

# Tests unitaires

## Développement dirigé par les tests

### Utilisation de JUnit

Gauthier Picard

École Nationale Supérieure des Mines de Saint-Étienne

[gauthier.picard@emse.fr](mailto:gauthier.picard@emse.fr)



## Test

- ▶ Vérifier par l'exécution
- ▶ Confronter une réalisation à sa spécification
- ▶ Critère d'arrêt (état ou sorties à l'issue de l'exécution) et permet de statuer sur le succès ou sur l'échec d'une vérification

## Test unitaire

- ▶ Test d'un bloc (unité) de programme (classe, méthode, etc.)
  - ▶ Vérification des comportements corrects
  - ▶ Vérification des comportements incorrects (valeurs incohérentes des paramètres, ...)
- ▶ Approche incrémentale
- ▶ Doivent être rejoués pour vérifier la non-régression



## *Test Driven Development (TDD)*

- ▶ Initialement conceptualisé par *Erich Gamma* et *Kent Beck*
- ▶ Plus généralement intégré aux approches de développement **agile** : *eXtreme Programming*, *Scrum*, etc.
- ▶ Utilisation de tests unitaires comme spécification du code
- ▶ De nombreux langages possède leur canevas de test unitaires (SUnit, JUnit, RUnit, etc.)



## Cycle de TDD

- 1 Ecrire un premier test
- 2 Vérifier qu'il échoue (car le code qu'il teste n'existe pas), afin de vérifier que le test est valide
- 3 Ecrire juste le code suffisant pour passer le test
- 4 Vérifier que le test passe
- 5 Réviser le code (*refactoring*), i.e. l'améliorer tout en gardant les mêmes fonctionnalités



## Avantages

- ▶ Ecrire les tests d'abord  $\Rightarrow$  on utilise le programme avant même qu'il existe
- ▶ Diminue les erreurs de conception
- ▶ Augmente la confiance en soi du programmeur lors de la révision du code
- ▶ Construction conjointe du programme et d'une suite de tests de non-régression
- ▶ Estimer l'état d'avancement du développement d'un projet (vélocité)

Référence :

Kent Beck, *Test Driven Development : By Example*, Addison-Wesley, 2002.



## JUnit 4

- ▶ *framework* Java
- ▶ open source : [www.junit.org](http://www.junit.org)
- ▶ intégré à Eclipse

## Principe

- ▶ Une classe de tests unitaires est associée à une autre classe
- ▶ Une classe de tests unitaires hérite de la classe `junit.framework.TestCase` pour bénéficier de ses méthodes de tests
- ▶ Les méthodes de tests sont identifiées par des *annotations* Java



## Méthodes de tests

- ▶ Nom quelconque
- ▶ visibilité `public`, type de retour `void`
- ▶ pas de paramètre, peut lever une exception
- ▶ annotée `@Test`
- ▶ utilise des instructions de test



Instruction	Description
<code>fail(String)</code>	fait échouer la méthode de test
<code>assertTrue(true)</code>	toujours vrai
<code>assertEquals(expected, actual)</code>	teste si les valeurs sont les mêmes
<code>assertEquals(expected, actual, tolerance)</code>	teste de proximité avec tolérance
<code>assertNull(object)</code>	vérifie si l'objet est <code>null</code>
<code>assertNotNull(object)</code>	vérifie si l'objet n'est pas <code>null</code>
<code>assertSame(expected, actual)</code>	vérifie si les variables référencent le même objet
<code>assertNotSame(expected, actual)</code>	vérifie que les variables ne référencent pas le même objet
<code>assertTrue(boolean condition)</code>	vérifie que la condition booléenne est vraie

L'instruction la plus importante est **`fail()`** :  
les autres ne sont que des raccourcis d'écriture !



## Example

```
class Money {
    private int fAmount;
    private String fCurrency;

    public Money(int amount, String currency) {
        fAmount = amount;
        fCurrency = currency;
    }

    public int amount() {
        return fAmount;
    }

    public String currency() {
        return fCurrency;
    }

    public Money add(Money m) {
        return new Money(amount() + m.amount(), currency());
    }
}
```



## Example

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

public class MoneyTest {

    @Test
    public void testSimpleAdd() {
        Money m12CHF = new Money(12, "CHF"); // création de données
        Money m14CHF = new Money(14, "CHF");
        Money expected = new Money(26, "CHF");
        Money result = m12CHF.add(m14CHF); // exécution de la méthode testée
        assertTrue(expected.equals(result)); // comparaison
    }
}
```



Les annotations sont à placer avant les méthodes d'une classe de tests unitaires

Annotation	Description
@Test	méthode de test
@Before	méthode exécutée <i>avant chaque test</i>
@After	méthode exécutée <i>après chaque test</i>
@BeforeClass	méthode exécutée <i>avant le premier test</i>
@AfterClass	méthode exécutée <i>après le dernier test</i>
@Ignore	méthode qui ne sera pas lancée comme test



### Example

```
public class MoneyTest {
    private Money f12CHF;
    private Money f14CHF;
    private Money f28USD;

    @Before
    public void setUp() {
        f12CHF= new Money(12, "CHF");
        f14CHF= new Money(14, "CHF");
        f28USD= new Money(28, "USD");
    }
}
```



## Appel des tests

- ▶ en ligne de commande :

```
java org.junit.runner.JUnitCore TestClass1 [...other test classes...]
```

- ▶ depuis un code Java :

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

- ▶ depuis Eclipse : Run > Run As > JUnit test

## Ordre d'exécution

- 1 La méthode annotée `@BeforeAll`
- 2 Pour chaque méthode annotée `@Test` (ordre indéterminé)
  - 1 Les méthodes annotées `@Before` (ordre indéterminé)
  - 2 La méthode annotée `@Test`
  - 3 Les méthodes annotées `@After` (ordre indéterminé)
- 3 La méthode annotée `@AfterAll`



## Suite de tests

- ▶ utilisation des annotations
- ▶ utilisation d'un autre programme d'exécution que celui par défaut :  
`org.junit.runners.Suite`
- ▶ pour changer de programme d'exécution : `@RunWith(Class)`
- ▶ classe vide annotée `@RunWith(Suite.class)`
- ▶ pour indiquer comment former la suite de test :  
`@SuiteClasses(Class[])`



## Example

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;

@RunWith(Suite.class)
@SuiteClasses(value = { MoneyTest.class, MoneyBagTest.class })
public class AllTests {
}
```

