



# Erlang: Тесты программ

**Андрей Ушаков** начинает цикл, посвященный качеству нашего кода и программ, создаваемых на любимом языке Erlang.



Наш эксперт

Андрей Ушаков активно приближает тот день, когда функциональные языки станут мейнстримом.

**X**ороший способ улучшить качество кода — это проверить его работоспособность согласно некоторому набору тестов. Чем мы и займемся в данной серии уроков.

Что такое тестирование? Это процесс проверки работоспособности созданного программного обеспечения (некоторого продукта) и соответствия этого продукта заданным требованиям. Допустим, мы занимаемся разработкой такого программного продукта, как калькулятор. Тогда требования будут определять поддерживаемые нашим калькулятором функции: обычный это калькулятор (с поддержкой только четырех арифметических операций) или инженерный (с поддержкой тригонометрии и т. д.), должны ли мы поддерживать работу с системами счисления с основаниями, отличными от 10, и т. д. При проверке наличия в нашем калькуляторе тех или иных функций (например, функции `sin` для инженерного калькулятора) мы проверяем соответствие созданного нами калькулятора требованиям. А проверка, что  $2+2$  будет  $4$  — это проверка работоспособности нашего калькулятора.

Давайте бегло рассмотрим, как выполняется тестирование программного продукта. Обычно для этого применяются т. н. пользовательские сценарии: когда повторяются действия пользователей для достижения того или иного результата. При наличии достаточного количества пользовательских сценариев, мы можем проверить как выполнение всех требований к нашему продукту, так и правильность его работы (хотя осуществить такую проверку не всегда просто). Давайте вернемся к нашему примеру с калькулятором. Так, действия пользователя, состоящие из ввода некоторого числа и нажатия на кнопку `sin` (с последующей проверкой полученного результата), служат для проверки поддерживаемых калькулятором функций (для проверки соответствия требованиям). А действия пользователя, состоящие из ввода числа 2, нажатия на кнопку `+`, ввода числа 2, нажатия на кнопку `=` и проверки полученного результата (который, естественно, должен быть 4), служат для проверки работоспособности нашего калькулятора. Очевидно, что нет особой разницы между двумя этими сценариями: они оба состоят из некоторых действий пользователя и проверки получаемого результата. В более сложных случаях мы также должны подготовить некий набор исходных данных для того, чтобы получать один и тот же результат при одних и тех же действиях пользователя (но это не всегда возможно). Поэтому для проведения тестирования абсолютно неважно, откуда появились сценарии для тестирования: из требований, из проверки работоспособности или при воспроизведении той или иной ошибки.

У многих читателей может возникнуть вопрос: а зачем разработчикам вникать в процесс тестирования, если заниматься тестированием должны специальные люди (QA-специалисты)? Да, при правильной постановке процесса разработки программисты не занимаются тестированием всего продукта согласно набору пользовательских сценариев. Но при этом они все же должны принимать участие в тестировании создаваемого ими программного

обеспечения. Как может разработчик принять в этом участие? Ответ очевиден: создать тесты на разрабатываемый им код, компоненты, подсистемы, библиотеки и т. д. При этом тестирование, которое осуществляет разработчик (при помощи написания тестов), называют автоматизированным тестированием, в отличие от ручного тестирования приложения, которое осуществляет QA-специалист (хотя и QA-специалисты могут разрабатывать тесты для автоматизированного тестирования).

Вполне закономерен вопрос о плюсах и минусах такого подхода к разработке. Начнем с тех преимуществ, которые он нам дает. Главное его преимущество в том, что все внутренние составляющие нашего приложения (компоненты, подсистемы и т. д.) оказываются протестированы (естественно, при 100% покрытии тестами исходного кода). Вернемся снова к пользовательским сценариям работы: каждый сценарий — это та или иная комбинация возможных входных данных для всех компонентов, из которых состоит наше приложение. Если мы хотим полностью протестировать наше приложение, то должны составить все возможные комбинации входных данных для всех компонентов нашего приложения, после чего каждой такой комбинации сопоставить некоторый пользовательский сценарий работы. Очевидно, что число таких комбинаций будет настолько велико, что подобную работу будет невозможно выполнить. Поэтому полностью протестировать наше приложение невозможно: можно проверить лишь основные сценарии работы и отсутствие известных ошибок (обычно сценарий воспроизведения ошибки становится сценарием проверки ее исправления).

