компонентов версии с помощью интерфейсов со строгими контрактами, разделения задач, тщательного тестирования, своевременного обмена информацией, а также соглашения о внедрении новых функций.

Заключение

За долгие годы работы мы обнаружили, что, как ни странно, «чем быстрее, тем безопаснее». Состояние продукта и скорость разработки на самом деле не противоречат друг другу, и продукты, которые выпускаются чаще и с небольшим количеством изменений, имеют более высокое качество. Они быстрее приспосабливаются к ошибкам, встречающимся в продакшене, и к неожиданным изменениям рынка. Кроме того, «чем быстрее, тем *дешевле*», потому что наличие предсказуемой и частой последовательности версий снижает стоимость каждого выпуска и делает стоимость отмены любого выпуска очень низкой.

Простое наличие структур, *обеспечивающих* непрерывное развертывание, приносит большую выгоду, *даже если развертываемые выпуски не распространяются среди пользователей*. Что мы имеем в виду? На самом деле мы не выпускаем каждый день совершенно разные версии Search, Maps или YouTube. Но, чтобы быть готовыми к этому, нам нужны надежный и хорошо документированный процесс непрерывного развертывания, точные и актуальные показатели удовлетворенности пользователей и состояния продукта, а также скоординированная команда с четкой политикой в отношении процессов в продукте. Как показывает практика, для этого часто требуется настраивать двоичные файлы в продакшене, управлять конфигурациями (в VCS) и использовать набор инструментов, позволяющий предпринимать меры безопасности, такие как пробный прогон, механизмы отката и добавления исправлений.

Итоги

- *Скорость — это командная победа*: чтобы получить оптимальный рабочий процесс для большой команды, работающей с единой базой кода, необходимо использовать модульные архитектуры и почти непрерывную интеграцию.
- *Оценивайте изолированные изменения*: добавляйте защитные флаги к любым функциям, чтобы иметь возможность изолировать проблемы на раннем этапе обновления.
- *Сделайте реальность эталоном*: используйте приемы поэтапного внедрения, чтобы справиться с проблемой разнообразия устройств и широкого круга пользователей. Специализация версии для искусственной среды, не похожей на

продакшен, может привести к появлению неприятных сюрпризов на поздних стадиях обновления.

- *Поставляйте только то, что используется:* контролируйте стоимость и ценность любой существующей функции, чтобы знать, имеет ли она достаточную актуальность и ценность для пользователей.
- *Сдвиг влево* позволяет быстрее принимать обоснованные решения на более ранних этапах обновления с помощью непрерывной интеграции и непрерывного развертывания.
- *Чем быстрее, тем безопаснее:* поставляйте новые версии чаще и с небольшим количеством изменений, чтобы снизить риски каждого конкретного выпуска и минимизировать время выхода версии на рынок.

Вычисления как услуга

Автор: Онуфрий Войтащук

Редактор: Лиза Кэри

Я не пытаюсь понять компьютеры, я пытаюсь
понять программы.

Барбара Лисков

После того как код будет написан, вам понадобится оборудование для его запуска. То есть вы должны будете купить или арендовать это оборудование. По сути, это и есть *вычисления как услуга* (SaaS, compute as a service), где под словом «вычисления» подразумеваются вычислительные мощности, необходимые для фактического выполнения ваших программ.

В этой главе мы расскажем, как простая идея «взять оборудование для выполнения кода»¹ отображается в жизнеспособную систему, которая будет масштабироваться по мере роста и развития организации. Глава получилась довольно длинной, потому что описывает сложную тему, и поэтому разделена на четыре раздела:

- «Приручение вычислительной среды» описывает, как компания Google приручила вычислительную среду, и объясняет некоторые ключевые понятия SaaS.
- «Написание ПО для управляемых вычислений» показывает, как решение для управляемых вычислений влияет на подходы к разработке ПО. Мы считаем, что подход «скот, а не домашние любимцы» (гибкое планирование) сыграл важнейшую роль в успехе Google и является необходимым инструментом в арсенале программистов.
- «SaaS во времени и масштабе» более подробно рассматривает некоторые уроки, извлеченные компанией Google из применения разных вариантов вычислительной архитектуры, и их влияние на рост и развитие организации.

¹ Оговорка: для некоторых приложений под «оборудованием для выполнения» подразумевается оборудование ваших клиентов (представьте, например, игру в упаковке, которую вы купили десять лет назад). Это совсем другая проблема, которая не рассматривается в этой главе.

- Наконец, раздел «Выбор вычислительной услуги» предназначен в первую очередь для инженеров, которые будут принимать решение о выборе вычислительной услуги для своей организации.

Приручение вычислительной среды

Система Borg¹, разработанная в Google для внутренних нужд, стала предшественницей многих современных архитектур CaaS (таких, как Kubernetes или Mesos). Чтобы лучше понять, как ее отдельные аспекты отвечают потребностям растущей и развивающейся организации, мы проследим, как развивалась Borg и как инженеры Google приручали вычислительную среду.

Автоматизация труда

Представьте, что вы студент университета на рубеже веков. Чтобы развернуть новый код, вы должны переслать его по SFTP на одну из машин в компьютерной лаборатории университета, выполнить вход на эту машину через SSH, скомпилировать и запустить его. Это решение подкупает своей простотой, но с течением времени и увеличением масштаба оно сталкивается со значительными проблемами. Однако поскольку с него начинаются многие проекты, часто организации создают процессы, представляющие упрощенную эволюцию этой системы, по крайней мере для некоторых задач — количество машин увеличивается (вы используете SFTP и SSH для работы со многими из них), но основная технология остается неизменной. Например, в 2002 году Джефф Дин, один из старших инженеров Google, так охарактеризовал выполнение задач автоматической обработки данных в процессе подготовки к выпуску новой версии:

[Выполнение задачи] — это логистический кошмар, отнимающий много времени. В настоящее время требуется получить список из 50+ машин, и на каждой из 50+ машин запустить процесс и следить за его выполнением. Отсутствует возможность автоматического переноса вычислений на другую машину, если одна из машин выйдет из строя, а мониторинг выполнения заданий осуществляется ситуативно <...> Кроме того, поскольку процессы могут мешать друг другу, существует сложный, реализованный человеком файл «регистрации» для ограничения использования мощности машин, что приводит к неоптимальному планированию и усилению конкуренции за использование ограниченных машинных ресурсов.

Названные сложности стали одним из первых стимулов для Google направить усилия на приручение вычислительной среды. Они демонстрируют, как простое решение становится неудобным с увеличением масштаба.

¹ Verma A., Pedrosa L., Korupolu M. R., Oppenheimer D., Tune E., Wilkes J. Large-scale cluster management at Google with Borg. EuroSys, Article No.: 18 (April 2015): 1–17.

Простая автоматизация

Организация может предпринять довольно простые шаги, чтобы немного ослабить проблемы выполнения задач. Процесс развертывания двоичного файла на каждой из 50+ машин и его запуск можно автоматизировать с помощью простого сценария командной оболочки или — если это должно быть многоразовое решение — с использованием более надежного инструмента, чтобы получить более простое средство, которое будет выполнять развертывание параллельно (тем более что «50+» со временем наверняка увеличится).

Мониторинг каждой машины тоже можно автоматизировать. Прежде всего инженер, ответственный за процесс мониторинга, должен иметь возможность своевременно узнавать о проблемах, возникающих на машинах. Показатели (например, «процесс активен» и «количество обработанных документов») он должен получать из процесса и записывать в общее хранилище или передавать в службу мониторинга, чтобы потом быстрее находить аномалии в процессе. Для этой цели можно использовать решения с открытым исходным кодом, такие как Graphana или Prometheus, позволяющие создавать панели мониторинга.

Типичная последовательность действий при обнаружении аномалии: подключиться к машине по SSH, прервать процесс (если он продолжает выполняться) и запустить его снова. Это утомительно, и есть риск допустить ошибку (нужно убедиться, что подключение выполняется к нужной машине и прерывается нужный процесс), поэтому эти действия тоже желательно автоматизировать:

- Вместо мониторинга сбоев вручную можно запустить программу-агент на машине, которая будет определять аномалии (например, «процесс не сообщал о работоспособности в течение последних 5 минут» или «процесс не обработал ни одного документа за последние 10 минут») и завершать процесс.
- Чтобы избавиться от необходимости входить в систему для повторного запуска процесса после его аварийной остановки, можно заключить его запуск в простой сценарий оболочки `while true; do run && break; done`.

Эквивалент этой процедуры в облачном мире — определение политики автоматического восстановления (уничтожение и повторное создание виртуальной машины или контейнера после того, как они не пройдут проверку работоспособности).

Эти относительно простые улучшения решают часть проблем, упомянутых Джеффом Дином и описанных выше. Но ручное регулирование и автоматический перенос задания на новые машины требуют более сложных решений.

Автоматизация планирования

Следующий шаг — автоматизация распределения заданий между машинами. Для этого нужна первая настоящая «услуга», которая в итоге вырастет в SaaS. То есть, чтобы автоматизировать планирование, нужна некая центральная служба, которая имеет полный список доступных машин и может по запросу выбрать незанятые машины и автоматически развернуть на них двоичный файл. Это избавит от необходимости

вручную поддерживать файл «регистрации» и делегировать эту работу компьютерам. Такая система сильно напоминает ранние архитектуры с разделением времени.

Продолжением этой идеи является объединение планирования с реакцией на отказ. Сканируя журналы машин на наличие выражений, сигнализирующих о проблемах (например, массовых ошибках чтения с диска), можно выявить проблемные машины, сообщить (сотрудникам) о необходимости ремонта таких машин и одновременно исключить планирование каких-либо работ на этих машинах. Для избавления от рутинного труда можно сначала попробовать автоматически исправить проблемы, прежде чем привлекать к их устранению инженера, например перезагрузить машину в надежде, что проблема исчезнет, или запустить автоматическое сканирование диска.

Последнее замечание в цитате Джеффа касается необходимости вручную переносить вычисления на другую машину, если машина, на которой они выполнялись до этого, вдруг сломается. Эта проблема имеет простое решение: поскольку уже существуют автоматизация планирования и возможность выявлять неработоспособные машины, можно просто заставить планировщик выделить новую машину и запустить задание на ней. Сигнал к этому может исходить от демона мониторинга машины или отдельного процесса.

Все эти улучшения напрямую связаны с растущим масштабом организации. Когда парк состоял из одной машины, SFTP и SSH были идеальными решениями, но в масштабе сотен или тысяч машин без автоматизации не обойтись. Цитата, с которой мы начали, взята из проектного документа 2002 года «*Global WorkQueue*» — раннего внутреннего решения по применению SaaS для некоторых рабочих нагрузок в Google.

