Mockist (Outside-in) TDD Overview

Перевод статьи: [Mocking as a Design Tool](https://codurance.com/2018/10/18/mocking-as-a-design-tool/) (by Sandro Mancuso)

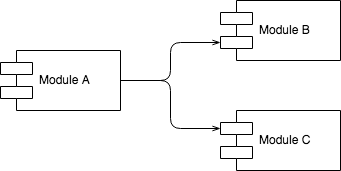
Использование Mock-объектов все еще является предметом напряженных споров в кругах практиков TDD. Cамая большая проблема заключается в том, что в случае использования Mock’ов, тесты в конечном итоге слишком много знают о внутреннем устройстве тестируемого модуля, что затрудняет будующий рефакторинг. Из-за этого рекомендуется избегать использования Mock’ов и использовать их только в качестве механизма изоляции в следующих сценариях:

* **Domain model**: Mock’и используются для иммитации портов в таких архитектурах как hexagonal, onion, clean с целью защиты/отделения ядра приложения (application и domain уровни) от инфраструктуры и механизмов доставки;
* **System**: Mock’и используются на системных границах (I/O и т.п.);
* Layering: Mock’и используются на границах уровней приложения.

Однако, при построении сложных систем, где бизнес-потоки не являются линейными (потоки, которые затрагивают различные части модели предметной области и каждая часть может инициировать свои собственные подпотоки), ограничение использования Mock’ов сценариями, описанными выше, является недостаточным.

**Association vs Composition vs Aggregation**

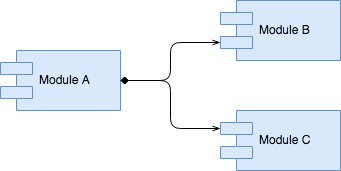
Рассмотрим отношения между модулями **A**, **B** и **C**:



**Association**

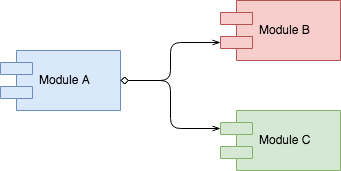
На диаграмме видно, что модуль **A** связан с модулями **B** и **C**. По направлению связей можно сказать, что **А** использует **B** и **C**. **B** и **C**, в свою очередь, ничего не знают об **A**. Вопрос в том, являются ли **B** и **C** частью **A**, или же **A** просто использует **B** и **C**.

**Composition**



Данная диаграмма описывает, что **B** и **C** являются частью **A**, т.е. их жизненный цикл непосредственно контролируется жизненным циклом **А**. Если **A** удаляется (из памяти или даже из кодовой базы), **B** и **C** также удаляются, так как они не имеют смысла без **A**.

**Aggregation**



Данная диаграмма описывает, что **A** просто использует **B** и **C**, т.е. их жизненный цикл не контролируется жизненным циклом **A**. Если **A** удаляется, **B** и **C** будут продолжать существовать.

Composition и Aggregation являются типами/разновидностями Association. А различие меджу ними важно, поскольку оно может помочь нам понять границы наших модулей или компонентов.

**Boundaries (границы)**

Прежде чем мы поговорим об использовании Mock’ов в контексте Outside-in TDD, мы должны поговорить о границах. Некоторые границы такие, как пакеты, пространства имен, классы, методы или функции являются очевидными. Они обуславливаются используемым языком и парадигмой программирования. Другие границы менее очевидны, это означает, что они определяются нашими **Macro Design** решениями, такими как определение слоев, компонентов, модели предметной области, механизмов доставки, инфраструктуры и шаблонов проектирования в целом. Практики TDD, в том числе те, которые обычно стараются избегать Mock’ов, с удовольствием используют их на границах того, что они разрабытывают используя TDD. Единственная проблема заключается в том, что они редко соглашаются, где эти границы должны быть.

**Рассмотрим пример**

Представьте, что мы создаем e-commerce web application и после обсуждения с владельцем продукта, мы определили следующие высокоуровневые требования:

*Scenario: выполнение платежа*

Для того, чтобы мои товары были доставлены на мой домашний адрес, в качестве покупателя, мне необходимо заплатить за все товары в моей корзине.

*Acceptance criteria*

* Сценарий будет инициирован со страницы выписки, после того, как пользователь заполнит платежные реквизиты;
* Оплата должна быть обработана согласно:
* Выбранный способ оплаты (кредитная карта, дебетовая карта, Paypal, Apple Pay);
* Разные страны имеют разные платежные шлюзы;
* Обнаружение мошенничества должно применяться ко всем платежам по кредитным и дебетовым картам.
* Должен быть создан заказ, содержащий информацию о пользователе, всех товарах в корзине, скидках, способе оплаты и адрес доставки:
* Статус заказа должен быть установлен в "open" при создании; в "paid" после успешной оплаты или в "payment failed" в противном случае.
* Уведомление для складской системы должно быть отправлено со всеми проданными товарами. Складская система инициирует процесс доставки;
* Дату(ы) доставки следует проверить еще раз, как только платеж будет отправлен и показан пользователю на странице подтверждения, т.к. различные товары могут быть доставлены в разные даты;
* Подтверждение оплаты и дата(ы) доставки, должно быть отправлено пользователю по электронной почте;
* Пользователь долже увидеть страницу подтверждения с номером заказа, которая ссылается на детали заказа, дату(ы) доставки с разбивкой по элементам и способам оплаты и итоговой суммой.

