**Автоматизированное Юнит и Интеграционное тестирование Java приложений с использованием Junit 5 И Mockito**

**Тестирование: ручное или автоматизированное?**

Сначала надо понять различия между ручными и автоматизированными тестами. Ручное тестирование проводится непосредственно человеком, который нажимает на кнопочки в приложении или взаимодействует с программным обеспечением или API с необходимым инструментарием. Это достаточно затратно, так как это требует от тестировщика установки среды разработки и выполнения тестов вручную. Имеет место вероятность ошибки за счет человеческого фактора, например опечатки или пропуска шагов в тестовом сценарии.

Автоматизированные тесты, с другой стороны, производятся машиной, которая запускает тестовый сценарий, который был написан заранее. Такие тесты могут сильно варьироваться в зависимости от сложности, начиная от проверки одного единственного метода в классе до отработки последовательности сложных действий в UI, чтобы убедиться в правильности работы. Такой способ считается более надежным, однако его работоспособность все еще зависит от того насколько скрипт для тестирования был хорошо написан.

Автоматизированные тесты – это ключевой компонент непрерывной интеграции ([Continuous Integration](https://www.atlassian.com/continuous-delivery/continuous-integration/how-to-get-to-continuous-integration)) и непрерывной доставки ([continuous delivery](http://https/www.atlassian.com/continuous-delivery/pipeline)), а также хороший способ масштабировать ваш QA процесс во время добавления нового функционала для вашего приложения.

Во время ручного тестирования мы подготавливаем окружение (зависимости), запускаем приложение, вводим тестовые данные, проверяем работоспособность, вводим другие тестовые данные и так пока не проверим все возможные варианты входов и выходов, разумеется, если мы хотим, чтобы наше приложение работало. Все это занимает очень много времени и чем больше становиться приложение, тем сложнее выполнять этот процесс. В результате, программист практически перестает заниматься проверкой функциональности, особенно старой, что может привести к неожиданному поведению программы, а возможно даже к критическим для бизнеса ошибкам и потерям денег и времени. Похожий сценарий может произойти с автоматизированными тестами плохого качества, но об этом позже.

**Цель юнит-тестирования**

Главная цель юнит-тестирования – обеспечение стабильного роста программного проекта. В начале жизненного цикла проекта, его развивать довольно просто, но намного сложнее поддерживать развитие с прошествием времени

На рис. 1.1 изображена динамика роста типичного проекта без тестов. Все начинается быстро, потому что ничего вас не тормозит. Еще не приняты неудачные архитектурные решения; еще нет существующего кода, который необходимо прорабатывать и поддерживать. Однако с течением времени вам приходится тратить все больше времени, чтобы написать тот же по объему функционал, что и в начале проекта. Со временем скорость разработки существенно замедляется — иногда даже до состояния, в котором проект вообще перестает двигаться вперед. (Рисунок)

Если не принять должных мер (например, постоянной чистки и рефакторинга), код постепенно усложняется и дезорганизуется. Исправление одной ошибки приводит к появлению новых ошибок, а изменение в одной части проекта нарушает работоспособность в нескольких других — возникает своего рода «эффект домино». Со временем код становится ненадежным. И что еще хуже, его становится все труднее вернуть в стабильное состояние. Тесты помогают справиться с этой тенденцией. Они становятся своего рода «подушкой безопасности» — средством, которое обеспечивает защиту против большинства регрессий. Тесты помогают удостовериться в том, что существующая функциональность работает даже после разработки новой функциональности или рефакторинга кода.

Недостаток юнит-тестирования заключается в том, что тесты требуют начальных вложений, и иногда весьма значительных. Но в долгосрочной перспективе они окупаются, позволяя проекту расти на более поздних стадиях. Разработка большинства нетривиального программного обеспечения без помощи тестов практически невозможна.

