Mock-объекты

Часто тестируемый метод может вызывать методы других классов, которые в данном случае тестировать не нужно. Unit-тест потому и называется модульным, что тестирует отдельные модули, а не их взаимодействие. Причем, чем меньше тестируемый модуль – тем лучше с точки зрения будущей поддержки тестов. Для тестирования взаимодействия используются интеграционные тесты, где вы уже тестируете скорее полные use cases, а не отдельную функциональность.

Однако наши классы очень часто используют другие классы в своей работе. Например, слой бизнес логики (Business Logic layer) часто работает с другими объектами бизнес логики или обращается к слою доступа к данным (Data Access layer). В трехслойной архитектуре веб-приложений это вообще постоянный процесс: Presentation layer обращается к Business Logic layer, тот, в свою очередь, к Data Access layer, а Data Access layer – к базе данных. Как же тестировать подобный код, если вызов одного метода влечет за собой цепочку вплоть до базы данных?

В таких случаях на помощь приходят так называемые mock-объекты, предназначенные для симуляции поведения реальных объектов во время тестирования. Вообще, понятие mock-объект достаточно широко: оно может, с одной стороны, обозначать любые тест-дублеры (Test Doubles) или конкретный вид этих дублеров – mock-объекты.

Понятие тест-дублеров введено неким Gerard Meszaros в своей книге «XUnit Test Patterns». Джерард и Мартин делят все тест-дублеры на 4 группы:

•Dummy – пустые объекты, которые передаются в вызываемые внутренние методы, но не используются. Предназначены лишь для заполнения параметров методов.

•Fake – объекты, имеющие работающие реализации, но в таком виде, который делает их неподходящими для production-кода (например, In Memory Database).

•Stub – объекты, которые предоставляют заранее заготовленные ответы на вызовы во время выполнения теста и обычно не отвечающие ни на какие другие вызовы, которые не требуются в тесте. Также могут запоминать какую-то дополнительную информацию о количестве вызовов, параметрах и возвращать их потом тесту для проверки.

•Mock – объекты, которые заменяют реальный объект в условиях теста и позволяют проверять вызовы своих членов как часть системы или unit-теста. Содержат заранее запрограммированные ожидания вызовов, которые они ожидают получить. Применяются в основном для т.н. interaction (behavioral) testing.

•Spy - у шпионов те же задачи, что и у моков, т.е. наблюдение и проверка взаимодействий с зависимостями во время выполнения теста. Разница в том, что шпионы используют функциональную реализацию для работы и могут записывать более сложные состояния, которые можно использовать для последующей проверки или утверждения (assertion).

Dummy

Предположим, что вам нужно протестировать метод Foo() класса TestFoo, который делает вызов другого метода Bar() класса TestBar. Предположим, что метод Bar() принимает какой-нибудь объект класса Bla в качестве параметра и потом ничего особого с ним не делает. В таком случае имеет смысл создать пустой объект Bla, передать его в класс TestFoo (сделать это можно при помощи широко применяемого паттерна Dependency Injection или каким-либо другим приемлемым способом), а затем уже Foo() при тестировании сам вызовет метод TestBar.Bar() с переданным пустым объектом. Это и есть иллюстрация использования dummy-объекта в unit-тестировании.

Fake

Метод Bar() выполняет какие-то действия с ним (допустим, Bar() сохраняет данные в базу или вызывает веб-сервис, а мы этого не хотим). В таких случаях наш объект класса TestBar должен быть уже не таким глупым. Мы должны научить его в ответ на запрос сохранения данных просто выполнить какой-то простой код (допустим, сохранение во внутреннюю коллекцию). В таких случаях можно выделить интерфейс ITestBar, который будет реализовывать класс TestBar и наш дополнительный класс FakeBar. При unit-тестировании мы просто будем создавать объект класса FakeBar и передавать его в класс с методом Foo() через интерфейс. Естественно, при этом класс Bar будет по-прежнему создаваться в реальном приложении, а FakeBar будет использован лишь в тестировании. Это иллюстрация fake-объекта

Stub

Stub-объекты (стабы) – это типичные заглушки. Они ничего полезного не делают и умеют лишь возвращать определенные данные в ответ на вызовы своих методов. В нашем примере стаб бы подменял класс TestBar и в ответ на вызов Bar() просто бы возвращал какие-то левые данные. При этом внутренняя реализация реального метода Bar() бы просто не вызывалась. Реализуется этот подход через интерфейс и создание дополнительного класса StubBar, либо просто через создание StubBar, который является унаследованным от TestBar. В принципе, реализация очень похожа на fake-объект с тем лишь исключением, что стаб ничего полезного, кроме постоянного возвращения каких-то константных данных не требует. Типичная заглушка. Стабам позволяется лишь сохранять у себя внутри какие-нибудь данные, удостоверяющие, что вызовы были произведены или содержащие копии переданных параметров, которые затем может проверить тест.

