ADVANCED PROGRAMMING PRACTICES

Unit Testing Frameworks

JUnit

Software testing

- Software testing is meant to avoid software **failure**.
- A **failure** is caused by a **fault** in the code base.
- A **symptom** is an **observable behavior** of the system that enables us to observe a **failure** and possibly find its corresponding **fault**.
- The process of discovering what caused a failure is called **fault identification**.
- The process of ensuring that the failure does not happen again is called **fault correction**, or fault removal.
- Fault identification and fault correction is popularly called **debugging**.
- Software testing, in practice, is about identifying a certain possible system failure and design a **test case** that proves that this particular failure is **not** experienced by the software.
- **“testing can reveal only the presence of faults, never their absence.” [Edsger Dijkstra]**

Software testing

- There are many driving sources for software testing:
  - Requirements-driven testing, Structure-driven testing, Statistics-driven testing, Risk-driven testing.
- There are many levels and kinds of software testing:
  - Unit Testing, Integration Testing, Function Testing, Acceptance Testing, Installation Testing.
- Unit testing can easily be integrated in the programming effort by using a **Unit Testing Framework**.
- However, unit testing cannot be applied for higher-level testing purposes such as function testing or acceptance testing, which are system-level testing activities.

Unit testing

- **Defintion:** A unit test is a piece of code written by a developer that exercises a very small, specific area of functionality applied to one of the units of the code being tested. Usually a unit test exercises some particular method in a particular context.

- **Example:** add a large value to a sorted list whose content is known, then confirm that this value appears at the end of the list.

- The goal of unit testing is to isolate important parts (i.e. units) of the program and show that the individual parts are free of certain faults.

Unit testing: benefits

- Facilitates change:
  - Unit testing allows the programmer to change or refactor code later, and make sure the module still works correctly after the change (i.e. **regression testing**).
- Simplifies integration:
  - Unit testing helps to eliminate uncertainty in the units and can be used in a bottom-up integration testing style approach.
- Documentation:
  - Unit testing provides a sort of living documentation of the specifications of the units of the system. Developers looking to learn what functionality is provided by a unit and how to use it can look at the unit tests to gain understanding of the unit's API specifications.

Unit testing: benefits

- Distributes feedback:
  - Identifies defects early and regularly in the development cycle and avoids to cluster the finding of bugs in the later stages.
- Confidence building:
  - Successful (and meaningful) tests builds up confidence in the code to everyone involved, which then may lead to write more tests and thus even more completeness in testing.
- Forces analysis:
  - Writing unit tests forces the programmers to read and analyze their code, thus removing defects through constant code verification.

Unit testing framework

- For a large system, there can be thousands of unit tests, which can be tedious to maintain and execute.
- **Automated** testing supports **maintainability** and **extensibility** along with **efficiency**.
- A xUnit Testing Framework lets a programmer associate Classes and Methods to corresponding Test Classes and Test Methods.
- Automation is achieved by automatically setting up a testing **context**, calling each test case, verifying their corresponding **expected result**, and **reporting** the status of all tests.
- Can be combined with the use of a Software Versioning Repository: prior to any commit being made, unit testing is re-applied to make sure that the committed code is still working properly.
- Build automation tools can be used to integrate the testing process in the building process and be connected to the revision control process. e.g. Make, Ant, Maven, Gradle.

Unit testing framework: components

- **Tested Class** – the class that is being tested.
- **Tested Method** – the method that is tested.
- **Test Case** – the testing of a class's method against some specified conditions.
- **Test Case Class** – a class performing some test cases.
- **Test Case Method** – a Test Case Class's method implementing a test case.
- **Test Suite** – a collection of test cases that can be tested in a single batch.

Java unit testing framework: JUnit

- In Java, the standard unit testing framework is known as JUnit.
- Test Cases and Test Results are Java objects.
- JUnit was created by Erich Gamma and Kent Beck, two authors best known for Design Patterns and eXtreme Programming, respectively.
- Using JUnit you can easily and incrementally build a test suite that will help you measure your progress, spot unintended side effects, and focus your development efforts.

