Mockito: Как создаются Mock'и

Работа с Mockito начинается, как ни странно, с класса org.mockito.Mockito. Этот класс содержит в себе несколько статичных методов, которые обычно и импортируются в тест-класс. Первое же, что бросается в глаза, это огромные JavaDoc комментарии. Разработчики решили вести настоящий Tutorial прямо в коде. Обнаружить код среди такого обилия комментариев без использования инструментов IDE весьма непросто. Класс Mockitoделегирует все вызовы классу org.mockito.internal.MockitoCore. Если класс Mockito - точка входа внешнего API, то MockitoCore - точка входа во внутреннюю реализацию проекта.

Создание mock объекта начинается с вызова метода mock(Class<T> typeToMock, MockSettings settings), где

typeToMock - класс типа будущего mock’а

settings - настройки mock’а

Типичный вызов создания мока выглядит так:

```
1 Mockito.mock(SomeClass.class)
```

При этом настройки mock’а инициализируются по умолчанию с помощью метода Mockito.withSettings(), который возвращает реализацию интерфейсов настроек. А интерфейсов класса с настройками mock объектов два - org.mockito.MockSettings и org.mockito.mock.MockCreationSettings. Первый определяет методы установки параметров mock’ов, второй определяет интерфейс для доступа к этим параметрам без возможности их изменения. Соответственно, запись значений в объект производится через первый интерфейс, а все остальные классы работают с параметрами через второй интерфейс, чтобы ничего в нем случайно не изменить. Настройки можно задавать вручную при создании mock объекта. В частности можно указать произвольное имя mock’а, чтобы логи выполнения тестов стали понятнее.

При создании mock’а, все его методы stubb’ируются. Результат вызова метода определяется Answer объектом. Для важных методов тестирования мы задаем Answer объект вручную через конструкцию Mockito.when(Matcher).thenAnswer(Answer), для других же объектов вызывается Answer по умолчанию. Конкретная реализация ответа также определяется в настройках mock’а и её можно переопределить. В настройках по умолчанию используется org.mockito.internal.stubbing.defaultanswers.GloballyConfiguredAnswer. Этот ответ в свою очередь делегирует запрос реализации ответа, которая определена в глобальном объекте конфигурации, который был рассмотрен в статье Mockito: Обработка аннотаций. В конфигурации же определена реализация org.mockito.internal.stubbing.defaultanswers.ReturnsEmptyValues, которая возвращает пустые значения для разных типов объектов.

Помимо GloballyConfiguredAnswer, в Mockito, реализовано ещё несколько реализаций ответов, ссылки на эти объекты хранятся в Enum’е org.mockito.Answers. Также в Mockitoещё имеются и внутренние реализации ответов, такие как ReturnsEmptyValues и ReturnsMoreEmptyValues. Последний делегирует запрос ReturnsEmptyValues и дополняет его логику определением еще нескольких типов данных. Дело в том, что ReturnsEmptyValues возвращает null для всех объектов, кроме определенных реализаций коллекций, примитивных типов и их оберток. В какой-то момент разработчики решили поддержать еще строки и массивы, но так как кто-то, скорее всего, успел написать тесты с учетом этой специфики, то просто обновить существующую реализацию уже недостаточно. Пришлось создавать новый тип ответа ReturnsMoreEmptyValues. Новый ответ сейчас используется в реализации ответа ReturnsSmartNulls, который вместо null пытается создать и вернуть mock’реализацию данного типа.

Есть и другие внутренние реализации ответов, но в них нет ничего интересного. Так же имеется возможность задать в настройках объекты - слушатели, которые будут оповещены в момент вызова метода. Эти объекты реализуют интерфейс org.mockito.MockSettings.InvocationListener.

Помимо прочего, базовый класс настроек имеет метод самопроверки, который в аргументе принимает класс объекта, mock которого планируется создать, и проверяет возможность создания mock'а с данными настройками. В случае успешной проверки создается новый экземпляр объекта настроек, куда устанавливается объект имени mock'а и обработанные настройки, определенные раннее. Объект имени мока с интерфейсом org.mockito.internal.util.MockName служит для хранения имени мока и для выполнения некоторых операций по обработке имени мока.

