Esercitazioni
Prova Finale 2019

Part 2 - Testing in Practice

Docente Gianpaolo Cugola

Esercitatore Mario Scrocca

Code: https://github.com/marioscrock/ingsoft-prova-finale-19

Why Testing?

- Systematically writing tests improves your **confidence** on your code
- Is it an overkill or a non-sense to write *code that tests code*?
- Yes, it requires time and resources, but…
- Writing **automatic tests improves productivity**: once you change a line of code, you can run tests again with a simple "click" and see if you broke something
- You can diminish the **gap between what your application *is meant to do* and what it really *does*!**

What?

- **<u>JUnit is a framework for unit testing</u>**

- It provides an **API** for writing classes that test others by means of *asserting* the results of the operations. For example: *assertEquals(4, sqrt(16))*

- It provides also ***runners*** that run the tests and provide their result in a convenient format

- It provides maven plugins for running tests by executing the "**test" goal**

- Eventually, it provides plugins for running tests in IDEs and for displaying them in a nice way

https://junit.org/junit4/

Demo: Quick start with JUnit

- Open es1 folder from the repo
- Write a simple test class
- Run it

https://github.com/marioscrock/ingsoft-prova-finale-19

Deep Dive - A Test Class

```java
public class AssertTests {
  @Test
  public void testAssertArrayEquals() {
    byte[] expected = "trial".getBytes();
    byte[] actual = "trial".getBytes();
    assertArrayEquals("failure - byte arrays not same", expected, actual);
  }

  @Test
  public void testAssertEquals() {
    assertEquals("failure - strings are not equal", "text", "text");
  }

  @Test
public void testAssertFalse() {
  assertFalse("failure - should be false", false);
}

@Test
public void testAssertNotNull() {
  assertNotNull("should not be null", new Object());
}

@Test
public void testAssertNotSame() {
  assertNotSame("should not be same Object", new Object(), new Object());
}

@Test
public void testAssertNull() {
  assertNull("should be null", null);
}
```

https://github.com/junit-team/junit4/wiki/Assertions

Writing a Test Class

- Nothing special, it is a “normal” class with some *annotations* on top of methods

- When you write *@Test* on top of a method it becomes a test and it will be run as a test (we will see later what it means)

- Assertions are procedures provided by JUnit that raise an exception in case they are wrong and make the test **fail**

- The standard signature for assertions is:

```
assertSomething(“Message in case of failure”,
expected, actual)
```

```
import static org.hamcrest.CoreMatchers.allOf;
import static org.hamcrest.CoreMatchers.anyOf;
import static org.hamcrest.CoreMatchers.both;
import static org.hamcrest.CoreMatchers.containsString;
import static org.hamcrest.CoreMatchers.equalTo;
import static org.hamcrest.CoreMatchers.everyItem;
import static org.hamcrest.CoreMatchers.hasItems;
import static org.hamcrest.CoreMatchers.not;
import static org.hamcrest.CoreMatchers.sameInstance;
import static org.hamcrest.CoreMatchers.startsWith;
import static org.junit.Assert.assertArrayEquals;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertNotNull;
import static org.junit.Assert.assertNotSame;
import static org.junit.Assert.assertNull;
import static org.junit.Assert.assertSame;
import static org.junit.Assert.assertThat;
import static org.junit.Assert.assertTrue;

import java.util.Arrays;

import org.hamcrest.core.CombinableMatcher;
import org.junit.Test;
```

Matchers

- Matchers are POJOs;
- They are objects that encapsulate a property for some class of objects in a declarative way;
- Signature:
  `assertThat(String reason, actual, Matcher)`

```
// Core Hamcrest Matchers with assertThat
@Test
public void testAssertThatHamcrestCoreMatchers() {
  assertThat("good", allOf(equalTo("good"), startsWith("good")));
  assertThat("good", not(allOf(equalTo("bad"), equalTo("good"))));
  assertThat("good", anyOf(equalTo("bad"), equalTo("good")));
  assertThat(7, not(CombinableMatcher.<Integer> either(equalTo(3)).or(equalTo(4))));
  assertThat(new Object(), not(sameInstance(new Object())));
}
```

https://github.com/junit-team/junit4/wiki/Matchers-and-assertthat

Assumptions

- Making dependencies on assumptions *explicit* can improve design
- If the assumption is not satisfied, ignore the test!

```
import static org.junit.Assume.*
@Test public void filenameIncludesUsername() {
    assumeThat(File.separatorChar, is('/'));
    assertThat(new User("optimus").configFileName(), is("configfiles/optimus.cfg"));
}

@Test public void correctBehaviorWhenFilenameIsNull() {
   assumeTrue(bugFixed("13356"));  // bugFixed is not included in JUnit
   assertThat(parse(null), is(new NullDocument()));
}
```

https://github.com/junit-team/junit4/wiki/Assumptions-with-assume

Fixtures

- “A fixed state of a set of objects used as a baseline for running tests”

