ГЛАВА 2

Разработчик в тестировании

Давайте представим идеальный процесс разработки. Все начинается с теста. Вот код, который построил разработчик Джек. А вот — тест, который разработчик Джек придумал еще до того, как написал код. Другими словами, до того, как написать первую строчку кода, разработчик прикидывает, что ему понадобится для тестирования. Затем он напишет тесты для граничных значений, для слишком больших и слишком малых входных данных, для значений, нарушающих граничные условия, и для множества других предположений. Какие-то из этих тестов станут частью функций, превратятся в самотестируемый код или юнит-тесты. На этом уровне лучше всех тестирует код тот, кто его написал. Иначе говоря, в коде, который построил Джек, хорошо разбирается сам разработчик Джек, и он же протестирует его лучше всех.

Другие тесты требуют знаний, выходящих за рамки кода, и зависят от внешней инфраструктуры. Например, у нас есть тест, который возвращает данные из удаленного хранилища (с сервера баз данных или из облака). Для тестирования нам нужна либо сама база данных, либо ее имитация. В индустрии уже выработались инструменты для этого: *тестовая оснастка* (harness), *тестовая инфраструктура*, *подставные объекты* (mocks) и *имитации* (fakes). В мире идеального процесса разработки эти макеты сразу существуют для любого интерфейса, с которым имеет дело разработчик, и каждый аспект любой функции можно протестировать в любое время. Но не увлекайтесь, мы находимся в воображаемом мире.

Мы подошли к первой развилке, где нашему сказочному миру разработки потребовался еще один герой, то есть тестировщик. При написании кода функциональности и кода тестов важны разные образы мышления. Это разные

типы разработки. При разработке функционального кода на первом плане стоит *создание*. Нужно принимать во внимание пользователей, их сценарии использования продукта, последовательности действий и т. д. А при написании тестового кода есть ориентир на *разрушение*, то есть разработку такого кода, который выявит случаи, мешающие эффективной работе пользователя и его действиям. Так как мы находимся в сказочной стране идеальной разработки, мы можем нанять двух Джеков, один из которых построит дом, а другой покажет, как его можно сломать.

НА ЗАМЕТКУ

При написании кода функциональности и кода тестов важны разные образы мышления.

В нашем сказочном мире идеального процесса разработки у нас будет сколько угодно разработчиков функциональности и разработчиков тестов, которые идут рука об руку и вместе строят сложный программный продукт. Это еще присказка, ведь настоящая сказка позволит нам выделить по целому разработчику на каждую фичу, к каждому разработчику приставить столько разработчиков тестов, сколько нужно. Они занимались бы глобальной тестовой инфраструктурой, помогали бы решить проблемы, найденные юнит-тестированием, чтобы разработчики не отвлекались на них от процесса создания, требующего полной концентрации.

Итак, одни разработчики пишут функциональный код, другие разработчики пишут тестовый код, и тут мы вспоминаем про третью сторону: сторону пользователя.

Естественно, в нашем сказочном мире эта задача упадет на плечи отдельных инженеров. Их задачи связаны с пользователем: сценарии использования продукта, пользовательские истории, исследовательское тестирование и т. д. Разработчики со стороны пользователя рассматривают то, как разные фичи связываются воедино и образуют единое целое. Они работают над проблемами всей системы и обычно принимают точку зрения пользователя, проверяя, действительно ли совокупность частей образует что-то полезное для них.

Итак, вот наше представление об идеальной разработке ПО: три разные роли разработчиков работают вместе над надежным, практичным совершенством, причем каждый специализируется на чем-то своем и все взаимодействуют на равных.

Кто хочет работать в компании, в которой программные продукты создаются подобным образом? Мы точно хотим!

К сожалению, никто из нас в таких компаниях не работает. Google, как и многие компании до нее, потратил множество усилий на то, чтобы приблизиться к идеалу. Возможно, потому что мы поздно начали, мы учились на ошибках предшественников. Google повезло поймать момент перехода модели программных продуктов от огромных клиентских приложений с многолетними циклами выпуска к облачным сервисам, которые выпускаются каждые несколько недель, дней или

часов[1]. Благодаря удачному стечению обстоятельств у нас получилось хоть как-то приблизить разработку в Google к идеальному процессу разработки ПО.

