Содержание

Об авторе

Мориц Ленц (Moritz Lenz) – плодовитый блогер, автор и участник проектов с открытым исходным кодом. Он работает архитектором программного обеспечения и главным инженером-программистом для ИТ среднего бизнеса аутсорсинговой компании, где создал систему непрерывной интеграции и доставки для более пятидесяти программных библиотек и приложений.

О техническом рецензенте

Майкл Томас (Michael Thomas) более 20 лет трудился в области разработки программного обеспечения в качестве индивидуального участника, руководителя группы, менеджера программ и вице-президента по проектированию. Майкл имеет более 10 лет опыта работы с мобильными устройствами. В настоящее время он работает в медицинском секторе, используя мобильные устройства для ускорения обмена информацией между пациентами и поставщиками медицинских услуг.

Благодарность

Написание книги не является одиночным начинанием и возможно только с помощью многих людей и организаций. Я хотел бы поблагодарить всех моих бета-читателей (beta reader), которые предоставили обратную связь. К ним относятся, в произвольном порядке, Карл Фогель (Karl Vogel), Михаил Иткин (Mikhail Itkin), Карл Мясак (Carl Mäsak), Мартин Турн (Martin Thurn), Шломи Фиш (Shlomi Fish), Ричард Липпманн (Richard Lippmann), Ричард Фоли (Richard Foley) и Роман Филиппов (Roman Filippov). Пол Кокрейн (Paul Cochrane) заслуживает особой благодарности за рецензирование и предоставление обратной связи по сообщениям в блоге и рукописи, а также за доступность для обсуждения контента, идей и организационных вопросов.

Я также хочу поблагодарить мою издательскую команду в Apress: Стив Англин (Steve Anglin), Марк Пауэрс (Mark Powers) и Мэттью Муди (Matthew Moodie), – а также всех, кто делает потрясающую работу на заднем плане, например дизайн обложки, набор текста и маркетинг.

Наконец, спасибо моим родителям за то, что они разожгли мою любовь к книгам и технике. И самое главное, моей семье: Сигне (Signe) – моей жене, за постоянную поддержку; и моим дочерям – Иде (Ida) и Ронье (Ronja), за то, что они поддерживали меня в реальном мире и приносили радость в мою жизнь.

Введение

Одним из ключей к успешной разработке программного обеспечения является получение быстрой обратной связи. Это помогает разработчикам избежать тупиков, и в случае ошибки, которая быстро обнаруживается, ее можно исправить, пока код еще свеж в памяти разработчика.

С точки зрения бизнеса быстрая обратная связь также помогает заинтересованным сторонам и продуктовому менеджеру не создавать функциональность, которая оказывается бесполезной, что позволяет избежать напрасных усилий. Достижение взаимопонимания о желаемом продукте является очень сложной задачей в любом программном проекте. Показ работающего (хотя и частично) продукта на ранней стадии часто помогает устранить недопонимание между заинтересованными сторонами и разработчиками.

Существует множество способов добавления обратной связи на разных уровнях, от добавления компоновки (linting) и других проверок кода в IDE до гибких процессов (agile processes), которые подчеркивают повышенную стоимость доставки. Первая часть этой книги посвящена тестам программного обеспечения и их автоматическому выполнению, практике, известной как *непрерывная интеграция* (Continuous Integration – CI).

При реализации CI вы настраиваете сервер, который автоматически проверяет каждое изменение исходного кода, возможно, в нескольких средах, например в комбинациях версий операционной системы и языка программирования.

Следующим логическим шагом и темой второй части этой книги является *непрерывная доставка* (Continuous Delivery – CD). После создания и тестирования кода вы добавляете больше шагов к автоматизированному процессу: автоматическое развертывание в одной или нескольких тестовых средах, дополнительные тесты в установленном состоянии и, наконец, развертывание в производственной среде. Последний шаг обычно охраняется воротами ручного одобрения.