- Create a well known fixed environment on which tests run —> improves repeatability

- Used for loading/preparing input data or mock objects

- Specified with annotation: *@Before** and *@After**

Fixtures

```
import java.io.Closeable;
import java.io.IOException;

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class TestFixturesExample {
  static class ExpensiveManagedResource implements Closeable {
    @Override
    public void close() throws IOException {}
  }

  static class ManagedResource implements Closeable {
    @Override
    public void close() throws IOException {}
  }

  @BeforeClass
  public static void setUpClass() {
    System.out.println("@BeforeClass setUpClass");
    myExpensiveManagedResource = new ExpensiveManagedResource();
  }

  @AfterClass
  public static void tearDownClass() throws IOException {
    System.out.println("@AfterClass tearDownClass");
    myExpensiveManagedResource.close();
    myExpensiveManagedResource = null;
  }

  private ManagedResource myManagedResource;
  private static ExpensiveManagedResource myExpensiveManagedResource;
```

```
  private void println(String string) {
    System.out.println(string);
  }

  @Before
  public void setUp() {
    this.println("@Before setUp");
    this.myManagedResource = new ManagedResource();
  }

  @After
  public void tearDown() throws IOException {
    this.println("@After tearDown");
    this.myManagedResource.close();
    this.myManagedResource = null;
  }

  @Test
  public void test1() {
    this.println("@Test test1()");
  }

  @Test
  public void test2() {
    this.println("@Test test2()");
  }
}
```

https://github.com/junit-team/junit4/wiki/Test-fixtures

Fixture - the output

```
@BeforeClass setUpClass
@Before setUp
@Test test2()
@After tearDown
@Before setUp
@Test test1()
@After tearDown
@AfterClass tearDownClass
```

Rules

Rules —> “flexible addition or redefinition of the behaviour of each test method in a test class”. Fosters object composition over using annotations. Rules can be reused in different tests (for more: JUnit wiki);

```
public static class HasTempFolder {
  @Rule
  public final TemporaryFolder folder = new TemporaryFolder();

  @Test
  public void testUsingTempFolder() throws IOException {
    File createdFile = folder.newFile("myfile.txt");
    File createdFolder = folder.newFolder("subfolder");
    // ...
  }
}
```

e.g. The **TemporaryFolder** Rule allows creation of files and folders that are deleted when the test method finishes (whether it passes or fails). By default no exception is thrown if resources cannot be deleted

Theories

Theories —> A theory is a test that, given some assumptions, is executed over a set of data (for more: JUnit wiki)

```
@RunWith(Theories.class)
public class UserTest {
    @DataPoint
    public static String GOOD_USERNAME = "optimus";
    @DataPoint
    public static String USERNAME_WITH_SLASH = "optimus/prime";

    @Theory
    public void filenameIncludesUsername(String username) {
        assumeThat(username, not(containsString("/")));
        assertThat(new User(username).configFileName(), containsString(username));
    }
}
```

Mocking

- It happens that we have to execute some other class code in order to test one class

- This is bad, because in **unit testing**, we want to test a single, independent unit of work

- Mock objects are **fake objects** that help in avoiding testing dependencies

- A **mock object implements the same interface** of its “real” counterpart, but **can verify the behaviour of the calls against it and return predefined values**

- **Mock vs Stubs (**easy: this link, detailed: this link)

