Mockist (Outside-in) TDD Overview

Перевод статьи: [Mocking as a Design Tool](#) (by Sandro Mancuso)

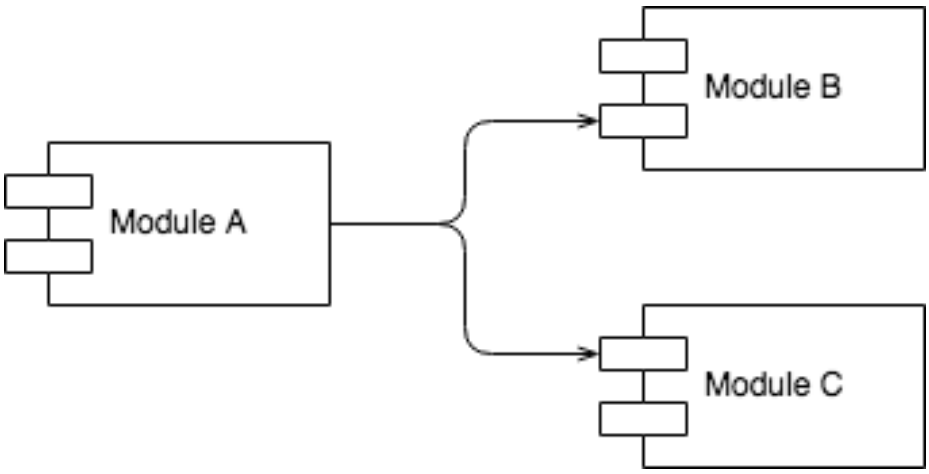
Использование Mock-объектов все еще является предметом напряженных споров в кругах практиков TDD. Самая большая проблема заключается в том, что в случае использования Mock'ов, тесты в конечном итоге слишком много знают о внутреннем устройстве тестируемого модуля, что затрудняет будующий рефакторинг. 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

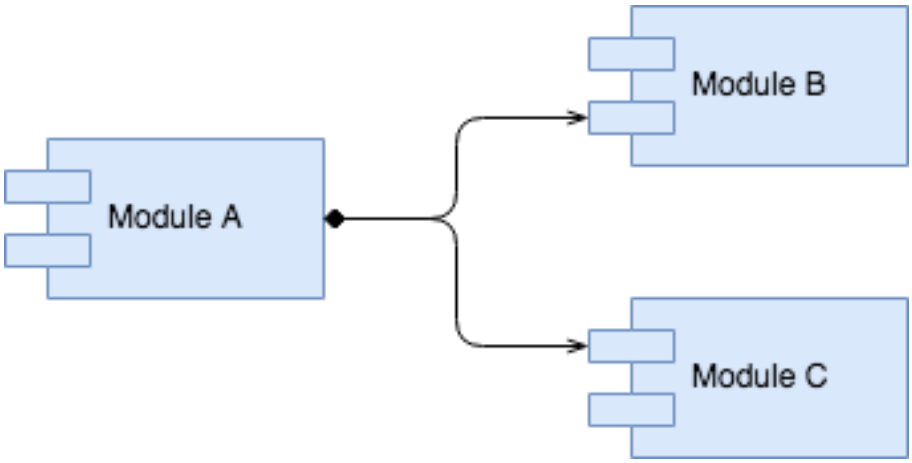
Рассмотрим отношения между модулями **A**, **B** и **C**:



Association

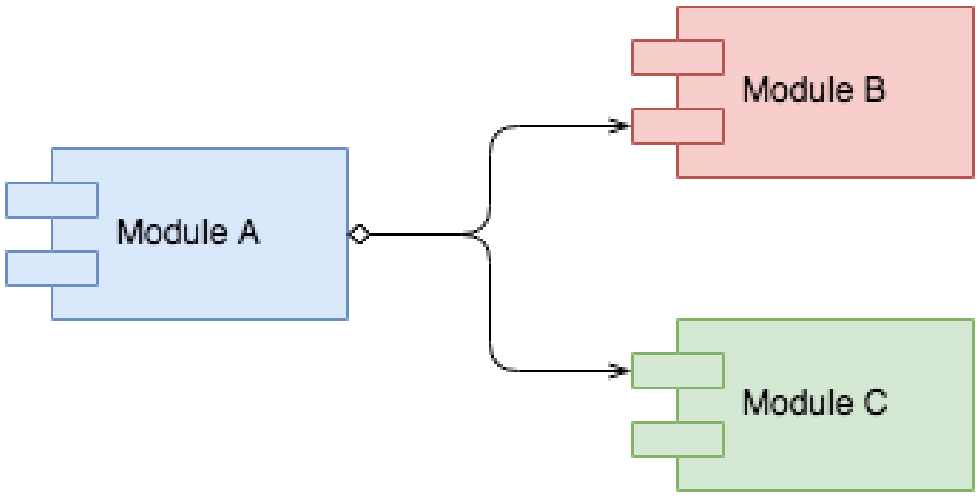
На диаграмме видно, что модуль **A** связан с модулями **B** и **C**. По направлению связей можно сказать, что **A** использует **B** и **C**. **B** и **C**, в свою очередь, ничего не знают об **A**. Вопрос в том, являются ли **B** и **C** частью **A**, или же **A** просто использует **B** и **C**.