CD расширяет автоматизацию и таким образом предоставляет возможность быстрых итерационных циклов вплоть до производственной среды, где программное обеспечение может принести

пользу. С помощью такого механизма вы можете быстро получить обратную связь или данные об использовании от реальных клиентов и оценить, полезно ли расширять функциональность или обнаруживать баги, пока разработчики все еще помнят код, который они написали.

Примеры кода в этой книге используют Python. Благодаря своей динамической природе Python хорошо подходит для небольших экспериментов и быстрой обратной связи. Хорошо укомплектованная стандартная библиотека и обширная экосистема доступных библиотек и фреймворков, а также чистый синтаксис Python делают его хорошим выбором даже для более крупных приложений. Python обычно используется во многих областях, например в веб-разработке, науке о данных и машинном обучении, интернете вещей (IoT) и автоматизации систем. Это становится лингва франка профессиональных программистов и тех, кто просто коснулся автоматизировать какую-то часть своей работы или хобби.

Python поставляется в двух основных языковых версиях, 2 и 3. Поскольку поддержку Python 2 планируется завершить в 2020 году, и почти все основные библиотеки теперь поддерживают Python 3, новые проекты следует начинать в Python 3, а устаревшие приложения нужно перенести на эту языковую версию, если это возможно. Следовательно, в этой книге предполагается, что Python относится к Python 3, если явно не указано иное. Если вы знаете только Python 2, будьте уверены, что вы легко поймете исходный код, содержащийся в этой книге, и с легкостью перенесете знания в Python 3.

Целевая аудитория

Эта книга предназначена для технических специалистов, вовлеченных в процесс доставки программного обеспечения: разработчиков программного обеспечения, архитекторов, инженеров по релизу и инженеров DevOps.

Главы, в которых используются примеры исходного кода, предполагают базовое знакомство с языком программирования Python. Если вы знакомы с другими языками программирования, потратьте несколько часов на чтение вводного материала по Python. Вы, вероятно, достигнете уровня, на котором сможете легко следовать примерам кода в книге.

В примере инфраструктуры используется Debian GNU/Linux, поэтому знакомство с этой операционной системой полезно, хотя и не обязательно.

Примеры кода

Примеры кода, используемые в этой книге, доступны на GitHub под организацией `python-ci-cd` по адресу https://github.com/python-ci-cd или по ссылке **Download Source Code**, расположенной по адресу www.apress.com/9781484242803.

Поскольку некоторые примеры основаны на автоматическом извлечении кода из определенных репозиториев Git, необходимо разделить их на несколько репозиториев. Несколько глав ссылаются на отдельные репозитории в этом пространстве имен.

Так будут оформляться предупреждения и важные примечания.

Так будут оформляться советы или рекомендации.

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Скачивание исходного кода примеров

Скачать файлы с дополнительной информацией для книг издательства «ДМК Пресс» можно на сайте www.dmkpress.com или www.дмк.рф на странице с описанием соответствующей книги.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в основном тексте или программном коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу dmkpress@gmail.com, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и Apress очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Глава 1

Автоматическое тестирование

Перед погружением в примеры тестирования кода Python необходимо более подробно обсудить саму природу тестов.

1.1. Что же мы хотим от тестов

Зачем вообще заморачиваться над написанием тестов? Существует множество причин, почему мы хотим написать или хотя бы иметь тесты.

Это не необычно иметь несколько тестов в одном наборе тестов (test suite), написанных в ответ различным потребностям.

Быстрая обратная связь

Каждое изменение в коде содержит риск внести баги. Исследования показывают, что от 7 до 20 % исправлений багов привносят новые баги[1].

Не лучше ли, если бы мы могли найти больше багов перед тем, как они найдут путь к клиенту? Или даже до того, как ваши коллеги увидят их? Это не просто вопрос тщеславия. Если вы быстро получаете обратную связь о том, что у вас баг, то проще вспомнить все детали той части кода, над которой только что работали, так что исправление багов будет в разы быстрее.