С другой стороны, тестируя внутренние компоненты приложения по отдельности, мы сможем проверить гораздо больше возможных сценариев использования компонентов (и по входным данным, и по манипуляциям с этими данными), т. е. протестировать больше аспектов работы нашего приложения. Конечно, наличие тестов на компоненты не означает, что нет нужды в тестировании приложения согласно пользовательским сценариям; правильная работы отдельных компонентов не гарантирует правильности работы приложения в целом. Наличие тестов на компоненты нашего приложения означает, что можно безболезненно заниматься развитием и сопровождением кода (добавлять новые функции, исправлять ошибки в имеющемся коде, выполнять рефакторинг).

Вполне возможна ситуация, что в процессе работы над кодом тесты перестанут проходить. Значит, у нас либо тесты перестали соответствовать действительности, либо мы что-то добавили некорректно, в результате чего наша реализация перестала удовлетворять проверкам в тестах. В первом случае мы изменяем тесты — таким образом, чтобы они соответствовали действительности. Во втором случае мы исправляем наши изменения в коде, чтобы они снова начали удовлетворять всем проверкам во всех тестах. Нет ничего страшного, если во время работы над задачей тесты перестанут проходить: достаточно, чтобы они проходили по окончании работы над задачей.

Разработка тестов для существующего кода часто приводит к пониманию того, что код не очень удобен для написания тестов к нему. После этого нередко приходится менять исходный код. Обычно проблемы с исходным кодом заключаются в сильной связности между его частями. Для решения этой проблемы создают интерфейсную часть и реализацию для некоторой функциональности, чтобы взаимодействие с этой функциональностью происходило через интерфейс. Это приводит к уменьшению связности кода, а также к тому, что мы в будущем при желании сможем изменить реализацию функциональности, не затрагивая использующий ее код. Как итог, разработка тестов часто вынуждает нас к некоторой декомпозиции исходного кода, что приводит к уменьшению его связности и увеличению гибкости дальнейшего использования. А это улучшает общую архитектуру системы.

С разработкой тестов связан еще один немаловажный момент: раннее обнаружение ошибок. Если ошибку обнаруживает QA-специалист, то он ищет стабильный сценарий ее воспроизведения, после чего регистрирует ее в баг-трекере. В какой-то момент разработчик берется за ее исправление. Для этого он воспроизводит ошибку у себя, после чего ищет в коде причину возникновения ошибки и исправляет ее. На все эти шаги может уйти достаточно много времени и сил. Если же разработчик при помощи тестов находит ошибку в коде, то он просто исправляет ее — и все.

И еще один плюс наличия тестов: они являются хорошей документацией по использованию кода и по внутреннему его поведению. С документацией на код очень часто бывает много проблем: ее могут забыть актуализировать; ее может писать отнюдь не разработчик, и в результате — как работать с кодом, совсем не понятно; в конце концов, она может содержать ошибки, которые никто не заметил. Тесты на код, конечно, не могут заменить документацию (особенно хорошую), но, тем не менее, они неплохо ее дополняют. К тому же тесты обычно всегда актуализированы относительно кода, их пишут разработчики (во многих случаях это тот же человек, который писал код) и в них меньше вероятность появления ошибки, так как при наличии ошибок код, скорее всего, не будет удовлетворять проверкам в тестах, что будет поводом к проверке самих тестов.

Рассмотрев плюсы написания тестов, определимся с ценой, которую мы вынуждены платить при таком подходе к разработке.

Во-первых, при таком подходе время разработки увеличивается в несколько раз (обычно в 2–3 раза). Это связано с тем, что сложность и объем кода тестов обычно сопоставимы со сложностью и объемом соответствующего им основного кода. Учтем, что при таком подходе в несколько раз увеличивается и общий объем кода, несколько усложняется сопровождение и дальнейшую модификацию кода. Поэтому, когда есть потребность очень быстро получить код (например, по пилотному проекту или макету приложения), тестов совсем не пишут либо пишут их по минимуму (например, общий интеграционный тест на все приложение).

Во-вторых, так как время жизни тестов сопоставимо со временем жизни кода, то к написанию тестов следует подходить точно так же, как и к написанию кода. Это означает, что нужно продумывать их архитектуру, возможное повторное использование каких-то частей тестов, избегать дублирования кода и т.п. Что

опять же увеличивает время, необходимое на написание тестов существующего кода.