Контейнерная и многоарендная архитектура

До сих пор мы неявно предполагали взаимно-однозначное соответствие между машинами и программами, выполняющимися на них. Это очень неэффективно с точки зрения потребления вычислительных ресурсов (ОЗУ, ЦП):

- Количество разных типов заданий (с разными требованиями к ресурсам) почти наверняка будет превышать количество типов машин (с разным объемом доступных ресурсов), поэтому для многих заданий придется использовать машины одного и того же типа (пригодного для выполнения заданий, самых требовательных к ресурсам).
- Потребности в программных ресурсах со временем растут, а на развертывание новых машин требуется много времени. Если на приобретение новых, более крупных машин в организации уходят месяцы, то вы должны приобретать достаточно мощные машины, чтобы удовлетворить ожидаемый рост потребностей в ресурсах. Это приведет к напрасным расходам, потому что первое время мощности новых машин не используются полностью¹.

¹ Этот и следующий пункты не так актуальны, если ваша организация арендует машины у поставщика облачных услуг.

- После поступления новых машин старые никуда не денутся (выбрасывать их, вероятно, будет слишком расточительно), поэтому вам придется управлять разнородным парком, плохо адаптирующимся к вашим потребностям.

Естественное решение — определить для каждой программы ее требования к ресурсам (ЦП, ОЗУ, дисковое пространство), а затем попросить планировщик подыскать для программы подходящий набор машин.

Соседская собака лает в моем ОЗУ

Решение, описанное выше, отлично работает, если все компоненты действуют слаженно. Но если я укажу в конфигурации, что каждая программа в моем конвейере обработки данных потребляет один ЦП и 200 Мбайт ОЗУ, а затем — из-за ошибки или естественного роста — они начнут потреблять больше, то возникнет риск исчерпания ресурсов на машинах, на которых эти программы запланированы. В случае нехватки процессоров соседние задания будут испытывать всплески задержек, а в случае нехватки ОЗУ — процесс может быть аварийно завершен ядром или испытывать жуткие задержки из-за подкачки с диска¹.

Две программы могут плохо уживаться друг с другом на одном компьютере и по другим причинам. Многие программы требуют установки зависимостей какой-то конкретной версии, и эти требования могут противоречить требованиям к версиям другой программы. Программа может ожидать, что определенные системные ресурсы (например, `/tmp`) будут доступны ей для монопольного использования. В отношении безопасности программа может обрабатывать конфиденциальные данные и должна быть уверена, что другие программы на том же компьютере не смогут получить к ним доступ.

То есть многоарендная вычислительная услуга должна обеспечивать определенную степень изоляции, своего рода гарантию, что процесс сможет безопасно выполняться, не вступая в конфликты с другими арендаторами машины.

Классическим решением проблемы изоляции является использование виртуальных машин (VM). Однако оно сопряжено со значительными накладными расходами² с точки зрения ресурсов (необходимы ресурсы для запуска полноценной ОС внутри каждой VM) и времени запуска (требуется загрузить полную ОС). Это делает их не самым идеальным решением для контейнеризации пакетных заданий, быстро выполняющихся и требующих небольшой объем ресурсов. По этой причине инженеры Google, разработавшие Borg в 2003 году, начали искать другие решения и в итоге пришли к *контейнерам* — легковесному механизму, основанному на контрольных группах (cgroups, предложенных инженерами Google для включения в ядро Linux

¹ В Google уже давно решили, что задержки из-за подкачки с диска настолько ужасны, что в случае нехватки памяти лучше прервать процесс и перенести его на другую машину, поэтому в Google нехватка памяти всегда вызывает завершение процесса.

² Несмотря на значительное количество исследований, направленных на снижение накладных расходов, эти расходы никогда не будут такими же низкими, как в случае процесса, выполняющегося непосредственно.

в 2007 году) и chroot-клетках (chroot jails), который использует технологию монтирования связей (bind mounts) и/или объединения/наложения файловых систем для их изоляции. В числе известных реализаций контейнеров с открытым исходным кодом можно назвать Docker и LMCTFY.

С течением времени и развитием организации будет возникать все больше проблем, связанных с недостаточной изоляцией. Вот конкретный пример: в 2011 году инженеры, работающие над Borg, обнаружили, что исчерпание пространства идентификаторов процессов (которое по умолчанию ограничивалось 32 000 идентификаторов) вызывает ошибки изоляции и ограничивает общее количество процессов (поток) в одной реплике. Чуть позже мы рассмотрим этот пример подробнее.

Выбор оптимальной конфигурации и автоматическое масштабирование

В 2006 году система Borg планировала задания, опираясь на параметры, такие как количество реплик и требования к ресурсам, которые определялись инженером.

Если взглянуть на проблему со стороны, то идея просить людей определить потребности в ресурсах выглядит ошибочной, потому что это не те величины, с которыми они имеют дело ежедневно. То есть со временем эти параметры конфигурации сами становятся источником неэффективности. Инженеры должны потратить время на их определение при первом запуске службы, и по мере того как организация накапливает все больше и больше услуг, стоимость их определения будет расти. Более того, с течением времени программы тоже развиваются (и, вероятно, увеличиваются их потребности), а параметры конфигурации не успевают за ними. В конце концов возникает аварийная ситуация, при разборе которой выясняется, что новые выпуски требовали все больше ресурсов и уменьшили резерв, оставшийся на случай неожиданных всплесков, а когда такой всплеск произошел, оставшегося резерва оказалось недостаточно.

Естественное решение — автоматизировать настройку этих параметров. К сожалению, реализовать его на удивление сложно. Например, в Google лишь недавно была достигнута точка, когда более половины потребностей ресурсов всего парка Borg стали определяться автоматически. Тем не менее, несмотря на то что охвачена только половина, она включает большую часть конфигураций, а это означает, что большинству инженеров не нужно тратить силы и время на определение потребностей своих контейнеров. Мы рассматриваем это как успешное применение идеи «простое должно быть легким, а сложное — возможным» — сам факт, что некоторая доля рабочих нагрузок Borg слишком сложна для автоматического управления их параметрами, совсем не означает, что автоматизация не дает преимуществ в простых случаях.

Итоги

По мере роста организации и популярности продуктов будут расти и эти направления:

- Количество приложений, которыми необходимо управлять.
- Количество копий приложений, которые требуется запустить.
- Размер наибольшего приложения.

Для эффективного управления масштабом необходима автоматизация, охватывающая все эти направления роста. Со временем можно ожидать, что сама автоматизация станет более активной как в обработке новых типов требований (например, распределение заданий между графическими и тензорными процессорами — это одно из существенных изменений в Borg за последние 10 лет), так и в увеличении масштаба. Действия, которые легко выполнить вручную в меньшем масштабе, необходимо автоматизировать, чтобы избежать коллапса организации с увеличением нагрузки.

Один из примеров — продолжающийся в Google процесс автоматизации управления *центрами обработки данных*. Десять лет назад каждый центр был отдельным объектом. Мы управляли ими вручную. Включение центра обработки данных было сложным и рискованным ручным процессом, требующим специальных навыков и занимавшим недели (при полной готовности машин). Однако с увеличением числа центров обработки данных Google мы перешли к модели, в которой включение центра превратилось в автоматизированный процесс.

Написание ПО для управляемых вычислений

Переход от списков машин, управляемых вручную, к автоматизированному планированию и настройке значительно упростил управление парком машин, но также потребовал глубоких изменений в подходах к проектированию и разработке ПО.

Проектирование с учетом возможности отказов

Представьте, что инженер должен обработать пакет, включающий миллион документов, и проверить их правильность. Если обработка одного документа занимает одну секунду, то на обработку всего пакета на одной машине понадобится примерно 12 дней. Чтобы сократить время обработки до более приемлемых 100 минут, мы распределяем работу между 200 машинами.

Как обсуждалось в разделе «Автоматизация планирования» выше, планировщик Borg может в одностороннем порядке остановить любую из 200 рабочих реплик и переместить вычисления на другую машину¹. То есть он может запустить новую реплику без участия человека, настройки переменных окружения или установки пакетов.

Переход от «инженер должен вручную контролировать каждую из 100 задач и вмешиваться в случае поломки» к «если что-то пойдет не так с одной из задач, система автоматически остановит вычисления и запустит их на другой машине» был описан много лет спустя с помощью аналогии «домашние любимцы против скота»².

¹ Планировщик делает по конкретным причинам (например, если появится необходимость обновить ядро, обнаружится неисправность диска на машине или в ходе перепланирования для более равномерного распределения рабочих нагрузок в центре обработки данных). Однако смысл наличия вычислительной услуги в том, что автор ПО не должен думать о причинах, по которым это может произойти.

² Автором метафоры «домашние любимцы и скот», по мнению Рэнди Биаса (Randy Bias), считается Билл Бейкер (Bill Baker, <https://oreil.ly/ILYjI>). Она чрезвычайно популярна как

Если сервер — это домашний любимец, то, когда он ломается, приходит человек (обычно в панике), чтобы посмотреть, что произошло, и исправить проблему, если получится. Заменить любимца сложно. Если серверы — это рабочий скот, то им даются безликие имена от replica001 до replica100, и когда один из них выходит из строя, автоматический планировщик удалит соответствующий экземпляр и создаст взамен новый. Отличительная черта «рабочего скота» — простота запуска нового экземпляра для выполнения задания: эта процедура не требует ручной настройки и может выполняться полностью автоматически. Это обеспечивает свойство самовосстановления, описанное выше, — в случае сбоя автоматизация может удалить неисправный экземпляр и заменить его новым без участия человека. Обратите внимание: оригинальная метафора относится к серверам (виртуальным машинам), но она в равной степени применима и к контейнерам: если есть возможность запустить новую версию контейнера из образа без участия человека, то автоматизация сможет поддерживать услугу в работоспособном состоянии.

Если серверы — это домашние любимцы, то бремя их обслуживания будет расти линейно или даже сверхлинейно с размером парка серверов, и ни одна организация не должна легкомысленно относиться к этому бремени. Если же серверы — это рабочий скот, то ваша система сможет сама вернуться в стабильное состояние после сбоя и вам не придется тратить силы и время на восстановление ее работоспособности.

Однако недостаточно просто превратить виртуальные машины или контейнеры в скот, чтобы гарантировать автоматическое восстановление системы после сбоя. Управляя парком из 200 машин, система Borg почти наверняка остановит хотя бы одну реплику, возможно, даже не один раз, и каждая такая остановка будет увеличивать общую продолжительность вычислений на 50 минут (или на время, потраченное на остановленную обработку). Чтобы преодолеть эту проблему, нужна другая архитектура обработки: вместо статического распределения задач нужно разбить весь набор из миллиона документов, например на 1000 блоков по 1000 документов в каждом. Когда какая-то реплика закончит обработку фрагмента, она сообщит полученные результаты и возьмет другой фрагмент. То есть в случае сбоя мы потеряем времени не больше, чем требуется на обработку одного фрагмента. Это очень хорошо согласуется с архитектурой обработки данных, которая была стандартной в Google в то время: работа не распределялась равномерно между репликами в начале вычислений — задания назначались динамически в ходе обработки, что позволяло снизить затраты из-за отказов реплик.