Composition



Данная диаграмма описывает, что **B** и **C** являются частью **A**, т.е. их жизненный цикл полностью контролируется жизненным циклом **A**. 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** продолжают существовать.

Composition и Aggregation являются типами/разновидностями Association. А различие меджу ними важно, поскольку оно может помочь нам понять границы наших модулей/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 разрабатывают 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 платежа, после того, как пользователь заполнит платежные реквизиты;
- Оплата должна быть обработана согласно:
    - Выбранному способу 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 адресе 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, 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

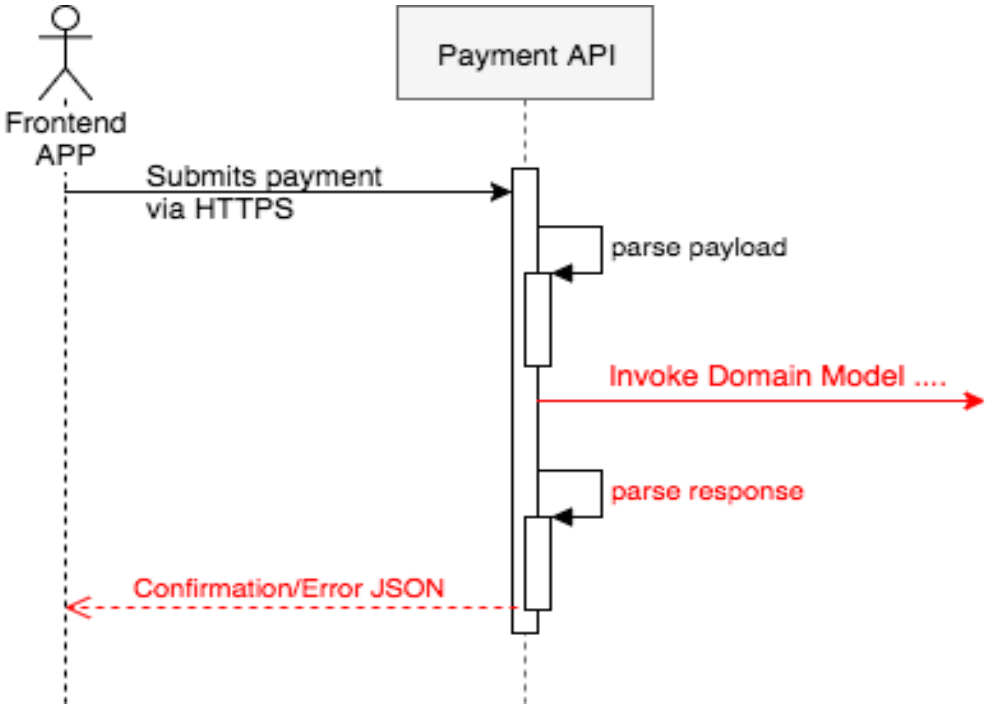
При построении сложного потока, содержащего много «подвижных» частей, попытка построить все за один раз может быть чрезвычайно сложной, поскольку нам придется постоянно переключать свое внимание между элементами логики на разных уровнях абстракции. Например, придется фокусироваться на деталях выявления 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'а, и далее перейдем к sub-flow'ам.