Mock

Mock-объект (мок), в свою очередь, является, грубо говоря, более умной реализацией заглушки, которая уже не просто возвращает предустановленные данные, но еще и записывает все вызовы, которые проходят через нее, чтобы вы могли дальше в unit-тесте проверить, что именно эти методы вот этих вот классов были вызваны тестируемым методом и именно в такой последовательности (хотя учет последовательности и строгость проверки, в принципе, настраиваемая вещь). То есть мы можем сделать мок MockFoo, который будет каким-то образом вызывать реальный метод Foo() класса TestFoo и затем смотреть, какие вызовы тот сделал. Или сделать мок MockBar и затем проверить, что при вызове метода Foo() реально произошел вызов метода Bar() с нужными нам параметрами.

Unit-тестирование условно делится на два подхода:

- state-based testing, в котором мы тестируем состояние объекта после прохождения unit-теста

•interaction (behavioral) testing, в котором мы тестируем взаимодействие между объектами, поведение тестируемого метода, последовательность вызовов методов и их параметры и т.д.

То есть в state-based testing нас интересует в основном, в какое состояние перешел объект после вызова тестируемого метода, или, что более часто встречается, что в реальности вернул наш метод и правилен ли этот результат. Подобные проверки проводятся при помощи вызова методов класса Assert различных unit-тест фреймворков: Assert.AreEqual(), Assert.That(), Assert.IsNull() и т.д.

В interaction testing нас интересует прежде всего не статическое состояние объекта, а те динамические вызовы методов, которые происходят у него внутри. То есть для нашего примера с классами TestFoo и TestBar мы будем проверять, что тестируемый метод Foo() действительно вызвал метод Bar() класса TestBar, а не то, что он при этом вернул и в какое состояние перешел. Как правило, в случае подобного тестирования программисты используют специальные mock-фреймворки (TypeMock.Net, EasyMock.Net, MoQ, Rhino Mocks, NMock2), которые содержат определенные конструкции для записи ожиданий и их последующей проверки через методы Verify(), VerifyAll(), VerifyAllExpectations() или других (в зависимости от конкретного фреймворка).

Spy

Spy (шпион) — самый неоднозначный тестовый дублер из всех, так как его определение разнится от автора к автору. Резюмируя первоначальное определение Джерарда Месароша и перефразируя его своими словами:

> Мы можем смело утверждать, что у шпионов те же задачи, что и у моков, т.е. наблюдение и проверка взаимодействий с зависимостями во время выполнения теста. Разница в том, что шпионы используют функциональную реализацию для работы и могут записывать более сложные состояния, которые можно использовать для последующей проверки или утверждения (assertion).

Шпион может заменить или даже расширить конкретную реализацию зависимости, переопределив некоторые методы в целях записи соответствующей информации для проверки теста. Независимо от того, сгенерирован ли шпион с помощью инструментов или создан вручную, по определению, под капотом всегда будет работающая реализация.

Подводя итог, мы можем разделить каждый из пяти типов дублеров по следующим категориям:

- Те, что не имитируют поведение и не наблюдает за взаимодействиями: пустышки.
- Те, что имитируют поведение: стабы и фейки.
- Те, что наблюдают за взаимодействиями: моки и шпионы.
- Те, что не располагают функциональной реализации под капотом: пустышки, стабы и моки.
- Те, что имеют функциональную реализацию под капотом: фейки и шпионы.

Все тестовые дублеры могут быть сгенерированы вручную или с помощью внешних инструментов. Те, которые конфигурируются с помощью инструментов, с большей вероятностью связывают детали реализации зависимости с самим тестом. Использование таких инструментов, как Mockito или Mockk, может упростить конфигурирование тестов на начальном этапе, но также может привести к более высоким затратам на обслуживание в будущем. Сгенерированные вручную дублеры имеют тенденцию увеличивать размер базы тестового кода в начале, но они также упрощают поддержку тестирования в долгосрочной перспективе.