Точно так же для систем, обслуживающих пользовательский трафик, желательно, чтобы перепланирование контейнеров не приводило к ошибкам у пользователей. Когда по каким-то причинам планировщик Borg решает остановить контейнер и запустить новый, он заранее уведомляет контейнер о своем намерении. Контейнер может отреагировать на это, отклонив новые запросы, и при этом у него останется

способ описания идеи «тиражируемого программного модуля». Ее можно использовать также для описания других понятий, а не только серверов (глава 22).

время, чтобы завершить текущие запросы. Это, в свою очередь, требует, чтобы система балансировки нагрузки понимала ответ реплики: «Я не могу принять новые запросы» и перенаправляла трафик на другие реплики.

Итак, если контейнеры или серверы действуют как рабочий скот, то ваша служба сможет автоматически вернуться в работоспособное состояние, но, чтобы обеспечить бесперебойную работу при умеренном уровне сбоев, необходимы дополнительные усилия.

Пакетные задания и задания обслуживания

Документ «*Global WorkQueue*» (описанный в первом разделе этой главы) решает проблему так называемых «пакетных заданий» — программ, которые, как ожидается, будут выполнять конкретные задания (например, обработку данных) от начала и до конца. Каноническим примером пакетных заданий может служить анализ журналов или обучение модели машинного обучения. Пакетные задания отличаются от «заданий обслуживания» — программ, которые, как ожидается, будут выполняться бесконечно и обслуживать входящие запросы. Каноническим примером таких заданий может служить задание, обслуживающее поисковые запросы пользователей на основе предварительно созданного индекса.

Эти два типа заданий имеют (обычно) разные характеристики¹, в частности:

- Пакетные задания в первую очередь ориентированы на высокую скорость обработки. Обслуживающие задания ориентированы на сокращение задержки в обслуживании одного запроса.
- Пакетные задания недолговечны (выполняются минуты или максимум часы). Обслуживающие задания обычно долговечны (по умолчанию перезапускаются только после развертывания новых версий).
- Поскольку обслуживающие задания долговечны, они обычно требуют больше времени для запуска.

До сих пор мы приводили в пример в основном пакетные задания, и, как мы видели, чтобы адаптировать пакетное задание к сбоям, достаточно разбить работу на небольшие части и динамически распределять эти части между рабочими репликами. Исторически для этого в Google использовался MapReduce², позже замененный фреймворком Flume³.

¹ Эта классификация, как и любые другие, не идеальна, и есть программы, которые не вписываются ни в одну из категорий или обладают характеристиками, типичными как для обслуживающих, так и для пакетных заданий. Однако, как и в большинстве других случаев, эта классификация все же фиксирует различие, наблюдаемое во многих реальных ситуациях.

² Dean J., Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. 6th Symposium on Operating System Design and Implementation (OSDI), 2004.

³ Chambers C., Raniwala A., Perry F., Adams S., Henry R., Bradshaw R., Weizenbaum N. Flume-Java: Easy, Efficient Data-Parallel Pipelines. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2010.

Обслуживающие задания во многих отношениях лучше подходят для обеспечения отказоустойчивости, чем пакетные задания. Их работа естественным образом разбивается на небольшие части (индивидуальные запросы пользователей), которые динамически назначаются репликам в соответствии со стратегией обработки большого потока запросов посредством балансировки нагрузки в кластере серверов, которая использовалась с первых дней обслуживания интернет-трафика.

Однако некоторые обслуживающие приложения не укладываются в этот паттерн. Каноническим примером может служить любой сервер, который интуитивно описывается как «лидер» определенной системы. Такой сервер обычно хранит состояние системы (в памяти или в своей локальной файловой системе), и если машина, на которой он работает, выйдет из строя, вновь созданный экземпляр обычно не может воссоздать состояние системы. Другой пример: обработка больших объемов данных — больше, чем умещается на одной машине, — из-за чего данные приходится сегментировать, например, между 100 серверами, каждый из которых хранит 1 % данных, и обрабатывать запросы к этой части данных. Это похоже на статическое распределение работы между пакетными заданиями: если один из серверов выйдет из строя, на время теряется возможность обслуживать часть данных. Последний пример: ваш сервер известен в других частях системы по имени хоста. В этом случае, независимо от устройства сервера, если сетевое соединение с этим конкретным хостом будет потеряно, то другие части системы не смогут с ним связаться¹.

Управление состоянием

В предыдущем описании мы обращали особое внимание на *состояние* как источник проблем в подходе «реплики как скот»². Каждый раз, когда одно задание заменяется другим, теряется все его текущее состояние (а также все, что находилось в локальном хранилище, если задание было перемещено на другой компьютер). Это означает, что текущее состояние следует рассматривать как временное, а «реальное хранилище» должно находиться в другом месте.

Проще всего организовать хранение во внешнем хранилище. То есть все, что должно существовать дольше, чем обслуживается один запрос (в случае обслуживающих заданий) или обрабатывается один блок данных (в случае пакетных заданий), необходимо хранить вне компьютера, выполняющего задание, в надежном постоянном

¹ Adya A. et al. Auto-sharding for datacenter applications, OSDI, 2019; Atul Adya, Daniel Myers, Henry Qin, and Robert Grandl, «Fast key-value stores: An idea whose time has come and gone», HotOS XVII, 2019.

² Обратите внимание, что кроме распределенного состояния существуют и другие требования к настройке эффективного решения «серверы как скот», такие как наличие систем обнаружения и балансировки нагрузки (чтобы приложение, которое перемещается по центру обработки данных, оставалось доступным). Поскольку эта книга посвящена не построению полной инфраструктуры SaaS, а тому, как такая инфраструктура соотносится с искусством программной инженерии, мы не будем вдаваться в дополнительные подробности.

хранилище. Если локальное состояние не изменяется в ходе работы, то сделать приложение отказоустойчивым можно относительно безболезненно.

К сожалению, большинство приложений не настолько просты. Возникает вопрос: «Как реализовать долговременное и надежное хранение — тоже как рабочий скот?» Ответ — «да». Управлять хранимым состоянием скот может посредством его репликации (копирования). Аналогичную идею, хотя и на другом уровне, представляют RAID-массивы: диски интерпретируются как временные хранилища (учитывая, что любой из них может выйти из строя), но все вместе они хранят общее состояние. В мире серверов эту идею можно реализовать с помощью нескольких реплик, хранящих один фрагмент данных, и механизма синхронизации, гарантирующего создание достаточного количества копий каждого фрагмента данных (обычно от 3 до 5). Обратите внимание, что правильно настроить такую систему очень сложно (для работы с записями требуется их согласовать), поэтому в Google был разработан ряд специализированных решений для хранения данных¹, подходящих для большинства приложений, использующих модель, в которой все состояния являются временными.

Другие типы локальных хранилищ, которые может использовать скот, — «воссоздаваемые» данные, которые хранятся локально для уменьшения задержки обслуживания. Наиболее очевидным примером является кеширование: кеш — это не что иное, как временное локальное хранилище, где хранится временное состояние, но хранилища с состоянием не исчезают постоянно, что позволяет улучшить характеристики производительности. Ключевым уроком для производственной инфраструктуры Google стала организация кеша для достижения целевых показателей по задержке при полной нагрузке основного приложения. Это позволило нам избежать сбоев при потере кеша, потому что имелся другой путь для обработки общей нагрузки (хотя и с большей задержкой). Однако требуется компромисс в решении, сколько ресурсов потратить на избыточность, чтобы снизить риск сбоя при потере кеша.

При «разогреве» приложения данные могут извлекаться из внешнего хранилища в локальное, чтобы уменьшить задержку обслуживания запросов.

Еще один случай использования локального хранилища — на этот раз для данных, которые записываются чаще, чем читаются, — пакетная запись. Эта стратегия широко используется для мониторинга данных (представьте сбор статистики об использованном процессорном времени в парке виртуальных машин с поддержкой автоматического масштабирования). Ее можно использовать везде, где допустимо исчезновение части данных, например, потому что не требуется 100%-ный охват данных (случай мониторинга) или исчезающие данные можно воссоздать (характерно

¹ См., например, *Ghemawat S., Gobioff H., Leung S.-T.* The Google File System. Proceedings of the 19th ACM Symposium on Operating Systems, 2003; *Chang F. et al.* Bigtable: A Distributed Storage System for Structured Data. 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI); *Corbett J. C. et al.* Spanner: Google's Globally Distributed Database. OSDI, 2012.

для пакетных заданий, которые обрабатывают данные по частям и записывают некоторые результаты для каждой части). Обратите внимание, что во многих случаях, даже если конкретное вычисление занимает много времени, его можно разделить на более мелкие временные окна, периодически записывая контрольные точки с состоянием в долговременное хранилище.

Подключение к услуге

Как упоминалось выше, если какой-то узел в системе, где выполняется программа, имеет жестко определенное имя (или даже определяемое в виде параметра конфигурации при запуске), реплики программы перестают быть скотом. Однако для подключения к вашему приложению другому приложению необходимо откуда-то получить ваш адрес. Но откуда?

Эта проблема решается введением дополнительного уровня косвенности, чтобы другие приложения могли получать ссылку на ваше приложение, используя идентификатор, который не изменяется при перезапуске «серверных» экземпляров. Определение ссылки по идентификатору может выполняться другой системой, в которой планировщик регистрирует ваше приложение, запуская его на определенном компьютере. Чтобы избежать операций поиска в распределенном хранилище при выполнении запросов к вашему приложению, клиенты, скорее всего, будут определять адрес вашего приложения, устанавливая соединение с ним во время запуска и следить за работоспособностью соединения в фоновом режиме. Этот прием обычно называется *обнаружением услуг*, и многие вычислительные услуги имеют свои встроенные или модульные решения. Большинство таких решений также включают некоторую форму балансировки нагрузки, которая еще больше снижает привязку к конкретным сетевым узлам.