---

[1] *Jim Bird*. Bugs and Numbers: How Many Bugs Do You Have in Your Code? // Building Real Software: Developing and Maintaining Secure and Reliable Software in the Real World. URL: http://swreflections.blogspot.de/2011/08/bugs-and-numbers-how-many-bugs-do-you.html. August 23, 2011.

Много тест-кейсов (test case), чтобы предоставить цикл быстрых обратных связей. Вы можете запустить их до того, как отправить свои изменения в систему контроля версий, и они сделают вашу работу более эффективной и будут держать историю вашего кода чистой.

Уверенность

Ссылаясь на предыдущий абзац, следует упомянуть отдельно, можете быть уверены, зная, что простые ошибки за вас будут отловлены набором тестов. В большинстве компаний, ориентированных на разработку программного обеспечения, существуют критические зоны, в которых баги могут подвергнуть опасности весь бизнес. Только представьте, что разработчик случайно ошибся в системе входа (login system) в программе управления данными о здоровье и теперь клиенты видят чужие диагнозы. Или система автоматической выплаты перечислила на клиентскую кредитную карту неправильную сумму.

Даже компании, не разрабатывающие программное обеспечение, могут катастрофически пострадать от ошибок в программах. Оба – климатический орбитальный аппарат Mars[1] и первый запуск ракеты «Ariane 5»[2] – понесли потери соответствующего транспортного средства из-за проблем с программным обеспечением.

Ответственность их работы создает эмоциональный стресс для разработчиков программного обеспечения. Автоматизированные тесты (automated tests) и хорошая методология разработки могут помочь уменьшить этот стресс.

Даже если разрабатываемое программное обеспечение не является критически важным, неблагоприятные факторы риска могут привести к тому, что разработчики или сопровождающие лица внесут наименьшие возможные изменения и отложат необходимый рефакторинг (refactoring), который сохранит код в рабочем состоянии. Уверенность, которую обеспечивает хороший набор тестов, может позволить разработчикам сделать то, что необходимо, чтобы кодовая база не стала пресловутым большим комком грязи[3].

---

[1] Wikipedia. Mars Climate Orbiter // https://en.wikipedia.org/wiki/Mars_Climate_Orbiter. 2018.

[2] *J. L. Lions*. Ariane 5: Flight 501 Failure. Report by the Inquiry Board // http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html. July 1996.

[3] Wikipedia. Большой комок грязи // https://ru.wikipedia.org/wiki/Большой_комок_грязи. 2018.

Помощь в отладке

Когда разработчики изменяют код, что, в свою очередь, приводит к сбою теста, они хотят, чтобы тест помог найти ошибку. Если тест просто говорит «что-то не так», это знание лучше, чем незнание ошибки. Было бы еще полезнее, если бы тест мог дать подсказку для начала отладки.

Если, например, сбой теста показывает, что функция `find_Short-test_path` вызвала исключение, вместо того чтобы возвращать путь, как ожидалось, и мы знаем, что или эта функция (либо та, которую она вызвала) сломалась, или она получила неправильный ввод. Это гораздо лучшее средство отладки.

Справка по проектированию

Движение Extreme Programming (XP)[1] отстаивает необходимость практической *разработки на основе тестов* (test-driven development – TDD). То есть прежде чем писать какой-либо код, который решает проблему, вы сначала пишете неудачный тест. Затем пишете достаточно кода, чтобы пройти тест. Либо вы закончили, либо пишете следующий тест. Промыть и повторить.