Outside-in проектирование с использование Mock'ов

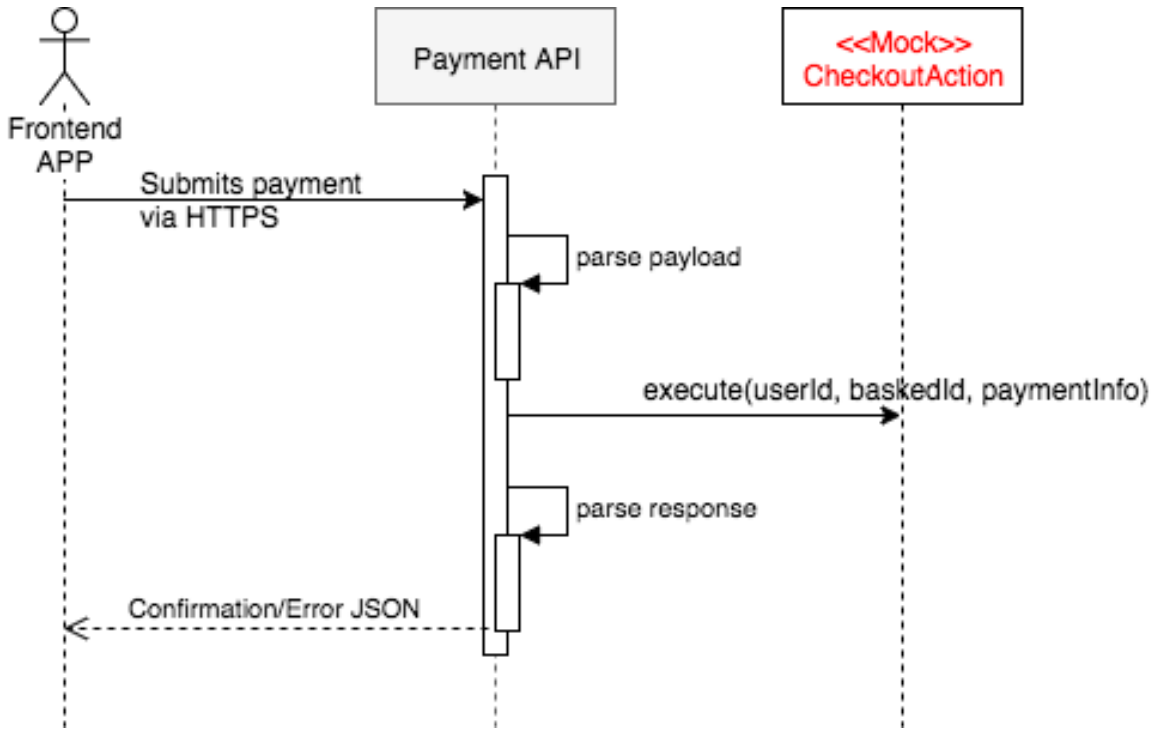
Предположим, что у нас будет endpoint под названием PaymentsAPI, который будет анализировать JSON, полученный через HTTP-запрос, запускать Main flow и возвращать JSON через HTTP-ответ во внешнее приложение (front-end app).



Теперь нам нужно определить модуль ядра приложения, который будет вызываться из кода, кот. обрабатывает запрос к endpoint'у PaymentsAPI. На данный момент мы можем использовать Mock для проектирования интерфейса

этого модуля и завершения реализации endpoint'а PaymentsAPI. Причина, по которой мы можем безопасно сделать это, заключается в том, что мы уже решили, какое поведение останется в коде endpoint'а PaymentsAPI, а что будет делегировано проектируемому модулю ядра приложения.

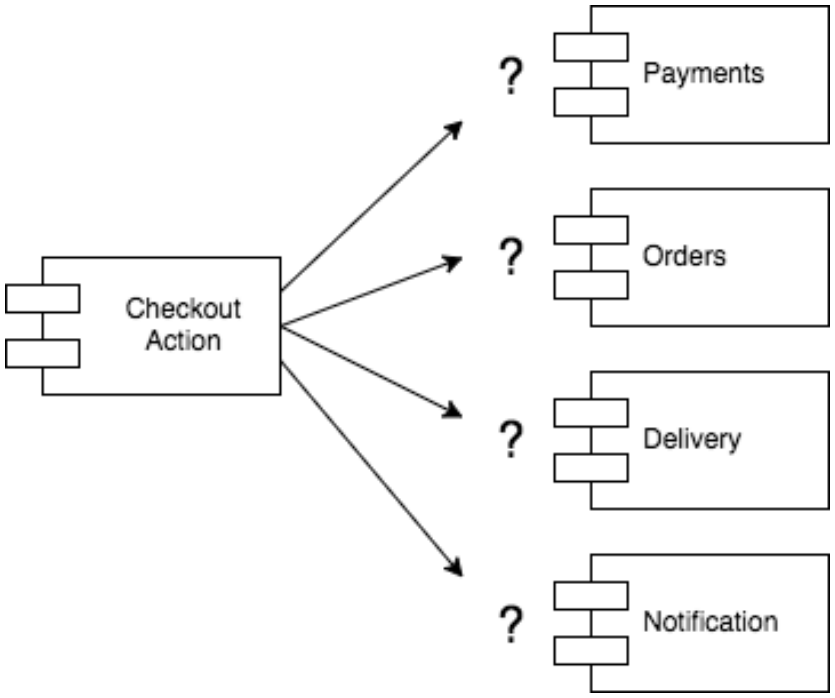
Если следовать Clean Architecture, Domain Driven Design (DDD), или Interaction Driven Design (IDD), нам понадобится модуль, который будет управлять подпотоками (sub-flows) основного потока (main flow). Этот модуль играет роль Use Case'а (в Clean Architecture), Application Service'а (в DDD) или Action'а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



