Имитации (mocks) и заглушки (stubs) — это два разных типа тестовых дублёров (вообще их больше). К объектам из продакшна тестовые дублёры создают реализацию для тестов.
Проще говоря, вы заменяете реальную вещь (например, класс, модуль или функцию) поддельной копией. Подделка выглядит и действует как оригинал (даёт такие же ответы на те же вызовы методов), но это заранее установленные ответы, которые вы сами определяете для юнит-теста.

«Заглушки обеспечивают постоянные ответы на вызовы, сделанные во время теста, обычно вообще не реагируя ни на что, кроме того, что запрограммировано для теста. Заглушки могут также записывать информацию о вызовах, такую как заглушка шлюза электронной почты, которая запоминает сообщения, которые она «отправила», или, возможно, только сколько сообщений она «отправила»

Существует несколько видов объектов, которые позволяют симулировать поведение реальных объектов во время тестирования:

1. **Dummy** — пустые объекты, которые передаются в вызываемые методы, но не используются.

Предназначены лишь для заполнения параметров методов.

2. **Fake** — объекты, которые имеют реализации, но в таком виде, который делает их неподходящими для использования в рабочей ситуации.
3. **Stub** — предоставляют заранее заготовленные ответы на вызовы во время теста и не отвечают ни на какие другие вызовы, которые не требуются в тесте.
4. **Mock** — объекты, которые заменяют реальный объект в условиях теста и позволяют проверять вызовы своих методов. Содержат заранее подготовленные описания вызовов, которые они ожидают получить. Применяются в основном для тестирования поведения пользователя.

Тестовые дублёры используются не только в юнит-тестах. Более сложные дублёры применяются для контролируемой имитации целых частей вашей системы. Однако в юнит-тестах используется особенно много имитаций и заглушек (в зависимости от того, предпочитаете вы общительные или одиночные тесты) просто потому что множество современных языков и библиотек позволяют легко и удобно их создавать.

Ваши юнит-тесты будут работать очень быстро.

На приличной машине можно прогнать тысячи модульных тестов за нескольких минут.
Тестируйте изолированно небольшие фрагменты кодовой базы и избегайте контактов с БД, файловой системой и HTTP-запросов (ставя здесь имитации и заглушки), чтобы сохранить высокую скорость.
Поняв основы, со временем вы начнёте всё более свободно и легко писать юнит-тесты. Заглушка внешних участников, настройка входных данных, вызов тестируемого субъекта — и проверка, что возвращаемое значение соответствует ожидаемому. Посмотрите на разработку через тестирование , и пусть юнит-тесты направляют вашу разработку; если они применяются правильно, это поможет попасть в мощный поток и создать хорошую поддерживаемую архитектуру, автоматически выдавая всеобъемлющий и полностью автоматизированный набор тестов.

При юнит-тестировании, поскольку тестируются отдельные модули, необходимо их максимально изолировать, чтобы избежать получения неверных результатов из-за случайного внешнего воздействия. Примером такой ситуации может быть тестирование метода, который обращается к удаленному источнику информации.

В этой ситуации сбой сервера, с которого должны загружаться данные, может привести к несоответствию ожидаемого и действительного результата и мы получим fake-negative результат. Вместо этого мы можем создать объект-заглушку (mock-объект), который будет заменять собой внешний источник данных, возвращая нашему тестируемому методу определенный, строго заданный результат, обеспечивая таким образом предсказуемость результатов работы тестируемого модуля

тупик

Подход с заглушкой прост в использовании и не требует дополнительных зависимостей для модульного теста. Основной метод заключается в реализации соавторов как конкретных классов, которые демонстрируют лишь небольшую часть общего поведения соавторов, которая необходима тестируемому классу. В качестве примера рассмотрим случай, когда реализация службы находится на стадии тестирования

Это экономит нам много времени на поддержку классов-заглушек в виде отдельных объявлений, а также помогает избежать распространенных ошибок реализации заглушек: повторного использования

заглушек в модульных тестах и резкого увеличения количества конкретных заглушек в проекте.

Подход с встроенной заглушкой очень удобен и быстр в реализации, но для большего контроля над тестовым примером и обеспечения того, чтобы при изменении реализации объекта службы тестовый пример также изменялся соответствующим образом, подход с фиктивным объектом лучше.

Макетные объекты

Используя mock-объекты (например, из EasyMock или JMock), мы получаем высокий уровень контроля над тестированием внутренностей реализации тестируемого модуля. Если реализация изменяется, чтобы использовать соавтора по-другому, то модульный тест немедленно дает сбой, сигнализируя разработчику, что его необходимо переписать. В тупиковой реализации тест может провалиться, а может и не провалиться: в случае провала сообщение об ошибке будет загадочным; если бы это было не так, то это было бы только случайно.

Чтобы исправить неудачный тест, мы должны изменить его, чтобы отразить внутреннюю реализацию службы. Постоянная переработка тестовых случаев для отражения внутреннего

устройства реализации рассматривается некоторыми как бремя, но на самом деле это заложено в самой природе модульного тестирования. Мы тестируем реализацию модуля, а не его контракт с остальной системой. Для проверки контракта мы использовали бы интеграционный тест и относились бы к сервису как к черному ящику, определяемому его интерфейсом, а не его реализацией.