Это имеет очевидные преимущества: вы убедитесь, что весь код, который пишется, имеет тестовое покрытие, и что вы не пишете ненужный или недостижимый код. Тем не менее специалисты по TDD также сообщили, что подход, основанный на тестировании, помог им написать лучший код. Один из аспектов заключается в том, что написание теста заставляет задуматься о программном интерфейсе приложения (Application programming interface – API), который будет иметь реализацию, и поэтому вы начнете реализацию с лучшим представлением. Другая причина заключается в том, что чистые (pure) функции (функции, возвращаемое значение которых зависит только от ввода, они не вызывают побочных эффектов и не считывают данные из баз данных и т. д.) очень просто проверить. Таким образом, подход, основанный на тестировании, направляет разработчика к лучшему разделению алгоритмов или бизнес-логики от поддержки логики и вспомогательной. Такое разделение интересов является аспектом хорошего проектирования программного обеспечения.

---

[1] Wikipedia. Extreme programming // https://en.wikipedia.org/wiki/Extreme_programming. 2018.

Следует отметить, не все согласны с этими наблюдениями, учитывая опыт или аргументы о том, что тестировать некоторый код гораздо сложнее, чем писать, это приводит к трате усилий, требуя тестов для всего. Тем не менее помощь в разработке, которую могут предоставить тесты, является причиной, по которой разработчики пишут код, и поэтому не должна быть здесь упущена.

Спецификация продукта

Дни больших унифицированных технических документов для программных проектов в основном прошли. Большинство проектов следуют некоторой итеративной модели разработки, и даже если есть подробный документ спецификации, он часто устарел.

Когда нет подробной и актуальной спецификации, тестовый набор может взять на себя роль спецификации. Когда люди не уверены, как программа должна вести себя в определенной ситуации, тест может дать ответ. Для языков программирования, форматов данных, протоколов и прочего может даже иметь смысл предложить набор тестов, который можно использовать для проверки более чем одной реализации.

1.2. Недостатки тестов

Было бы нечестно молчать о недостатках тестов. Эти недостатки не должны отвлекать вас от написания тестов, но знание о них поможет решить: что тестировать, как писать тесты и, возможно, сколько тестов писать.

Усилия

Написание тестов требует времени и усилий. Таким образом, когда вам поручено реализовать функцию, нужно не только реализовать эту функцию, но и написать тесты для нее, что приведет к большей работе и меньшему количеству времени для выполнения других задач, которые могут принести прямую пользу бизнесу. Если, конечно, тесты не обеспечивают достаточной экономии времени (например, за счет того, что нет необходимости исправлять ошибки в производственной среде и очищать данные, которые были повреждены в результате ошибки), чтобы амортизировать время, затрачиваемое на написание тестов.

Дополнительный код для поддержки

Тесты сами по себе являются кодом и должны поддерживаться, как и код, который тестируется. В общем, вы хотите наименьшее количество кода, которое может решить вашу проблему, потому что чем меньше у вас кода, тем меньше кода нужно поддерживать. Думайте о коде (включая тестовый код) как об обязательстве, а не как об активе.

Если вы пишете тесты вместе с вашими функциями и исправлениями ошибок, то должны изменить эти тесты при изменении требований. Некоторые тесты также требуют изменения при рефакторинге, что затрудняет изменение базы кода.

Хрупкость

Некоторые тесты могут быть хрупкими, то есть они иногда дают неправильный результат. Тест, который не проходит, даже если рассматриваемый код является правильным, называется *ложноположительным*. Такой сбой теста требует времени для отладки, не предоставляя никакой ценности. *Ложноотрицательный тест* – это тест, который не дает сбоя, когда тестируемый код не работает. Ложноотрицательный тест также не имеет никакой ценности, но его гораздо труднее обнаружить, чем ложноположительный тест, поскольку большинство инструментов привлекают внимание к неудачным тестам.

Хрупкие тесты подрывают доверие к тестам. Если развертывание продукта с ошибочными тестами становится нормой – поскольку все считают, что эти неудачные тесты являются ложноположительными,– то значение результатов набора тестов упало до нуля. Вы можете по-прежнему использовать его для отслеживания информации, какой из тестов не прошел по сравнению с последним прогоном, но это приводит к большой ручной работе, которую никто не хочет делать.

