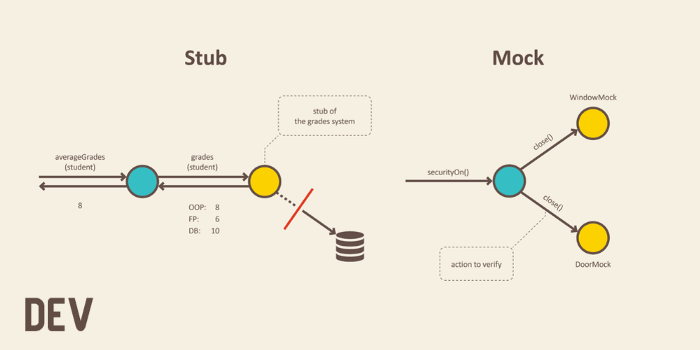
**1. Провести Research на тему “Что такое Stubs, Mocks”.**

**2. Дать примеры, когда Stubs и Mocks могут использоваться в тестах.**

1. **Провести Research на тему “Что такое Stubs, Mocks”.**  
   Очень часто наш код (функция, модуль) имеют внешние зависимости. Внешняя зависимость — это все, что делает ваши тесты не правдивыми и сложно-поддерживаемыми. Файловая система — зависимость: структура каталогов может быть другой на другой машине. База данных — зависимость, ее может не быть на другой машине. Веб-сервис — зависимость: может не быть интернета или может присутствовать фаервол и.т.д

Если на вопрос: «будет ли этот компонент вести себя так же на другой машине?» вы отвечаете нет, то его необходимо “подменить” и тут вам на помощь как раз придут стабы и моки. Но есть и другая сторона медали, когда разработчик начинает увлекаться и приходит к тому, что подменяет вообще все. Соответственно тесты перестают проверять само приложение и начинают тестировать стабы, моки. Это в корне не верно. Если «живых» реализаций в тесте нет, то этот тест не тестирует ничего.

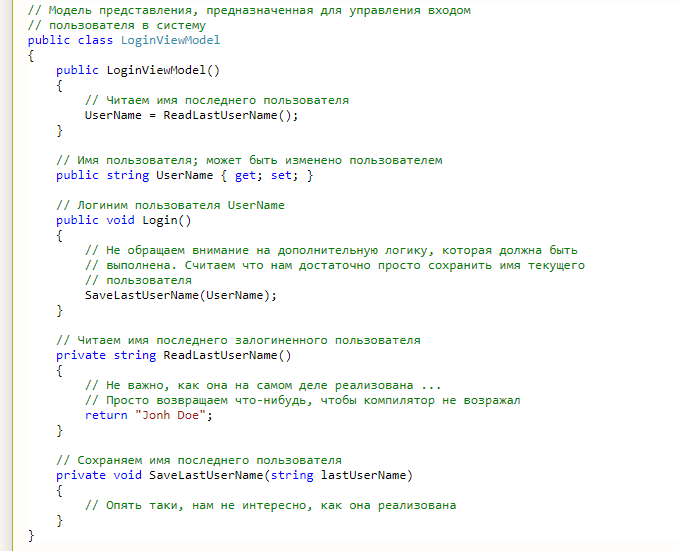
Иногда эти термины stubs и mock путают: разница в том, что стаб ничего не проверяет, а лишь имитирует заданное состояние. А мок — это объект, у которого есть ожидания. Например, что данный метод класса должен быть вызван определенное число раз. Иными словами, ваш тест никогда не сломается из-за «стаба», а вот из-за мока может.

С технической точки зрения это значит, что, используя стабы, мы проверяем состояние тестируемого класса или результат выполненного метода. При использовании мока мы проверяем, соответствуют ли ожидания мока поведению тестируемого класса. Также лучше использовать не более одного мока на тест. Иначе с высокой вероятностью вы нарушите принцип «тестировать только одну вещь». При этом, в одном тесте может быть сколько угодно стабов или же мок и стабы.  
  
2. **Дать примеры, когда Stubs и Mocks могут использоваться в тестах.**

Существует категория классов, которые тестировать весьма просто. Если класс зависит только от примитивных типов данных и не имеет никаких связей с другими бизнес-сущностями, то достаточно создать экземпляр этого класса, «пнуть» его некоторым образом путем изменения свойства или вызова метода и проверить ожидаемое состояние.