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

Проектирование ядра приложения

Теперь, когда мы знаем, что модуль CheckoutAction является точкой входа в ядро нашего приложения, нам необходимо его реализовать. Согласно нашему предыдущему анализу, у нас есть по крайней мере четыре группы поведения, которые модуль CheckoutAction должен вызвать/запустить: Payments, Orders, Delivery, Notification. И нам необходимо решить как эти группы поведения будут ассоциированы с модулем CheckoutAction: **Composition** или **Aggregation**?



Т.е. необходимо понять являются ли они неотъемлемой частью CheckoutAction или просто им используются. Ответ на этот вопрос напрямую влияет на положение границы, которые мы проводим вокруг наших компонентов, что, в свою очередь, влияет на стратегию тестирования. Разработка кода с использованием Outside-in TDD намного проще и эффективнее, когда границы компонентов четко определены и немного обдуманы наперед.

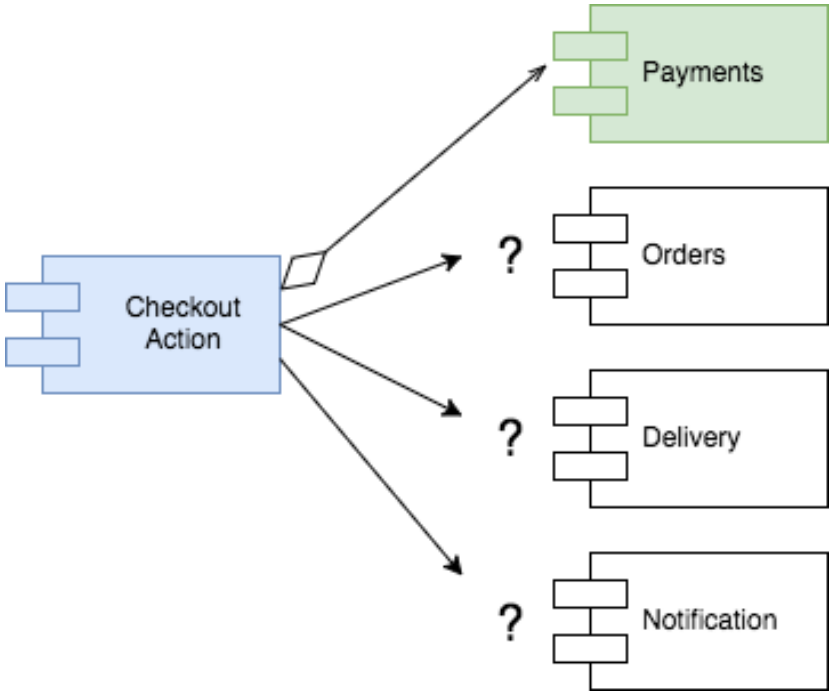
Проще всего было бы поразмышлять о жизненных циклах, а именно, если мы удаляем CheckoutAction, должны ли мы также удалить логику, связанную с Payments, Orders, Delivery и Notification? Но нам нужно быть очень осторожными используя такой подход, потому что теоретически, если CheckoutAction является единственной частью кода, вызывающей эти другие части логики, они будут

сиротами и должны быть удалены. Получается, что этого недостаточно, чтобы решить, какой должна быть ассоциация: **Composition** или **Aggregation**. Лучший подход – это думать об изменяемости, т.е. изменяются ли модули/компоненты по одной и той же причине? Заинтересованы ли в них другие компоненты системы? Должны ли изменения в них повлиять на другие части системы?