JUnit: test case procedure

- Every test case aims at testing that a certain method has the effect of returning or changing a value according to a certain correct behavior in a certain context.
- Thus, every test case has three phases:
  1. Set up the context in which the method will be called.
  2. Call the method.
  3. Observe if the method call resulted in the correct behavior using one or more JUnit assertion method.

JUnit: setting up the testing context

- All test cases represent an execution in a certain context.
- The context is represented by:
  - The values passed to the method as parameters.
  - Other variables that the method may be using internally, which are not passed as parameters.
- All of these must be set up before the call to the tested method.
- The values passed to or used by the method can be declared as:
  - Local variables in the test case method.
  - Data members of the test class.
- Declaring the contextual data as data members of the test class allows several test case methods to share the same contextual data elements.

JUnit: setting up the testing context

- If more than one test case share the same context, group them in a test class and use **`setUp()`**/**`tearDown()`** (JUnit 3) or **`@Before`**/**`@After`** (JUnit 4) to set the context to the right values before each test case is run.
- JUnit 4 also enables you to set a number of static values and methods that are shared across all tests if that is desirable. These are managed using **`@BeforeClass`** and **`@AfterClass`**.

JUnit: assertions

- The JUnit framework provides a complete set of assertion methods that can be used in different situations:

`void assertEquals(boolean expected, boolean actual)`
Check that two primitive values or objects are equal.

`void assertTrue(boolean condition)`
Check that a condition is true.

`void assertFalse(boolean condition)`
Check that a condition is false.

`void assertNotNull(Object object)`
Check that an object isn't null.

`void assertNull(Object object)`
Check that an object is null.

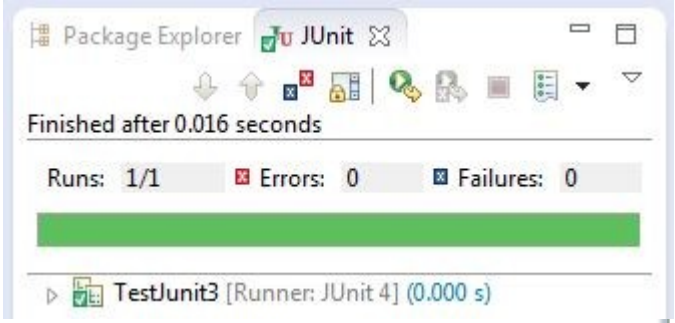
`void assertSame(boolean condition)`
Tests if two object references point to the same object.

`void assertNotSame(boolean condition)`
Tests if two object references do not point to the same object.

`void assertArrayEquals(expectedArray, resultArray)`
Tests whether two arrays are equal to each other.

JUnit: examples of context and assertions

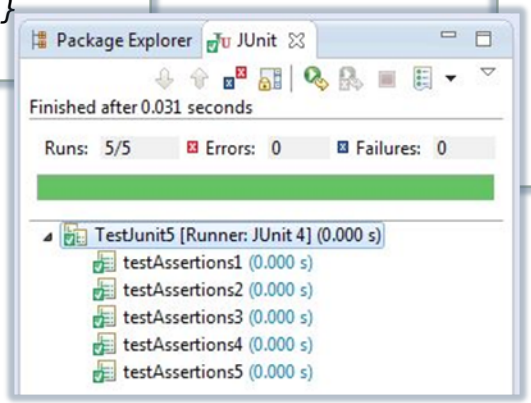
- A JUnit test class is executable by a test runner.
- Most Java IDEs have a JUnit test runner




```java
import static org.junit.Assert.*;
public class TestJunit3 {
   @Test public void testAssertions() {
      String str1 = new String ("abc");
      String str2 = new String ("abc");
      String str3 = null;
      String str4 = "abc";
      String str5 = "abc";
      int val1 = 5;
      int val2 = 6;
      String[] expectedArray = {"one", "two", "three"};
      String[] resultArray =  {"one", "two", "three"};
      //Check that two objects are equal
      assertEquals(str1, str2);
      //Check that a condition is true
      assertTrue (val1 < val2);
      //Check that a condition is false
      assertFalse(val1 > val2);
      //Check that an object isn't null
      assertNotNull(str1);
      //Check that an object is null
      assertNull(str3);
      //Check if two object references point to the same object
      assertSame(str4,str5);
      //Check if two object references not point to the same object
      assertNotSame(str1,str3);
      //Check whether two arrays are equal to each other.
      assertArrayEquals(expectedArray, resultArray);
   }
}
```