К сожалению, некоторые виды тестов очень сложно выполнить надежно. Тесты графического пользовательского интерфейса (Graphical User Interface – GUI), как правило, довольно чувствительны к изменениям макета или технологии. Тесты, в которых используются компоненты, находящиеся вне вашего контроля, также могут быть источником хрупкости.

Ложное чувство безопасности

Безупречный запуск набора тестов может дать вам ложное чувство безопасности. Это может быть связано или с ложноотрицательными результатами (тесты, которые должны были провалиться, но не провалились), или с отсутствием тестовых сценариев. Даже если тестовый набор достигает 100%-го покрытия тестируемого кода, он может пропустить некоторые пути кода (code path) либо сценарии. Таким образом, вы видите успешно прошедший тестовый прогон и воспринимаете это как указание на то, что ваше программное обеспечение работает правильно только для того, чтобы быть заваленным отчетами об ошибках, после того как реальные клиенты вступают в контакт с продуктом.

Не существует прямого решения для чрезмерной уверенности, которую может обеспечить набор тестов. Только благодаря опыту работы с кодовой базой и ее тестами вы почувствуете реалистичные уровни достоверности, которые обеспечивает зеленый (т. е. проходящий) тестовый прогон.

1.3. Характеристики хорошего теста

Хороший тест – это тест, который сочетает в себе несколько причин написания тестов, избегая при этом как можно больше недостатков. Это означает, что тест должен быть быстрым, простым для понимания и поддержки, давать хорошую и конкретную обратную связь в случае неудачи и быть надежным.

Может быть, несколько удивительно, но иногда он должен давать сбои, хотя можно ожидать, что тест не пройден. Тест, который никогда не проходит, также никогда не дает обратной связи и не может помочь с отладкой. Это не означает, что вы должны удалить тест, для которого вы никогда не регистрировали сбой. Возможно, это не удалось на компьютере разработчика, и он или она исправили ошибку перед проверкой изменений.

Не все тесты могут соответствовать всем критериям хороших тестов, поэтому давайте рассмотрим некоторые из различных типов тестов и компромиссы, которые им присущи.

1.4. Виды тестов

Существует традиционная модель классификации тестов, основанная на их области действия (объем кода, который они охваты-

вают) и их назначении. Эта модель разделяет код тестов на модульные, интеграционные и системные тесты. Он также добавляет дымовое тестирование (smoke tests), тесты производительности (performance tests) и другие тесты для различных целей.

Модульные тесты (Unit Tests)

Модульный тест (выполняется изолированно) – наименьший блок программы, которую имеет смысл охватить. В процедурном или функциональном программировании язык, который имеет тенденцию быть подпрограммой или функцией. В объектно ориентированном языке, таком как Python, это может быть метод. В зависимости от того, насколько строго вы интерпретируете определение, это также может быть класс или модуль.

Модульный тест должен избегать запуска кода за пределами тестируемого модуля. Поэтому, если вы тестируете бизнес-приложение, интенсивно использующее базу данных, модульный тест по-прежнему не должен выполнять вызовы базы данных (доступ к сети для вызовов API) или файловой системы. Существуют способы замены таких внешних зависимостей для целей тестирования, о которых я расскажу позже, хотя если вы сможете структурировать свой код, чтобы избежать таких вызовов, по крайней мере в большинстве модулей, тем лучше.

Поскольку доступ к внешним зависимостям делает большую часть кода медленным, модульные тесты обычно бывают невероятно быстрыми. Это делает их подходящими для тестирования алгоритмов или основной бизнес-логики.