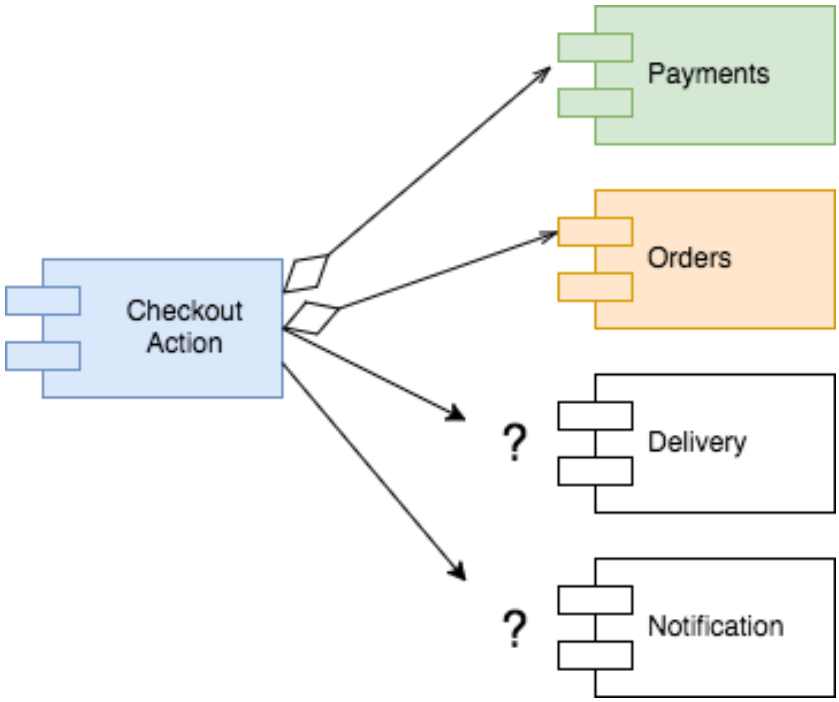
Анализ изменяемости

Какова должна быть ответственность модуля CheckoutAction? Если мы попытаемся в нем разместить всю логику, связанную с платежами, заказами, доставкой, уведомлением, плюс координацию этим направлениям функционала, то CheckoutAction будет нарушать SRP и десяток других принципов проектирования. Чтобы избежать этого, обязанностью модуля CheckoutAction должно быть управление Main flow'ом, а каждый шаг Main flow'а CheckoutAction должен делегировать соответствующему модулю.

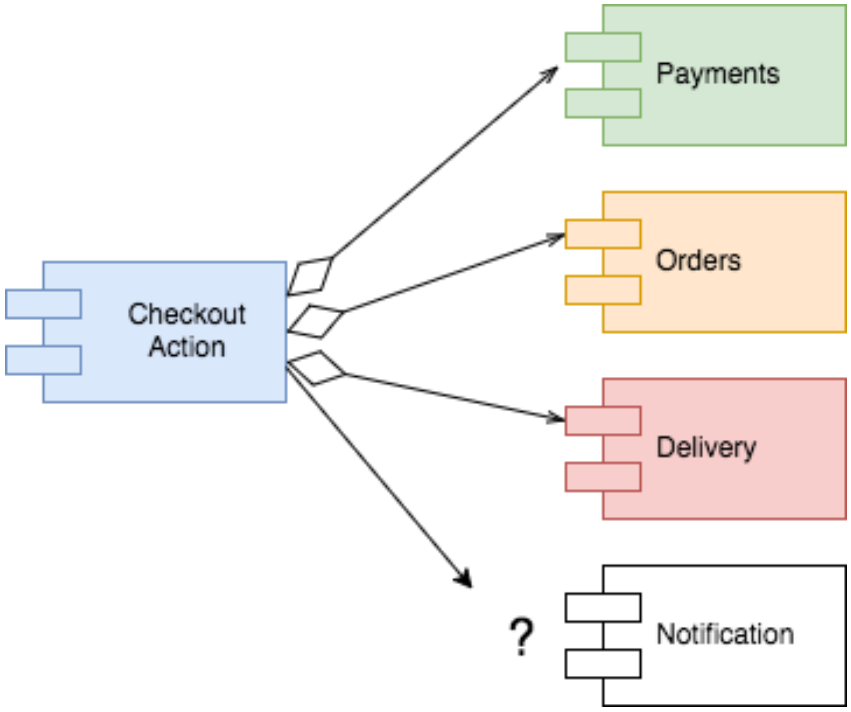
Теперь давайте посмотрим на модуль Payments. Мы можем добавить или удалить способы оплаты, платежные шлюзы, логику оплаты для конкретной страны и логику обнаружения мошенничества. Из разговора с предствителями бизнеса, мы выяснили, что перед добавлением нового способа оплаты или шлюза, определенная группа людей будет участвовать в подписании контракта с поставщиками, и это не те люди, которые будут работать над другими частями системы. Очевидно, что необходимо, чтобы изменения в модуле Payments не повлияли на другие части системы. Это сделало бы модуль Payments соответствующим OCP, то есть мы могли бы добавлять или удалять способы оплаты, шлюзы и т. д., без какого-либо влияния на остальную систему. Имея это в виду, я вполне уверен, что модуль Payments должен быть полностью независимым от модуля CheckoutAction, что подразумевает ассоциацию **Aggregation**.



