# Лекция 14. Тестирование в Java

# Библиотека Mockito

Mockito – это библиотека создания заглушек для облегчения процессов тестирования отдельных модулей программы.

Для начала работы с этой библиотекой необходимо загрузить ее jar c официального сайта <https://code.google.com/p/mockito/> и добавить в качестве зависимости в проект.

Основным классом для работы с фреймворком является класс ***org.mockito.Mockito***, который содержит множество статических методов. Для удобства работы с методами данного класса, его лучше импортировать статически (import static org.mockito.Mockito.\*), что позволит вызывать методы класса без указания имени класса.

Класс ***org.mockito.Mockito*** содержит следующие методы (все методы можно найти в документации на класс):

1. mock(Class mockToClass) – метод создания объекта-заглушки типа переданного класса или интерфейса.
2. verify(T mock) – метод проверки определенного поведения переданной заглушки.

|  |
| --- |
| **import** **static** org.mockito.Mockito.\*;  **import** java.util.List;  **import** org.junit.Test;  **public** **class** TestClass {  **@Test**  **public** **void** testVerify() {  // создание объекта-заглушки типа List  List<Object> mockedList = *mock*(List.**class**); //метод Mockito.mock()  // наполнение заглушки данными и поведением  mockedList.add("one");  mockedList.add("two");  mockedList.clear();  // проверка, что в список добавлялся  // объект типа String со значением “two”  *verify(mockedList).add("two");*  // проверка, что список был очищен  *verify*(mockedList).clear();  }  } |
| Тест выполняется успешно. Однако, если модифицировать метод проверки выполнения действий объекта mockedList, то тест не будет пройден успешно: |
| **import** **static** org.mockito.Mockito.\*;  **import** java.util.List;  **import** org.junit.Test;  **public** **class** TestClass {  **@Test**  **public** **void** testVerify() {  …  // verification  *verify*(mockedList).add("ааа");  *verify*(mockedList).add("two");  *verify*(mockedList).clear();  }  } |
|  |
| Тест не пройден, так как в объект-заглушку никогда не добавлялся объект типа String со значением “aaa”. |

1. when(T methodCall) – «запоминает» поведение, которое должно быть реализовано при вызове на объекте-заглушке указанного метода.

|  |
| --- |
| **public** **class** TestClass {  **@Test**  **public** **void** testStubbing() {  LinkedList mockedList = *mock*(LinkedList.**class**);  // задание поведения:  // когда необходимо вернуть элемент в позиции 0, вернуть «first»  *when*(mockedList.get(**0**)).thenReturn("first");  // когда необходимо вернуть элемент в позиции 1,вызывть исключение  *when*(mockedList.get(**1**)).thenThrow(**new** RuntimeException());  // following prints "first"  System.***out***.println(mockedList.get(**0**));  // following throws runtime exception  System.***out***.println(mockedList.get(**1**));  // following prints "null" because get(999) was not stubbed  System.***out***.println(mockedList.get(**999**));  }  } |
| Во время выполнения кода, при вызове mockedList.get(1) будет генерироваться исключение RuntimeException. |

1. Методы times(), never(), atLeastOnce(), atLeast(int number), atMost() позволяют произвести проверку количества вызовов методов с определенными параметрами.

|  |
| --- |
| **public** **class** TestClass {  **@Test**  **public** **void** testInvokationNumber() {  LinkedList mockedList = *mock*(LinkedList.**class**);  mockedList.add("once"); // 1 раз «once»  mockedList.add("twice"); // 2 разa «twice»  mockedList.add("twice");  mockedList.add("three times");// 3 разa «three times»  mockedList.add("three times");  mockedList.add("three times");  *verify*(mockedList, *times*(**1**)).add("once");  *verify*(mockedList, *times*(**2**)).add("twice");  *verify*(mockedList, *times*(**3**)).add("three times");  *verify*(mockedList, *never*()).add("never happened");  *verify*(mockedList, *atLeastOnce*()).add("three times");  *verify*(mockedList, *atMost*(**5**)).add("three times");  }  } |
| При последовательном вызове verify()-методов в качестве параметра передается количество вызовов посредством метода times(number), а после – сам метод со значением. Так, метод add() с параметром «once» действительно вызывается один раз, однако если изменить ожидаемое количество вызовов, например, на 2, то верификация не будет пройдена и тест закончится ошибкой. |

1. Методы doReturn(Obejct), doNothing(), doThrow(), doCallRealMethod() выполняют заданное поведение, когда (when()) этого требует указанный метод.

|  |
| --- |
| **public** **class** TestClass {  **@Test**  **public** **void** testDoThrowWhen() {  LinkedList mockedList = *mock*(LinkedList.**class**);  *doThrow*(**new** RuntimeException("Once could not be added"))  .when(mockedList).add("once");  mockedList.add("once");  }  } |
| При выполнении теста doThrowWhen() будет сгенерировано исключение, так как вызов метода add() с параметром «once» невозможен. |

В Mockito также есть класс, позволяющий точно проверить, с какими параметрами был вызван тот или иной метод. Этот класс ***org.mockito.Matchers*** содержит набор методов, использование которых позволяет настроить проверку аргументов методов для тестирования.