Это самый простой и эффективный способ тестирования, и любой толковый дизайн отталкивается от подобных классов, которые являются «строительными блоками» нижнего уровня, на основе которых затем уже строятся более сложные абстракции. Но количество классов, которые живут в такой «изоляции» не много по своей природе. Даже если мы по нормальному выделили всю логику по работе с базой данных (или сервисом) в отдельный класс (или набор классов), то рано или поздно появится кто-то, кто эти классы будет использовать для получения более высокоуровневого поведения и этого «кого-то» тоже нужно будет тестировать.

Но для начала давайте рассмотрим более типичный случай, когда логика по работе с базой данных или внешним сервисом, а также логика обработки этих данных сосредоточена в одном месте.

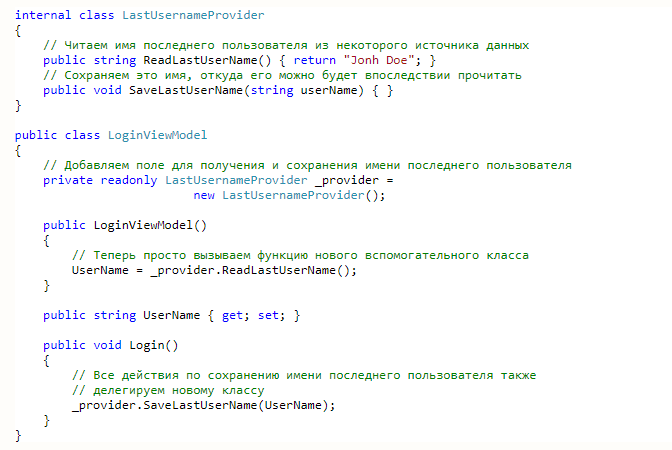
  
  
Когда речь заходит о тестировании подобных классов, то обычно эта вью-модель помещается на форму, которая затем тестируется руками Если вместо вью-модели подобное смешивание логики происходит при реализации серверных компонент, то они тестируются путем создания простого консольного приложения, которое будет вызывать необходимые высокоуровневые функции, тестируя, таким образом, весь модуль целиком. В обоих случаях такой вариант тестирования нельзя назвать очень уж автоматическим.

ПРИМЕЧАНИЕ

Не нужно бросать в меня камнями с криками «Да кто сегодня вообще такую хрень написать можно? Ведь уже столько всего написано о вреде такого подхода, да и вообще, у нас есть юнити-шмунити и другие полезности, так что это нереальный баян двадцатилетней давности!». Кстати, да, это баян, но, во-первых, речь не юнитях и других контейнерах, а о базовых принципах, а во-вторых, подобное «интеграционное» тестирование все еще невероятно популярно, во всяком случае, среди многих моих «зарубежных» коллег.

Создания «швов» для тестирования приложения

Даже если не задумываться о том, какое количество новомодных принципов проектирования нарушает наша вью-модель, четко видно, что ее дизайн несколько … убог. Ведь даже если проектировать старым дедовским бучевским методом, то становится понятно, что всю работу по сохранению имени последнего пользователя, логику по работе с базой данных (или другим внешним источником данных) нужно спрятать подальше с глаз долой и сделать это «проблемой» кого-то другого и использовать уже этого «кого-то» в качестве «строительного блока» для получения более высокоуровневого поведения:

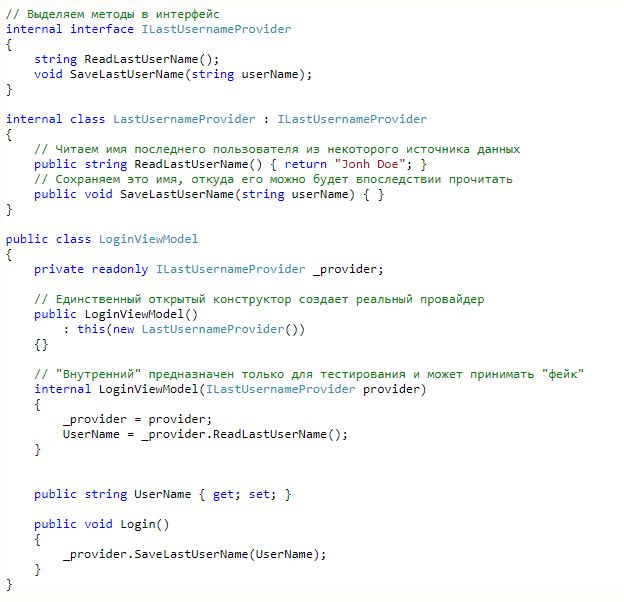
  
Пока что написание модульного теста все еще остается затруднительным, но становится понятным, как можно достаточно просто «подделать» реальную реализацию класса LastUsernameProvider и сымитировать нужное для нас поведение. Достаточно выделить методы этого класса в отдельный интерфейс или просто сделать их виртуальными и переопределить в наследнике. После чего останется лишь «прикрутить» нужный нам объект в нашу вью-модель.