JUnit: examples of context and assertions

```java
import org.junit.Test;
import static org.junit.Assert.*;

public class TestJunit4 {

   String str1 = new String ("abc");
   String str2 = new String ("abc");
   String str3 = null;
   String str4 = "abc";
   String str5 = "abc";

   @Test public void testAssertions1() {
      assertEquals(str1, str2);}

   @Test public void testAssertions4() {
      assertNotNull(str1);}

   @Test public void testAssertions5() {
      assertNull(str3);}

   @Test public void testAssertions6() {
      assertSame(str4,str5);}

   @Test public void testAssertions7() {
      assertNotSame(str1,str3);}
}
```




```java
import org.junit.Test;
import org.junit.Before;
import static org.junit.Assert.*;

public class TestJunit5 {

  String str1, str2, str3, str4, str5;

  @Before public void beforeEachTest(){
     str1 = new String ("abc");
     str2 = new String ("abc");
     str3 = null;
     str4 = "abc";
     str5 = "abc";}
  @Test public void testAssertions1() {
     assertEquals(str1, str2);}

  @Test public void testAssertions2() {
     assertNotNull(str1);}

  @Test public void testAssertions3() {
     assertNull(str3);}

  @Test public void testAssertions4() {
     assertSame(str4,str5);}

  @Test public void testAssertions5() {
     assertNotSame(str1,str3);}
```

Tested class

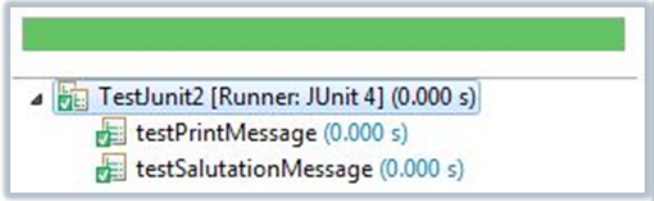
- Nothing needs to be added to the tested class.
- Every test case tests a call to a method of the tested class in a predefined context and asserts whether it resulted in a predicted behavior.
- Each test case is implemented as a method of a JUnit test case class.
- Test cases that share the same context elements can be gathered in a single test class.

```java
/*
* This class prints the given message on console.
*/
public class MessageUtil {

   private String message;

   //Constructor
   //@param message to be printed
   public MessageUtil(String message){
      this.message = message;
   }

   // prints the message
   public String printMessage(){
      System.out.println(message);
      return message;
   }

   // add "Hi!" to the message
   public String salutationMessage(){
      message = "Hi!" + message;
      System.out.println(message);
      return message;
   }
}
```

JUnit 3 test class

- JUnit 3 test class:
  - Must be a subclass of **`TestCase`**.
  - Every test case method must start with "**`test`**".
  - If all tests in a test class share the same context of execution, it can be set using the **`setUp()`** method, which executes before every test method's execution.
  - If resources need to be freed after the test case, it may be done in the **`tearDown()`** method, which executes after every test method's execution.

```
import junit.framework.TestCase;

public class TestJunit2 extends TestCase{
   String message;
   MessageUtil messageUtil;

   public void setUp(){
      System.out.println("inside setUp()");
      message = "Robert";
      messageUtil = new MessageUtil(message);
   }
   public void tearDown(){
      System.out.println("inside tearDown()");
   }
   public void testSalutationMessage() {
      System.out.println("Inside testSalutationMessage()");
      message = "Hi!" + "Robert";
      assertEquals(message,messageUtil.salutationMessage());
   }
   public void testPrintMessage() {
      System.out.println("Inside testPrintMessage()");
      assertEquals(message, messageUtil.printMessage());
   }
}
```