**СВЯЗЬ МЕЖДУ ЮНИТ-ТЕСТИРОВАНИЕМ И СТРУКТУРОЙ КОДА**

Сама возможность покрытия кода тестами — хороший критерий определения качества этого кода, но он работает только в одном направлении. Это хороший негативный признак — он выявляет низкокачественный код с относительно высокой точностью. Если вдруг обнаружится, что код трудно протестировать, это верный признак того, что код нуждается в улучшении. Плохое качество обычно проявляется в сильной связности (tight coupling) кода; это означает, что части кода недостаточно четко изолированы друг от друга, что в свою очередь создает сложности с их раздельным тестированием. Но, к сожалению, возможность покрытия кода тестами является плохим позитивным признаком. Тот факт, что код проекта легко тестируется, еще не означает, что этот код написан хорошо. Качество кода может быть плохим даже в том случае, если он не страдает сильной связностью.

**Определение юнит-теста**

Юнит-тестом называется автоматизированный тест, который:

* проверяет правильность работы небольшого фрагмента кода (также называемого юнитом);
* делает это быстро;
* и поддерживая изоляцию от другого кода;

Неоднозначность может вызывать первый и третий пункт. Вопрос изоляции и вопрос того, что является юнитом - это основное различие между лондонской и классической школой юнит-тестирования.

**Структура юнит-теста**

Юнит тест состоит из 3ех частей: Arrange – подготовка, Act – действие, Assert – проверка. Этот паттерн называется AAA или 3A.

Пример

В секции подготовки тестируемая система (system under test, SUT) и ее зависимости приводятся в нужное состояние; Секция подготовки самая большая.

В секции действия вызываются методы SUT, передаются подготовленные зависимости и сохраняется выходное значение (если оно есть); Секция действия должна состоять из одной строки. Если она состоит из двух и более, это может указывать на проблемы с API тестируемой системы. Иногда это правило можно нарушить, но это относится не к бизнес-логике, а к служебному или инфраструктурному.

В секции проверки проверяется результат, который может быть представлен возвращаемым значением, итоговым состоянием тестируемой системы и ее коллабораторов или методами, которые тестируемая система вызывает у этих коллабораторов.

**Лондонская школа**

Подход лондонской школы подразумевает изоляцию тестируемого кода, от его зависимостей (коллабораторов). То есть все зависимости класса нужно заменить на тестовые заглушки. Тестовая заглушка – объект, который имеет такой же интерфейс, как и заменяемый, но ведет себя так, как нам нужно для тестирования. Разновидности и создание тестовых заглушек рассмотрено позже. Под юнитом в лондонской школе подразумевается один класс или методов.

**Классический подход**

В классическом подходе изолируются друг от друга не классы, а тесты. То есть последовательность выполнения не влияет на результат. Тестируемые классы не должны обращаться к общему состоянию, через которое они могу влиять друг на друга. Примерами таких зависимостей являются БД и файловая система.

При использовании классического подхода юнит не обязан ограничиваться классом.

**Итог**

В лондонской школе, происходит изоляция юнитов, юнит – это класс и заглушки используются для коллабораторов. Допускается не использовать заглушки для объектов без изменяемого состояния (объектов-значений)

В классической школе, изолируют юнит-тесты, юнит – это класс или набор классов, а заглушки используются для совместных зависимостей.

Картинки

Лондонская школа обладает следующими преимуществами:

* Улучшенная детализация. Тесты высоко детализированы и проверяют только один класс за раз.
* Упрощение юнит-тестирования большого графа взаимосвязанных классов. Так как все коллабораторы заменяются тестовыми заглушками, вам не придется беспокоиться о них при написании теста.
* Если тест падает, вы точно знаете, в какой функциональности произошел сбой. Так как все коллабораторы заменены на заглушки, не может быть других подозреваемых, кроме самого тестируемого класса. Конечно, все еще возможны ситуации, в которых тестируемая система использует объект-значение, и изменение в этом объекте-значении приводит к падению теста. Однако такие случаи встречаются не так часто, потому что все остальные зависимости устранены в тестах.