При использовании этой модели иногда может потребоваться повторять запросы, потому что сервер с приложением, которому вы посылаете запросы, может отключиться до того, как ему удастся ответить¹. Повторные запросы являются стандартной практикой в сетевых взаимодействиях (например, между мобильным приложением и сервером) из-за возможных ошибок в сети, но они не используются, например, во взаимодействиях сервера с базой данных. По этой причине важно проектировать API серверов так, чтобы они правильно обрабатывали такие сбои. В случае изменяющихся запросов организация повторных попыток запросов может оказаться сложной задачей. В таких ситуациях важно гарантировать *идемпотентность* — результат, возвращаемый в ответ на запрос, не должен зависеть от количества от-

¹ Обратите внимание, что повторные попытки должны быть реализованы правильно — с отсрочкой, постепенным увеличением задержки между попытками и инструментами, позволяющими избежать каскадных сбоев. По этой причине такие реализации, вероятно, должны быть частью библиотеки вызова удаленных процедур, а не создаваться вручную каждым разработчиком. См., например, главу 22 в «Site Reliability Engineering. Надежность и безотказность как в Google» (<https://oreil.ly/aVCy4>).

правленных запросов. Одним из полезных инструментов, помогающих добиться идемпотентности, являются идентификаторы, назначаемые клиентом: если вы заказываете пиццу домой, клиент присваивает заказу идентификатор, и если заказ с этим идентификатором уже был зафиксирован, сервер предполагает, что это повторный запрос, и сообщает об успешном выполнении (он также может проверить соответствие параметров заказа).

Бывает, что планировщик теряет связь с определенной машиной из-за какой-то сетевой проблемы. В этом случае он решает, что вся работа потеряна, и переносит ее на другие машины, а затем соединение с машиной восстанавливается! В результате появляются два экземпляра программы на двух разных машинах, и обе думают, что они, например, «герлица072». Для устранения неоднозначности нужно проверить, на какую из них ссылается система разрешения адресов (а другая должна остановить программу или прекратить работу), — это еще один вид идемпотентности: две реплики, выполняющие одну и ту же работу, являются еще одним потенциальным источником дублирования запросов.

Одноразовый код

Выше большая часть обсуждения была сосредоточена на надежных заданиях, обслуживающих пользовательский трафик или конвейеры, производящие данные в продакшене. Однако инженерам-программистам нередко приходится выполнять однократный анализ, проверять прототипы и нестандартные конвейеры обработки данных и многое другое. Для этого нужны вычислительные ресурсы.

Часто рабочая станция инженера обладает достаточными вычислительными ресурсами для этого. Если инженер решит, например, проанализировать 1 Гбайт журналов, созданных службой за день, чтобы проверить, всегда ли подозрительная строка *A* появляется перед строкой ошибки *B*, он может просто загрузить журналы, написать короткий сценарий на Python и дать ему поработать минуту или две.

Но если он решит проанализировать 1 Тбайт журналов, созданных службой за год (с той же целью), ему придется ждать результатов в течение примерно одного дня, что, вероятно, неприемлемо. Вычислительная услуга, которая позволяет инженеру выполнить анализ в распределенной среде за несколько минут (с использованием нескольких сотен ядер), находится в пределах между «сейчас» и «завтра». Для повторяющихся задач (например, уточнения запроса после просмотра результатов) вычислительная услуга попадет в пределы между «к концу дня» и «никогда».

Иногда разрешение инженерам выполнять разовые задания в распределенной среде может привести к напрасному расходованию ресурсов. Маловероятно, что обработка, выполняемая инженером, будет стоить дороже, чем время, затраченное инженером на написание кода обработки. Выбор того или иного компромисса зависит от вычислительной среды организации и сколько она платит своим инженерам, но едва ли тысячу процессорных часов можно сравнить со стоимостью одного дня работы инженера. Вычислительные ресурсы в этом отношении похожи на маркеры, о кото-

рых мы говорили в начале книги; компания может получить некоторую экономию, внедрив процесс для получения дополнительных вычислительных ресурсов, но стоимость упущенных технических возможностей и времени наверняка превысит экономию.

И все же вычислительные ресурсы отличаются от маркеров — их легко по случайности взять слишком много. Маловероятно, что кто-то унесет домой тысячу маркеров, но вполне возможно, что кто-то случайно напишет программу, которая задействует тысячу машин, и не заметит этого¹. Естественным решением этой проблемы является введение ограничений на ресурсы, доступные отдельным инженерам. Альтернативный вариант, используемый в Google: благодаря эффективному выполнению низкоприоритетных пакетных рабочих нагрузок (см. раздел о многоарендности далее в этой главе) мы можем дать нашим инженерам практически неограниченное количество вычислительных ресурсов для низкоприоритетных пакетных заданий, что вполне подходит для большинства разовых инженерных задач.

СаaS во времени и масштабе

Выше мы говорили о том, как развивались СаaS в Google, и о том, как простая потребность «дай мне ресурсы для выполнения моих заданий» трансформируется в реальную архитектуру, подобную Borg. Некоторые аспекты влияния архитектуры СаaS на жизнь ПО во времени и масштабе заслуживают более пристального внимания.

Контейнеры как абстракция

Контейнеры, как уже отмечалось выше, в первую очередь играют роль механизма изоляции и способа поддержки многоарендности при минимальном взаимовлиянии различных задач друг на друга, совместно выполняющихся на одной машине. Это было первоначальной мотивацией их создания, по крайней мере в Google. Но, как оказалось, контейнеры также играют очень важную роль в абстрагировании вычислительной среды.

Контейнер обеспечивает границу абстракции между развернутым ПО и реальной машиной, на которой оно выполняется. То есть по мере изменения машины со временем необходимо адаптировать только ПО поддержки контейнеров (предположительно управляемое одной командой), тогда как прикладное ПО (управляемое отдельными командами на разных этапах развития организации) может оставаться без изменений.

Давайте обсудим, как контейнерная абстракция позволяет организации управлять изменениями.

¹ В Google такое случалось несколько раз, например из-за того, что кто-то оставил инфраструктуру нагрузочного тестирования, занимающую тысячу виртуальных машин Google Compute Engine, уйдя в отпуск, или из-за того, что новый сотрудник отлаживал главный двоичный файл на своей рабочей станции, не заметив, что тот породил 8000 реплик.

Абстракция файловой системы дает возможность внедрения ПО, написанного за пределами компании, без необходимости управлять нестандартными конфигурациями. Это может быть ПО с открытым исходным кодом, которое организация использует в своем центре обработки данных, или приобретения, которые она хочет встроить в СааS. Без абстракции файловой системы запуск двоичного файла, который ожидает другой конфигурации файловой системы (например, наличие вспомогательного двоичного файла в `/bin/foo/bar`), потребует изменения базовой конфигурации всех машин в парке, либо фрагментации парка, либо изменения ПО (что может быть затруднительно или даже невозможно из-за лицензионных требований).

Все эти решения могут быть вполне осуществимыми, если импорт внешнего ПО происходит раз в жизни, но ни одно из них не годится, если импорт ПО является обычной (или даже редкой) практикой.

Абстракция файловой системы также помогает в управлении зависимостями, потому что позволяет их предварительно объявлять и упаковывать (например, определенные версии библиотек, необходимые для запуска ПО). Если появится протекающая абстракция, вынуждающая всех использовать одну и ту же версию скомпилированных библиотек, обновление любого компонента станет трудным или вообще невозможным.

Контейнеры также дают простую возможность управления *именованными ресурсами* на машине. Канонический пример — сетевые порты. В числе других именованных ресурсов можно назвать графические процессоры и другие ускорители.

Изначально в Google сетевые порты не были включены в абстракцию контейнеров, поэтому двоичным файлам приходилось искать свободные порты самостоятельно. В результате в кодовой базе на C++ в Google функция `PickUnusedPortOrDie` использовалась более чем в 20 000 местах. Механизм контейнеров Docker, созданный после добавления пространств имен в ядро Linux, использует эти пространства для поддержки виртуальных и частных сетевых адаптеров (NIC), благодаря чему приложения могут прослушивать любые порты по своему выбору. Сетевой стек Docker отображает порт на несущей машине с портом в контейнере. Фреймворк Kubernetes, изначально основанный на Docker, пошел еще дальше и требует от сетевой реализации интерпретировать контейнеры («поды» в терминологии Kubernetes) как «реальные» IP-адреса, доступные в сети хоста. Теперь каждое приложение может прослушивать любой порт, не опасаясь конфликтов.

Эти улучшения особенно важны для ПО, не предназначенного для выполнения в конкретном вычислительном стеке. Многие популярные программы с открытым исходным кодом имеют параметры настройки портов, но эти способы настройки не согласованы.

Контейнеры и неявные зависимости

К абстракции контейнера, как и к любой абстракции, применяется закон Хайрама о неявных зависимостях. К контейнерам этот закон применим *даже в большей сте-*

пени из-за огромного количества пользователей (в Google весь продакшен и многое другое выполняется под управлением Borg), а также из-за того, что пользователи не чувствуют, что обращаются к API, используя, например, файловую систему (и еще менее вероятно, что они задумываются над тем, является ли этот API стабильным, версионированным и т. д.).

Для иллюстрации вернемся к примеру исчерпания пространства идентификаторов процессов, с которым система Borg столкнулась в 2011 году. Возможно, вам интересно, как может исчерпаться пространство идентификаторов процессов. Разве это не просто целые числа из 32- или 64-битного пространства? Не совсем так. В Linux идентификаторы назначаются из диапазона $[0, \dots, \text{PID_MAX} - 1]$, где константа `PID_MAX` по умолчанию равна 32 000. Однако `PID_MAX` можно увеличить простым изменением конфигурации (и значительно). Задача решена?

Увы, нет. Согласно закону Хайрама, тот факт, что идентификаторы процессов, выполняющихся в Borg, ограничены диапазоном $0 \dots 32\,000$, стал неявной гарантией API, от которой люди начали зависеть. Например, процессы хранения журналов зависели от того факта, что PID уместается в пять символов, и их работа нарушалась с появлением шестизначных PID, потому что имена записей превышали максимально допустимую длину. Решение проблемы вылилось в длительный двухэтапный проект. Во-первых, была увеличена верхняя граница идентификаторов PID, которые может использовать один контейнер (чтобы одно задание, испытывающее утечки потоков выполнения, не смогло сделать всю машину непригодной для использования). Во-вторых, пространство PID было разделено на два подпространства — для потоков выполнения и процессов. (Потому что, как оказалось, очень немногие пользователи полагались на ограничение 32 000 в отношении идентификаторов PID, присваиваемых потокам. Благодаря этому мы смогли увеличить верхнюю границу идентификаторов для потоков и оставить ее на уровне 32 000 для процессов.) На третьем этапе предполагалось внедрить в Borg пространство имен PID, чтобы дать каждому контейнеру свое собственное полное пространство PID. Но, как и следовало ожидать (опять же согласно закону Хайрама), многочисленные системы стали исходить из предположения, что тройка $\{\text{имя_хоста}, \text{время}, \text{pid}\}$ однозначно идентифицирует процесс, которое станет недействительным после введения пространства имен PID. Попытки выявить все эти места и исправить их (а также исправить любые соответствующие данные) все еще продолжают восемь лет спустя.