В-третьих, написание хороших тестов, покрывающих все возможные сценарии выполнения кода со всеми возможными вариантами данных, достаточно сложно и нетривиально.

И, наконец, в-четвертых — это человеческий фактор. Как показывает практика, для многих разработчиков написание тестов к своему коду гораздо менее интересно, чем написание самого кода. Поэтому бывают ситуации, когда разработчики относятся к написанию тестов формально и спустя рукава. Понятно, что ценность тестов, созданных при таком отношении, сильно ниже, чем могла бы быть. Еще вариант, связанный с человеческим фактором — это когда написанием тестов к коду занимается специальный человек (часто — новичок в разработке), а не разработчик, который пишет сам код. В итоге тесты получаются не очень хорошего качества и, возможно, с неполным покрытием кода. Связано это с тем, что для написания качественных тестов необходимо понимать исходный код, а стороннему человеку обычно сложнее вникнуть в код, нежели автору. Следует сказать, что подход с отдельным разработчиком для написания тестов на приложениеывает и оправданным: например, для написания интеграционных тестов или функциональных тестов на подсистемы.

Также следует сказать, что в некоторых ситуациях без написания тестов на существующий код никак не обойтись. Представьте такую ситуацию, что вашим продуктом является библиотека (причем неважно, для внутреннего использования или для внешнего мира) с некоторой реализованной функциональностью. Как ее можно протестировать без написания кода (в данном случае, тестов), только по набору пользовательских сценариев? Ответ очевиден: никак. Поэтому для тестирования такого продукта нам придется написать некоторый объем кода. При написании тестов нам понадобятся сценарии использования этой библиотеки. Только, в отличие от пользовательских сценариев, это будут не действия пользователей, а набор сценариев для выполнения всех возможных типичных задач.

Мы оценили плюсы и минусы написания тестов при разработке кода (надеюсь, что плюсы для вас перевесили минусы, и вы решили, что будете писать тесты на код, если уже не делаете этого). Рассмотрим более пристально, какие бывают виды тестов. Начнем мы с наиболее, пожалуй, популярного вида тестов, которые пишут разработчики: модульных тестов (они же unit-тесты). Это тесты для проверки отдельных единиц кода (процедур, функций, классов и т.д.), причем при проверке мы можем работать с проверяемыми единицами кода как с «белым ящиком» (об этом ниже). Модульные тесты никак не должны зависеть от окружения; им достаточно для работы только исходной единицы кода (и дополнительных библиотек для создания самих тестов). Так, например, если для успешного выполнения модульных тестов необходимо существование некоторого файла в некоторым содержимым в некоторой директории, то такой тест не может считаться модульным. Также, модульные тесты должны работать независимо друг от друга. Это означает, что результат выполнения любого теста должен быть всегда одним и тем же, вне зависимости от того как мы запускаем этот тест: одного или вместе »

## Разработка через тестирование

Разработка через тестирование [test-driven development, TDD] — это подход к разработке программного обеспечения, при котором сначала пишутся тесты, а потом уже код, который эти тесты проверяет. Весь созданный код (практически весь) при этом автоматически оказывается покрыт тестами.

При этом тесты определяют требования к коду и его дизайну. При таком подходе разработка основывается на повторении очень коротких циклов, состоящих из создания тестов, написания кода, удовлетворяющего этим тестам, и рефакторинга тестов и кода. Следует заметить, что для покрытия всего кода

тестами не обязательно разрабатывать программное обеспечение согласно этому подходу. Вполне можно сначала разработать код, а потом создать для него тесты; но нужно быть готовым к тому, что для создания тестов (в особенности модульных) уже написанный код необходимо будет изменить.



## Непрерывная интеграция

Непрерывная интеграция [continuous integration] — это подход к разработке программного обеспечения, при котором сборка проекта осуществляется периодически, в автоматическом режиме (обычно при помощи специальных средств, таких как Hudson, Jenkins и т.д.). При этом обычно выполняются все существующие тесты в проекте. Если во время процесса сборки возникли какие-либо проблемы или какие-то из тестов выполнились с ошибкой, это является сигналом, что с проектом что-то не в порядке. Поскольку сборка

проекта осуществляется периодически (обычно каждую ночь), то о проблемах со сборкой разработчики узнают практически сразу же (и сразу же все эти проблемы правят), а не на финальной стадии разработки проекта (когда необходимо собрать проект для заказчика/конечного пользователя). Еще одним большим плюсом такого подхода является тот факт, что QA-специалисты могут приступать к проверке новой функциональности или исправлений ошибок сразу же, как только разработчик завершит свой труд.

