Classicist And Mockist TDD Comparison

Перевод статьи: Mocks Aren't Stubs (by Martin Fowler)

Обычные тесты

Начнем с иллюстрации Classicist и Mockist TDD-стилей на простом примере. Мы хотим взять объект заказа (Order) и заполнить его товарами со склада (Warehouse). Order очень прост и содержит только один продукт и его количество. Warehouse имеет запасы различных продуктов. Когда мы просим Order заполниться со склада, есть два возможных сценария:

- Если Warehouse содержит достаточно товара для выполнения Order'а, Order становится заполненным, а количество товара в Warehouse уменьшается на соответствующее число;
- Если Warehouse содержит недостаточно товара, то Order не заполняется и в Warehouse ничего не происходит.

Эти два сценария подразумевают несколько тестов, которые выглядят как обычные JUnit-тесты.

```
public class OrderStateTester extends TestCase {
  private static String TALISKER = "Talisker";
  private static String HIGHLAND_PARK = "Highland Park";
  private Warehouse warehouse = new WarehouseImpl();

  protected void setUp() throws Exception {
    warehouse.add(TALISKER, 50);
    warehouse.add(HIGHLAND_PARK, 25);
  }

  public void testOrderIsFilledIfEnoughInWarehouse() {
    Order order = new Order(TALISKER, 50);
    order.fill(warehouse);
    assertTrue(order.isFilled());
    assertEquals(0, warehouse.getInventory(TALISKER));
  }
```

```
  public void testOrderDoesNotRemoveIfNotEnough() {
    Order order = new Order(TALISKER, 51);
    order.fill(warehouse);
    assertFalse(order.isFilled());
    assertEquals(50, warehouse.getInventory(TALISKER));
  }
}
```

xUnit-тесты следуют типичной четырехэтапной последовательности:

- Setup;
- Exercise;
- Verify;
- Teardown.

В этом случае этап Setup выполняется частично в методе `.setup()` (инициализация Warehouse'а) и частично в методе `.test…()` (инициализация Order'а). Вызов `.fill(…)` экземпляра Order – это этап Exercise. Здесь мы просим объек типа Order сделать то, что мы хотим проверить. Утверждения assert – это этап Verify, и они необходимы, для проверки, правильности функционирования тестируемого метода. В данном случае нет явного этапа Teardown, т.к. сборщик мусора в Java делает это за нас неявно.

Таким образом для тестирования данных сценариев потребуется system-under-test или SUT (Order) и один Collaborator (Warehouse). Warehouse необходим по двум причинам: первая – это вообще позволить тестируемому поведению работать (метод `.fill(…)` вызывает методы экземпляра Warehouse), а второе – это необходимость верификации, т.к. метод `.fill(…)` может изменить состояние экземпляра Warehouse. Между SUT и Collaborator'ами существуют определнные различия, которые рассматриваются далее.

При таком способе тестирования используется проверка состояния: это означает, что мы определяем, правильно ли работал тестируемый метод, проверяя состояние SUT и его Collaborator'ов после того, как тестируемы метод был вызван.

Тесты с использованием Mock'ов

Теперь мы протестирует то же поведение с использованием Mock'ов, для создания которых будет использоваться JMock Framework.

```
public class OrderInteractionTester extends MockObjectTestCase {
  private static String TALISKER = "Talisker";

  public void testFillingRemovesInventoryIfInStock() {
    //setup - data
    Order order = new Order(TALISKER, 50);
    Mock warehouseMock = new Mock(Warehouse.class);

    //setup - expectations
    warehouseMock.expects(once()).method("hasInventory")
      .with(eq(TALISKER),eq(50))
      .will(returnValue(true));
    warehouseMock.expects(once()).method("remove")
      .with(eq(TALISKER), eq(50))
      .after("hasInventory");

    //exercise
    order.fill((Warehouse) warehouseMock.proxy());

    //verify
    warehouseMock.verify();
    assertTrue(order.isFilled());
  }

  public void testFillingDoesNotRemoveIfNotEnoughInStock() {
    Order order = new Order(TALISKER, 51);
    Mock warehouse = mock(Warehouse.class);

    warehouse.expects(once()).method("hasInventory")
      .withAnyArguments()
      .will(returnValue(false));

    order.fill((Warehouse) warehouse.proxy());

    assertFalse(order.isFilled());
  }
}
```

Легко заметить, что этап Setup очень отличается от предыдущего подхода. Он разделен на две части: данные и ожидания. В части данных настраивается объект Order, поведение которого и будет тестироваться, и эта часть этапа Setup похожа на традиционный. Разница заключается в том, что Collaborator не является

экземпляром Warehouse, вместо него используется Mock Warehouse'а – технически, это экземпляр класса Mock.

Во второй части настраиваются ожидания, которые указывают какие методы и как должны быть вызваны у Mock'а во вренмя работы тестируемого метода.