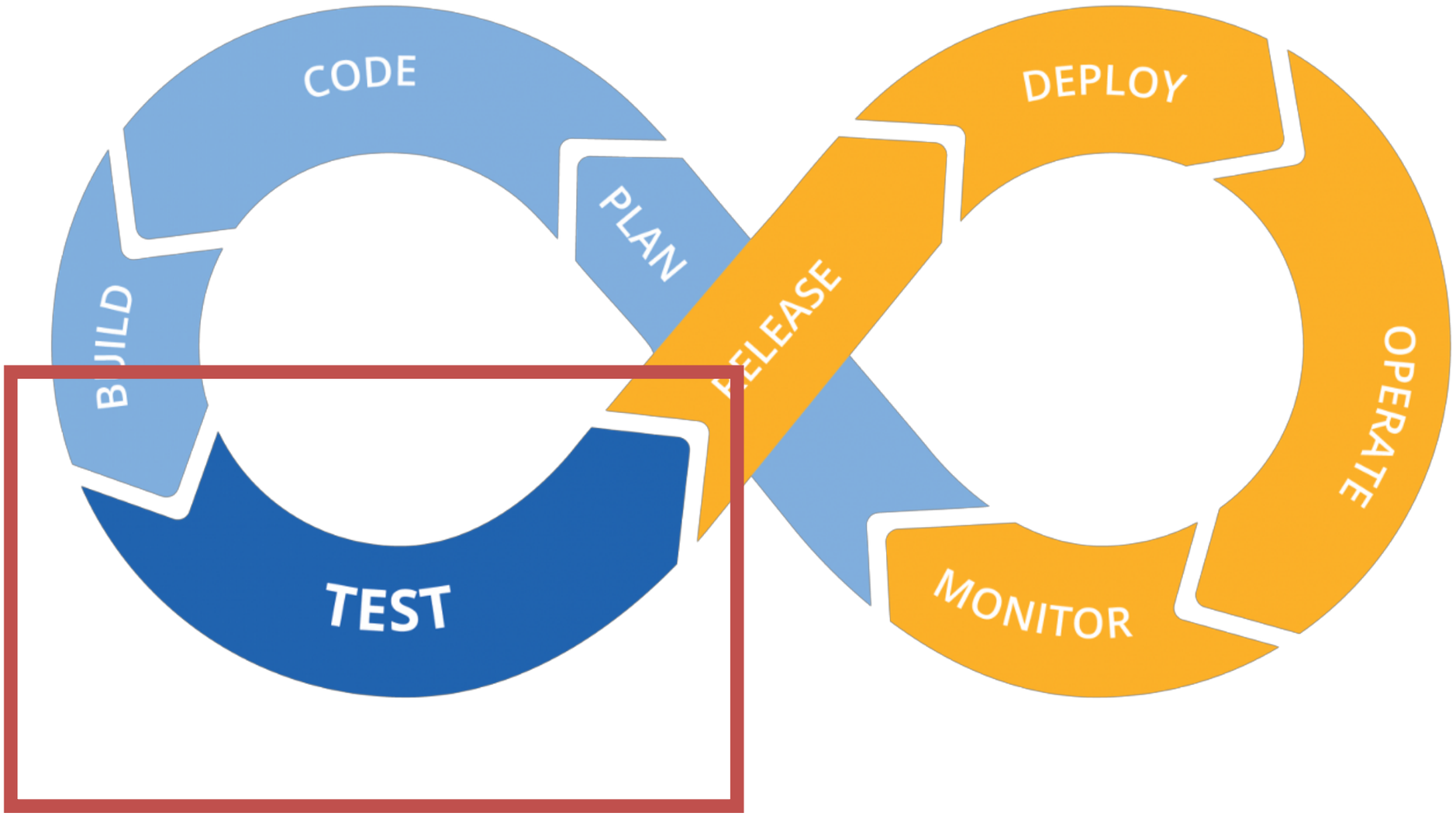
Examples

```
// You can mock concrete classes and interfaces
TrainSeats seats = mock(TrainSeats.class);

// stubbing appears before the actual execution
when(seats.book(Seat.near(WINDOW).in(FIRST_CLASS))).thenReturn(BOOKED);

// the following prints "BOOKED"
System.out.println(seats.book(Seat.near(WINDOW).in(FIRST_CLASS)));

// the following prints "null" because
// .book(Seat.near(AISLE).in(FIRST_CLASS))) was not stubbed
System.out.println(seats.book(Seat.near(AISLE).in(FIRST_CLASS)));

// the following verification passes because
// .book(Seat.near(WINDOW).in(FIRST_CLASS)) has been invoked
verify(seats).book(Seat.near(WINDOW).in(FIRST_CLASS));

// the following verification fails because
// .book(Seat.in(SECOND_CLASS)) has not been invoked
verify(seats).book(Seat.in(SECOND_CLASS));
```

https://github.com/mockito/mockito/wiki

CI/CD

Continuous Integration/Delivery



Credits to https://blog.gds-gov.tech/that-ci-cd-thing-principles-implementation-tools-

Time for coding guys/girls!

Chat Exercise

Do you know MVC?

- **Model**: User, Group, Message
- **Controller**: SimpleClient
- **View**: ViewClient

Interactions?

What problems can the Observer design pattern solve?

The **Observer pattern** addresses the following problems:

- A decoupled **one-to-many dependency**
- Changes on the observed object are **automatically reported.**
- O**pen-ended number** of observing objects.

Observer Pattern