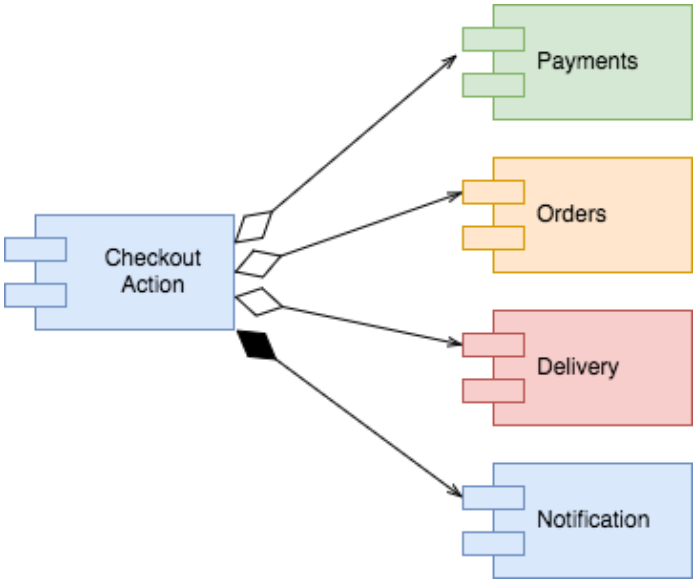
Далее рассмотрим модуль Orders. Из разговора с предствителями бизнеса, мы выяснили, что заказы имеют свой собственный жизненный цикл (новые, ожидающие подтверждения, оплаченные, отклоненные, выполненные и т. д.), а различные части системы и внутренние пользователи нуждаются в информации о заказах, например, бэк-офис. Этого достаточно для принятия решения об отделении логики заказов от модуля CheckoutAction, что подразумевает ассоциацию **Aggregation**.



Модуль Delivery является сложной частью системы, поскольку он включает в себя различных поставщиков услуг в разных частях мира. Он включает в себя разные модели договорных обязательств, логику принятия решений, какие варианты доставки доступны пользователям по всему миру в соответствии с их адресом доставки и т.д. Эта информация поддерживается другой командой бэк-офиса. Выглядит довольно безопасно предположить, что модуль Delivery также должен быть изолирован от модуля CheckoutAction – т.е. **Aggregation**.