Итак, разработчики в Google занимаются реализацией функциональности, отвечают за построение компонентов, строят основу приложения, поставляемого пользователю. Они пишут функции и код юнит-тестов для этих функций. *Джек строит дом.*

Разработчики в тестировании в Google отвечают за создание средств тестирования. Они помогают разработчикам с написанием юнит-тестов и создают большие тестовые инфраструктуры, упрощая разработчикам процесс создания малых и средних тестов и помогая выявлять больше проблем. *Джек делает дом устойчивым.*

Инженеры по тестированию в Google встают на сторону пользователя во всех аспектах, связанных с качеством. В контексте разработки они создают автоматизацию пользовательских сценариев, а в контексте продукта — оценивают универсальность и эффективность всех действий по тестированию, которые выполняют другие инженеры. *Джек готовит дом к приходу гостей.*

Это уже не сказка, а наша попытка сделать ее былью.

В этой книге мы рассказываем в основном о работе разработчиков в тестировании и инженеров по тестированию. Роль разработчиков мы затрагиваем только там, где она соприкасается с тестированием. На самом деле разработчики вовлечены в тестирование довольно сильно, но обычно эти процессы курируют специалисты, в должности которых есть слово «тестирование».

Жизнь разработчика в тестировании

Когда компания только появляется, тестировщиков в ней, как правило, нет. Точно так же как нет руководителей проектов, системных администраторов и других должностей. Каждый сотрудник выполняет все эти роли одновременно. Мы любим представлять, как Ларри и Сергей[2] ломали головы над пользовательскими сценариями и юнит-тестами на заре Google. С ростом компании появились разработчики в тестировании, которые первыми привнесли фокус на качество в строго технологический дух программирования[3].

1 Заметим, что даже для клиентских продуктов Google старается делать частые и надежные обновления с помощью автообновления, встроенного во все клиентские приложения.

2 Речь идет о Ларри Пейдже и Сергее Брине — основателях Google.

3 О том, как появилась роль разработчика в тестировании, Патрик Коупленд рассказывает в предисловии.

Как организованы процессы разработки и тестирования

Прежде чем мы взглянем поближе на задачи разработчиков в тестировании, давайте посмотрим, в каких условиях они работают. Разработчики в тестировании и разработчики связаны очень тесно, у них много общих задач в работе над любым продуктом. Так происходит, потому что в Google тестирование — обязанность всей инженерной команды, а не только людей, в должности которых есть слово «тестирование».

Готовый код — основной артефакт, над которым работают все участники команды. Организовать структуру, написать код и сопровождать его — их ежедневные задачи. Большая часть кода Google хранится в едином репозитории и использует общие инструменты. Все процессы сборки и выпуска построены вокруг этих инструментов и репозитория. Каждый инженер Google знает эту среду как свои пять пальцев, поэтому любой участник команды независимо от своей роли может залить новый код, создать и запустить тест, собрать версию и т. д.

НА ЗАМЕТКУ

Готовый код — основной артефакт, над которым работают все участники команды. Организовать структуру, написать код и сопровождать его — их ежедневные задачи.

Еще один плюс единого репозитория: инженерам, переходящим из проекта в проект, не нужно переучиваться, а так называемые «двадцатипроцентные сотрудники» могут нормально работать с первого дня в проекте[1]. Кроме того, весь исходный код доступен любому инженеру. Разработчики веб-приложений могут просмотреть интересующий их код браузера, не спрашивая ни у кого разрешения, или узнать, как другие, более опытные сотрудники решали аналогичные задачи. Можно взять код для повторного использования на уровне модулей или даже на уровне структур контролов или данных. Google — един, и он использует общий репозиторий с удобными (еще бы!) средствами поиска.

1 В Google есть официальная система распределения графика, благодаря которой любой сотрудник может тратить 20% своего рабочего времени на любой другой проект Google. Иными словами, четыре дня в неделю вы занимаетесь своей основной работой, а один день экспериментируете и изобретаете. Необязательно именно так распределять свое время, многие бывшие сотрудники Google вообще считали такой рабочий график мифическим. Однако каждый из авторов этой книги участвовал в «двадцатипроцентных проектах», значит, они — реальность. Более того, многие инструменты, описанные в книге, — результат «двадцатипроцентной» работы, которая со временем воплотилась в полноценные финансируемые проекты. Правда, многие сотрудники Google в свое «двадцатипроцентное время» просто работают с другими проектами, а не занимаются экспериментами, поэтому от такой концепции рабочего времени выигрывают многие. (К моменту издания книги на русском языке Google официально отменил эту систему. — *Примеч. перев.*)

Благодаря тому что наша кодовая база открыта, доступ к ней есть у всей компании, а технический инструментарий един, мы разработали огромный набор библиотек и сервисов для общего использования. Общий код надежно работает на боевой среде Google и ускоряет работу над проектами, а использование общих библиотек при разработке сокращает количество багов.