Основной объект системы, выполняющий создание объекта, реализует интерфейс org.mockito.plugins.MockMaker, но напрямую с этим объектом работает только утилитарный класс org.mockito.internal.util.MockUtil. Именно метод createMock класса MockUtil вызывается в MockitoCore#mock. В первую очередь создается объект с интерфейсом org.mockito.invocation.MockHandler, который перехватывает вызов стаббированных методов в mock объекте. Объекты перехватчиков создаются фабрикой org.mockito.internal.handler.MockHandlerFactory. Создается базовая реализация обработчика org.mockito.internal.handler.MockHandlerImpl и оборачивается разными wrapper'ами (паттерн Wrapper) для навешивания дополнительного функционала. На данный момент используются следующие обертки:

org.mockito.internal.handler.NullResultGuardian для предотвращения возвращения null'овых значений из методов, которые возвращают примитивные типы или их обертки.

org.mockito.internal.handler.MockHandlerFactory оповещает слушателей, объявленных в настройках о вызове метода.

Далее вызывается метод MockMaker#createMock, куда передаются настройки и MockHandler.
Объект MockMaker'а создается в статичном поле класса MockUtil с помощью ClassPathLoader, который уже использовался для загрузки пользовательской реализации класса конфигурации.
Метод ClassPathLoader#getMockMaker возвращает заранее загруженную реализацию MockMaker'а. Так же как и с объектом конфигурации, имеется возможность использовать свою реализацию MockMaker'а, но принцип загрузки отличается:

```
1  static <T> List<T> loadImplementations(Class<T> service) {
2      ClassLoader loader = Thread.currentThread().getContextClassLoader();
3      if (loader == null) {
4          loader = ClassLoader.getSystemClassLoader();
5      }
6
7      Enumeration<URL> resources;
8      try {
9          resources = loader.getResources("mockito-extensions/" +
10 service.getName());
11     } catch (IOException e) {
12         throw new MockitoException("Failed to load " + service, e);
13     }
14
15     List<T> result = new ArrayList<T>();
16     for (URL resource : Collections.list(resources)) {
17         InputStream in = null;
18         try {
19             in = resource.openStream();
20             for (String line : readerToLines(new InputStreamReader(in, "UTF-8"))) {
21                 String name = stripCommentAndWhitespace(line);
22                 if (name.length() != 0) {
23                     result.add(service.cast(loader.loadClass(name).newInstance()));
24                 }
25             }
26         } catch (Exception e) {
27             throw new MockitoConfigurationException(
28                     "Failed to load " + service + " using " + resource, e);
29         } finally {
30             closeQuietly(in);
31         }
32     }
33     return result; }
```

С помощью ClassLoader’а загружается файл с именем класса, переданного в аргументе service. На данный момент в качестве service используется класс MockMaker. Файл ищется в папке с именем “mockito-extensions”. В файле указываются полные имена классов - один класс на строчку. Далее эти классы загружаются и создаются их объекты. Несмотря на возможность загрузить несколько MockMaker'ов, будет применен только первый найденный. Если пользовательских классов нет, то загружается MockMaker по умолчанию org.mockito.internal.creation.CglibMockMaker.

Создание прокси классов, а mock объекты фактически ими и являются, по умолчанию обеспечивается библиотекой CGLib. Исходники этой библиотеки включены в проект, чтобы не заморачиваться с версией проекта и перепаковкой исходников. Конечно, Maven облегчил бы задачу и структура проекта стала бы яснее, но что сделано, то сделано. Разработчики не модифицируют эти исходники и в целом это запрещается. CGLib - библиотека, позволяющая создавать, расширять классы и интерфейсы в runtime’е.

Внутри MockMaker‘а обработчик вызова стаббированных методов - MockHandlerоборачивается внутрь класса MethodInterceptorFilter, реализующего интерфейс MethodInterceptor, который является частью CGLib библиотеки.

Как ни странно, логика создания объектов с использованием CGLib размыта и описана в неком утилитарном классе org.mockito.internal.creation.jmock.ClassImposterizer, который уже использовался при проверке настроек mock’объекта. Этот класс позаимствован из библиотеки jMock.

#Thanks to jMock guys for this handy class that wraps all the cglib magic.

Кстати, это не единственный класс в Mockito с похожим комментарием.