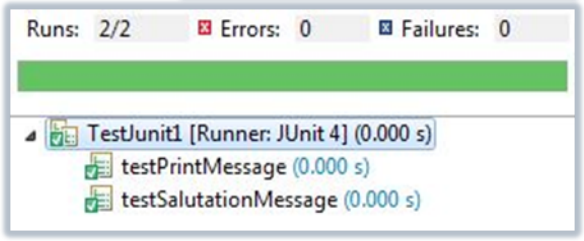



```
inside setUp()
Inside testSalutationMessage()
Hi!Robert
inside tearDown()
inside setUp()
Inside testPrintMessage()
Robert
inside tearDown()
```

JUnit 4 test class

- JUnit 4 test class:
  - Not a subclass of TestCase.
  - JUnit 4, uses the notion of annotations, which are keywords preceded by @, used by the JUnit test runner to define the execution behavior of a test class.
  - Provides class-level and test-level context setting
  - Test cases with @Tes

```
in before TestJunit1 class
in before TestJunit1 test case
Inside testPrintMessage()
Robert
in after TestJunit1 test case
in before TestJunit1 test case
Inside testSalutationMessage()
Hi!Robert
in after TestJunit1 test case
in after TestJunit1 class
```




```java
import static org.junit.Assert.assertEquals;
import org.junit.Test;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;

public class TestJunit1 {
   String message = "Robert";
   MessageUtil messageUtil = new MessageUtil(message);

   @BeforeClass public static void beforeClass() {
      System.out.println("in before TestJunit1 class");
   }
   @AfterClass public static void  afterClass() {
      System.out.println("in after TestJunit1 class");
   }
   @Before public void before() {
      System.out.println("in before TestJunit1 test case");
   }
   @After public void after() {
      System.out.println("in after TestJunit1 test case");
   }
   @Test public void testPrintMessage() {
      System.out.println("Inside testPrintMessage()");
      assertEquals(message, messageUtil.printMessage());
   }
   @Test public void testSalutationMessage() {
      System.out.println("Inside testSalutationMessage()");
      message = "Hi!" + "Robert";
      assertEquals(message,messageUtil.salutationMessage());
   }
}
```

JUnit 4: test class annotations

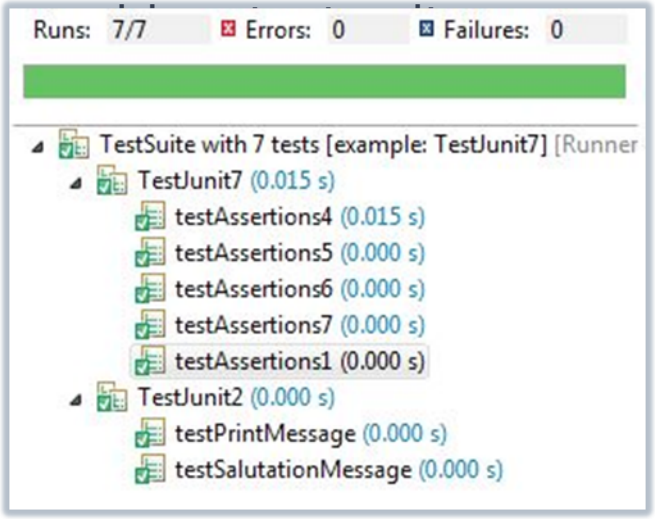
- **`@Before`**
  - Several tests need similar objects created before they can run. Annotating a **`public void`** method with **`@Before`** causes that method to be run before each test method.
- **`@After`**
  - If you allocate external resources in a **`@Before`** method you need to release them after the test runs. Annotating a **`public void`** method with **`@After`** causes that method to be run after each test method.
- **`@BeforeClass`**
  - Annotating a **`public static void`** method with **`@BeforeClass`** causes it to be run once before any of the test methods in the class. As these are static methods, they can only work upon static members.
- **`@AfterClass`**
  - Perform the following **`public static void`** method after all tests have finished. This can be used to perform clean-up activities. As these are static methods, they can only work upon static members.

