Test unitaire

Test Unitaire

➔ Pourquoi tester ?

- Moyen d'exécuter le code d’une fonctionnalité développée
- Exécutable facilement (via l’IDE, maven et donc intégration continue)
- Repérer les bugs au plus tôt
- **Et surtout** pour ne pas casser du code existant

Test Unitaire

- Type de test
  - Test unitaire (test d'une portion du code isolée du reste de l'application)
  - Test d'intégration (test d'une fonctionnalité dans un environnement proche de la réalité)
  - Test de régression (test mettant en évidence un bug)
  - Test boite noire (test sans connaître l'implémentation de la fonctionnalité testé)
  - Test boite blanche (test en tenant compte du code de la fonctionnalité testé)
  - Test de performance
  - Test fonctionnel (test sur un véritable environnement par un utilisateur avec l'UI par exemple)

Test Unitaire

- TDD (Test Driven Development)
  - Développement piloté par les tests
  - On écrit le test avant la fonctionnalité elle même
  - Force à faire du code testable

Test Unitaire

- Framework de test Java
  - JUnit
  - TestNG
- Outil complémentaire de test
  - Matchers (Hamcrest, Fest, AssertJ)
  - Mock (Mockito, EasyMock, PowerMock)

Test Unitaire

- JUnit
  - Mettre les classes de test dans src/test/java
  - Suffixer la classe par ‘Test’ (ex: MyServiceTest)
    - *‘mvn test’ lance les tests*
  - @Test sur une méthode (public void) pour déclarer un test

```
public class MyService {

  public String surroundBracket(String name) {
    return "(" + name + ")";
  }
}
```

```
public class MyServiceTest {

  @Test
  public void testAddBracket() {
    MyService myService = new MyService();
    String name = "toto";
    String bracketName = myService.surroundBracket(name);
    Assert.assertEquals("(" + name + ")", bracketName);
  }
}
```

Test Unitaire

➔ **@Before** sur une méthode (**public void**) pour exécuter cette méthode avant chaque test

```
public class MyService {

   public String surroundBracket(String name) {
      return "(" + name + ")";
   }
}
```

```
public class MyServiceTest {

   private MyService myService;

   @Before
   public void before() {
      myService = new MyService();
   }

   @Test
   public void testAddBracket() {
      String name = "toto";
      String bracketName =
myService.surroundBracket(name);
      Assert.assertEquals("(" + name + ")", bracketName);
   }

   @Test
   public void testAddBracketNull() {
      String bracketName = myService.surroundBracket(null);
      Assert.assertEquals("(null)", bracketName);
   }
}
```

Test Unitaire

➔ **@BeforeClass** sur une méthode (**public static void**) pour exécuter cette méthode avant **tous** les tests de cette classe de test

```
public class MyService {

   public String surroundBracket(String name)
{
      return "(" + name + ")";
   }
}
```

```
public class MyServiceTest {

   private static MyService myService;

   @BeforeClass
   public static void beforeClass() {
      myService = new MyService();
   }

   @Test
   public void testAddBracket() {
      String name = "toto";
      String bracketName = myService.surroundBracket(name);
      Assert.assertEquals("(" + name + ")", bracketName);
   }

   @Test
   public void testAddBracketNull() {
      String bracketName = myService.surroundBracket(null);
      Assert.assertEquals("(null)", bracketName);
   }
}
```

Test Unitaire

- **@After** (méthode exécutée après chaque test)
- **@AfterClass** (méthode exécutée après tous les tests de la classe de test)
- **@Ignore** (ignore le test même s'il est annoté @Test)

- Suite Test
  - Pour exécuter un groupe de test

```
@RunWith(Suite.class)
@Suite.SuiteClasses({MyServiceTest.class, MyService2Test.class})
public class AllTest {

}
```

Test Unitaire

- ➔ Assert JUnit
  - ◆ Avec l'API de JUnit on a dans la classe Assert.java

```
static public void assertEquals(long expected, long actual)
static public void assertEquals(Object expected, Object actual)
static public void assertTrue(boolean condition)
```

```
Assert.assertEquals("ok", "ok");
Assert.assertEquals(true, true);
Assert.assertEquals(5, 5);

Assert.assertNotEquals(4f, 5f);

String[] strings = {"a", "b", "c"};
Assert.assertArrayEquals(strings, strings);

Assert.assertTrue("test".contains("s"));
```

Test Unitaire

- ➔ Hamcrest
  - JUnit intègre Hamcrest qui introduit la notion de Matcher

```
public static <T> void assertThat(T actual, Matcher<? super T> matcher)
```

  - Matchers : **Is**, **IsNot**, **IsNull**, **AllOf**, **AnyOf**, ...

```
Assert.assertThat("ok", Is.is("ok"));

Assert.assertThat("this", IsNot.not("that"));

Assert.assertThat("this", IsNot.not(IsNull.nullValue()));

Assert.assertThat("this", IsNull.notNullValue());

Assert.assertThat("test", StringContains.containsString("s"));

Assert.assertThat(Arrays.asList(1, 2, 3, 4, 5, 6, 7), IsCollectionContaining.hasItem(3));
```

Test Unitaire

- Hamcrest
  - En utilisant des import static
  - Plus simple à lire

```
Assert.assertThat("ok", is("ok"));

Assert.assertThat("this", not("that"));

Assert.assertThat("this", not(nullValue()));

Assert.assertThat("this", notNullValue());

Assert.assertThat("test", containsString("s"));

Assert.assertThat(Arrays.asList(1, 2, 3, 4, 5, 6, 7), hasItem(3));
```