Дело здесь не в том, что контейнеры должны запускаться в пространствах имен PID. Это хорошая идея, но не самая интересная. Когда создавались контейнеры Borg, пространств имен PID не существовало. И даже если бы это было не так, едва ли инженеры, разработывавшие Borg в 2003 году, могли осознавать ценность их внедрения. Даже сейчас на машинах есть ресурсы, которые недостаточно изолированы, что, вероятно, когда-нибудь вызовет проблемы. Это особенно наглядно подчеркивает сложность проектирования системы контейнеров, которая со временем будет оставаться простой в обслуживании и ценной для более широкого сообщества, где эти типы проблем уже возникали и были учтены.

Одна услуга для управления всем

Как обсуждалось выше, раннее решение WorkQueue было нацелено только на некоторые пакетные задания, которые совместно использовали набор машин, управляемых этим решением, а для обслуживающих заданий использовалась другая архитектура, в которой каждое конкретное обслуживающее задание выполнялось отдельно, в выделенном наборе машин. Эквивалентное решение с открытым исходным кодом можно представить как запуск отдельного кластера Kubernetes для каждого типа рабочей нагрузки (+ один пул для всех пакетных заданий).

В 2003 году был запущен проект Borg с целью (успешно достигнутой) создать вычислительную услугу, которая объединяет эти разрозненные наборы машин в один большой набор. Набор Borg охватывал и обслуживающие, и пакетные задания и стал единственным набором в любом центре обработки данных (представьте один большой кластер Kubernetes для всех рабочих нагрузок в каждом географическом местоположении). Здесь стоит отметить два значительных достижения с точки зрения эффективности.

Во-первых, машины для обслуживающих заданий превратились в скот (как отмечено в проектом документе Borg: «Машины анонимны: программам не важно, на какой машине они выполняются, если она имеет требуемые характеристики»). Если бы каждой команде, сопровождающей обслуживающее задание, приходилось управлять своим собственным набором машин (своим кластером), то каждая несла бы одни и те же организационные издержки по обслуживанию и администрированию этого набора. Со временем методы управления этими наборами неизбежно стали бы расходиться, делая изменения в масштабах компании (например, переход на новую серверную архитектуру или переключение центров обработки данных) все более сложными. Унифицированная инфраструктура управления — то есть *общая* вычислительная услуга для всех рабочих нагрузок в организации — позволяет Google избежать линейного роста издержек. Нет различий в методах управления физическими машинами в парке, есть только Borg¹.

Второе достижение, более тонкое, вероятно, применимое не ко всем организациям, но очень важное для Google. Различные потребности пакетной и обслуживающей обработки оказываются взаимодополняющими. Обслуживающие задания обычно требуют выделения избыточных ресурсов, потому что должны обладать достаточной емкостью для обслуживания пользовательского трафика без значительного уменьшения задержки даже в часы пик или в случае частичного сбоя инфраструктуры. Это означает, что машина, на которой выполняются только обслуживающие задания, будет недостаточно загружена. Заманчиво попытаться воспользоваться этими избыточными ресурсами, увеличив нагрузку на машину, но это сведет на нет цель

¹ Как и в любой сложной системе, здесь есть свои исключения. Не все машины в Google управляются системой Borg, и не каждый центр обработки данных охватывается одной ячейкой Borg. Но большинство инженеров работают в среде, в которой все машины управляются системой Borg.

избыточности, потому что, если произойдет всплеск или отключение, необходимые ресурсы окажутся недоступными.

Однако это рассуждение относится только к обслуживающим заданиям! Если на машине действует несколько обслуживающих заданий, которые в сумме потребляют полный объем ОЗУ и вычислительной мощности, то на эту машину больше нельзя помещать никаких обслуживающих заданий, даже если реальное потребление ресурсов остается на уровне 30 % от общей емкости. Но мы можем (и в Borg это будет реализовано) распределить неиспользуемые 70 % между пакетными заданиями с условием, что если какое-то из обслуживающих заданий потребует выделить дополнительный объем ОЗУ или процессорное время, мы освободим ресурсы, занятые пакетными заданиями (приостановив их, если потребуется процессорное время, или прервав, если потребуется ОЗУ). Для пакетных заданий важна пропускная способность (измеряемая по совокупности сотен рабочих реплик, а не отдельных задач), а отдельные реплики — это скот, и они охотно используют свободную мощность для обслуживающих заданий.

В зависимости от профиля рабочих нагрузок в заданном наборе машин либо вся пакетная рабочая нагрузка эффективно работает на свободных ресурсах (потому что мы все равно платим за них из-за нехватки обслуживающих заданий), либо вся обслуживающая рабочая нагрузка платит только за то, что ею используется, но не за резервную мощность, необходимую для обеспечения отказоустойчивости (потому что в этом резерве выполняются пакетные задания). Как показывает опыт Google, в большинстве случаев мы получаем пакетную обработку бесплатно.

Многоарендность для обслуживающих заданий

Выше мы обсудили ряд требований, которым должна удовлетворять вычислительная услуга, чтобы быть пригодной для выполнения обслуживающих заданий. Как уже отмечалось, управление обслуживающими заданиями с помощью общего вычислительного решения дает множество преимуществ, но также сопряжено со сложностями, например с требованием наличия механизма обнаружения служб (раздел «Подключение к услуге»). Существуют также другие требования, которые начинают действовать при расширении управляемых вычислений до обслуживающих заданий, например:

- Перепланирование заданий должно регулироваться: теоретически возможно уничтожить и вновь запустить 50 % реплик пакетного задания (потому что это вызовет только временный сбой в обработке, тогда как нас больше волнует пропускная способность), но вряд ли это будет приемлемо в отношении обслуживающих заданий (потому что оставшихся заданий, вероятно, будет слишком мало, чтобы обслуживать пользовательский трафик, пока вновь запускаемые задания не вернутся к работе).
- Пакетное задание обычно можно прервать без предупреждения. При этом мы потеряем часть выполненной работы, которую придется повторить. Но прерывая без предупреждения обслуживающее задание, мы рискуем вернуть ошибку в от-

вет на запрос пользователя или (в лучшем случае) выполнить его с задержкой. Желательно заранее предупредить задание и дать ему несколько секунд, чтобы оно могло завершить обработку уже полученных запросов и не принимать новые.

Borg охватывает пакетные и обслуживающие задания, но большинство предложений по организации вычислений поддерживают два понятия — общий набор машин для пакетных заданий и выделенные наборы машин для обслуживающих заданий. И независимо от того, используется ли для обоих типов заданий одна и та же архитектура, обе группы выигрывают от обращения с ними как со скотом.

Отправляемая конфигурация

Планировщик Borg получает конфигурацию для службы или пакетного задания, запускаемого в ячейке, как результат RPC. Оператор службы может управлять ею с помощью интерфейса командной строки, выполняя RPC вручную. Параметры для этих вызовов обычно хранятся в общей документации или в их голове.

Зависимость от документации и личных знаний в противоположность коду, хранящемуся в репозитории, редко бывает хорошей идеей еще и потому, что и документация, и личные знания имеют свойство ухудшаться со временем (глава 3). Однако и следующий естественный шаг в эволюции — оформление необходимых команд в виде сценария — тоже уступает использованию специального языка описания конфигураций для определения настроек службы.

Со временем присутствие службы обычно перестает ограничиваться рамками одного набора реплицированных контейнеров в одном центре обработки данных и разрастается сразу по нескольким направлениям:

- Служба будет распространяться по нескольким центрам обработки данных (как для приближения к пользователям, так и для повышения устойчивости к сбоям).
- Служба будет развертываться не только в продакшене, но и в тестовой среде, и в среде разработки.
- Служба будет накапливать дополнительные реплицированные контейнеры разных типов в виде прикрепленных вспомогательных служб, таких как `metacached`.

Управление службой значительно упростится, если появится возможность выразить сложную настройку на стандартном языке описания конфигураций, позволяющем свободно выражать стандартные операции (например, «обновить службу до новой версии, затрачивая на это не более 5 % ресурсов в любой конкретный момент»).

Стандартный язык описания конфигураций позволит другим командам включить стандартную конфигурацию в определения своих служб. Как обычно, во времени и в масштабе стандартная конфигурация приобретает особую ценность. Если каждая команда напишет свой код для поддержки службы `metacached`, то будет очень сложно в масштабах всей организации выполнять такие задачи, как переход на новую реализацию `metacache` (например, по причинам производительности или лицензирования) или установка обновлений безопасности во всем развертывании

memcache. Также обратите внимание, что наличие стандартного языка описания конфигураций является требованием для автоматизации развертывания (глава 24).

Выбор вычислительной услуги

Маловероятно, что какая-то организация пойдет по пути, по которому пошла Google, создавая свою вычислительную архитектуру с нуля. В наши дни доступно множество современных предложений как с открытым исходным кодом (например, Kubernetes и Mesos, или, на другом уровне абстракции, OpenWhisk и Knative), так и в виде общедоступных управляемых облачных решений (разной степени сложности, от Managed Instance Groups в Google Cloud Platform или автоматически масштабируемого Amazon Web Services Elastic Compute Cloud (Amazon EC2) до управляемых контейнеров, подобных Borg, таких как Microsoft Azure Kubernetes Service (AKS) или Google Kubernetes Engine (GKE), и бессерверных предложений, таких как AWS Lambda или Google Cloud Functions).

Однако большинство организаций *предпочтут* организовать вычислительную услугу, как и Google, внутри компании. Обратите внимание, что вычислительная инфраструктура имеет сильное привязывающее действие, поскольку код будет писаться так, чтобы использовать все свойства системы (закон Хайрама). Например, в случае выбора решения на основе виртуальных машин команды будут настраивать свои образы виртуальных машин, а в случае выбора решения на основе контейнеров команды будут обращаться к API диспетчера кластера. Если архитектура позволит коду обращаться с виртуальными машинами (или контейнерами) как с домашними любимцами, команды будут делать это, и тогда будет сложно перейти к решению, в котором с этими машинами и контейнерами обращаются как со скотом (или даже с обоими видами домашних животных).

Чтобы показать, как даже мельчайшие детали вычислительного решения могут оказать связывающее действие, рассмотрим, как Borg выполняет пользовательские команды, указанные в конфигурации. В большинстве случаев команды запускают двоичный файл (возможно, с дополнительными аргументами). Однако для удобства авторы Borg включили возможность передачи сценария командной оболочки, например `while true; do ./my_binary; done`¹. Однако, в отличие от двоичного файла, который можно запустить вызовом пары функций `fork` и `exec` (что и делает Borg), сценарий должен запускаться командной оболочкой, такой как Bash. То есть фактически Borg выполняет команду `/usr/bin/bash -c $USER_COMMAND`, которая также позволяет запускать простые двоичные файлы.