Этот класс вносит разнородность в Mockito, что очень некрасиво. Например, для создания экземпляров классов используется библиотека objenesis. В Mockito же эта процедура выполняется напрямую, с помощью рефлексии.

В первую очередь устанавливается видимость конструкторов, класс для которого создается Mock объект. Это мы уже проходили. Решается парой методов рефлексии. Далее создается класс proxy объекта. Эта работа выполняется с помощью CGLib класса Enhancer. Этот класс конструирует класс для будущего proxy по параметрам, которые мы в него передали.

```
1  private Class<?> createProxyClass(Class<?> mockedType, Class<?>...interfaces) {
2      if (mockedType == Object.class) {
3          mockedType = ClassWithSuperclassToWorkAroundCglibBug.class;
4      }
5
6      Enhancer enhancer = new Enhancer() {
7          @Override
8          @SuppressWarnings("unchecked")
9          protected void filterConstructors(Class sc, List constructors) {
10             // Don't filter
11         }
12     };
13
14enhancer.setClassLoader(SearchingClassLoader.combineLoadersOf(mockedType));
15     enhancer.setUseFactory(true);
16     if (mockedType.isInterface()) {
17         enhancer.setSuperclass(Object.class);
18         enhancer.setInterfaces(prepend(mockedType, interfaces));
19     } else {
20         enhancer.setSuperclass(mockedType);
21         enhancer.setInterfaces(interfaces);
22     }
23     enhancer.setCallbackTypes(new Class[]{MethodInterceptor.class, NoOp.class});
24     try {
25         return enhancer.createClass();
26     } catch (CodeGenerationException e) {
27     }
   }
```

Я опустил некоторые моменты по управлению секьюрностью классов и обработки исключения. Они незначительны.

Для начала решаем проблему с багом CGLib, который отказывается обрабатывать класс Object, так как где-то, видимо, завязывается на родителе класса. Т.е. если тип mock объекта - Object, то мы создаем пустой объект ClassWithSuperclassToWorkAroundCglibBug, с которым дальше работаем. Затем создаем объект Enhancer'a, и ему передаются classloader’ы классов будущих mock’ов, обернутых во внутренний класс CGLib для работы со всеми найденными classloader’ами как с одним. Класс, помимо прочего, будет реализовывать интерфейс Factory, который определяет методы установки обработчиков вызовов методов и инстанцирования этих объектов. Это внутренняя особенность CGLib. Далее указываем Enhancer'у типы классов и интерфейсов, которые он должен будет унаследовать, и типы обработчиков-перехватчиков методов классов. В данном случае это стандартные классы MethodInterceptor, который позволяет определять обработчики, и NoOp, который передает управление методам объекта напрямую - нужно для Spy моков.

Следующим шагом создается объект прокси:

```
1  private Object createProxy(Class<?> proxyClass, final MethodInterceptor
   interceptor) {
2      Factory proxy = (Factory) objenesis.newInstance(proxyClass);
3      proxy.setCallbacks(new Callback[] {interceptor,
4  SerializableNoOp.SERIALIZABLE_INSTANCE });
5      return proxy;
   }
```

Как видно, objensis намного упрощает создание объектов. У разработчиков, видимо, не доходят руки, чтобы переписать старый код с использованием этой библиотеки. В качестве обработчиков событий указываем созданный ранее InternalMockHandler и пустую реализацию интерфейса NoOp.

Теперь остается только оповестить все компоненты системы о событии создания мока:

```
1
    mockingProgress.mockingStarted(mock, typeToMock);
```

и вернуть его в класс-тест.

Хочу заметить, что в коде немало хардкода. Немало мест, где метод принимает параметры по общему интерфейсу, а внутри метода проверяется объект на соответствие определенному типу, который реализует этот интерфейс:

```
1  private InternalMockHandler cast(MockHandler handler) {
2      if (!(handler instanceof InternalMockHandler)) {
3          throw new MockitoException("At the moment you cannot provide own
4  implementations of MockHandler." +
5                  "\nPlease see the javadocs for the MockMaker interface.");
6      }
7      return (InternalMockHandler) handler;
   }
```

Но стоит отдать разработчикам должное за заботу о пользователях - расширяя проект, они стараются не менять старую часть API. Обидно было бы переписывать кучу тестов при переходе на новую версию *Mockito*.