НА ЗАМЕТКУ

Благодаря тому что наша кодовая база открыта, доступ к ней есть у всей компании, а технический инструментарий един, мы разработали огромный набор библиотек и сервисов для общего использования.

Так как код вливается в общую инфраструктуру, к его разработке инженеры относятся очень осторожно. Подтверждают такое особое отношение неписаные, но соблюдаемые правила работы с кодом.

— Используйте уже написанные библиотеки по максимуму. Пишите свою только в том случае, если у вас есть на то серьезная причина из-за специфических особенностей проекта.

— Когда вы пишете код, помните, что он должен быть легко читаем. Сделайте так, чтобы его можно было без труда найти в репозитории. С вашим кодом будут работать другие люди, поэтому нужно сделать его понятным и легко изменяемым.

— Общий код должен быть автономным и пригодным для повторного использования. Инженеров поощряют за создание сервисов, которые могут использовать другие команды. Возможность повторного использования кода гораздо важнее, чем его сложность.

— Сделайте зависимости настолько очевидными, что их невозможно будет пропустить. Если проект зависит от общего кода, то пусть о любых его изменениях узнают участники всех зависимых проектов.

Если инженер предлагает более удачное решение, то он же проводит рефакторинг всех существующих библиотек и помогает перевести все зависимые проекты на новые библиотеки. Работа на благо сообщества должна поощряться[1].

Google серьезно относится к процедуре код-ревью, и любой код, особенно общий, просматривает специалист с черным поясом по читаемости кода. Специальная комиссия выделяет таких специалистов за умение писать чистый и сти-

1 Самым распространенным механизмом такого поощрения в Google является премия «от коллег». Любой инженер, которому помогла работа другого инженера, может назначить ему премию в благодарность. У руководителей тоже есть свои способы поощрения сотрудника. Смысл в том, что работу на благо сообщества нужно подпитывать! Конечно, не стоит забывать, что есть и неформальный способ поблагодарить коллегу — «услуга за услугу».

листически точный код для четырех основных языков Google (C++, Java, Python и JavaScript).

Код в общем репозитории устанавливает более высокую планку для тестирования, — об этом мы расскажем чуть позже.

Для решения проблем с зависимостями платформ мы минимизируем их различия на машинах сотрудников. У всех инженеров на компьютерах стоит та же версия OS, что и на боевых машинах. Мы тщательно следим за актуальностью сборок Linux, чтобы разработчик, проводящий тестирование на своем компьютере, получил такие же результаты, как и при тестировании в боевой среде. Различия в процессорах и операционных системах между компьютерами инженеров и дата-центрами минимальны[1]. Если баг возник на компьютере тестировщика, то он, скорее всего, воспроизведется и на компьютере разработчика, и в условиях реальной эксплуатации.

Весь код, связанный с зависимостями платформ, собирается в библиотеки на самом нижнем уровне стека. Управляют этими библиотеками те же ребята, которые отвечают за дистрибутивы Linux. Наконец, для каждого из языков программирования, на которых пишут в Google, мы используем только один компилятор, который поддерживается и постоянно тестируется на одной из сборок Linux. Ничего сложного, но эта простая схема экономит силы на финальных стадиях тестирования, а заодно сокращает количество проблем, связанных со спецификой среды, которые сложны в отладке и отвлекают от разработки новых функций. Простота и надежность.

НА ЗАМЕТКУ

Вся платформа Google построена на простоте и единообразии: одинаковые сборки Linux для компьютеров инженеров и машин с боевой средой, общие базовые библиотеки под централизованным управлением, общая инфраструктура хранения исходного кода, сборки и тестирования, один компилятор для каждого из языков программирования, общая спецификация сборки для всех языков. И самое важное, культура, которая уважает и поощряет поддержку этих общих ресурсов.

Тему единообразия платформ и единства репозитория продолжает единая система сборки, не зависящая от языка, на котором написан проект. Не важно, на каком языке работает команда (C++, Python или Java), она все равно будет использовать общие «файлы сборки».

Чтобы сборка состоялась, нужно указать «цель сборки». Это может быть библиотека, бинарный файл или набор тестов, который состоит из некоторого количества исходных файлов.

1 Это правило в Google нарушено только в лабораториях локального тестирования Android и Chrome OS, где для проверки новых сборок нужно иметь широкий спектр оборудования.

Последовательность шагов следующая.

