# План презентации

## Блок 1. Модульное тестирование.

Кто пишет модульные тесты? Зачем писать модульные тесты?

**Основные преимущества:**

* Показывает, что отдельные части программы работоспособны
* Упрощает поиск ошибок
* Позволяет вносить изменения и проводить рефакторинг будучи уверенным, что модуль работает корректно.
* Модульные тесты можно рассматривать как «живой документ» для тестируемого класса

**Недостатки:**

* Написание тестов занимает довольно много времени.
* Количество кода для тестирования превышает, код самой программы
* Отбираем работу у QA ☺

Если сравнить время, на написание тестов и время на дебаг кода и локализацию бага, то модульное тестирование выгоднее.

## Блок 2. Модульное тестирование на Java.

Java предоставляет выбор библиотек для модульного тестирования.

Мы будем использовать: JUnit, Mockito, PowerMock

## Блок 3. JUnit.

**JUnit** — библиотека для модульного тестирования программ Java.

### Список основных аннотаций

|  |  |
| --- | --- |
| **Аннотация** | **Описание** |
| @Test public void testMethod() | Метод является тестовым |
| @Test(timeout=100) public void testMethod() | Если время выполнения превысит параметр timeout, то тест будет завершен неудачно |
| @Test (expected = MyException.class) public void testMethod() | Метод должен выбросить исключение принадлежащие к классу MyException, в противном случае тест будет завершен неудачно |
| @BeforeClass public static void testMethod() | Метод вызывающийся один раз для класса перед выполнением тестовых методов; здесь можно разместить инициализацию, которую нужно выполнять только один раз, например, прочитать данные, которые будут использоваться в тестовых методах или создать соединение с базой данных |
| @AfterClass public static void testMethod() | Метод вызывающийся один раз для класса после выполнения тестовых методов; здесь можно разместить деинициализацию которую нужно выполнять только один раз, например, закрыть соединение с базой данных или удалить данные, которые больше не нужны |
| @Before public static void beforeMethod() | Метод, вызывающийся перед каждым тестовым методом в тестовом классе; здесь можно выполнить необходимую инициализацию, например, выставить начальные параметры |
| @After public static void afterMethod() | Метод, вызывающийся после каждого тестового метода в тестовом классе; здесь можно выполнить необходимую деинициализацию, например, удалить данные, которые больше не нужны |

### Список типов проверок Asserts

|  |  |
| --- | --- |
| **Тип проверки** | **Описание** |
| assertTrue(boolean condition) | Проверка на равенство условия condition значению true |
| assertFalse(boolean condition) | Проверка на равенство условия condition значению false |
| assertEquals(<тип> expected, <тип> actual) | Проверка на равенство |
| assertNotNull(Object object) | Проверка, что Object не null |
| assertNull(Object object) | Проверка, что Object null |
| assertSame(Object expected, Object actual) | Проверка на равенство двух объектов expected и actual, т.е. один и тот же объект |

### Пример

|  |
| --- |
| **public class** CalculatorTest {   **private** Calculator **calculator**;   **@Before  public void** initTest() {  **calculator** = **new** Calculator();  }   **@After  public void** afterTest() {  **calculator** = **null**;  }   **@Test  public void** getSum() {  *assertEquals*(4, **calculator**.getSum(2, 2));  *assertTrue*(**calculator**.getSum(2, 2) == 4);  *assertFalse*(**calculator**.getSum(2, 2) == 5);  } } |

## Блок 4. Mockito

Становится сложнее, когда у тестируемого кода есть внешние зависимости, например код работает с базой данных или другим модулем.

Для ликвидации внешних зависимостей в модульных тестах используются тестовые объекты.

*Стоит заметить, что существует классический труд по модульным тестам за авторством Жерарда Месароша под названием "xUnit test patterns: refactoring test code", в котором автор вводит аж 5 видов тестовых объектов, которые могут запросто запутать неподготовленного человека:****— dummy object****, который обычно передается в тестируемый класс в качестве параметра, но не имеет поведения, с ним ничего не происходит, никакие методы не вызываются. Примером таких dummy-объектов являются new object(), null, «Ignored String» и т.д.****— test stub****(заглушка), используется для получения данных из внешней зависимости, подменяя её. При этом игнорирует все данные, могущие поступать из тестируемого объекта в stub. Один из самых популярных видов тестовых объектов. Тестируемый объект использует чтение из конфигурационного файла? Передаем ему ConfigFileStub возвращающий тестовые строки конфигурации для избавления зависимости на файловую систему.*

***— test spy****(тестовый шпион), используется для тестов взаимодействия, основной функцией является запись данных и вызовов, поступающих из тестируемого объекта для последующей проверки корректности вызова зависимого объекта. Позволяет проверить логику именно нашего тестируемого объекта, без проверок зависимых объектов.*

***— mock object****(мок-объект), очень похож на тестовый шпион, однако не записывает последовательность вызовов с переданными параметрами для последующей проверки, а может сам выкидывать исключения при некорректно переданных данных. Т.е. именно мок-объект проверяет корректность поведения тестируемого объекта.*

***— fake object****(фальшивый объект), используется в основном чтобы запускать (незапускаемые) тесты (быстрее) и ускорения их работы. Эдакая замена тяжеловесного внешнего зависимого объекта его легковесной реализацией. Основные примеры — эмулятор для конкретного приложения БД в памяти (fake database) или фальшивый вебсервис.*

Можно писать тестовые объекты в ручную, но это очень долго.