Ключевым отличием здесь является то, как мы проверяем правильность взаимодействия Order и Warehouse. В предыдущем подходе мы использовали **проверку состояния** экземпляров Order и Warehouse. В подходе с использованием Mock'ов используется **проверка поведения**, т.е. описываются ожадиния на этапе Setup и далее на этапе Verify проверяется выполнение этих ожиданий. И только для экземпляра Order мы используем проверку состояния с помощью asserts. А если тестируемый метод не изменяет состояние Order'а, то данная проверка вообще не нужна.

Различия между Mock'ами и Stub'ами

В первом, описанном выше, подходе к тестированию используется реальный объект Warehouse, а во втором – Mock Warehouse'а, который, конечно же, не является реальным объектом Warehouse. Использование Mock'ов – это один из способов подменять реальный объект в тесте, но есть и другие способы.

Существует понятие Test Double (тестовый дублер), которое используется как обобщение для всех видов объектов, используемых в тестах вместо раельных объектов. Существуют следующие виды Test Double'ов:

- **Dummy** – это объект, который передается, но никогда не используется. Используется для заполнения списка параметров метода;

- **Fake** – это объект, который имеет реальную реализацию, но из-за определенных ограничений не может быть использован в Prod'е. Хорошим примером является встроенная (in-memory) база данных;
- **Stub** – это объект, возвращающий предопределенные ответы на вызовов методов, используемых в тесте и обычно никак не отвечающий на вызовы остальных методов;
- **Spy** – это Stub, который ко всему прочему записывает информацию о том, как его методы вызывались;
- **Mock** – это объект с запрограммированными ожиданиями того, как методы должны быть вызваны.

Из всего этого списка только Mock'и используются для проверки поведения, тогда как остальные используются в сочетании с проверкой состояния. Mock'и на самом деле ведут себя так же, как и остальные Test Double'ы на этапе Exercise, так как необходимо заставить SUT поверить, что он взаимодействует с настоящими Collaborator'ами. Однако Mock'и отличаются от остальных на этапах Setup и Verify.

Чтобы глубже исследовать Test Double'ы необходимо расширить наш пример. Многие разработчики используют Test Double'ы в случаях, если с реальным объектом неудобно работать. Более распространенным случаем для использования Test Double'ов может быть ситуация, в которой необходимо отправить email клиенту, если возникают проблемы с его заказом. Проблема в том, что мы не хотим отправлять email'ы клиентам во время тестирования. Поэтому вместо реального Email-сервиса мы будем использовать Test Double, которым мы можем манипулировать в тестовых сценариях.

Здесь мы можем начать видеть разницу между Mock'ами и Stub'ами. Если бы мы писали тест для описанного выше поведения, мы могли бы использовать следующий Stub:

```
public interface MailService {
  public void send(Message msg);
}

public class MailServiceStub implements MailService {
  private List<Message> messages = new ArrayList<Message>();
  public void send(Message msg) {
    messages.add(msg);
  }
  public int numberSent() {
    return messages.size();
  }
}
```

И далее мы можем использовать проверку состояния этого Stub'а в тесте:

```
 public void testOrderSendsMailIfUnfilled() {
    Order order = new Order(TALISKER, 51);
    MailServiceStub mailer = new MailServiceStub();
    order.setMailer(mailer);
    order.fill(warehouse);
    assertEquals(1, mailer.numberSent());
 }
```

С использованием Mock'ов этот тест будет выглядеть иначе:

```
 public void testOrderSendsMailIfUnfilled() {
    Order order = new Order(TALISKER, 51);
    Mock warehouse = mock(Warehouse.class);
    Mock mailer = mock(MailService.class);
    order.setMailer((MailService) mailer.proxy());

    mailer.expects(once()).method("send");
    warehouse.expects(once()).method("hasInventory")
      .withAnyArguments()
      .will(returnValue(false));

    order.fill((Warehouse) warehouse.proxy());
 }
```

В обоих случаях используются Test Double'ы вместо реального MailService'а, однако различие в том, что используя Stub проверяется состояние, в то время как используя Mock проверяется поведение.

Так же, чтобы проверить состояние Stub'а, в нем необходимо создавать дополнительные методы. В результате Stub реализует интерфейс MailService, но добавляет дополнительный метод `.numberSent()`, необходимый для тестирования.

Mock’и всегда используются для проверки поведения, в то время как Stub может использоваться для проверки как поведения так и состояния. Stub, проверяющий поведение называется Spy.

Classicist и Mockist стили TDD

В соответствии с **Classicist TDD** стилем в тестах используются реальные объекты. Test Double’ы же используются только, если с реальным объектом неудобно работать. Таким образом приверженец данного стиля использовал бы реальный объект Warehouse и Test Double для MailService’а (вид Test Double’а значения не имеет).

Приверженец **Mockist TDD** стиля предпочтет использовать Mock’и для всех Collaborator’ов.