В какой-то момент разработчики Borg осознали, что в масштабе Google оболочка Bash потребляет значительные ресурсы (в основном память), и решили перейти

¹ Эта конкретная команда представляет особую опасность для Borg, потому что препятствует работе механизмов обработки ошибок в Borg. Однако для отладки проблем все еще используются сложные обертки, например записывающие настройки среды в журнал.

к использованию более легкой оболочки `ash`. Они внесли изменения в код запуска процессов, чтобы выполнялась команда `/usr/bin/ash -c $USER_COMMAND`.

Можно подумать, что это изменение не несет никаких рисков: мы контролируем среду, знаем, что оба файла существуют и это изменение должно сработать. Но оно не сработало, потому что инженеры Borg были не первыми, кто заметил повышенный расход памяти при запуске `Bash`. Некоторые команды проявили изобретательность в своем желании ограничить использование памяти и заменили (в своих образах файловых систем) команду `Bash` специально написанным кодом «запустить второй аргумент». Они хорошо знали об использовании памяти, и поэтому, когда команда Borg изменила способ запуска процессов, задействовав для этого оболочку `ash` (которая не подменялась нестандартным кодом), использование памяти увеличилось (потому что вместо экономного нестандартного кода выполнялся интерпретатор `ash`). Это вызвало предупреждения, откат изменения и недовольство.

Еще одна причина, по которой выбор вычислительной услуги трудно изменить со временем, заключается в том, что любой выбор в итоге будет окружен большой экосистемой вспомогательных служб — инструментов журналирования, мониторинга, отладки, оповещения, визуализации, оперативного анализа, языков описания конфигураций и метаязыков, пользовательских интерфейсов и многого другого. Эти инструменты придется переписать при переходе на другую вычислительную услугу, и даже простое перечисление этих инструментов, вероятно, станет проблемой для средней или большой организации.

Таким образом, выбор вычислительной архитектуры играет важную роль. Как и большинство решений в программной инженерии, это решение предполагает компромиссы. Обсудим некоторые из них.

Централизация против индивидуализации

С точки зрения накладных расходов на управление вычислительным стеком (а также эффективности использования ресурсов) лучшее, что может сделать организация, — выбрать единое решение `SaaS` для управления всем парком машин и использовать только доступные инструменты. В этом случае по мере роста организации затраты на управление парком будут оставаться в разумных пределах. Это практически тот же путь, который прошла компания Google с Borg.

Необходимость индивидуализации

Однако по мере роста в организации будут появляться все более разнообразные потребности. Например, в 2012 году Google запустила `Google Compute Engine` (общедоступное облачное предложение «виртуальная машина как услуга»). В Google каждая виртуальная машина работала в отдельном контейнере под управлением Borg. Однако «скотоводческий» подход к управлению задачами был малопригоден для рабочих нагрузок `Cloud`, потому что каждый конкретный контейнер на самом деле был вир-

туальной машиной, которую арендовал конкретный пользователь, а пользователи Cloud редко относились к виртуальным машинам как к скоту¹.

Устранение этих различий потребовало значительных усилий с обеих сторон. Организация Cloud позаботилась о поддержке бесшовной миграции между виртуальными машинами: для виртуальной машины, выполняющейся на одном сервере, создается копия на другом сервере, которая доводится до идеального состояния, и затем в нее перенаправляется весь трафик без заметного перерыва в обслуживании². В Borg, с другой стороны, потребовалось внести изменения, чтобы избежать случайного уничтожения контейнеров, содержащих виртуальные машины (и дать время для копирования данных из старой виртуальной машины в новую), а также, с учетом ресурсоемкости процесса миграции, алгоритмы планирования Borg были оптимизированы для уменьшения вероятности перепланирования³. Конечно, эти изменения были развернуты только для машин, на которых выполняются облачные рабочие нагрузки, что привело к раздвоению (небольшому, но все же заметному) предложения Google для внутренних вычислений.

Другой пример — который тоже привел к раздвоению — Google Search. Примерно в 2011 году один из контейнеров, обслуживающих веб-трафик Google Search, имел гигантский индекс на локальных дисках, в котором хранилась редко используемая часть веб-индекса Google (более распространенные запросы обслуживались кешами в памяти в других контейнерах). Для создания этого индекса на конкретном компьютере требовалось несколько жестких дисков и несколько часов времени. Однако в то время Borg предполагала, что если какой-то из дисков, хранящий данные в конкретном контейнере, вышел из строя, то контейнер не может продолжать работу и его необходимо перенести на другую машину. Эта комбинация (наряду с относительно высокой частотой отказов вращающихся дисков по сравнению с другим оборудованием) вызвала серьезные проблемы с доступностью: контейнеры постоянно сталкивались с проблемами, и требовалась целая вечность, чтобы снова запустить их. Для исправления этого недостатка в Borg пришлось добавить возможность, позволяющую контейнерам самостоятельно обрабатывать отказы дис-

¹ Задание визуализации графики не является взаимозаменяемым с моим почтовым сервером, даже если обе задачи выполняются в одной и той же форме виртуальной машины.

² Это не единственный повод для внедрения бесшовной миграции пользовательских виртуальных машин. Она также предлагает значительные преимущества для пользователя, позволяя обновлять ОС или аппаратное обеспечение хоста без нарушения работы виртуальной машины. Альтернативой (которая используется другими крупными поставщиками облачных услуг) является доставка «уведомлений о событиях обслуживания», например о том, что виртуальная машина может быть перезагружена или остановлена, а затем вновь запущена поставщиком облачных услуг.

³ Это особенно актуально, учитывая, что не все виртуальные машины клиентов включены в бесшовную миграцию. Для некоторых рабочих нагрузок даже кратковременное снижение производительности во время миграции недопустимо. Эти клиенты будут получать уведомления о техническом обслуживании, а Borg со своей стороны постарается избежать вытеснения контейнеров с этими виртуальными машинами, разве что в случае крайней необходимости.

ков, в обход стандартной процедуры в Borg, то есть команде разработчиков Search пришлось адаптировать процесс, чтобы иметь возможность продолжать работу с частичной потерей данных.

Множество других раздвоений, охватывающих такие области, как форма файловой системы, доступ к ней, управление памятью, локальность ЦП и памяти, специальное оборудование, особые ограничения планирования и т. д., привели к тому, что Borg API стал большим и громоздким, а результаты пересечения разных вариантов поведения стало труднее предсказывать и еще труднее проверять. Никто не знал, что произойдет, если контейнер *одновременно* запросит у Cloud замену *и* специальную обработку отказа диска в Search.

После 2012 года команда Borg посвятила значительное время чистке Borg API, в ходе которой выяснилось, что некоторые функции Borg вообще вышли из употребления¹. Еще одну важную группу образовали функции, которые используются несколькими контейнерами, но было неясно, действительно ли их применение было осознанным — процесс копирования файлов конфигурации между проектами привел к распространению функций, которые изначально предназначались только для опытных пользователей. Чтобы ограничить распространение таких функций и четко обозначить их предназначение только для опытных пользователей, был введен белый список. Однако чистка все еще продолжается, и некоторые изменения (например использование меток для идентификации групп контейнеров) еще не выполнены².

Но, как это обычно бывает, даже если есть возможность приложить усилия для индивидуализации и получить некоторые преимущества, не добавляя недостатков (таких, как вышеупомянутый белый список для функциональных возможностей), рано или поздно придется сделать трудный выбор: нужно ли расширить явно (или, что еще хуже, неявно) поверхность API, чтобы приспособить ее для конкретного пользователя нашей инфраструктуры, или лучше доставить ему значительные неудобства, но сохранить единообразие?

Уровень абстракции: бессерверные вычисления

Описание, как Google приручает вычислительную среду, можно воспринимать как рассказ о совершенствовании абстракции — более продвинутые версии Borg взяли на себя больше управленческих функций и еще больше изолировали контейнеры от базовой среды. Может показаться, что здесь все просто: больше абстракции — хорошо, а меньше абстракции — плохо.

¹ Хорошее напоминание о том, что мониторинг и контроль за использованием функций со временем приобретут особую ценность.

² Фреймворк Kubernetes, извлекая выгоду из опыта чистки Borg, но не испытывавший ограничений из-за обширной базы пользователей, с самого начала был более современным (например, в отношении меток). Однако, получив более широкое распространение, Kubernetes начал испытывать похожие проблемы.

Конечно, не все так просто. Пространство со множеством предложений имеет очень сложный ландшафт. Выше, в разделе «Приручение вычислительной среды», мы обсудили переход от домашних любимцев, выполняющихся на физических машинах (принадлежащих организации или арендованных в вычислительном центре), к управлению контейнерами как скотом. Между ними находится целый спектр предложений на основе виртуальных машин, которые могут играть самые разные роли, от гибкого средства замены «голого железа» (в предложениях «инфраструктура как услуга», таких как Google Compute Engine (GCE) или Amazon EC2) до тяжеловесных заменителей контейнеров (с автоматическим масштабированием и другими инструментами управления).

По опыту Google, решение проблемы масштабного управления заключается в использовании подхода к управлению машинами как скотом. Повторю: если каждой из ваших команд потребуется всего одна машина — домашний любимец — в каждом из ваших центров обработки данных, то затраты на управление ими будут увеличиваться сверхлинейно с ростом организации (потому что количество команд и количество центров обработки данных тоже будут увеличиваться). А после перехода к управлению машинами как скотом контейнеры станут естественным выбором, поскольку они легче (в том смысле, что требуют меньше ресурсов и быстрее запускаются) и могут настраиваться в широких пределах, так что если вам потребуется организовать особый доступ к оборудованию для определенного типа рабочей нагрузки, вы с легкостью сможете (если захотите) сделать это.

Преимущество подхода к виртуальным машинам как к скоту заключается, прежде всего, в способности использовать свою ОС, что имеет значение, если для рабочих нагрузок требуется разнообразный набор операционных систем. Кроме того, некоторые организации уже имеют опыт управления виртуальными машинами, а также знают про конфигурации и рабочие нагрузки на их основе, поэтому могут отдать предпочтение виртуальным машинам вместо контейнеров, чтобы снизить затраты на миграцию.

Что такое бессерверные вычисления?

Еще более высокий уровень абстракции — *бессерверные* вычисления¹. Допустим, что организация обслуживает веб-контент и использует (или хочет использовать) универсальную серверную инфраструктуру для обработки HTTP-запросов и отправки ответов. Ключевой чертой инфраструктуры является поддержка инверсии управления, поэтому пользователь будет отвечать только за реализацию какого-либо «действия» или «обработчика» — функции на выбранном языке, которая принимает параметры запроса и возвращает ответ.