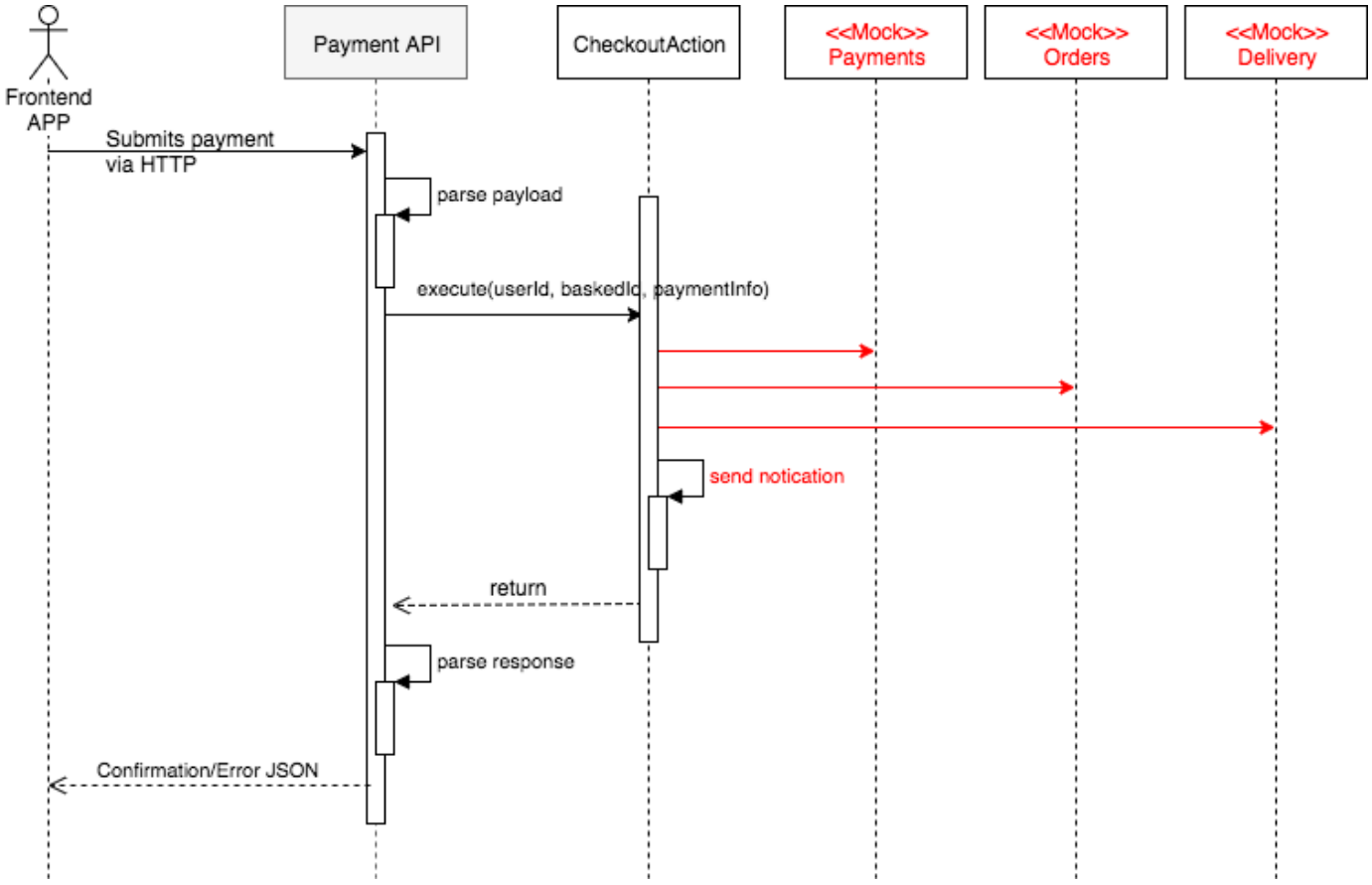


И наконец мы дошли до модуля Notifications. От представителей бизнеса мы узнали только лишь, что нам нужно будет отправить уведомление пользователю с результатом процесса оформления заказа — либо платеж был получен и заказ должен быть обработан, либо платеж был отклонен. Несмотря на то, что я чувствую, что эта логика может расти и использоваться повторно, у меня недостаточно доказательств этого, поэтому в этом случае я сохраню ее как подмодуль модуля CheckoutAction – т.е. **Composition**.