Например, если ваше приложение является помощником по навигации, в нем есть хотя бы один фрагмент алгоритмически сложного кода: маршрутизатор, который (учитывая карту, начальную точку и цель) создает маршрут или список возможных маршрутов, с такими показателями, как длина и ожидаемое время прибытия. Этот маршрутизатор или даже его части – это то, что вы хотите охватить модульными тестами как можно тщательнее, включая странные крайние случаи, которые могут вызвать бесконечные циклы, или проверить, что путешествие из Берлина в Мюнхен не отправляет вас через Рим.

Огромный объем тестовых случаев, которые вы хотите для такого устройства, делает другие виды тестов нецелесообразными. Кроме того, вы не хотите, чтобы такие тесты проваливались из-

за несвязанных компонентов, поэтому их сосредоточенность на устройстве повышает специфичность.

Интеграционные тесты (Integration Tests)

Если вы собрали сложную систему, такую как автомобиль или космический корабль, из отдельных компонентов, и каждый компонент работает нормально в отдельности, каковы шансы, что все это работает? Существует множество причин, по которым что-то может пойти не так: некоторые провода могут быть неисправны, компоненты хотят общаться по несовместимым протоколам, или, возможно, соединения не могут противостоять вибрации во время работы.

По программному обеспечению ничего не отличается, поэтому пишут интеграционные тесты. Интеграционный тест проверяет сразу несколько блоков. Это делает несоответствия на границах между блоками очевидными (через сбои теста), позволяя исправлять такие ошибки на ранней стадии.

Системные тесты (System Tests)

Системный тест помещает часть программного обеспечения в среду и проверяет ее там. Для классической трехуровневой архитектуры системный тест начинается с ввода через пользовательский интерфейс и тестирует все уровни вплоть до базы данных.

Если модульные тесты и интеграционные тесты являются тестами белого ящика (тесты, которые требуют и используют знания о том, как реализовано программное обеспечение), системные тесты, как правило, являются тестами черного ящика. Они принимают точку зрения пользователя и не заботятся о внутренностях системы.

Это делает системные тесты наиболее реалистичными с точки зрения того, как программное обеспечение подвергается тестированию, но они имеют несколько недостатков.

Во-первых, управление зависимостями для системных тестов может быть очень сложным. Например, если вы тестируете веб-приложение, обычно сначала требуется учетная запись, использующаяся для входа в систему, а затем для каждого тестового примера требуется фиксированный набор данных, с которыми он может работать.

Во-вторых, системные тесты часто используют столько компонентов одновременно, что сбой теста не дает четкого представле-

ния о том, в каком месте на самом деле неправильно, и требует, чтобы разработчик смотрел на каждый сбой теста, нередко для выяснения, что изменения не связаны с ошибками теста.

В-третьих, системные тесты обнаруживают сбои в компонентах, которые вы не собирались тестировать. Системный тест может завершиться неудачей из-за неправильно настроенного сертификата безопасности транспортного уровня (Transport Layer Security – TLS) в API, который используется программным обеспечением и который может быть полностью вне вашего контроля.

Наконец, системные тесты обычно намного медленнее, чем модульные и интеграционные тесты. Тесты белого ящика позволяют вам тестировать только те компоненты, которые вам нужны, поэтому вы можете избежать запуска неинтересного кода. В тесте системы для веб-приложения вам, возможно, придется выполнить вход в систему, перейти на страницу, ту, что хотите проверить, ввести некоторые данные, а затем, наконец, выполнить тест, который вы действительно хотите сделать. Системные тесты часто требуют гораздо больше настроек, чем модульные или интеграционные, увеличивая время их выполнения и более длительный интервал до получения обратной связи о коде.

Дымовые тесты (Smoke Test)

Дымовой тест похож на системный в том, что он тестирует каждый слой в вашем технологическом стеке, хотя это не является тщательным тестом для каждого. Обычно он написан не для проверки правильности какой-либо части вашего приложения, а скорее для того, чтобы приложение вообще работало в его текущем контексте.

