Объекты заглушки (MockObjects) и их использование (цели, применимость)

Любой человек, начинающий изучать юнит тестирование рано или поздно сталкивается со следующей проблемой: согласно идеологии юнит тестирования каждый класс должен быть протестирован отдельно. В простых книжных примерах это выглядит легко, но на практике между многими классами имеется тесная взаимосвязь. Для разрыва такой связи как раз и может служить. А для облегчения тестирования используются библиотеки — библиотеки, создающие объекты-заглушки.

MockObjects - это автоматически генерируемые заглушки, которые позволяют имитировать некоторую функциональность и управлять ею прямо из теста. MockObjects (мок) создаются на базе какого-то интерфейса или класса. В этом случае мок поддерживает все методы эмулируемого интерфейса, плюс методы по настройке поведения мока.

***Управление MockObjects включает:***

* возврат определенных значений из методов при наступлении определенных условий;
* ожидание определенных выводов методов, определенное количество раз;
* учет порядка вызовов методов.

Преимущество разработки с мок-объектами заключается в том, что они позволяют уже на ранних этапах разрабатывать программное обеспечение, имея на руках только интерфейсы, но не имея реальные классы. MockObjects также позволяют заметно ускорить исполнение тестов, изолируя ресурсоемкие участки кода. Однако не стоит забывать, что нельзя слишком увлекаться MockObjects, т.к. это может привести к неадекватности тестов и реализации.

***Когда использовать Mock Objects***

Мнимые объекты или моки (Mock Objects) используются:

* для изоляции тестируемого кода,
* для тестирования делегирования.

При использовании Mock Objects для изоляции они заменяют реальные объекты, используемые в тестируемом коде. При этом в тесте появляется возможность контролировать используемые внутри тестируемого кода (делегируемые) объекты, а именно указывать, какой результат они должны возвращать при вызове тех или иных методов с теми или иными параметрами.

При использовании Mock Objects для тестирования делегирования мы можем генерировать ожидания тех или иных вызовов в результате работы тестируемого кода. Если эти вызовы не будут совершены, Mock Objects сгенерируют ошибки.

***Причины необходимости использования мок-объектов вместо реальных классов:***

* Тесты выполняются слишком медленно. Медленный тест - это [запах](http://wiki.agiledev.ru/doku.php?id=tdd:smells). Обычно это связано с использованием внешних, не слишком быстрых ресурсов.
* Тесты зависят от внешних ресурсов, которые вы не всегда можете контролировать.
* Тестирование при помощи реальных классов не может быть осуществлено принципиально, например, если у вас еще их нет, а их подготовкой занимаются другие разработчики.
* При необходимости тестирования поведения при исключительных ситуациях, в случаях, когда генерация исключений потребует больших усилий. Дать здесь ссылку на статью про ручное создание мок-объектов.
* При необходимости проверки правильности взаимодействия тестируемого объекта с другими объектами, в случаях, когда это взаимодействие составляет суть работы класса. Например, при проверке правильности кеширования информации, нам требуется удостовериться, что информация будет затребована из источника именно 1 раз.

**Фреймворки, которые поддерживают Mock Objects**

Для PHP мок-объекты поддерживаются SimpleTest, а также PHPUnit2.

Для .Net мок-объекты поддерживаются Nunit, Visual Studio Team System, MbUnit, Mногофункциональным Mock Фреймворком является Typemock Isolator, также одним из лидирующих и не менее функциональным является Rhino Mock.

### Передача зависимого объекта через конструктор. Constructor Injection

Constructor injection - самый распространенный вид внедрения зависимостей и самый простой. Большинство ознакомительных примеров по использованию мок-объектов используют именно этот способ передачи моков в тестируемые классы. Действительно, все наглядно и просто: создаем мок-объект, передаем его в конструктор - и все, мы изолировали класс. Класс становится менее связанным, увеличивается гибкость, все довольны:

|  |
| --- |
| class SomeClass()  {  protected $\_delegated\_object;    public function \_\_construct(InterfaceA $delegated\_object){  $this->\_delegated\_object = $delegated\_object;  }    public function someMethodToBeTested(){  [...some code collecting $params...]  $this->\_delegated\_object->doSomething($params);  [..any other code...]  }  } |

Пример теста:

|  |
| --- |
| Mock::generate('InterfaceA', 'MockDelegatedClass');    class SomeClassTest extends TestCase{    function testSomeMethodToBeTested() {  $delegated\_object = new MockDelegatedClass();  $params = [array](http://www.php.net/array)('some expected params');  $delegated\_object->expectOnce('doSomething', [array](http://www.php.net/array)($params));    // Passing delegated object in tested class constructor  $some\_class = new SomeClass($delegated\_object);  $some\_class->someMethodToBeTested();  }  } |

Еще одно достоинство Constructor injection заключается в том, что все зависимости класса прописываются явно. То есть, взглянув на один лишь конструктор класса, мы уже понимаем, какими ресурсами остальной системы он пользует. Это несомненно плюс на этапе изучения системы.