1. any() – «любой» параметр метода.
2. eq() – возвращает эквивалентный значение параметра метода.
3. isNotNull() / isNull()– возвращает (не) null-аргумент.

|  |
| --- |
| **public** **class** TestClass {  **@Test**  **public** **void** testMatcher() {  Foo mock1 = *mock*(Foo.**class**);  mock1.someMethod(**9**, **9**, "third argument");  *verify*(mock1).someMethod(*eq*(**9**), *eq*(**9**), *anyString*());    Foo mock2 = *mock*(Foo.**class**);  mock2.someMethod(**10**, **9**, **null**);  *verify*(mock2).someMethod(*anyInt*(), *anyInt*(), (String)*isNotNull*());  }  } |
| Тест для mock1 выполнится успешно, так как при проверке (вызове метода verify() указывается, что параметры должны быть (9, 9, anyString). А вот при выполнении проверки вызова метода someMethod() на заглушке mock2 произойдет ошибка, так как в качестве третьего параметра передан null, тогда как Matcher настроен на не null-объект типа String. |

# Создание клиента веб-сервиса в Java

Web Service (веб-сервис) – это технология обеспечения обмена сообщениями между двумя компонентами в сети, путем предоставления друг-другу публичного контракта взаимодействия (обычно, WSDL). Общение с web service происходят посредством SOAP-сообщений по HTTP протоколу.

Веб сервисы используются для реализации архитектур типа SOA (Service oriented architecture). Главное отличие таких систем в том, что неделимой «единицей» в них является не операция, а сообщение. Чаще всего веб-сервисы используются для интеграции систем в самых различных сферах деятельности человека.

Для того, чтобы создать клиент веб-службы, необходимо знать сетевой адрес веб-сервиса и таким образом получить доступ к его публичному контракту (WSDL-файлу). Далее, с помощью Java-утилиты wsimport сгенерировать исходный код Java для создания общих сущностей (общей предметной области), которую использует сервер и клиент.

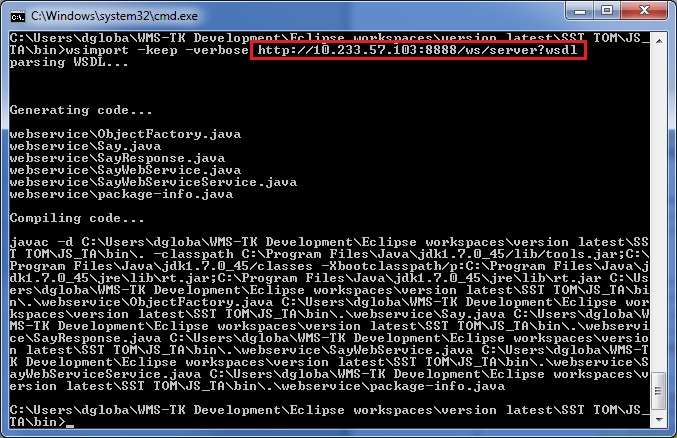
Для генерации файлов исходного кода необходимо указать папку, куда будут сохранены генерируемые файлы, и адрес WSDL-контракта.