В мире Borg для запуска этого кода создается контейнер, каждая реплика которого содержит сервер, состоящий из кода инфраструктуры и функций. Чтобы справиться

¹ К бессерверным вычислениям относятся, например, технологии FaaS (function as a service — функция как услуга) и PaaS (platform as a service — платформа как услуга). Границы между этими терминами размыты.

с увеличением трафика, можно добавлять новые реплики или расширять центры обработки данных. Если трафик уменьшится, можно уменьшить количество работающих реплик до минимума (мы в Google обычно поддерживаем не меньше трех реплик в каждом центре обработки данных, где работает сервер).

Однако если одну и ту же инфраструктуру использует несколько команд, возможен другой подход: многоарендность можно присвоить не машинам, а серверной инфраструктуре. Такой подход предполагает запуск большого количества экземпляров инфраструктуры, динамическую загрузку и выгрузку кода, определяющего действия и динамическую отправку запросов тем экземплярам, где загружен соответствующий код действия. В такой ситуации командам не нужно запускать свои серверы, а значит, они становятся «бессерверными».

Бессерверные инфраструктуры часто сравнивают с моделью «виртуальные машины как домашние любимцы». Идея бессерверных вычислений — революционная, потому что дает все преимущества управления скотом: автоматическое масштабирование, сокращение накладных расходов, отсутствие необходимости явно настраивать серверы. Однако, как отмечалось выше, переход к совместной многоарендной модели управления машинами как скотом уже должен быть целью организации, планирующей дальнейший рост, поэтому лучше сравнивать бессерверные архитектуры с архитектурой «постоянных контейнеров», такой как Borg, Kubernetes или Mesosphere.

Достоинства и недостатки

Во-первых, обратите внимание, что бессерверная архитектура требует, чтобы код *не имел состояния*; едва ли можно запускать виртуальные машины пользователей или реализовать Spanner в бессерверной архитектуре. Никакие способы управления локальным состоянием (за исключением его неиспользования), о которых мы говорили выше, не применимы в бессерверной среде. В мире контейнеров можно потратить несколько секунд или минут во время запуска, чтобы настроить подключения к другим службам, заполнить кэши из холодного хранилища и т. д. и ожидать, что в типичном случае у вас будет время, чтобы предпринять какие-то действия перед завершением. В бессерверной модели нет локального состояния между запросами; все необходимые данные должны подготавливаться в области видимости запроса.

Как показывает практика, большинство потребностей организаций невозможно удовлетворить с использованием рабочих нагрузок без состояния. Это может привести к зависимости от конкретных решений (собственных или сторонних) конкретных задач (например, решений управляемых баз данных, которые часто сопровождают бессерверные облачные предложения) или к внедрению двух решений: контейнерного и бессерверного. Стоит упомянуть, что многие или большинство фреймворков бессерверных вычислений основаны на других вычислительных решениях: AppEngine — на Borg, Knative — на Kubernetes, Lambda — на Amazon EC2.

Управляемая бессерверная модель привлекательна своей способностью *адаптивного масштабирования* потребления ресурсов, особенно с низким трафиком. Например, контейнер в Kubernetes не может масштабироваться до нуля контейнеров (предполагается, что развертывание контейнера и узла — слишком медленная процедура, чтобы выполнять ее в ходе обслуживания запроса). Это означает, что существуют минимальные затраты, выражающиеся в обязательном наличии приложения в постоянной кластерной модели. Бессерверное приложение, напротив, легко масштабируется до нуля, поэтому стоимость владения масштабируется в соответствии с трафиком.

При очень высоком трафике вступают в силу ограничения базовой инфраструктуры независимо от вычислительного решения. Если вашему приложению необходимо 100 000 ядер для обслуживания трафика, то на любом физическом оборудовании, поддерживающем вашу инфраструктуру, должно быть доступно 100 000 физических ядер. На более низком уровне, когда приложение получает достаточный объем трафика, чтобы поддерживать занятость нескольких серверов, но недостаточный, чтобы создать проблемы для поставщика услуги, оба решения — постоянное контейнерное и бессерверное — могут масштабироваться для его обработки, причем бессерверное решение будет масштабироваться активнее и точнее.

Наконец, выбор бессерверного решения подразумевает определенную потерю контроля над средой. В некотором смысле это хорошо: наличие контроля означает необходимость его осуществлять и нести дополнительные накладные расходы. И конечно, если вам понадобятся какие-то дополнительные функции, отсутствующие в используемой инфраструктуре, то это станет для вас проблемой.

Возьмем один конкретный пример: команда Google Code Jam (проводившая соревнование по программированию с тысячами участников и реализовавшая для этого веб-интерфейс на Google AppEngine) запустила специальный сценарий за несколько минут до начала состязания, генерирующий искусственный трафик к веб-интерфейсу конкурса, чтобы разогреть достаточное количество экземпляров приложения для обслуживания фактического трафика после начала конкурса. Это сработало, но от такой своеобразной ручной подстройки можно было бы избавиться, выбрав бессерверное решение.

Компромисс

Компания Google не стала вкладывать большие средства в бессерверные решения. Ее решение на основе постоянных контейнеров, Borg, достаточно продвинутое и может предложить большинство преимуществ бессерверных решений (например, автоматическое масштабирование, различные инфраструктуры для различных типов приложений, инструменты развертывания, унифицированные инструменты журналирования и мониторинга и т. д.). Единственное, чего не хватает Borg, — более агрессивного масштабирования (в частности, возможности масштабирования до нуля), но подавляющее большинство ресурсов в Google приходится на службы

с высоким трафиком, поэтому избыточность в предоставлении услуг обходится сравнительно дешево. В то же время многие приложения в Google не способны действовать в мире «без состояния», от GCE до собственных систем баз данных, таких как BigQuery (<https://cloud.google.com/bigquery>) или Spanner, и серверов, которым требуется много времени для заполнения кеша, как вышеупомянутые задания по обслуживанию поиска с длинным хвостом. То есть преимущества единой унифицированной архитектуры для всех этих приложений перевешивают потенциальную выгоду от наличия отдельного бессерверного стека для части рабочих нагрузок.

Мы не утверждаем, что выбор Google является единственно правильным для всех организаций: многие другие организации успешно используют смешанные (контейнерные и бессерверные) или чисто бессерверные архитектуры и сторонние решения для хранения данных.

Основные выгоды бессерверные системы несут не крупным, а небольшим организациям или командам. Бессерверная модель, хотя и является более ограничительной, позволяет поставщику услуги взять на себя большую часть общих издержек по управлению и *уменьшить эти издержки* для пользователей. Запустить код, принадлежащий одной команде, в общедоступной бессерверной архитектуре, такой как AWS Lambda или Google Cloud Run, значительно проще (и дешевле), чем настраивать кластер для запуска кода в службе контейнеров, такой как GKE или AKS, если только кластер не используется совместно многими командами. Если у вашей команды появится желание воспользоваться преимуществами управляемых вычислений, но ваша организация не захочет или не сможет перейти на решение с постоянными контейнерами, то бессерверные предложения поставщиков облачных услуг, вероятно, покажутся вам привлекательными, потому что стоимость (ресурсов и управления) общего кластера хорошо компенсируется, только если кластер действительно используется несколькими командами в организации.

Однако обратите внимание, что с ростом организации и распространения технологий управляемых вычислений вы, вероятно, перерастаете ограничения чисто бессерверных решений. Это сделает более привлекательными решения, в которых существует прорывной путь (например, от KNative к Kubernetes), потому что они обеспечивают естественный переход к унифицированной вычислительной архитектуре, такой как в Google. Используйте их, если ваша организация решит пойти по нашему пути.

Общедоступный или приватный

Когда компания Google только начинала свою деятельность, предложения SaaS были в основном доморощенными: если вам нужно было такое решение, то вы создавали его. Единственный выбор между общедоступным и приватным пространством заключался в выборе между приобретением своих машин и их арендой, но все управление вашим парком полностью возлагалось на вас.

В эпоху общедоступных облачных услуг появились более дешевые варианты. Но есть и другие аспекты, которые организациям придется учитывать, делая свой выбор.

Организация, использующая общедоступное облако, фактически перекладывает (часть) издержек по управлению на поставщика облачных услуг. Для многих организаций это привлекательное предложение — они могут сосредоточиться на создании ценностей в своей области знаний и не накапливать опыт в области организации инфраструктуры. Конечно, поставщики облачных услуг берут больше, чем стоит содержание чистого железа, чтобы возместить управленческие расходы, но у них уже есть накопленный опыт, и они делятся им со своими клиентами.

Кроме того, общедоступное облако упрощает масштабирование инфраструктуры. По мере увеличения уровня абстракции — от покупки времени виртуальных машин до управляемых контейнеров и бессерверных предложений — растет простота масштабирования — от необходимости подписывать договор аренды до запуска простой команды, чтобы получить еще несколько виртуальных машин, и предоставления инструментов автоматического масштабирования ресурсов с изменением трафика. Прогнозирование потребления ресурсов — сложная задача, особенно для молодых организаций или продуктов, поэтому отсутствие необходимости заранее выделять ресурсы дает значительные преимущества.

Одна из существенных проблем при выборе поставщика облачных услуг — боязнь оказаться в затруднительном положении: поставщик может внезапно повысить цены или просто уйти с рынка, поставив организацию в очень трудное положение. Один из первых поставщиков бессерверных вычислений, Zimki, предлагавший «платформу как услугу» для запуска JavaScript, закрылся в 2007 году, уведомив об этом клиентов за три месяца.

Эту проблему можно частично смягчить, выбрав общедоступное облачное решение на архитектуре с открытым исходным кодом (например, Kubernetes). Такой подход оставляет открытыми пути для миграции на тот случай, если услуги конкретного поставщика по какой-то причине окажутся неприемлемыми. Эта стратегия поможет значительно снизить риски, но ее нельзя назвать идеальной. Из-за закона Хайрама трудно гарантировать, что возможности, поддерживаемые только этим конкретным поставщиком, не будут использоваться иначе.

Эта стратегия имеет два возможных расширения. Одно из них — использовать общедоступное облачное решение нижнего уровня (например, Amazon EC2) и запустить на его основе более высокоуровневое решение с открытым исходным кодом (например, Open-Whisk или KNative). Это поможет гарантировать возможность повторного использования любых настроек для высокоуровневого решения, инструментов, созданных на его основе, и сопутствующих неявных зависимостей, если вдруг вы решите мигрировать. Другое расширение — использовать мультиоблако, то есть управляемые службы, основанные на одних и тех же решениях с открытым исходным кодом, двух или более поставщиков облачных услуг (скажем, GKE и AKS для Kubernetes). Это еще больше упростит миграцию между ними, а также поможет

избежать попадания в зависимость от конкретных деталей реализации, доступных у одного из поставщиков.