Остаются еще два различия между классической и лондонской школами:

* подход к проектированию системы на базе методологии разработки через тестирование (TDD, Test-Driven Development);
* проблема излишней спецификации (over-specification);

Лондонский стиль юнит-тестирования ведет к методологии TDD по схеме «снаружи внутрь» (outside-in): вы начинаете с тестов более высокого уровня, которые задают ожидания для всей системы. Используя заглушки, вы указываете, с какими коллабораторами система должна взаимодействовать для достижения ожидаемого результата. Затем вы проходите по графу классов, пока не реализуете их все. Заглушки делают этот процесс разработки возможным, потому что вы можете сосредоточиться на одном классе за раз. Вы можете отсечь всех коллабораторов тестируемой системы и такимобразом отложить реализацию этих коллабораторов.

Классическая школа такой возможности не дает, потому что вам приходится иметь дело с реальными объектами в тестах. Вместо этого обычно используется подход по схеме «изнутри наружу» (inside-out). В этом стиле вы начинаете с модели предметной области, а затем накладываете на нее дополнительные слои, пока программный код не станет пригодным для конечного пользователя. Но, пожалуй, самое принципиальное различие между школами — проблема излишней спецификации (over-specification), то есть привязки теста к деталям имплементации тестируемой системы. Лондонский стиль приводит к тестам, завязанным на детали имплементации, чаще, чем классический стиль. И это становится главным аргументом против повсеместного использования заглушек и лондонского стиля в целом.

Примеры юнит тестов с лондонским и класс

**Аспекты хорошего юнит-теста**

Хороший юнит-тест должен обладать следующими тремя атрибутами:

* защита от багов;
* устойчивость к рефакторингу;
* быстрая обратная связь;

Они могут использоваться для анализа любых автоматизированных тестов, будь то юнит-, интеграционные или сквозные(end-to-end) тесты. В каждом тесте до той или иной степени проявляется каждый из атрибутов.

Первый аспект: защита от багов. Начнем с первого атрибута хорошего юнит-теста: защиты от багов. Как правило, такие ошибки возникают после внесения изменений в код — обычно после написания новой функциональности.

Чтобы оценить, насколько хорошо тест проявляет себя в отношении защиты от багов, необходимо принять во внимание, следующее:

* объем кода, выполняемого тестом;
* сложность этого кода;
* важность этого кода с точки зрения бизнес-логики.

Как правило, чем больше кода тест выполняет, тем выше вероятность выявить в нем баг (если, конечно, он там есть). Само собой, тест также должен иметь актуальный набор проверок, просто выполнить код недостаточно. Важен не только объем кода, но и его сложность и важность с точки зрения бизнес-логики. Код, содержащий сложную бизнес-логику, важнее инфраструктурного кода —ошибки в критичной для бизнеса функциональности наносят наибольший ущерб. Как следствие, тестирование тривиального кода обычно не имеет смысла. Этот код слишком простой и не содержит сколько-нибудь значительного объема бизнес-логики. В тестах, покрывающих тривиальный код, вероятность нахождения ошибок невелика.

Примером тривиального кода служит однострочное свойство следующего вида: (Геттеры, сеттеры)

Так же помимо вашего кода должны учитываться библиотеки и фреймворки. Для обеспечения оптимальной защиты тест должен проверять, как ваш код работает в комбинации с этими библиотеками, фреймворками и внешними системами.

Второй аспект: устойчивость к рефакторингу. Эта устойчивость определяет, насколько хорошо тест может пережить рефакторинг тестируемого им кода без выдачи ошибок. Например, вы провели рефакторинг нескольких участков кода, все выглядит лучше, чем прежде, но тесты не проходят. Новая функциональность работает так же хорошо, как и прежде. Это тесты были написаны так, что они падают при любом изменении тестируемого кода. И это происходит независимо от того, внесли вы ошибку в этот код или нет. Такая ситуация называется ложным срабатыванием. Это ложный сигнал тревоги: тест показывает, что функциональность не работает, тогда как в действительности все работает как положено. Такие ложные срабатывания обычно происходят при рефакторинге кода, когда вы изменяете имплементацию, но оставляете поведение приложения без изменений.

