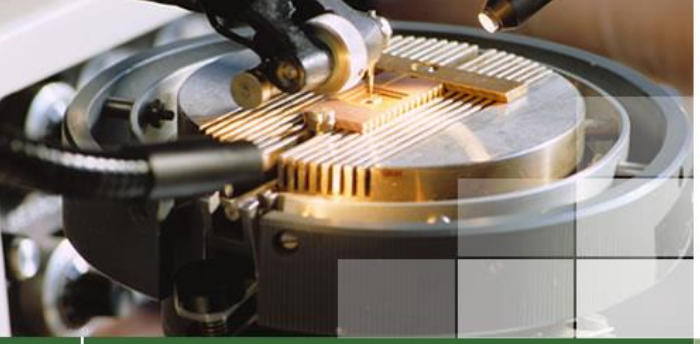


POLITECNICO DI MILANO

Dipartimento di
Elettronica e Informazione

Planning and Managing Software Projects 2012-13

Unit testing

Emanuele Della Valle, Lecturer: Daniele Dell'Aglio
http://dellaglio.org

Outline

- Unit testing
- JUnit
  - Test cases
  - Setup and Tear Down
  - Test suites

Unit testing

- The primary goal of unit testing is to take the *smallest piece of testable software* in the application, **isolate it** from the remainder of the code, and **determine whether it behaves exactly as you expect** [msdn]
- In computer programming, unit testing is a method by which *individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures*, are tested to determine if **they are fit for use**. [wikipedia]

Unit testing

- The goal is to write tests for the two situations
  - If the tested software component works properly when its inputs are valid and the pre-conditions are verified (Positive testing)
  - If the tested software component behaves in the proper way in the other case (Negative testing)
- Code and test, code and test!
  - Writing test after that all the code is written is complicated and time consuming
  - Tests help in checking if the development introduces errors

Unit testing – Key concepts

- Test case: method that can be executed to check if the behaviour of one or more objects is valid
- Test fixture: the context of the test – usually it is the set of objects on which the tests will run against
- Test suite: collection of test cases and related test fixtures that can be executed

Why Unit Testing?

- How to verify if the code works?
- Debug expressions

• Easy to do • Can be done at runtime	• Only one debug expression at a time • Human effort to verify the result

- Test expressions

• Easy to write • Can be inserted in every part of the code	• Scroll Blindness • Execution of the whole code • Human effort to verify the result

- Unit testing

• Focus on one component • Repeatable • Automatic verification of the results	• Require a lot of time to be written

JUnit

- JUnit is a unit testing framework for Java
- http://junit.org
- Today we focus on JUnit 4.x
  - Same concepts of previous versions
  - Based on Java annotations (instead of inheritance)

Example (example1 - TreeNode.java)

```
public class TreeNode{
   private String content;
   private TreeNode parent;
   private List<TreeNode> children;
   public TreeNode(){
      super();
      children=new ArrayList<TreeNode>();
   }
   public void addChild(TreeNode node){
      children.add(node);
   }
   public List<TreeNode> getChildren(){
      return children;
   }
}
```

JUnit – Test cases

- Test cases are Java methods such as

```
@Test public void testCaseName(){...}
```

- `@Test` is the JUnit annotation to indicate that the method is a test
- The method doesn't have parameter
- The method returns void

JUnit – Test cases

- The body of the test method contains the code to verify the satisfaction of the conditions
- It is done through Assertions
  - `assertNull(Object o)`
  - `assertNotNull(Object x)`
  - `assertEquals(Object a, Object b)`
  - `...`

(the full list is available at: http://junit.sourceforge.net/javadoc/org/junit/Assert.html)

Example (example2 - TreeNodeTest.java)

```
public class TreeNodeTest{
    @Test
    public void shouldAddChildInTail(){
       TreeNode root = new TreeNode();
       TreeNode firstChild = new TreeNode();
       firstChild.setContent("content1");
       root.addChild(firstChild);
       TreeNode secondChild = new TreeNode();
       secondChild.setContent("content2");
       root.addChildren(secondChild);
       int lastElement=root.getChildren().size()-1;
       TreeNode lastChild =
            root.getChildren().get(lastElement);
       assertEquals(secondChild, lastChild);
    }
}
```