1. Напишите класс или набор функций в одном или нескольких исходных файлах. Убедитесь, что весь код компилируется.
2. Укажите цель сборки (например, определенную библиотеку) для новой сборки.
3. Напишите юнит-тесты, которые импортируют библиотеку, имитируют нетривиальные зависимости и выполняют интересующие нас пути в коде для самых актуальных входных данных.
4. Создайте тестовую сборку для юнит-тестов.
5. Соберите и запустите сборку с тестами. Изменяйте код до тех пор, пока все тесты не будут проходить.
6. Запустите все обязательные инструменты статического анализа, которые проверяют соответствие кода гайдлайнам и выявляют стандартные баги.
7. Отправьте итоговый код на код-ревью (подробнее о код-ревью мы расскажем позже), внесите изменения и повторите все юнит-тесты.

В результате мы создаем две сборки: собственно библиотеку, представляющую новый сервис, и сборку тестов для этого сервиса. Учтите, что многие разработчики в Google применяют методологию TDD (Test-Driven Development, или разработка через тестирование), при которой шаг 3 предшествует шагам 1 и 2.

Если разработчик конструирует более крупный сервис, то он продолжает писать код, связывая постоянно увеличивающиеся сборки библиотек. Сборка бинарника создается из основного файла, который ссылается на нужные библиотеки. Так появляется продукт Google, у которого есть:

— хорошо протестированный автономный бинарный файл;

— легкочитаемая и приспособленная для повторного использования библиотека (с набором вспомогательных библиотек, которые можно использовать для создания других сервисов);

— набор юнит-тестов, покрывающих нужные аспекты всех сборок.

Типичный продукт Google — это набор нескольких сервисов. В любой команде мы стараемся добиться соотношения 1:1 между разработчиками и сервисами. Это означает, что сервисы собираются и тестируются параллельно, а затем интегрируются в итоговой сборке. Чтобы связанные сервисы могли создаваться одновременно, их интерфейсы взаимодействия согласовываются в начале проекта. Тогда разработчики могут реализовывать зависимости через такие интерфейсы, а не через библиотеки. В начале работы разработчики создают имитации таких интерфейсов, чтобы начать писать тесты на уровне всего сервиса.

Разработчики в тестировании вовлечены в создание большинства тестовых сборок и определяют, где нужно писать малые тесты. По мере того как маленькие сборки собираются в целое приложение, задачи растут, и уже нужно проводить более крупные интеграционные тесты. Если для сборки одной библиотеки достаточно выполнения малых тестов, написанных самим разработчиком, то с увеличением объема сборок разработчики в тестировании вовлекаются во все большей степени и пишут уже средние и большие тесты.

Сборка увеличивается в размерах, и малые тесты становятся частью регрессионного пакета. Они должны быть всегда актуальны. В противном случае в них инициируются баги, отладка которых ничем не отличается от отладки багов основного кода. Тесты — часть функциональности, а значит, баги в тестах относятся к функциональным багам и исправляются. Такая схема гарантирует, что новая функциональность не нарушит работу существующей, а изменения в коде не сломают тесты.

Разработчики в тестировании — центр всей этой деятельности. Они помогают разработчикам решить, какие юнит-тесты написать. Они создают подставные объекты и имитации. Они пишут средние и большие интеграционные тесты. Именно об этих задачах разработчиков в тестировании мы собираемся сейчас рассказать.

Кто такие разработчики в тестировании на самом деле?

Разработчики в тестировании — это инженеры, которые помогают тестировать на всех уровнях процесса разработки Google. Но все же в первую очередь они именно разработчики. Во всех наших руководствах по найму и внутренних документах написано, что их работа на 100% связана с программированием. Этот специфический, можно даже сказать гибридный, подход к тестированию позволяет нам рано привлекать тестировщиков к проектам. Причем они занимаются не составлением абстрактных тест-планов или моделей качества, а сразу погружаются в проектирование и написание кода. Это ставит на одну чашу весов и программистов, и тестировщиков. Это повышает производительность команды и создает доверие ко всем видам тестирования, включая ручное и исследовательское, которое потом проведут уже другие инженеры.

НА ЗАМЕТКУ

Тест — это еще одна фича приложения, и за нее отвечают разработчики в тестировании.

Разработчики в тестировании работают рука об руку с разработчиками продукта, причем в буквальном смысле. Мы стараемся, чтобы они даже сидели вместе. Тесты — это еще одна фича приложения, за которую отвечают разработчики в тестировании. Разработчики и разработчики в тестировании участвуют в ревью кода, написанного друг другом.

На собеседовании разработчики в тестировании должны продемонстрировать такие же знания по программированию, как и разработчики. Даже больше — они должны уметь тестировать код, который написали. Проще говоря, разработчик в тестировании должен ответить на те же вопросы по программированию, что и разработчик, а потом еще решить задачки по тестированию.

