TDD And Good Design

Перевод статьи: [Does TDD really lead to good design?](https://codurance.com/2015/05/12/does-tdd-lead-to-good-design/) (by Sandro Mancuso)

Я считаю, что TDD не может привести к хорошему дизайну, если мы не знаем, как хороший дизайн выглядит. Я также считаю, что мы, должны учить дизайну до TDD или, по крайней мере, одновременно.

Понаблядав за тем, как я и многие других разработчики работают, я понял, что далеко не многим из них удается создать хороший дизайн используя TDD. И несмотря на то, что мне нравится ритм RED-GREEN-REFACTORING, я считаю, что одного рефакторинга недостаточно, чтобы назвать TDD инструментом проектирования.

TDD не предписывает, как вы должны проектировать. Что оно действительно делает, так это постоянно раздражает вас вопросами: Вы уверены в этом? Достаточно ли это хорошо? Возможно ли сделать это лучше? Это постоянное напоминание посмотреть на получающийся дизайн и попытаться его улучшить – отличная вещь, но этого недостаточно.

На мой взгляд, TDD – это подход к процессу разработки программного обеспечения, который предоставляет много преимуществ, включая постоянное напоминание о том, что код необходимо улучшать. А то как именно улучшать код, не является частью TDD.

TDD – это рабочий процесс (а не инструмент проектирования), в котором на этапе рефакторинга вы применяете свои существующие знания в области разработки программного обеспечения в сочетании с принципами и методами проектирования, которые могут помочь вам достичь лучшего дизайна.

**Различные стили TDD**

Есть два основных стиля TDD со значительными различиями в части подхода к проектированию.

**Classicist TDD**

Classicist TDD – это оригинальный подход к TDD, созданный Кентом Беком. Он также известен как Detroit School TDD.

**Основные характеристики:**

* Проектирование происходит на этапе рефакторинга;
* Тесты основаны на проверке состояния;
* На этапе рефакторинга тестируемый модуль может разрастаться до нескольких классов;
* Mock’и используются редко, и только для изоляции внешних систем;
* Не принимается никаких предварительных проектных решений. Архитектурв/дизайн вытекает из кода путем рефакторинга;
* Это отличный способ избежать over-engineering’a;
* Хорошо подходит для исследовательских целей, когда мы знаем, каковы входные и желаемые выходные данные, но не знаем, как должна выглядеть реализация;
* Отлично подходит в случае, когда мы не можем полагаться на экспертов по предметной области.

**Недостатки:**

* Необходимо «выставлять на показ» состояние объектов только для целей тестирования, т.к. тесты основаны на проверке состояния;
* Фаза рефакторинга обычно больше по сравнению с Outside-in TDD;
* Тестируемый модуль разрастается и становится больше одного класса, когда классы появляются на этапе рефакторинга. Далее эти классы будут развиваться и могут сломать совершенно несвязанные с ними тесты, т.к. тесты используют их реальную реализацию вместо Mock’ов;
* Стадия рефакторинга часто пропускается неопытными членами команды, что приводит к циклу, который больше похож на RED-GREEN-RED-GREEN-…-RED-GREEN-MASSIVE REFACTORING;
* Из-за исследовательской природы Classicist TDD стиля некоторые тестируемые классы создаются в соответствии с предположениями “я думаю, что мне понадобится этот класс с этим интерфейсом (открытыми методами)”, что делает такие классы не очень подходящими при подключении к остальной системе;
* Хорошо подходит для новичков, но может стать пустой тратой времени для более опытных разработчиков.

**Outside-in TDD**

Outside-in TDD, также известный как London School или Mockist TDD, является стилем TDD, который был разработан и принят некоторыми из первых практиков XP в Лондоне. На основе данного стиля позже был создан BDD.

**Основные характеристики:**

* В отличие от Classicist TDD, Outside-in TDD предписывает направление, в котором мы начинаем TDD-процесс: извне (первый класс, который будет получать внешний запрос) внутрь (классы, которые будут содержать отдельные части поведения, которые необходимы для реализации требуемой функциональной возможности);
* TDD-процесс начинается с создания приемочного (acceptance) теста, который проверяет, работает ли функциональная возможность в целом. Приемочный тест также служит руководством для реализации;
* Имея непрошедший приемочный тест, который сообщает почему реализация функциональной возможности еще не завершена (не были возвращены необходимые данные, сообщение не было отправлено в очередь, данные не были сохранены в БД и т.п.), мы начинаем писать Unit-тесты и первым классом, который необходимо протестировать является класс, который будет обрабатывать внешний запрос (Controller, Queue Listener, Event Handler и т.п.);
* В отличии от Classicist TDD, в Outside-in TDD нам не нужно строить все приложение в одном классе. Вместо этого мы делаем некоторые предположения о том, с какими классами-коллабораторами должен взаимодействовать тестируемый класс. Затем мы пишем тесты, которые проверяют корректность этих взаимодействий;
* Классы-коллабораторы определяются в соответствии с тем, что должен делать тестируемый класс при вызове его открытого метода. Имена классов-коллабораторов и их методов должны исходить из языка предметной области (существительных и глаголов);
* После того, как текущий класс протестирован, мы выбираем первый класс-коллаборатор (который был создан без реализации) и начинаем TDD-процесс относительно него, следуя тому же подходу, который мы использовали для предыдущего класса. Вот почему данный стиль называется outside-in: мы начинаем с классов, которые ближе ко входу системы (outside), и двигаемся к внутренней части нашего приложения (in) по мере выявления классов-коллабораторов;
* Проектирование начинается в «красной» фазе, во время написания тестов;
* Тесты проверяют взаимодействие и поведение, а не состояние;
* Дизайн улучшается на этапе рефакторинга;
* Каждый класс-коллаборатор и его открытые методы всегда создаются исходя из потребностей существующего клиентского класса, что делает код очень хорошо читаемым;
* Фазы рефакторинга требуют намного меньше времени по сравнению с Classicist TDD;
* Способствует лучшей инкапсуляции, так как ни одно состояние не «выставляется на показ» только для целей тестирования;
* Способствует принципу проектирования: Tell, Don’t Ask;
* Хорошо подходит для бизнес-приложений, где существительные и глаголы могут быть извлечены из business story’ий и acceptance-критериев.