Мы могли бы добавить еще много поведения к acceptance criteria’м, но я думаю, этого достаточно, чтобы использовать в качестве примера, где один запрос из браузера может вызвать много разнородного поведения.

**Определение модулей и поведения**

Чтобы сохранить эту статью сфокусированной и простой, я сосредоточусь только на бизнес-логике, описанной выше, и опущу детали реализации.

Читая вышеизложенные требования, я считаю, что не стоит пытаться поместить всю функциональность в один модуль. Поэтому использование **Classicist TDD** в данном случае кажется мне пустой тратой времени. Т.к. приведенные выше требования дают мне достаточно информации для принятия некоторых **Macro Design** решений, которые позволят повысить эффективность разработки (создания дизайна) по средствам **Mockist (Outside-in) TDD**.

Глядя на требования, мы можем определить два различных типа потоков (flow):

* **Main flow:** Всеобъемлющий поток «выполнить платеж», который должен координировать различные подпотоки (sub-flows);
* **Sub-flows:** Шаги (подпотоки) всеобъемлющего потока. Каждый из этих шагов имеет свои собственные потоки и представляет/затрагивает различные части предметной области. Эти различные части можно назвать модулями.

Давайте посмотрим на потенциальные модули и обязанности:

* **Payments:** способы оплаты, платежные шлюзы, логика конкретной страны, обнаружение мошенничества;
* **Orders:** Создать, обновить заказ;
* **Delivery:** сложная логика, в которой разные элементы в корзине могут быть сгруппированы и доставлен в разные сроки в зависимости от местонахождения склада, варианта доставки и т. д;
* **Notification:** уведомления пользователей, связанные с оплатой, доставкой и т. п.

Всеобъемлющая логика также должна быть инкапсулирована в отдельном модуле, и эта логика должна быть доступна механизму доставки (в нашел случае HTTP-серверу).

**Использование Mock’ов в качестве инструмента проектирования**

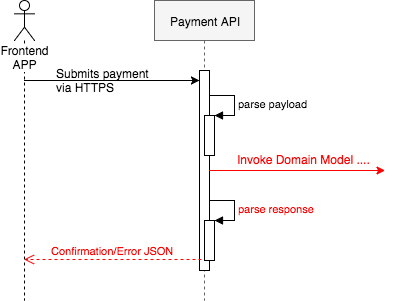
При построении сложного потока, содержащего много «подвижных» частей, попытка построить все за один раз может быть чрезвычайно сложной, поскольку нам придется постоянно переключать свое внимание между элементами логики на разных уровнях абстракции. Например, придется фокусироваться на деталях выявления мошенничества при тестировании Main flow’a.

Использование Mock’ов в качестве инструмента проектирования позволяет избежать описанных выше проблем и неудобств.

Использование Mock’ов в качестве инструмента проектирования имеет больше смысла при использовании **Outside-In TDD (London School)**. Этот стиль TDD сосредоточен на разделении большой проблемы на более мелкие. Работа начинается с рассмотрения Macro-поведения (main flow), разделяя его на более мелкие элементы поведения, пока элементы/части поведения не будут достаточно малы, чтобы быть достаточно сплоченными (High Cohesion) или/и иметь единственную причину для изменения/обязанность (SRP). В нашем примере мы начнем Outside-In TDD-процесс с main flow’a, и далее переходим к sub-flow’ам.

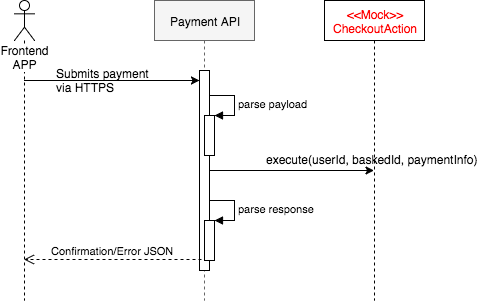
**Outside-in проектирование с использование Mock’ов**

Предположим, что у нас будет endpoint под названием PaymentsAPI, который будет анализировать JSON, полученный через HTTP-запрос, запускать Main flow и возвращать JSON через HTTP-ответ во внешнее приложение (front-end app).



Теперь нам нужно определить модуль ядра приложения, который будет вызываться из кода, кот. обрабатывает запрос к endpoint’y PaymentsAPI. На данный момент мы можем использовать Mock для проектирования интерфейса этого модуля и завершения реализации endpoint’a PaymentsAPI. Причина, по которой мы можем безопасно сделать это, заключается в том, что мы уже решили, какое поведение останется в в коде endpoint’a PaymentsAPI и что будет делегировано проектируемому модулю ядра приложения.

Если следовать Clean Architecture, Domain Driven Design (DDD), или Interaction Driven Design (IDD), нам понадобится модуль, который будет управлять подпотоками (sub-flows) основного потока (main flow). Этот модуль играет роль Use Case’a (в Clean Architecture), Application Service’a (в DDD) или Action’a (в IDD). Я буду использовать IDD и соответственно назову данный модуль CheckoutAction.



Использование Mock’a для разработки публичного интерфейса CheckoutAction помогло нам решить, что код endpoint’a PaymentsAPI должен отправлять и что он должен получать. Таким образом, мы можем завершить тестирование кода endpoint’a PaymentsAPI, не беспокоясь о деталях main flow’a. Это также позволяет нам жестко запрограммировать некоторое значение и протестировать весь пользовательский интерфейс.