Почему столько внимания уделяется ложным срабатываниям? Потому что они могути меть серьезные последствия для всего приложения. Как говорилось ранее, целью юнит-тестирования является обеспечение устойчивого роста проекта. Устойчивый рост становится возможным благодаря тому, что тесты позволяют добавлять новую функциональность и проводить регулярный рефакторинг без внесения ошибок в код. Здесь имеются два конкретных преимущества:

* Тесты становятся системой раннего предупреждения при поломке существующей функциональности. Благодаря таким ранним предупреждениям вы можете устранить ошибки задолго до того, как ошибочный код будет развернут, где исправление ошибок потребует значительно больших усилий.
* Вы получаете уверенность в том, что изменения в вашем коде не приведут к багам. Без такой уверенности вы будете проводить гораздо меньше рефакторинга, что в свою очередь приведет к постепенному ухудшению качества кода проекта.

Ложные срабатывания негативно влияют на оба эти преимущества:

* Если тесты падают без веской причины, они притупляют вашу готовность реагировать на проблемы в коде. Со временем вы привыкаете к таким сбоям и перестаете обращать на них внимание. А это может привести к игнорированию настоящих ошибок.
* С другой стороны, при частых ложных срабатываниях вы начинаете все меньше и меньше доверять вашим тестам. Они уже не воспринимаются как что-то, на что вы можете положиться. Отсутствие доверия приводит к уменьшению рефакторинга, так как вы пытаетесь свести к минимуму потенциальные ошибки. Эта история типична для большинства проектов с хрупкими тестами. Сначала разработчики серьезно относятся к падениям тестов и стараются их починить. Через какое-то время они устают от того, что тесты постоянно поднимают тревогу, и все чаще игнорируют их. Рано или поздно наступает момент, когда в работу попадают настоящие ошибки, так как разработчики проигнорировали их вместе сложными срабатываниям.

Количество ложных срабатываний, выданных тестом, напрямую связано со структурой этого теста. Чем сильнее тест связан с деталями имплементации тестируемой системы (system under test, SUT), тем больше ложных срабатываний он порождает. Уменьшить количество ложных срабатываний можно только одним способом: отвязав тест от деталей имплементации тестируемой системы. Тест должен проверять конечный результат — наблюдаемое поведение тестируемой системы, а не действия, которые она совершает для достижения этого результата. Тесты должны подходить к проверке SUT с точки зрения конечного пользователя и проверять только результат, имеющий смысл для этого пользователя.

**Картинка и возможно пример**

****

**Доделать вверху**

Третий аспект: быстрая обратная связь. Быстрая обратная связь является одним из важнейших свойств юнит-теста. Чем быстрее работают тесты, тем больше их можно включить в проект и тем чаще вы их сможете запускать. Быстро выполняемые тесты сильно ускоряют обратную связь. В идеальном случае тесты начинают предупреждать вас об ошибках сразу же после их внесения, в результате чего затраты на исправление этих ошибок уменьшаются почти до нуля. С другой стороны, медленные тесты увеличивают время, в течение которого ошибки остаются необнаруженными, что приводит к увеличению затрат на их исправление. Дело в том, что медленные тесты отбивают у разработчика желание часто запускать их, поэтому в итоге он тратит больше времени, двигаясь в ошибочном направлении.

К сожалению, создать идеальный тест невозможно. Дело в том, три атрибута являются взаимоисключающими. Невозможно довести их до максимума одновременно: одним из трех придется пожертвовать для максимизации двух остальных. Более того выдержать баланс еще сложнее. Нельзя просто обнулить один атрибут, чтобы сосредоточиться на остальных. Тест с нулевым значением в одной из трех категорий бесполезен. Следовательно, атрибуты нужно максимизировать так, чтобы ни один из них не падал слишком низко.