Различия

TDD-процесс

Согласно Mockist TDD стилю, разработка пользовательской истории начинается с написания теста для внешней части системы, сделав SUT’ом некоторый объект интерфейса (Controller, Message Queue Listener и т.д.). Далее продумывая ожидания от Collaborator’ов, мы исследуем взаимодействие между SUT и его Collaborator’ами.

После запуска этого теста ожидания от Mock'ов предоставляют руководство для следующего шага и отправную точку для дальнейшего TDD-процесса. Далее мы превращаем каждое ожидание в тест для соответствующего Collaborator’а и повторяем процесс, продвигаясь в глубь системы. Именно по-этому данный стиль

еще называют Outside-in TDD. Это очень структурированный и контролируемый подход к TDD, который хорошо подходит для систем со слоистой архитектурой.

Classicist TDD не предоставляет такого же руководства. Однако мы можем следовать тому же подходу, используя Stub'ы вместо Mock'в. Для этого, когда нам нужно что-то от Collaborator'а мы просто жестко кодируем ответ, который требует SUT в данном тесте. Затем, когда тест «зеленый», мы заменяем жестко закодированный ответ необходимой реализацией.

Также Classicist TDD может делать и другие вещи, например в рамках подхода Middle-out, согласно которому мы рассматриваем функциональную возможность и определяем, что нам нужно иметь в предметной области, чтобы эта функциональная возможность работала. И когда наша доменная модель готова, мы надстраиваем верхние/внешние слои (UI и т.д.). Многим нравится такой подход, потому что сначала он фокусирует внимание на модели предметной области, что помогает предотвратить утечку логики предметной области в пользовательский интерфейс.

Необходимо подчеркнуть, что оба стиля Mockist и Classicist TDD способствуют реализации функциональной возможности слой за слоем, не начиная один слой, пока другой не будет завершен. Оба стиля соответствуют Agile методологии и предпочитают короткие итерации. В результате они работают создавая фукцинальную возможность за функциональной возможностью, а не слой за слоем.

Создание Fixture'ы

Используя Classicist TDD нам необходимо создавать как SUT, так и все Collaborator'ы, которые требуются SUT в рамках конкретного теста. В реальных тестах часто приходится создавать много объектов, которые должны быть созданы перед тестом и удалены после теста.

Напротив, используя Mockist TDD нам необходимо создать SUT и Mock'и для всех его Collaborator'ов в рамках конкретного теста.

Изолированность тестов

Если в системе покрытой тестами согласно Mockist TDD появится ошибка, то «покраснеют» только те тесты, которые относятся к SUT, которая содержит эту ошибку.

И напротив, если в системе покрытой тестами согласно Classicist TDD появится ошибка, то могут сломаться любые тесты, которые используют SUT (как тестируемый объект или как Collaborator), что в результате может привести к волне «покрасневших» тестов.

Зависимоть между тестами и реализацией

Когда мы пишем тест согласно Mockist TDD, мы тестируем исходящие вызовы SUT, чтобы гарантировать, что он коммуницирует должным образом со своими Collaborator'ами. Тест согласно Classicist TDD проверяет только конечное состояние, а не то, как это состояние было получено. Таким образом, тесты согласно Mockist TDD больше связаны с реализацией тестируемого метода SUT, и, следовательно, изменения в вызовах Collaborator'ов, как правило, ломают такие тесты.

Зависимость между тестами и реализацией также препятствует рефакторингу, поскольку изменения в реализации с гораздо большей вероятностью нарушат тесты при Mockist TDD, чем при Classicist TDD.

Стиль приектирования

Как упоминалось ранее, приверженцы Mockist TDD используют Outside-in подход, а приверженцы Classicist TDD, в свою очередь, предпочитают Middle[Domain model]-out подход.

Практики Mockist TDD предпочитают создавать методы, которые принимают объект для сбора информации, вместо методов, возвращающих значение. Так же преверженцы этого стиля TDD призывают следовать «Закону Деметры», т.е. избегать цепочек вызовов `.getThis().getThat().getTheOther()`.

Также Mockist TDD способствует следованию «Tell, Don’t Ask» принципу, согласно которому необходимо просить объект, инкапсулирующий данные, выполнить определенные действия над этими данными, вместо того чтобы запрашивать данные из этого объекта и выполнять действия над ними в клиентском коде. Т.е. следование данному принципу позволяет избавиться от огромного количества getter’ов.

Использование Classicist TDD и, соответственно, проверки состояния, приводит к созданию getter’ов, которые нужны только в тестах. Mockist TDD и проверка поведения позволяет избежать данной проблемы.

Mockist TDD призывает к использованию ролевых интерфейсов, т.е. для каждого Collaborator’а сначала создается интерфейс, который является основой для создания Mock’а. И далее создается реализация, когда TDD-процесс доходит до данного Collaborator’а.