с другими тестами, а также вне зависимости от порядка выполнения разных тестов. Такое требование также может привести к тому, что возникнет необходимость в изменении кода, что приведет, в конечном итоге, к уменьшению связности внутри основного кода. Наиболее правильный подход при написании модульных тестов — это если такие тесты пишут разработчики, которые и создали тестируемый код. Одно большое исключение из этого правила — это работа с унаследованным кодом, в котором не было тестов. В этом случае разработчик, прежде чем решать некоторые задачи, пишет модульные тесты (возможно, и не только их) к уже существующему коду, созданному его предшественником.

Следующий вид тестов, о которых мы поговорим — это функциональные тесты, т. е. такие, которые проверяют некоторую функциональность всей системы или одной из ее подсистем (или некоторого компонента). При тестировании функциональности мы работаем со всей системой или некоторой подсистемой как с «черным ящиком» (об этом ниже).

Функциональные тесты работают с более крупными объектами, чем модульные. Поэтому создание функциональных тестов часто проще, чем модульных, т. к. мы работаем с абстракциями более высокого уровня. Но с другой стороны, достаточно часто для выполнения функциональных тестов необходимо сделать дополнительные действия по инициализации и очистке. Например, если мы тестируем подсистему работы с базой данных, то перед тестами необходимо эту базу создать, создать ее структуру и заполнить ее данными. Как и в случае модульных тестов, мы, конечно, желаем, чтобы результат выполнения каждого теста не зависел от результатов выполнения других тестов. Для достижения этого результата нам необходимо либо чистить базу перед выполнением каждого теста, либо заново создавать ее. Очевидно, что это усложняет инициализацию тестов и увеличивает время, потребное для их инициализации. А это приводит к увеличению времени прогона всех тестов. Поэтому модульным и функциональным тестам нужны разные сценарии запуска.

Обычно разработчик при работе над кодом периодически запускает на выполнение только модульные тесты (особенно перед сохранением результатов работы в системе контроля версий). А функциональные тесты (впрочем, как и модульные), обычно запускаются на выполнение на некотором выделенном сервере. За написание функциональных тестов не обязательно должен отвечать тот же человек, что и писал код. Часто бывает, что тестируемая подсистема или система настолько велика, что ее разрабатывают несколько программистов. При этом написание функциональных тестов может занимать достаточно большое количество времени. В такой ситуации вполне логично, чтобы за написание функциональных тестов отвечал отдельный разработчик (часто с более низкой квалификацией, по сравнению

с разработчиками основного кода). Хотя для небольших подсистем или систем такие тесты вполне может писать и разработчик данного программного обеспечения (или же в случае, когда с подсистемами работает всего один разработчик).

Из функциональных тестов можно выделить несколько подвидов тестов, к которым нужен особый подход. Начнем с интеграционных тестов. Это тесты на всю систему: при этом мы все подсистемы интегрируем в одну итоговую систему (как она будет выглядеть при реальном использовании). Очевидно, что развернуть такую систему перед тестами достаточно сложно. Если итоговая система обладает развитым пользовательским интерфейсом (на основе Web или некоторого оконного интерфейса), то ее крайне сложно протестировать без специальных средств. Например, для тестирования пользовательского интерфейса на основе Web применяются такие средства, как *Selenium* (см. <http://docs.seleniumhq.org/>). Эти средства эмулируют действия пользователя при работе с интерфейсом, что позволяет писать интеграционные тесты прямо на основе пользовательских сценариев использования системы. Причем такие тесты очень чувствительны даже к небольшим (иногда и незаметным для пользователя) изменениям в пользовательском интерфейсе: мелкое изменение способно привести к тому, что практически все тесты перестают выполняться. Если итоговая система не обладает развитым пользовательским интерфейсом, то ее тестирование гораздо проще и часто не требует специальных средств (как, например, в случае REST-сервисов). Но усилия на развертывание все равно придется затратить (а также усилия по обеспечению независимости результатов выполнения тестов).