При максимизации двух атрибутов за счет третьего, возникает следующая картина

****

**Перерисовать**

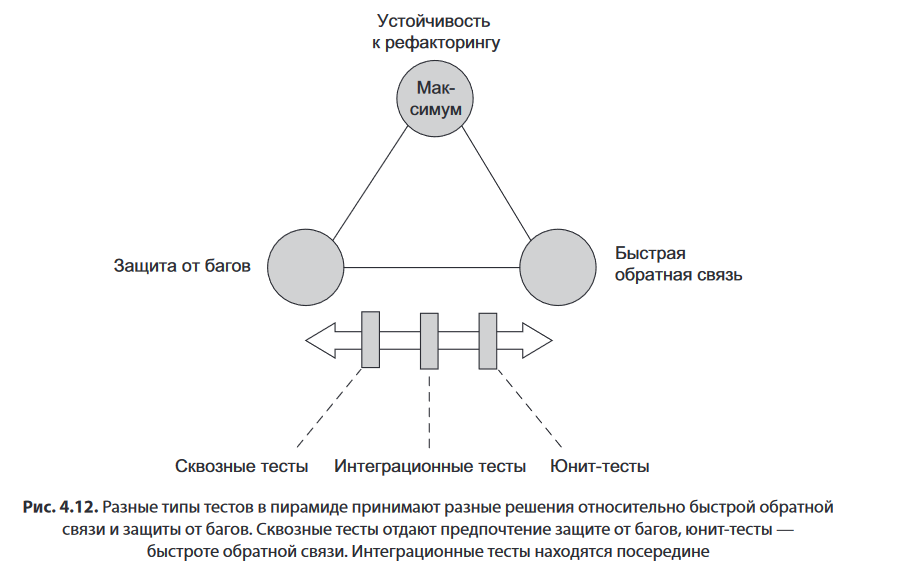
Сквозные тесты рассматривают систему с точки зрения конечного пользователя. Они обычно проходят через все компоненты системы, включая пользовательский интерфейс, базу данных и внешние приложения.Тем не менее наряду с преимуществами у сквозных тестов имеется крупный недостаток: они очень медленные.

В отличие от сквозных тестов, тривиальные тесты предоставляют быструю обратную связь. Кроме того, вероятность ложных срабатываний также мала, поэтому они обладают хорошей устойчивостью к рефакторингу. Тем не менее тривиальные тесты вряд ли смогут выявить какие-либо ошибки, потому что покрываемый ими код слишком прост. (сеттеры)

Также достаточно легко написать тест, который работает быстро и хорошо выявляет ошибки в коде, но делает это с большим количеством ложных срабатываний. Такие тесты называются хрупкими: они падают при любом рефакторинге тестируемого кода независимо от того, изменилась тестируемая ими функциональность или нет.

**Результаты поиска хорошего юнит-теста**

Выдержать баланс между атрибутами хорошего теста сложно. Тест не может иметь максимальных значений в каждой из первых трех категорий; а значит, придется идти на компромиссы. Рекомендуется максимизировать устойчивость к рефакторингу, так как тест либо устойчив, либо нет, а другие атрибуты можно варьировать, получив три вида тестов:

****

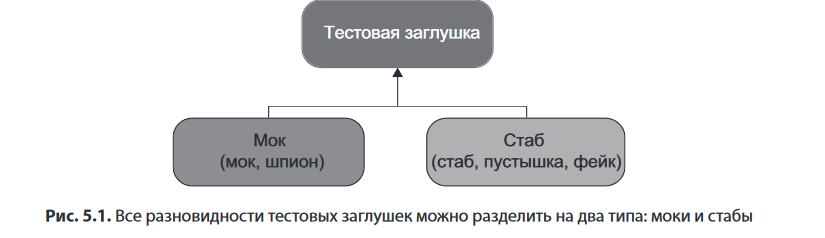
Ни один из уровней не делает устойчивость к рефакторингу предметом для компромисса.

****

Как правило в приложении должно поддерживаться отношение, как на рисунке сверху. Однако, оно справедливо не для всех. Если ваше приложение является CRUD скорее пирамида будет напоминать прямоугольник с юнит и интеграционными тестами Юнит-тесты менее полезны в ситуациях, в которых отсутствует алгоритмическая или бизнес сложность, — они быстро вырождаются в тривиальные тесты. В то же время интеграционные тесты полезны даже в таких случаях; каким бы простым код ни был, важно проверить, как он работает в интеграции с другими подсистемами (например, базой данных). В результате в CRUD-приложениях у вас будет меньше юнит-тестов и больше интеграционных. В самых тривиальных случаях интеграционных тестов может быть даже больше, чем юнит-тестов.