Мы будем использовать фреймворк Mockito*.*

**Mockito** предоставляет ряд возможностей для создания заглушек вместо реальных классов или интерфейсов при написании JUnit тестов.

Для того, чтобы сделать заглушку, достаточно вызвать метод mock, пример:

|  |
| --- |
| **iCalculator** = *mock*(ICalculator.**class**); |

Или применить аннотацию:

|  |
| --- |
| **@Mock** ICalculator **iCalculator**; |

## Основные возможности Mockito

1. **Определение поведения - when(mock).thenReturn(value)**

|  |
| --- |
| **@Test public void** add() {  *when*(**iCalculator**.add(10, 20)).thenReturn(30);   *assertEquals*(30, **calculatorWrapper**.add(10, 20)); } |

1. **Подсчет количества вызовов - atLeast, atLeastOnce, atMost, times, never**

Для проверки количества вызовов определенных методов Mockito предоставляет следующие методы :

atLeast (int min) - не меньше min вызовов;

atLeastOnce () - хотя бы один вызов;

atMost (int max) - не более max вызовов;

times (int cnt) - cnt вызовов;

never () - вызовов не было;

|  |
| --- |
| **@Test public void** division() {  *when*(**iCalculator**.division(26, 2)).thenReturn(13);   *assertEquals*(13, **calculatorWrapper**.division(26, 2));  *assertEquals*(13, **calculatorWrapper**.division(26, 2));   *verify*(**iCalculator**, *atLeast*(2)).division(26, 2);  *verify*(**iCalculator**, *never*()).subtraction(*anyInt*(), *anyInt*()); } |

1. **Обработка исключений - when(mock).thenThrow()**

|  |
| --- |
| **@Test**(expected = ArithmeticException.**class**) **public void** divisionByZero() {   *//when(iCalculator.division(anyInt(), eq(0))).thenThrow(ArithmeticException.class);  when*(**iCalculator**.division(6, 0)).thenThrow(ArithmeticException.**class**);   **calculatorWrapper**.division(6, 0); } |

1. **Использование интерфейса org.mockito.stubbing.Answer<T>**

Иногда описание поведения mock объекта требует определенной проверки с усложнением логики. В этом случае можно использовать интерфейс Answer<T>, который позволяет реализовать заглушки методов со сложным поведением.

|  |
| --- |
| **@Test public void** testThenAnswer() {  *when*(**iCalculator**.add(19, 25)).thenAnswer(**answer**);  *assertEquals*(**calculatorWrapper**.add(19,25), 44); }  **private** Answer<Integer> **answer** = **new** Answer<Integer>() {  **@Override  public** Integer answer(InvocationOnMock invocationOnMock) **throws** Throwable {   Object mock = invocationOnMock.getMock();  System.***out***.println (**"mock object : "** + mock.toString()); Object[] args = invocationOnMock.getArguments();  **int** d1 = (**int**) args[0];  **int** d2 = (**int**) args[1];  **int** sum = d1 + d2;  System.***out***.println (**""** + d1 + **" + "** + d2);   **return** sum;  } }; |

**Mockito** - гибкое и функциональное средство, но у него есть ограничения, например: final классы, private поля и методы, static методы и многое другое.

## Блок 5. PowerMock.

Эти задачи может решить PowerMock.

Пример создания заглушки для статики

|  |
| --- |
| **@RunWith**(PowerMockRunner.**class**) **@PrepareForTest**({StaticService.**class**}) **public class** CalculatorWrapperTest {   **@Before  public void** initTest() {  *mockStatic*(StaticService.**class**);  }  …  } |

Далее аналогично примерам выше.

## Блок 6. Совместимость.

При подключении перечисленных выше библиотек, надо обратить внимание на совместимость версий и отсутствие конфликтов: JUnit, Mockito, PowerMock, Javassist.

### Пример рабочей конфигурации для Maven:

|  |
| --- |
| <**properties**>  <**powermock.version**>1.5.6</**powermock.version**>  <**mockito.version**>1.9.5</**mockito.version**> </**properties**>  <**dependencies**>  <**dependency**>  <**groupId**>org.javassist</**groupId**>  <**artifactId**>javassist</**artifactId**>  <**version**>3.19.0-GA</**version**>  <**scope**>test</**scope**>  </**dependency**>  <**dependency**>  <**groupId**>org.mockito</**groupId**>  <**artifactId**>mockito-all</**artifactId**>  <**version**>${mockito.version}</**version**>  <**scope**>test</**scope**>  </**dependency**>  …  <**dependency**>  <**groupId**>org.powermock</**groupId**>  <**artifactId**>powermock-api-mockito</**artifactId**>  <**version**>${powermock.version}</**version**>  <**scope**>test</**scope**>  </**dependency**>  <**dependency**>  <**groupId**>org.powermock</**groupId**>  <**artifactId**>powermock-module-junit4</**artifactId**>  <**version**>${powermock.version}</**version**>  <**scope**>test</**scope**>  </**dependency**> </**dependencies**> |

## Блок 7. Резюме

https://github.com/kirill-of/nf-unit-testing-presentation