Classicist And Mockist TDD Comparison

Перевод статьи: [Mocks Aren't Stubs](https://martinfowler.com/articles/mocksArentStubs.html) (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’a, Order становится заполненным, а количество товара в Warehouse уменьшается на соответствующую сумму;
* 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’a) и частично в методе .test…() (инициализация Order’a). 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, правельности функционирования тестируемого метода. В данном случае нет явного этапа Teardown, т.к. сборщик мусора в Java делает это для 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’a – технически, это экземпляр класса Mock.

Во второй части настраиваются ожидания, которые указывают какие методы и как должны быть вызваны у Mock’a 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’a, 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 обопщение 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’e. Хорошим примером является встроенная (in-memory) база данных;
* **Stub** – это объект, возвращающий предопреденные ответы на вызовов методов, которые используются 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’a 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());

}