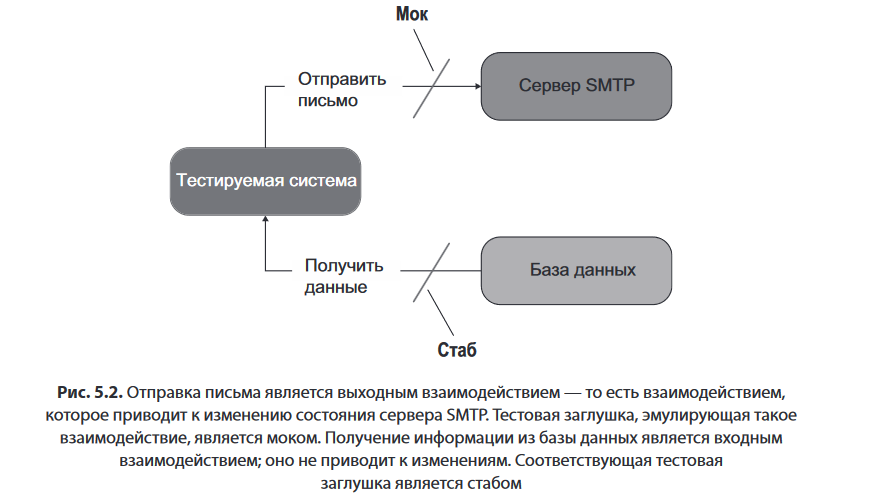
**Тестовые заглушки.**

Тестовая заглушка (test double) — общий термин, который описывает все разновид-ности фиктивных зависимостей в тестах. В действительностивсе разновидности можно разделить на два типа: моки и стабы



Различия между двумя типами сводятся к следующему:

* Моки помогают эмулировать и проверять выходные взаимодействия — то естьвызовы, совершаемые тестируемой системой к ее зависимостям для изменения их состояния.
* Стабы помогают эмулировать входные взаимодействия — то есть вызовы, совершаемые тестируемой системой к ее зависимостям для получения входных данных.



Примеры

**Список использованных источников:**

1) jUnit5 About: [Электронный ресурс]. URL: https://junit.org/junit5/ .2) jUnit5 JavaDocs: [Электронный ресурс]. URL:https://junit.org/junit5/docs/current/api/ .

3) jUnit5 исходный код [Электронный ресурс]. URL:https://github.com/junit-team/junit5 .

4) Mockito framework site [Электронный ресурс]. URL: <https://site.mockito.org/>.

5) Mockito исходный код [Электронный ресурс].URL: <https://github.com/mockito/mockito>.

6) Mockito JavaDocs [Электронный ресурс]. URL: <https://javadoc.io/doc/org.mockito/mockito-core/latest/org/mockito/Mockito.html>.

7) Статья: “Mockito и как его готовить” [Электронный ресурс]. URL: <https://habr.com/ru/post/444982/>.

8) Mockito Tutorial Baeldung [Электронный ресурс]. URL: https://www.baeldung.com/mockito-series

9) A Guide to Junit5 [Электронный ресурс]. URL: <https://www.baeldung.com/junit-5>.

10) Parallel Test Execution for Junit5 [Электронный ресурс]. URL: <https://www.baeldung.com/junit-5-parallel-tests>.

11) Эккель Б. Философия Java, 4-e изд. – Спб.: Питер, 2021. – 1168c.

12) Бек К. Экстремальное программирование: разработка через тестирование. – Спб.: Питер, 2022. – 224с.

13) Хориков В. Принципы юнит-тестирования. – Спб.: Питер, 2022. – 320с.

14) Mellor A. Test-Driven Development with Java. – Birmingham: Packt, 2023. – 325с.

15) Freeman S., Pryce N. Growing Object-Oriented Software, Guided by Tests. – Boston: Addison-Wesley, 2010. – 385с.

16) Мартин Р. Чистый код: создание, анализ и рефакторинг. – Спб.: Питер, 2022. – 464с

https://habr.com/ru/companies/otus/articles/443418/