Example (example2 - TreeNodeTest.java)

```
public class TreeNodeTest{
   @Test public void shouldAddChildInTail(){...}

   @Test
   public void childShouldHaveTheParent(){
      TreeNode root = new TreeNode();
      TreeNode child = new TreeNode();
      root.addChild(child);
      TreeNode parent =
            root.getChildren().get(0).getParent();
      assertEquals(root, parent);
   }
}
```

Let's run the test with Maven

- The maven plugin to execute Junit tests is Surefire: http://maven.apache.org/surefire/maven-surefire-plugin
- The main commands are:
- `mvn test`
  - Runs all the tests
- `mvn -Dtest=C test`
  - Runs all the test cases in the class *C*
- `mvn -Dtest=C#T test`
  - Runs the *T* test case in the class *C*
- `mvn -Dtest=C#T1,T2 test`
  - Runs the *T1* and *T2* tests case in the class *C*

Example (example2 - TreeNode.java)

Let’s fix the method code...

```
public class TreeNode{
    ...
    public void addChild(TreeNode node){
       node.setParent(this);
       childrens.add(node);
    }
    ...
}
```

Now the all the tests run and are successful!

JUnit – Test cases

- For Negative testings it is possible to set an expected exception as parameter of the `@Test` annotation

```
@Test(expected=ExpectedException.class)...
```

- It is possible to exclude the execution of a test case using the @Ignore annotation

```
@Ignore("reason")@Test public void ...
```

Example (example3 - TreeNodeNegTest.java)

```
public class TreeNodeNegTest{
    @Test(expected=NotOrphanException.class)
    public void nodeCannotHaveAnotherParent(){
        TreeNode root = new TreeNode();
        TreeNode root2 = new TreeNode();
        TreeNode child = new TreeNode();
        root.addChild(child);
        root2.addChild(child);
    }
}
```

Example (example3 - TreeNode.java)

```
public class TreeNode{
    ...
    public void addChild(TreeNode node){
       if(node.getParent()!=null)
            throw new NotOrphanException(node);
       node.setParent(this);
       childrens.add(node);
    }
    ...
}
```

Now the the test returns an exception!

JUnit – Setup

- The setup is done through `@Before` and `@BeforeClass` annotations
  - It allows to define the fixture
- Methods annotated with `@Before` are executed before the execution of each test case
- Methods annotated with `@BeforeClass` are the first to be executed
  - `@BeforeClass` methods should be static

JUnit – Tear Down

- The tear down is done through `@After` and `@AfterClass` annotations
  - Used to destroy the fixture and reset the status before the test execution
- Methods annotated with `@After` are executed after the execution of each test case
- Methods annotated with `@AfterClass` are executed after the execution of all the test methods and all the `@After` annotated methods
  - `@AfterClass` methods should be static

Example (example4 - TreeNodeTest.java)

```
public class TreeNodeTest{
   private TreeNode root;
   private TreeNode child;
   @Before
   public void setup(){
      root=new TreeNode();
      child=new TreeNode();
      root.addChild(child);
   }
   ...
}
```

Now we have to fix the code of the tests...

Example (TreeNodeTest.java)

```
public class TreeNodeTest{
   ...
   @Test public void shouldAddChildInTail(){
      TreeNode secondChild = new TreeNode();
      secondChild.setContent("content2");
      root.addChild(secondChild);
      TreeNode lastChild =
            root.getChildren().getLast();
      assertEquals(secondChild, lastChild);
   }
   @Test public void childShouldHaveTheParent(){
      TreeNode parent =
            root.getChildren().get(0).getParent();
      assertEquals(root, parent);
   }
}
```

Test suite

- In order to execute the test cases a test suite class is required
- `@RunWith` and `@SuiteClasses` are the annotations that define a test suite:
  - `@RunWith` indicates the test runner (a class implementing org.junit.runner.Runner)
  - `@SuiteClasses` defines the list of the classes containing the test cases of the test suite
- It is also possible to use the Java program:

```
org.junit.runner.JUnitCore.runClasses(
        TestClass1, TestClass2, ...);
```

Example (example5 - TreeNodeTestSuite.java)