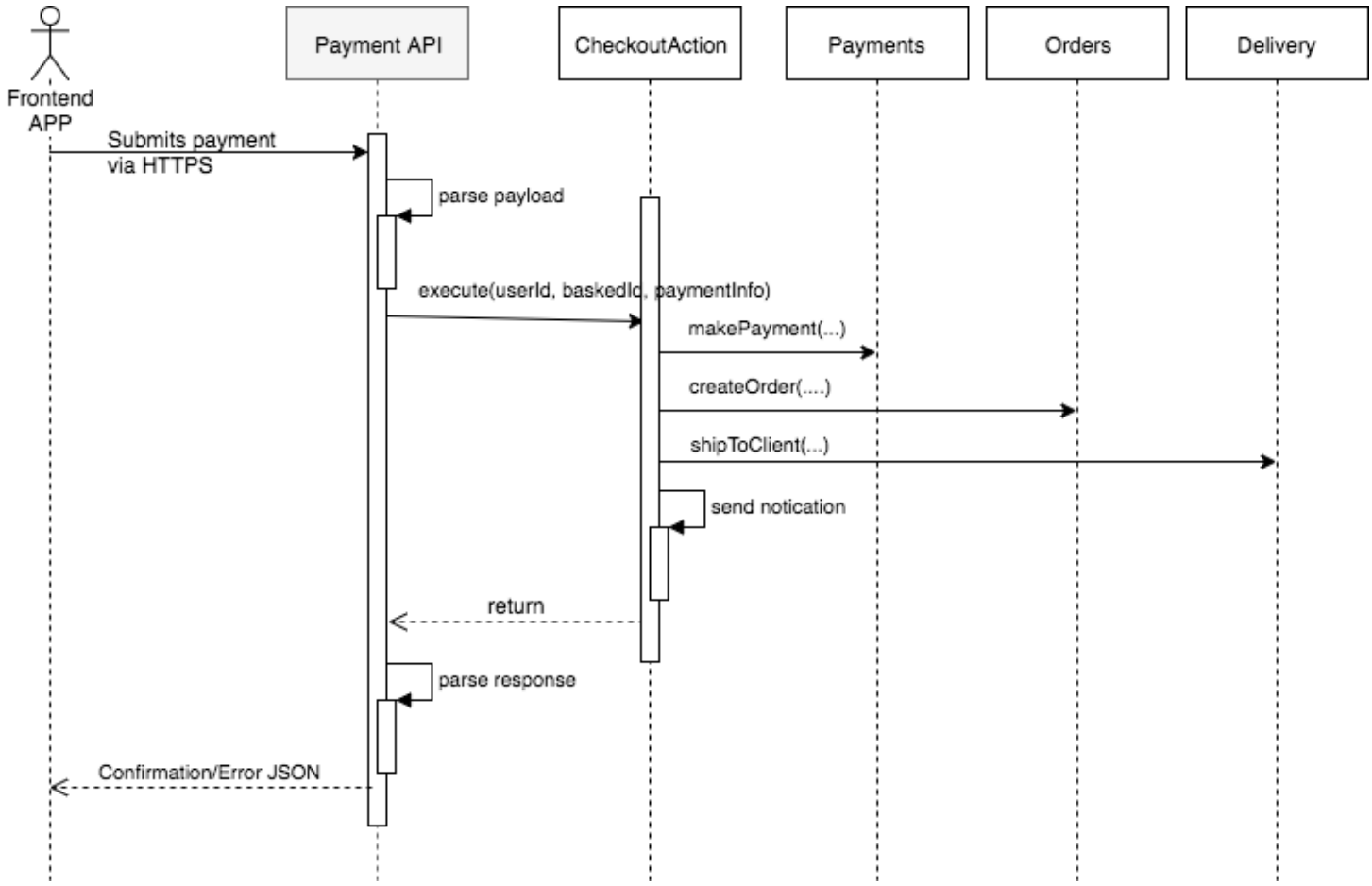


Использование Mock'ов для проектирования взаимодействий

На основе решения о том, какое поведение располагается вместе (Composition) и какое поведение будет располагаться в отдельном модуле (Aggregation), нам просто нужно выяснить, как эти модули будут взаимодействовать друг с другом.



Для этого я использую Outside-in TDD (London school) для реализации модуля CheckoutAction, т.е. использую Mock'и для определения публичного интерфейса модулей Payments, Orders и Delivery. Их интерфейс должен быть ограничен информацией, необходимой для CheckoutAction, уменьшая внешнюю зависимость (Low Coupling) между модулями и увеличивая внутреннюю связность (High Cohesion) каждого из них.



В процессе реализации модуля CheckoutAction мы также определим публичный интерфейс каждого модуля, с которым CheckoutAction взаимодействует. И далее необходимо будет думать о реализации каждого из этих модулей, будучи уверенным, что ничто другое не будет затронуто пока не будет изменен их публичный интерфейс.

Проектирование смежных модулей (Outside-in)

Далее мы можем рассмотреть один из смежных модулей, например, Payments. Могут возникнуть такие вопросы, как: должны ли мы отделить логику для кредитных карт, Paypal и т.д. друг от друга? Должны ли мы отделить логику обнаружения мошенничества от способов оплаты? Должны ли мы иметь область системы, в которой мы решаем, какой платежный шлюз будет использован в зависимости от местоположения пользователя? Отвечая на эти вопросы, мы решаем, сколько подмодулей будут иметь модуль Payments и тип ассоциации между ними – **Composition** или **Aggregation**. Это поможет нам решить, для чего, когда и где использовать Mock'и. Имея приблизительное представление о Macro-дизайне, мы можем начать разработку модуля Payments используя Outside-in TDD.

Заключительное слово

Для людей, привыкших к Outside-in TDD, использование Mock'ов является инструментом проектирования, а не тестирования. Как только мы это поймем, Mock'и станут отличным инструментом для управления Macro-дизайном наших приложений.

TDD становится намного проще, когда мы можем определить границы наших модулей. Быстрый анализ бизнеса и высокоуровневый дизайн непосредственно перед реализацией функциональной возможности поможет сэкономить много времени. И как только у нас появляется грубый высокоуровневый план, наши TDD усилия становятся более целенаправленными и эффективными, почти механическими — и это хорошо.

Но что делать, если границы не так очевидны? Иногда мы просто не можем увидеть, как должно выглядеть решение, и нам нужно исследовать. В подобных случаях забудьте о высокоуровневом проектировании, переключитесь прямо на

Classicist TDD и работайте маленькими шагами, пока решение не найдется. Если вам действительно нужно, вообще забудьте о TDD, начните печатать и посмотрите, что произойдет.

Все дело в уверенности. Если я смогу четко представить дизайн своих модулей и тип ассоциаций между ними, то я буду использовать TDD по направлению к этому дизайну и Mock'и для проектирования взаимодействия между различными модулями. Если же я не вижу решения или не уверен в том, какое решение наиболее предпочтительно, то я переключаюсь в режим исследования и работаю небольшими шагами, создавая небольшой беспорядок, а затем использую рефакторинг, чтобы решить, как лучше организовать свой код. Хотя это может показаться отличной идеей всегда использовать небольшие шаги и рефакторинг, но я нахожу это чрезвычайно медленным и неэффективным, поэтому я смешиваю разные стили TDD.