Еще одна стратегия — не столько ослабляющая зависимость, сколько упрощающая миграцию — использовать гибридное облако; то есть часть рабочей нагрузки разместить в приватной инфраструктуре, а часть — в общедоступном облаке. При таком подходе общедоступное облако можно использовать, например, для борьбы с переполнением. Типичная рабочая нагрузка организации может обрабатываться в приватном облаке, а в случае нехватки ресурсов некоторые рабочие нагрузки можно масштабировать в общедоступное облако. И снова, чтобы это решение было эффективным, в обеих областях следует использовать одно и то же решение с открытым исходным кодом для организации вычислительной инфраструктуры.

Обе политики — с использованием нескольких облаков или гибридного облака — требуют организации надежной связи между несколькими средами посредством прямых сетевых соединений между машинами и общих API в обеих средах.

Заключение

В ходе создания, совершенствования и эксплуатации своей вычислительной инфраструктуры мы в Google изучили детали проектирования таких структур. Наличие единой инфраструктуры для всей организации (например, одного или нескольких общих кластеров Kubernetes в каждом регионе) дает значительный выигрыш в эффективности управления и снижении затрат и позволяет разрабатывать общие инструменты поверх этой инфраструктуры. Ключевым инструментом в создании такой архитектуры являются контейнеры. Они позволяют разным задачам совместно использовать физическую (или виртуальную) машину, что способствует более эффективному расходованию ресурсов, а также обеспечивают уровень абстракции между приложением и ОС, дающий дополнительную устойчивость.

Для правильного использования контейнерной архитектуры приложение должно состоять из узлов, которые можно автоматически заменять (как скот), чтобы позволить ему масштабироваться до тысяч экземпляров. Разработка ПО с применением этой модели требует определенного отношения к ресурсам, например все локальные хранилища (включая диски) нужно воспринимать как эфемерные и ни в коем случае не использовать жестко определяемые имена хостов.

Мы в Google в целом удовлетворены выбором архитектуры, но многим организациям еще предстоит выбирать из широкого спектра вычислительных услуг — от модели «домашних любимцев» виртуальных или физических машин до модели «рабочего скота» реплицируемых контейнеров и абстрактной «бессерверной» модели, доступных в вариантах с управляемым и открытым исходным кодом. Ваш выбор будет основан на компромиссе, в котором будут учтены разные факторы.

Итоги

- Для масштабирования рабочих нагрузок необходима общая инфраструктура в продакшене.
- Вычислительное решение может предоставить стандартную и стабильную абстракцию и среду для ПО.
- ПО необходимо адаптировать для выполнения в распределенной управляемой вычислительной среде.
- Организация должна тщательно выбирать вычислительное решение, чтобы обеспечить необходимые уровни абстракции.

ЧАСТЬ V

Заключение

Послесловие

Программная инженерия в Google стала выдающимся экспериментом в области разработки и поддержки большой и постоянно развивающейся кодовой базы. За время своего пребывания в Google я видел, как команды инженеров делали первые шаги в этом направлении, продвигая Google вперед как компанию, обслуживающую миллиарды пользователей, и как лидера в индустрии высоких технологий. Едва ли это развитие было возможно без использования принципов, изложенных в данной книге, поэтому я очень рад, что она увидела свет.

Если что и доказали последние 50 лет (или предыдущие страницы этой книги), так это то, что программная инженерия далека от застоя. В среде, где технологии постоянно меняются, программная инженерия необходима любой мягкой организации. Современные принципы программной инженерии сводятся не только к эффективному управлению организацией — они определяют ответственность компании перед пользователями и миром в целом.

Решения типичных задач программной инженерии не всегда лежат на поверхности — нередко бывает нужно проявить недюжинную гибкость, чтобы определить решения, подходящие для текущих задач и учитывающие неизбежные изменения в технических системах. Такая гибкость — общее качество команд программной инженерии, с которыми я имел честь работать и учиться с момента моего прихода в Google в 2008 году.

Идея устойчивости тоже занимает одно из центральных мест в программной инженерии. В течение ожидаемого срока службы кодовой базы мы должны иметь возможность реагировать на изменения и адаптироваться к ним, будь то изменение направления развития продукта, изменения в технологических платформах, базовых библиотеках, операционных системах или в чем-то еще. В настоящее время мы полагаемся на принципы, изложенные в этой книге, чтобы достичь гибкости в изменении элементов нашей программной экосистемы.

Конечно, мы не можем доказать, что найденные нами пути достижения устойчивости подойдут для каждой организации, но я думаю, что было важно поделиться этими ключевыми уроками. Программная инженерия — это новая дисциплина, поэтому очень немногие организации смогли добиться в ней устойчивости и достичь большого масштаба. Предлагая этот обзор нашего пути и препятствий, встретившихся на нем, мы надеемся продемонстрировать ценность и осуществимость долгосрочного планирования работоспособности кода. Нельзя игнорировать ход времени и важность перемен.

В этой книге представлены некоторые из наших основных принципов, касающихся программной инженерии. Книга проливает свет на влияние технологий на обще-

ство. Как инженеры-программисты, мы несем ответственность за разработку кода с учетом инклюзивности, равенства и доступности для всех. Разработка только ради инноваций больше не приемлема, и технология, помогающая лишь узкому кругу пользователей, не может считаться инновационной.

Мы в Google всегда считали своей обязанностью показать разработчикам, как своим, так и сторонним, хорошо освещенный путь. С появлением новых технологий, таких как искусственный интеллект, квантовые вычисления и окружающие вычисления (ambient computing), нам, как компании, еще есть чему поучиться. Мне особенно интересно узнать, в каком направлении будет развиваться индустрия программной инженерии в ближайшие годы, и я уверен, что эта книга поможет сформировать этот путь.

*Асим Хусейн (Asim Husain),
вице-президент Google по разработке*

Об авторах

Титус Вингерс (Titus Winters) — главный инженер по ПО в Google. Гуглер с 2010 года. В настоящее время занимает пост председателя глобального подкомитета по развитию стандартной библиотеки C++. В Google возглавляет библиотеку кода на C++: 250 миллионов строк кода, которые ежемесячно правятся 12 000 инженеров. В течение последних семи лет Титус и его команда занимались организацией, обслуживанием и развитием основных компонентов кодовой базы на C++ в Google с использованием современных средств автоматизации и инструментов. Попутно он запустил несколько проектов, которые, как считается, входят в десятку крупнейших попыток реорганизации кода в истории человечества. Оказывая помощь в создании инструментов для рефакторинга и автоматизации, Титус на собственном опыте столкнулся с огромным количеством трюков и уловок, которые инженеры и программисты могут использовать, чтобы «просто заставить что-то работать». Этот уникальный опыт существенно изменил его взгляды на поддержку программных систем.

Том Маншрек (Tom Manshreck) — штатный технический писатель Google. Гуглер с 2005 года. Отвечает за разработку и сопровождение многих руководств по программированию в Google. С 2011 года является членом команды Google C++ Library Team, занимается разработкой набора документации Google по C++, участвовал в создании (вместе с Титусом Винтерсом) учебных курсов в Google по C++ и занимался документированием Abseil, открытого кода на C++ от Google. Имеет степени бакалавра политических наук и бакалавра истории, полученные в MIT. До Google работал управляющим редактором в Pearson/Prentice Hall и в различных стартапах.

Хайрам Райт (Hyrum Wright) — штатный инженер-программист Google. Гуглер с 2012 года, работающий в основном в области крупномасштабной поддержки кодовой базы на C++. Хайрам внес больше правок в кодовую базу Google, чем любой другой инженер за всю историю компании, и сегодня он возглавляет группу разработки инструментов поддержки автоматизированных изменений. Получил докторскую степень в области разработки ПО в Техасском университете в Остине, степень магистра в Техасском университете и степень бакалавра в Университете Бригама Янга. Время от времени читает лекции в Университете Карнеги—Меллона. Активно выступает на конференциях и пишет научные статьи по поддержке и развитию ПО.

Об обложке

На обложке «Программная инженерия в Google» изображен американский фламинго (*phoenicopterus ruber*). Эти птицы обитают на илистых отмелях и прибрежных лагунах с соленой водой вдоль побережий Центральной и Южной Америки и на берегах Мексиканского залива, хотя иногда добираются до Южной Флориды в США.

Знаменитое розовое оперение фламинго формируется по мере взросления птицы, и этот цвет обусловлен наличием каротиноидных пигментов в пище. Поскольку эти пигменты более распространены в естественных источниках, дикие фламинго, как правило, имеют более яркое оперение, чем их собратья в неволе, хотя иногда зоопарки добавляют в их рацион дополнительные пигменты. Фламинго обычно имеют рост около метра, а размах крыльев с черными кончиками составляет около полутора метров. У фламинго, болотной птицы, перепончатые трехпалые розовые лапы. Во внешнем виде самцов и самок фламинго нет заметных различий, но обычно самцы немного крупнее.

Фламинго питаются фильтратом и используют свои длинные ноги и шею для питания в глубокой воде. Большую часть дня они проводят в поисках пищи. В клюве имеются два ряда роговых пластинок с зубчиками, через которые они фильтруют свой рацион из семян, водорослей, микроорганизмов и мелких креветок. Фламинго живут большими группами до 10 000 особей и мигрируют после истощения кормовой базы. Фламинго не только стайные птицы, но и очень общительные. В их языке есть сигналы, помогающие им найти конкретных сородичей, и тревожные сигналы, чтобы предупредить о чем-то большую группу.

Когда-то американского фламинго относили к тому же виду, что и большого фламинго (*phoenicopterus roseus*), обитающего в Африке, Азии и Южной Европе, но сегодня он считается отдельным видом. В настоящее время американскому фламинго присвоен природоохранный статус «вызывающий наименьшие опасения», тем не менее многие животные, изображенные на обложках книг издательства O'Reilly, находятся под угрозой исчезновения. Они все важны для нашего мира.

Иллюстрацию для обложки нарисовала Карен Монтгомери (Karen Montgomery) на основе черно-белой гравюры из книги Касселя (Cassell) «Natural History».

Титус Винтерс, Том Маншрек, Хайрам Райт

**Делай как в Google.
Разработка программного обеспечения**

Перевел с английского А. Киселев

Заведующая редакцией	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>К. Тульцева</i>
Литературный редактор	<i>А. Руденко</i>
Обложка	<i>В. Мостипан</i>
Корректоры	<i>М. Одинокова, Г. Шкатова, Н. Сидорова</i>
Верстка	<i>Е. Неволайнен</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».

Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,

Б. Сапсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 06.2021. Наименование: книжная продукция.

Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,

58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск,

ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 27.05.21. Формат 70x100/16. Бумага офсетная. Усл. п. л. 43,860. Тираж 700. Заказ