Когда использовать заглушки и макеты?

Если фиктивные объекты лучше, зачем нам вообще использовать заглушки. Таким образом, простой ответ: «Делайте то, что подходит для вашего тестового примера, и создайте самый простой код для чтения и поддержки». Если тест с использованием заглушки быстро пишется и читается, и вы не слишком обеспокоены изменениями в соавторе или использованием соавтора внутри тестируемого модуля, тогда это нормально. Обычный случай, когда заглушки легче реализовать (и прочитать), чем макеты, — это когда тестируемому модулю необходимо использовать вызов вложенного метода для соавтора. Мок-версия более надежна . Конечно, истинно верящие в фиктивные объекты укажут, что это ложная экономия, и модульный тест будет более надежным и лучшим в долгосрочной перспективе, чем тест с использованием заглушек.

макеты — это лишь один из ряда «тестовых двойников», важно понимать, что каждый тип тестового двойника играет свою роль в тестировании. Точно так же, как вам нужно изучать различные шаблоны или рефакторинг, вам нужно понимать примитивные роли каждого типа тестового двойника. Затем их можно комбинировать для удовлетворения ваших потребностей в тестировании.

Mockito — это тестовый шпионский фреймворк, и его очень просто освоить. Примечательно, что в Mockito ожидания от любых фиктивных объектов не определяются перед тестом, как это иногда бывает в других фиктивных фреймворках. Это приводит к более естественному стилю когда вы начинаете издеваться.

Фиктивный объект

Это самый простой из всех тестовых двойников. Это объект, не имеющий реализации, который используется исключительно для заполнения аргументов вызовов методов, не имеющих отношения к вашему тесту.

Тестовая заглушка

Роль тестовой заглушки состоит в том, чтобы возвращать контролируемые значения тестируемому

объекту. Они описываются как косвенные входные данные для теста.

Пример диверсанта
Существует 2 распространенных варианта тестовых заглушек: ответчика и диверсанта.

Ответчики используются для проверки счастливого пути, как в предыдущем примере.

Диверсант используется для проверки исключительного поведения, как показано ниже.

Макет объекта

Мок-объекты используются для проверки поведения объекта во время теста. Под поведением объекта я имею в виду, что мы проверяем, что правильные методы и пути применяются к объекту при запуске теста.

Это очень отличается от вспомогательной роли заглушки, которая используется для предоставления результатов всему, что вы тестируете.

В заглушке мы используем шаблон определения возвращаемого значения для метода. В макете мы проверяем поведение объекта.

Тестовый шпион

в Mockito мне нравится использовать его, чтобы позволить вам обернуть реальный объект, а затем проверить или изменить его поведение для поддержки вашего тестирования.

Еще одна полезная функция testSpy — возможность заглушать обратные вызовы. Когда это будет сделано, объект будет вести себя как обычно, пока не будет вызван метод-заглушка.

Поддельный объект

Поддельные предметы обычно представляют собой предметы ручной работы или легкие предметы, используемые только для тестирования и не подходящие для производства. Хорошим примером может быть база данных в памяти или поддельный сервисный уровень.

Они, как правило, предоставляют гораздо больше функциональных возможностей, чем стандартные двойные тесты, и поэтому, вероятно, обычно не являются кандидатами для реализации с использованием Mockito. Это не значит, что они не могут быть созданы как таковые, просто их, вероятно, не стоит реализовывать таким образом.

+ **Обратите** внимание на простоту кода заглушки. Он должен быть легко читаемым, обслуживаемым, НЕ

содержать никакой логики и нуждаться в самостоятельном модульном тестировании. После того, как код заглушки был написан, затем следует модульный тест

Проблемы тестирования интеграции

Вообще, для использования заглушек веб-сервисов может быть ного причин. Например, следующих:

1. Тесты замедляются. Если поставщик сервиса находится далеко, сетевая среда нестабильная и при вызове сервиса происходит задержка, то время прохождения тестов может значительно увеличиваться.
2. Тесты работают нестабильно. Веб-сервисы могут быть не всегда доступны из-за технических обновлений, подвержены перегрузкам или ошибкам сетевых протоколов.
3. Тесты покрывают не все возможные варианты ответов сервиса. Не всегда есть возможность получить некоторые ответы от реального веб-сервиса и промоделировать все рабочие ситуации — следовательно, максимально полно протестировать взаимодействие.
4. Доступ к рабочим веб-сервисам ограничен. Часто встречается ситуация, когда рабочие сервисы недоступны из тестового контура, в

котором программисты ведут разработку. Это делает как по причине безопасности, так и для того, чтобы не подвергать лишней нагрузке рабочее окружение.

5. Из-за ошибок разработчика могут быть подвергнуты риску реальные данные. Например, есть веб-сервисы, которые позволяют добавлять или изменять данные в удаленной системе. Ошибка при вызове такого сервиса может привести к потере данных.