Следующий вид тестов, который можно выделить — это тесты на производительность, нагрузочные и стресс-тесты. Все они служат для проверки работы системы и отдельных подсистем под нагрузкой. Тесты на производительность служат для измерения времени выполнения, как отдельных фрагментов кода, так и целых подсистем. Нагрузочные тесты служат для тестирования системы (и, иногда, некоторых подсистем) под ожидаемой рабочей нагрузкой (т. е. под нагрузкой, которая описана в требованиях к проекту). Стресс-тесты служат для тестирования системы (и, иногда, некоторых подсистем) под высокой и даже пиковой нагрузкой. Все эти тесты объединяет их повышенная сложность (заключающаяся как в сложности развертывания окружения, так и в сложности самих тестов) и достаточно большое время выполнения.

А теперь взглянем более пристально на процесс написания тестов и на принципы, лежащие в основе этого процесса. Начнем с подхода к тестированию системы как «черного ящика». Как уже говорилось, данный подход используется при создании функциональных тестов на систему или на отдельные ее подсистемы.

Когда мы говорим о некоторой системе как о «черном ящике», это означает, что мы ничего не знаем о внутреннем устройстве данной системы. Все, что мы можем сделать с «черным ящиком» — это подать на вход одни данные и на выходе получить другие (здесь под входом и выходом мы понимаем любые точки для ввода и вывода данных: файлы, базы данных, некоторый набор API, логи, консоль и т. п.). При этом не гарантируется идемпотентность данной операции: т. е. если мы одни и те же данные подадим на вход два раза, то не факт, что во второй раз на выходе получим то же самое, что и в первый раз. В большинстве случаев, тестируемая нами подсистема является некоторым конечным автоматом: другими словами, у нее есть набор состояний, и при каждом взаимодействии с ней подсистема переходит из одного состояния в другое. Все возможные состояния и все возможные переходы мы можем получить из требований, ТЗ и прочей подобной документации к подсистеме (данная документация вполне может быть в некотором неформализованном виде).

Очевидно, что в итоге наши функциональные тесты (тестирование с точки зрения «черного ящика») будут выглядеть

следующим образом. Мы инициализируем подсистему, чтобы попасть в одно из ее состояний (не обязательно начальное). После этого при помощи одного или нескольких переходов мы переводим систему в некоторое другое состояние (не обязательно заключительное). И, наконец, мы проверяем выходные данные состояния, в которое мы в итоге перешли.

Мы можем использовать подход к тестируемому объекту как к «черному ящику» и при модульном тестировании. Но такой подход далек от оптимального. Во-первых, на отдельные части кода достаточно редко (хотя на некоторые критичные части так делают) пишутся требования и/или нечто подобное. Соответственно, это приводит к тому, что построить конечный автомат становится не так уж и просто. Во-вторых, мы никак не пользуемся тем фактом, что у нас есть доступ к исходному коду тестируемого фрагмента. Поэтому создание модульных тестов на основе некоторых знаний об исходном коде тестируемых фрагментов является оптимальным.

Можно сразу же предложить очень прямолинейный подход для создания необходимого набора тестов: для тестируемого фрагмента взять все его входные параметры, построить все возможные комбинации допустимых значений для этих параметров, после чего для каждой такой комбинации написать отдельный тест. Однако стоит учесть следующий факт: существует достаточно мало типов данных, диапазон возможных значений которых конечен и достаточно мал. Так, например, у булевского типа данных всего 2 значения. Но с другой стороны, у строк в большинстве языков программирования множество возможных значений бесконечно, а у 32-разрядного целого числа множество возможных значений лежит в диапазоне от -2,147,483,648 до -2,147,483,647 (всего 4,294,967,296 значений). Поэтому в жизни такой очень прямолинейный подход практически не применим. Конечно, возможны ситуации, когда, например, мы тестируем функцию, которая принимает несколько входных параметров булевского типа; в этом случае мы можем составить все возможные комбинации входных параметров и для каждой комбинации создать свой тест. Но такие ситуации достаточно редки.

Давайте воспользуемся тем фактом, что у нас есть исходный код: при его анализе мы можем выделить все возможные пути выполнения кода (а также все возможные варианты вычисления выражений). Если мы создадим тесты таким образом, чтобы проверялись все эти возможные пути (т.е. каждый тест выполняет код по одному из возможных путей), то этого будет достаточно для проверки тестируемого фрагмента кода. При этом следует внимательно учесть следующий момент: каждому из возможных путей выполнения кода соответствует некоторое подмножество из множества всех возможных комбинаций входных параметров.