Как вы уже догадались, специалистов на эту роль найти непросто. Скорее всего, это и есть причина относительно малого количества разработчиков в тестировании в Google. А вовсе не то, что мы нашли волшебную формулу производительности. Скорее, мы смирились с реальностью и адаптировали нашу работу, зная, что такое сочетание навыков встречается редко. Однако сходство ролей разработчика и разработчика в тестировании дало приятный побочный эффект: люди могут переходить из одной группы в другую. Google как раз старается поддерживать переходы между ролями. Представьте компанию, в которой все разработчики умеют тестировать, а все тестировщики умеют программировать. Нам далеко до этого, и, наверное, мы никогда такими не станем, но эти группы все-таки пересекаются. Мы находим разработчиков в тестировании со склонностью к разработке и разработчиков со склонностью к тестированию. Такие ребята становятся нашими лучшими инженерами и образуют самые эффективные команды разработки.

Ранняя стадия проекта

В Google нет правила, когда именно разработчики в тестировании должны присоединиться к проекту. Так же как нигде не прописано, когда именно проект становится «реальным». Типичный сценарий создания нового проекта такой: эксперимент, над которым работали в «двадцатипроцентное» время, набирается сил и становится самостоятельным продуктом Google. Именно так развивались Gmail и Chrome OS. Эти проекты начинались с неформальных идей, а со временем выросли в полноценные продукты со своими командами разработчиков и тестировщиков. Наш друг Альберто Савоя (написавший введение к этой книге) любит повторять, что «качество не имеет значения, пока ваш продукт не имеет значения».

В свое «двадцатипроцентное» время команды придумывают много нового. Часть этих идей ни к чему не приведет, другая часть станет новыми фичами других проектов, а некоторые смогут перерасти в официальные продукты Google. Ни одному проекту по умолчанию не полагается тестирование. Проект может потерпеть

неудачу, поэтому включать в него тестировщиков — напрасная трата ресурсов. Если проект закроют, что мы будем делать с готовой тестовой инфраструктурой?

Браться за качество еще до того, как концепция продукта дозрела и полностью сформирована, — это типичный пример неправильной расстановки приоритетов. Мы повидали много прототипов, созданных в «двадцатипроцентное» время, которые перерабатывались так сильно, что на стадии бета-версии или версии для внутренних пользователей от оригинального кода оставалась крохотная часть. Тестирование в экспериментальных проектах — безнадежная затея.

Конечно, не стоит впадать в крайности. Если продукт слишком долго развивается без тестирования, то становится тяжело менять архитектурные решения, плохо влияющие на его тестируемость. Это усложняет автоматизацию, а тестовые инструменты становятся ненадежными. Чтобы повысить качество, придется многое переделывать. Такой технический долг может затормозить разработку продукта на годы.

В Google не принято, чтобы тестирование появлялось в проектах рано. На самом деле разработчики в тестировании часто приходят на проекты на ранних этапах, но пока они больше разработчики, чем тестировщики. Это наше сознательное решение, но это не значит, что на ранних стадиях мы забываем о качестве. Это следствие неформального и одержимого новшествами процесса созидания в Google. Многомесячное планирование проекта перед разработкой, включающее контроль качества и тесты, — это не про нас. Проекты в Google начинаются намного менее формально.

Chrome OS — яркий пример такого подхода. На этом проекте все трое авторов этой книги работали больше года. Но задолго до того, как мы официально присоединились к работе, несколько разработчиков создали прототип. Он состоял в основном из скриптов и заглушек, но зато позволял продемонстрировать идею «чисто браузерного» приложения руководству Google, чтобы официально утвердить проект. На стадии прототипа команда концентрировалась на экспериментах и хотела доказать, что этот концепт вообще жизнеспособен. Тратить время на тестирование — или даже проектировать с оглядкой на тестируемость — было бы неразумным, особенно учитывая, что проект был еще неофициальным, а все демосценарии в будущем все равно заменили бы реальным кодом. Когда сценарии выполнили свое предназначение и продукт был утвержден, руководитель разработки обратился к нам за помощью в тестировании.

Все это — особая культура Google. Ни один проект не получит ресурсы тестирования просто так. Команды разработки обращаются к тестировщикам за помощью, убеждая их в том, что проект по-настоящему интересен и перспективен. После того как руководители разработки Chrome OS обрисовали свой проект, состояние дел и график выпуска, мы смогли выдвинуть свои требования по участию разработчиков в тестировании, уровню покрытия кода юнит-тестами и разделению обязанностей в процессе работы. Мы не участвовали в зарождении проекта, но когда он стал реальным, мы смогли серьезно повлиять на его реализацию.