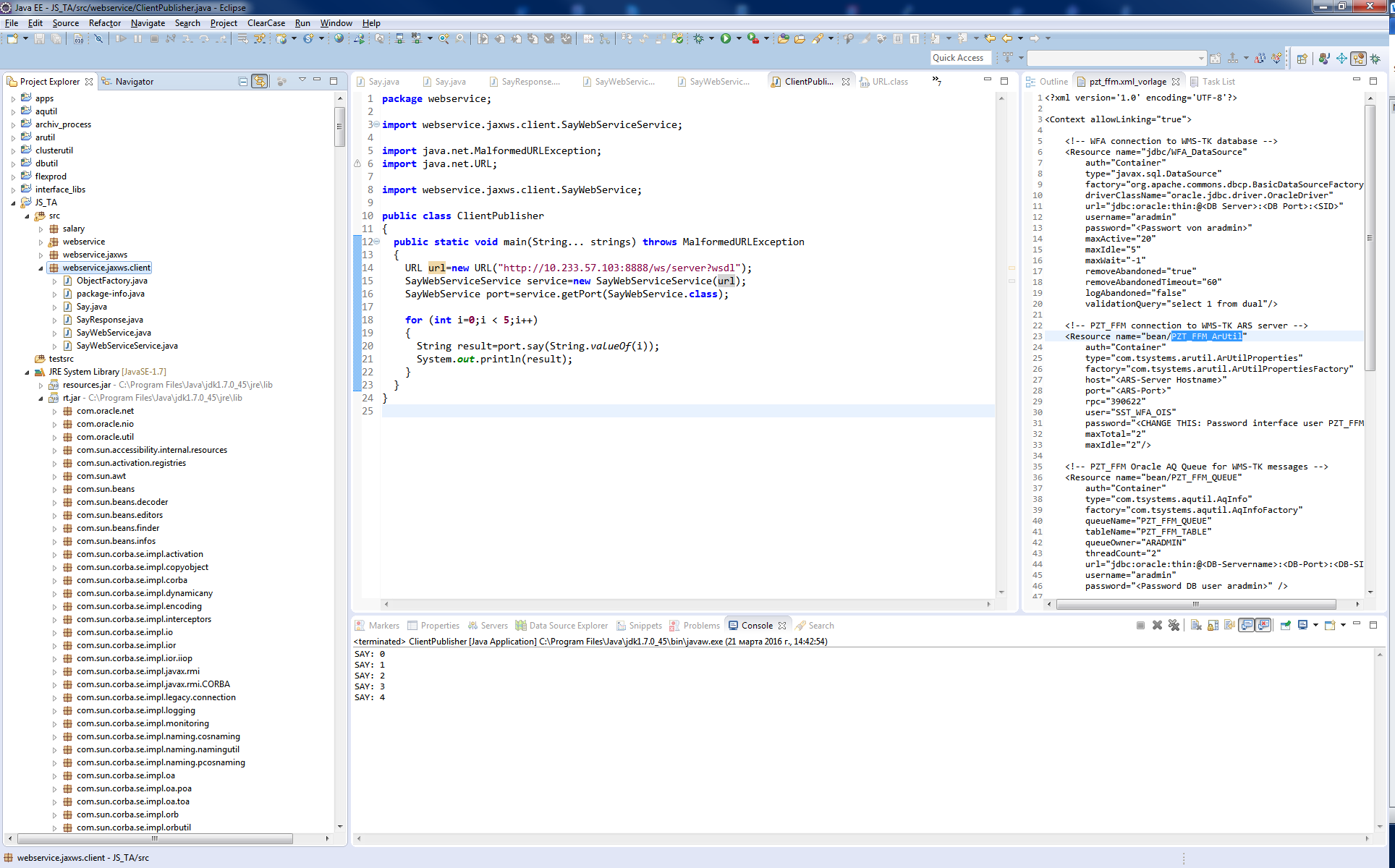
|  |
| --- |
| **wsimport -d** [путь\_к\_папке\_для\_сохранения\_исходных\_кодов][адрес\_веб\_сервиса]?WSDL  **-Xnocompile** |
| wsimport – утилита импорта сущностей веб-сервиса  -d – используемая директория для сохранения сгенерированных файлов  -Xnocompile – не производить компиляцию сгенерированных файлов на этапе импорта.  \*Указанные дополнительные ключи не являются обязательными |

Создание простого клиента ограничивается теперь наличием проекта, в который будут добавлены сгенерированные исходные коды, и подключением по сети для доступа к реальному веб-сервису, который будет обрабатывать запросы от клиента.

Пусть, в сети развернут веб-сервис по адресу http://10.233.57.103:8888/ws/server?wsdl . Необходимо создать клиента данного веб-сервиса и вызывать его методы.

С помощью утилиты wsimport генерируется клиентский код веб-сервиса. Сгенерированные файлы необходимо переместить в проект.





|  |
| --- |
| **import** java.net.MalformedURLException;  **import** java.net.URL;  **import** javax.xml.namespace.QName;  **import** javax.xml.ws.Service;  **public** **class** ClientPublisher{  **public** **static** **void** main(String... strings) **throws** MalformedURLException  {  URL url=**new** URL("http://10.233.57.103:8888/ws/server?wsdl");  SayWebServiceService service=**new** SayWebServiceService(url);  SayWebService port=service.getPort(SayWebService.**class**);  **for** (**int** i=**0**;i < **5**;i++)  {  String result=port.say(String.*valueOf*(i));  System.***out***.println(result);  }  }  } |
| Шаблон клиента веб-сервиса |