Понятно, что мы не можем взять все возможные комбинации входных параметров из такого подмножества. Поэтому обычно

поступают следующим образом. Обязательно берут некоторые осмысленные значения (например, для строк подключения к базе данных), берут граничные значения, если они существуют, и берут несколько любых произвольных значений из подмножества значений для входного параметра. После чего строят все возможные комбинации выбранных таким образом значений для всех входных параметров и проверяют в рамках одного теста для какого-либо пути выполнения исходного кода.

Но при написании модульных тестов может возникнуть следующая проблема: взаимодействие тестируемого кода с внешними подсистемами, такими как база данных, файловая подсистема и т.д. Если мы попробуем написать на такой код модульные тесты «в лоб», то, очевидно, ничего не получится. Причина в том, что для работы таких тестов необходимо сделать некоторую инициализацию внешних подсистем, от которых тестируемый код зависит. Как только мы добавим такую инициализацию в наши тесты, так сразу же наши тесты перестанут быть модульными.

Возникает известный русский вопрос: что делать? Ответ очевиден: наш код (который мы хотим тестиировать) имеет взаимосвязь с конкретной внешней подсистемой, т.е. у нашего кода и у внешней подсистемы сильная связность. И первый шаг, который необходимо сделать — это разорвать такую взаимосвязь (уменьшить связность нашего кода). Для этого в коде связь с конкретной внешней подсистемой мы заменяем связью с некоторым интерфейсом, реализующим необходимые нам операции. Для работы после такой модификации необходимо передавать конкретную реализацию для введенного интерфейса либо при инициализации кода, либо при его использовании. Такая модификация, помимо уменьшения связности, дает нам возможность заменить одну конкретную внешнюю подсистему на другую (например, мы можем заменить работу с файловой системой на работу с REST-сервисом).

Второй шаг, который необходимо сделать для написания модульных тестов — это создать некоторый объект, имитирующий некоторую реальную внешнюю подсистему (тест-объект). Этот объект не будет требовать инициализации и может проверять порядок вызова своих методов и передаваемые параметры. Таким образом мы, помимо создания заглушки для реальной внешней подсистемы, можем тестируировать то, как наш код взаимодействует с внешней подсистемой (т.е. поведение нашего кода). Но об этом мы поговорим очень подробно в одной из следующих статей.

В этом месяце мы рассмотрели концептуальное представление о том, что такое тестирование кода разработчиком. Мы узнали о принципах и подходах, о проблемах и их решении, о преимуществах и цене при написании тестов на код. Все, что было здесь сказано, применимо к любому языку программирования, любой платформе и операционной системе. О конкретике, связанной с языком Erlang, мы поговорим в следующий раз. [LXF](#)

## Скрытые зависимости и тесты

При создании тестов мы желаем, чтобы результат выполнения каждого теста никак не зависел от результата выполнения других тестов. Если результат выполнения тестов зависит от порядка их выполнения, это делает подобные тесты, по большому счету, бесполезными. Когда подобные зависимости между тестами возникают, необходимо найти причину этой зависимости и ее устраниить. Модульные тесты не зависят от внешнего окружения (файловой системы, баз данных и т.п.), поэтому зависимость между модульными тестами говорит о том, что между вызовами тестов сохраняется состояние

(например, в глобальной переменной или в статическом поле класса). Это говорит о высокой связности внутри кода.

Если проблема в коде теста, то достаточно переписать такие тесты. Если же проблема в тестируемом коде, то, возможно, потребуется переработка архитектуры кода, иногда затрагивающая весь проект. С другой стороны, выявление и устранение подобной зависимости делает код проще и снижает вероятность возникновения «side-эффектов». Функциональные тесты, в отличие от модульных тестов, зависят от окружения (они специально создаются

так, чтобы проверить работу подсистем в реальном окружении). Поэтому между такими тестами возможны зависимости и через код (как в случае с модульными тестами), и через данные (через внешнее окружение). В случае зависимости тестов через код мы поступаем точно так же, как и в случае модульных тестов. Зависимость тестов через данные говорит нам о том, что при инициализации внешних подсистем не все было инициализировано (или почищено). Это означает, что необходима доработка кода инициализации (или очистки) для функциональных тестов.