  - Matchers disponible dans org.hamcrest.*
  - Encore plus de matchers dans

```
<dependency>
  <groupId>org.hamcrest</groupId>
  <artifactId>hamcrest-all</artifactId>
  <version>1.3</version>
</dependency>
```

```
import static org.hamcrest.core.Is.*;
import static org.hamcrest.core.IsCollectionContaining.*;
import static org.hamcrest.core.IsNot.*;
import static org.hamcrest.core.IsNull.*;
import static org.hamcrest.core.StringContains.*;
```

Test Unitaire

➔ AssertJ

```
Assertions.assertThat("test").isEqualTo("test");
Assertions.assertThat(Lists.newArrayList("a", "b", "c", "d")).contains("b", Index.atIndex(1));
```

Test Unitaire

- ➔ Assertions sur le type **double**
  - ◆ Ne marche pas comme on voudrait

```
assertThat(0.1 + 0.2, is(0.3));

java.lang.AssertionError:
      Expected: is <0.3>
      but: was <0.30000000000000004>
      Expected :is <0.3>

      Actual   :<0.30000000000000004>
```

  - ◆ On peut utiliser IsCloseTo.closeTo(double operand, double error)

```
assertThat(0.1 + 0.2, closeTo(0.3, 0.1));
```

Test Unitaire

- ➔ Mock
  - Permet de “simuler” l’appel à une méthode
    - *méthodes de DAO car pas de BD lors des tests unitaires*
    - *méthodes qui font des requêtes http*
    - *...*
  - Ici on va vouloir mocker BracketService

```
public class MyService {

   private BracketService bracketService;

   public MyService(BracketService bracketService) {
      this.bracketService = bracketService;
   }

   public String surroundBracket(String name) {
      return bracketService.openingBracket() + name + bracketService.closingBracket();
   }
}
```

```
public class BracketService {
   public String openingBracket() {
      System.out.println("call openingBracket() but looooooong");
      try {
         Thread.sleep(10000l);
         return "(";
      } catch (InterruptedException e) {
         throw new RuntimeException(e);
      }
   }

   public String closingBracket() {
      System.out.println("call closingBracket() but looooooong");
      try {
         Thread.sleep(10000l);
         return ")";
      } catch (InterruptedException e) {
         throw new RuntimeException(e);
      }
   }

   public void useless() {

   }
}
```

Test Unitaire

- Avant
  - BracketService est la vraie implémentation du service

```
public class MyServiceTest {

  private static MyService myService;

  @BeforeClass
  public static void before() {
     BracketService bracketService = new
BracketService();
     myService = new MyService(bracketService);
  }

  @Test
  public void testAddBracket() {
     String name = "toto";
     String bracketName =
myService.surroundBracket(name);
     assertThat(bracketName, is("(" + name + ")"));
  }
}
```

Test Unitaire

- Avec un mock fait à la main... pas top
  - On override (surcharge) les méthodes de classe en renvoyant ce dont on a besoin

```
public class MyServiceTest {

  private static MyService myService;

  @BeforeClass
  public static void before() {
     BracketService bracketService = new BracketService() {
        @Override
        public String openingBracket() {
           return "(";
        }

        @Override
        public String closingBracket() {
           return ")";
        }
     };
     myService = new MyService(bracketService);
  }

  @Test
  public void testAddBracket() {
     String name = "toto";
     String bracketName = myService.surroundBracket(name);
     assertThat(bracketName, is("(" + name + ")"));
  }
}
```

Test Unitaire

➔ Avec Mockito... bien mieux

```
public class MyServiceTest {

    private MyService myService;
    private BracketService bracketServiceMock;

    @Before
    public void before() {
        bracketServiceMock = Mockito.mock(BracketService.class);
        Mockito.when(bracketServiceMock.openingBracket()).thenReturn("(");
        Mockito.when(bracketServiceMock.closingBracket()).thenReturn(")");
        myService = new MyService(bracketServiceMock);
    }

    @Test
    public void testAddBracket() {
        String name = "toto";
        String bracketName = myService.surroundBracket(name);
        assertEquals("(" + name + ")", bracketName);
        Mockito.verify(bracketServiceMock).openingBracket();
        Mockito.verify(bracketServiceMock).closingBracket();
        Mockito.verify(bracketServiceMock, Mockito.times(0)).useless();
    }
}
```

Test Unitaire

➔ **Important**

- Identifier les cas de tests à effectuer
- Ne pas tester ce que vous ne développez pas
  - *exemple: Ne pas tester les méthodes d'un EntityManager JPA telles que find(), persist(), ...*

Bonnes pratiques

Nommage

- Conventions de nommage
  - package
  - classe
  - méthode

Nommage

- Conventions de nommage
  - package **-> lowercase**
  - classe **-> PascalCase**
  - méthode **-> camelCase**

return

```
public boolean isEmpty() {
	if (size == 0) {
		return true;
	} else {
		return false;
	}
}
```

return

```
public boolean isEmpty() {
        return size == 0;
}
```

else

```
public String getVerdict(double grade) {
	if (grade < 10) {
		return "Raté";
	} else {
		return "Réussi";
	}
}
```

else

```
public String getVerdict(double grade) {
	if (grade < 10) {
		return "Raté";
	}
	return "Réussi";
}
```

interface vs classe

```
ArrayList list = new ArrayList();
list.add("element");
```

interface vs classe

```
List list = new ArrayList();
list.add("element");
```

generics

```
List list = new ArrayList();
list.add("element");
```

generics

```
List<String> list = new ArrayList<>();
list.add("element");
```

transformation

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> square = new ArrayList<>();
for (Integer i : list) {
    square.add(i * i);
}
```

transformation

```
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
List<Integer> square = list.stream()
    .map(i -> i*i)
    .collect(Collectors.toList());

// Reste à étudier les performances en fonction de la taille de list
```