JUnit 4: test class annotations

- **`@Test`**
  - Tells JUnit that the **`public void`** method to which it is attached can be run as a test case.
- **`@Ignore`**
  - Ignore the test and that test will not be executed.

Test suite: JUnit3

- A test suite is a **composite** of test cases.
- Allows to group test classes together.
- Defining a JUnit 3 test suite requires that test case classes be subclasses of TestCase.
- A JUnit 3 test case runner such as the one embedded in Eclipse will expect a **`suite()`** method to

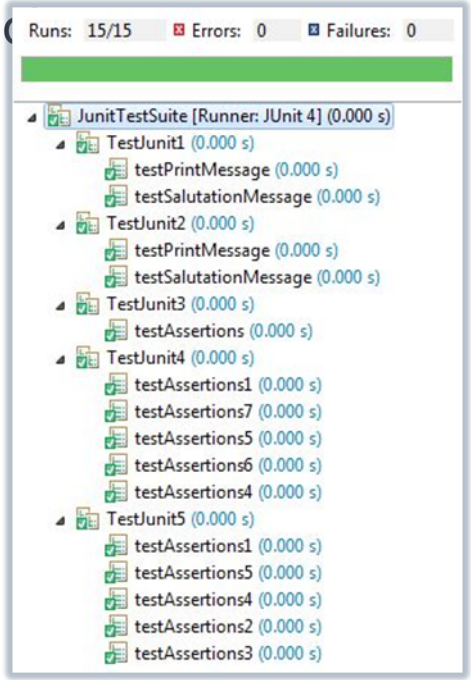



```java
// JUnit 3 test suite
import junit.framework.*;
//class that a test case runner uses
//to automatically run test cases
public class JunitTestSuiteRunner {
//needs to implement the suite() method
//that the test case runner uses
  public static Test suite(){
    // add the tests in the suite
    TestSuite suite = new TestSuite();
    //test classes need to extend TestCase
    suite.addTestSuite(TestJunit7.class);
    suite.addTestSuite(TestJunit2.class);
    return suite;
  }
}
```

Test suite: JUnit4

- Junit 4 uses annotations to achieve the same purpose.
- **@RunWith** specifies what test case runner will be used to run the test case class.
- **@SuiteClasses** specifies what test classes will be part of the test suite.
- Can include either JUnit 3 or JUnit 4 test c

```
// JUnit 4 test suite
//class that a test case runner uses
//to automatically run test cases

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;
@RunWith(Suite.class)
@SuiteClasses({TestJunit1.class,
               TestJunit2.class,
               TestJunit3.class,
               TestJunit4.class,
               TestJunit5.class})
public class JunitTestSuite {
}
```



JUnit: In the project

- You will be asked to write an increasing number of test cases for each build.
- Among these, some specific test cases will be asked for.
- Other test cases to your discretion, but all must be relevant and valid.
- All test cases should have the same structure:
  - Setting the context for the test
  - Call a method
  - Use JUnit assertions to verify that the method had correct result/effects.
- All test cases must be clearly documented using Javadoc.
  - Description of what the test case is testing.
  - Context (values set before the method is called, parameter values passed)
  - Expected result/effects.
- All JUnit classes should be in a separate folder whose structure should mirror the structure of your project.
- There should be a one-to-one relationship between your project classes and the JUnit classes.
- There should be one test suite for each module/folder, and one global test suite.

References

- TutorialsPoint. Junit Quick Guide.
- Junit.org. Frequently Asked Questions.