ПРИМЕЧАНИЕ

Честно говоря, я не большой фанат изменений в дизайне только ради «тестируемости» кода. Как показывает практика, нормальный ОО дизайн либо уже является достаточно «тестируемым» или же требует лишь минимальных телодвижений, чтобы сделать его таковым. Некоторые дополнительные мысли по этому поводу можно найти в заметке «Идеальная архитектура».

Даже не прибегая ни к каким сторонним библиотекам для «инджекта» зависимостей мы можем сделать это самостоятельно несколько простыми способами. Нужную зависимость можно передать через дополнительный конструктор, через свойство или создать фабричный метод, который будет возвращать интерфейс ILastUsernmameProvider.

Давайте рассмотрим вариант с конструктором, который является довольно простым и популярным (при небольшом количестве внешних зависимостей он работает просто прекрасно).

  
  
Поскольку дополнительный конструктор является внутренним (internal), то он доступен только внутри этой сборке, а также «дружеской» сборке юнит-тестов. Конечно, если тестируемые классы являются внутренними, то проблемы не будет ни какой, но поскольку все «клиенты» внутреннего класса находятся в одной сборке, то и контролировать их проще. Подобный подход, основанный на добавлении внутреннего метода для установки «фальшивого» поведения является разумным компромиссом упрощения тестирования кода, не налагая ограничения на использования более сложных механизмов управления зависимостями, типа IoC контейнеров.

ПРИМЕЧАНИЕ

Одним из недостатков при работе с интерфейсами является падение читабельности, поскольку не понятно, сколько реализаций интерфейса существует и где находится реализация того или иного метода интерфейса. Такие инструменты, как Решарпер существенно смягчают эту проблему, поскольку поддерживают не только навигацию к объявлению метода (Go To Declaration), но также и навигацию к реализации метода (Go To Implementation):

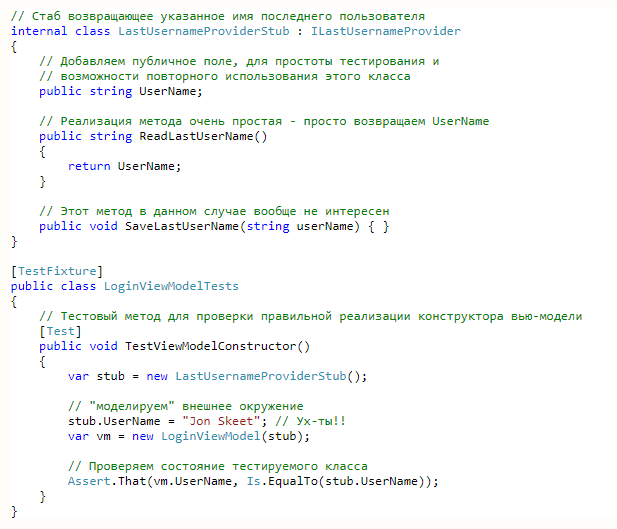
Проверка состояния vs проверка поведения

Теперь давайте попробуем написать юнит-тест вначале для конструктора класса LoginViewModel, который получает имя последнего залогиненного пользователя, а потом юнит-тест для метода Login, после выполнения которого, имя последнего пользователя должно быть сохранено.

Для нормальной реализации этих тестов нам нужна «фейковая» реализация интерфейса, при этом в первом случае, нам нужно вернуть произвольное имя последнего пользователя в методе ReadLastUserName, а во втором случае – удостовериться, что вызван метод SaveLastUserName.

Именно в этом и отличаются два типа «фейковых» классов: стабы предназначены для получения нужного состояния тестируемого объекта, а моки применяются для проверки ожидаемого поведения тестируемого объекта.

Стабы никогда не применяются в утверждениях, они простые «слуги», которые лишь моделируют внешнее окружение тестового класса; при этом в утверждениях проверяется состояние именно тестового класса, которое зависит от установленного состояния стаба.

  
  
У моков же другая роль. Моки «подсовываются» тестируемому объекту, но не для того, чтобы создать требуемое окружение (хотя они могут выполнять и эту роль), а прежде всего для того, чтобы потом можно было проверить, что тестируемый объект выполнил требуемые действия. (Именно поэтому такой вид тестирования называется behavior testing, в отличие от стабов, которые применяются для state-based testing).  
  