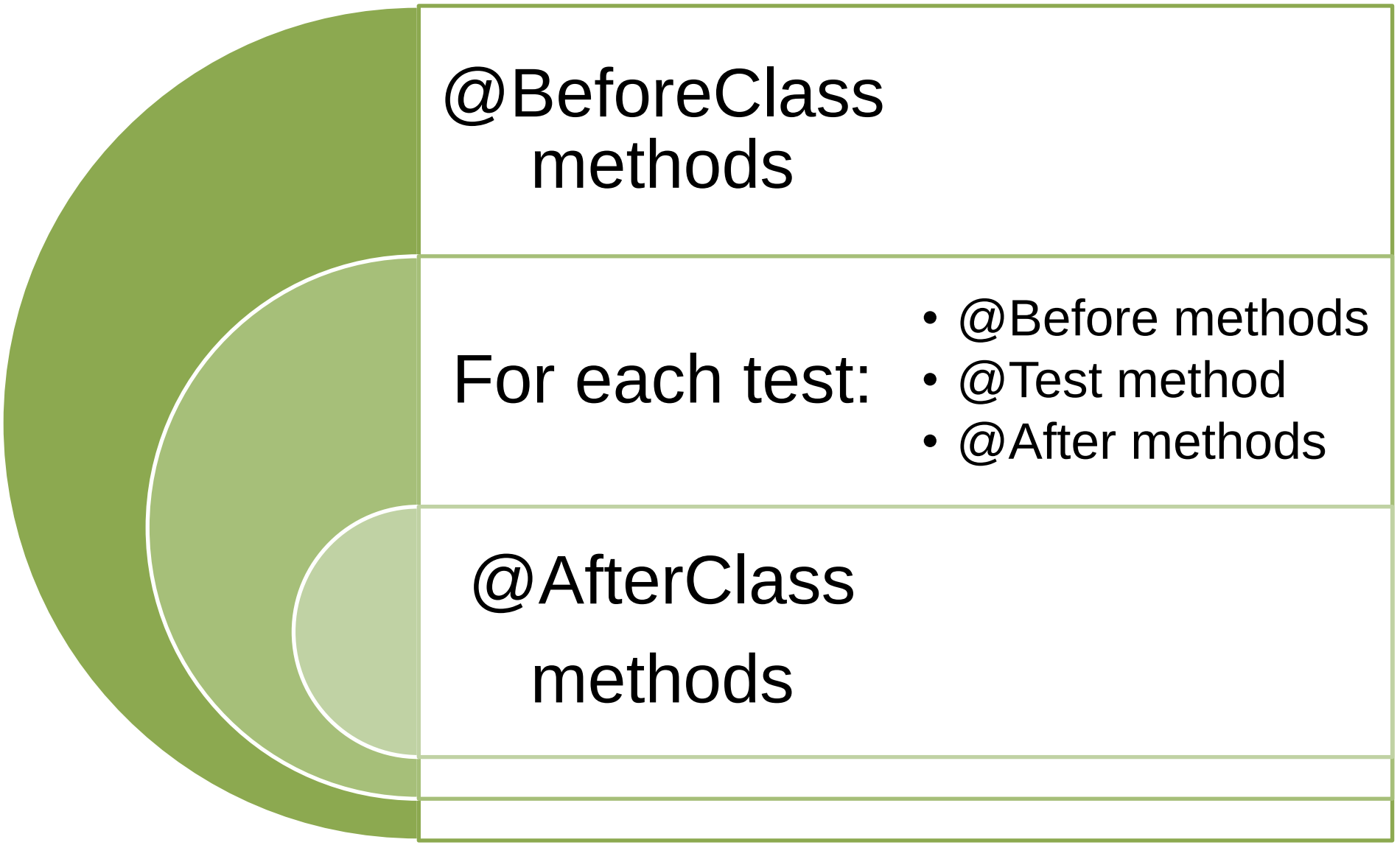
```
@RunWith(Suite.class)

@SuiteClasses({TreeNodeTest.class,
      TreeNodeNegTest.class})

public class TreeNodeTestSuite{

}
```

Test Suite lifecycle



JUnit – some best practices

- Write the tests before the tested code
  - Test-driven Development
- Put the test cases in the **same package** of the tested classes but in a **different source folder**
  - It enables the test of the protected methods
- Execute all the tests every time you change something!

References

- JUnit Test Infected: Programmers Love Writing Tests http://junit.sourceforge.net/doc/testinfected/testing.htm
- JUnit Cookbook http://junit.sourceforge.net/doc/cookbook/cookbook.htm
- JUnit and EasyMock (Dzone RefCard) http://refcardz.dzone.com/refcardz/junit-and-easymock
- JUnit Kung Fu: Getting More Out of Your Unit Tests http://weblogs.java.net/blog/johnsmart/archive/2010/09/30/junit-kung-fu-getting-more-out-your-unit-tests