Дымовой тест для веб-приложения может быть такой же простой, как вход в систему, после чего следует вызов страницы профиля пользователя, чтобы убедиться, что имя пользователя появляется где-то на этой странице. Это не проверяет какую-либо логику, но обнаруживает такие вещи, как неправильно настроенный веб-сервер или сервер базы данных, неверные файлы конфигурации или учетные данные.

Чтобы получить больше пользы от дымового теста, вы можете добавить страницу состояния или конечную точку API в свое приложение, которое выполняет дополнительные проверки, такие как наличие всех необходимых таблиц в базе данных, наличие зависимых служб и т. д. Только если все эти зависимости во время выпол-

нения будут реализованы, будет статус **OK**, который может легко определить дымовой тест. Обычно вы пишете только один или два дымовых теста для каждого развертываемого компонента, но запускаете их для каждого развертываемого экземпляра.

Тесты производительности

Обсуждаемые до сих пор тесты фокусируются на корректности, но нефункциональные качества, такие как производительность и безопасность, могут быть одинаково важны. В принципе, довольно просто запустить тест производительности: записать текущее время, выполнить определенное действие, снова записать текущее время. Разница между двумя временными записями заключается во времени выполнения этого действия. При необходимости повторите и рассчитайте некоторую статистику (например: медиана, среднее значение, стандартное отклонение) из этих значений.

Как обычно, дьявол кроется в деталях. Основными проблемами являются создание реалистичной и надежной тестовой среды, реалистичных тестовых данных и реалистичных тестовых сценариев.

Многие бизнес-приложения сильно зависят от баз данных. Итак, ваша среда тестирования производительности также требует базы данных. Репликация большого производственного экземпляра базы данных для среды тестирования может быть довольно дорогой, как с точки зрения аппаратного обеспечения, так и с точки зрения затрат на лицензирование. Таким образом, существует соблазн использовать уменьшенную базу данных тестирования, что может привести к аннулированию результатов. Если что-то идет медленно в тестах производительности, разработчики, как правило, говорят: «Это просто слабая база данных; Прод легко справится» (Прод – Production server, на котором будет произведен релиз. – *Прим. перев.*), и они могут быть правы. Или нет. Нет способа узнать.

Другим коварным аспектом настройки среды является множество движущихся частей, когда речь заходит о производительности. На виртуальной машине (ВМ) вы обычно не знаете, сколько циклов ЦП получила виртуальная машина от гипервизора, или среда виртуализации играла забавные трюки с памятью виртуальной машины (например, выгружая часть памяти виртуальной машины на диск), вызывая непредсказуемую производительность.

На физических машинах (которые также лежат в основе каждой виртуальной машины) вы сталкиваетесь с современными системами управления питанием, которые контролируют тактовую ча-

стоту, основываясь на тепловых соображениях, а в некоторых случаях даже на специальных инструкциях, используемых в ЦП[1].

Все эти факторы приводят к тому, что измерения производительности становятся гораздо более неопределенными, чем можно наивно ожидать от такой детерминированной системы, как компьютер.

1.5. Резюме

Как разработчики программного обеспечения мы хотим, чтобы автоматизированные тесты давали быструю обратную связь об изменениях, улавливали регрессии до того, как они достигли клиента, и давали достаточную уверенность в том, что мы сможем изменить код. Хороший тест – быстрый, надежный и имеет высокую диагностическую ценность в случае неудачи.

Модульные тесты обычно бывают быстрыми и имеют высокую диагностическую ценность, но охватывают только небольшие фрагменты кода. Чем больше кода охватывает тест, тем медленнее и хрупче он становится, и его диагностическая ценность уменьшается.

В следующей главе мы рассмотрим, как писать и запускать модульные тесты на Python. Затем разберем, как запускать их автоматически для каждого коммита.

---

[1] *Vlad Krasnov*. On the Dangers of Intel's Frequency Scaling // Cloudflare. URL: https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/. November 10, 2017.