Tests Unitaires avec JUNIT

Erick STATTNER

Maître de Conférences en Informatique

Université des Antilles

erick.stattner@univ-antilles.fr

www.erickstattner.com

Description de l'enseignement

Objectifs pédagogiques:

- Se familiariser avec la notion de Tests Unitaires
- Identifier les critères d'évaluation adaptés au contexte
- Mettre en place des tests unitaires à l'aide JUNIT
- Evaluer la couverture des tests

Organisation:

- 3h incluant CM, exercices

Sommaire

1. Introduction aux tests unitaires
2. Mise en place des tests avec JUnit
3. Couverture des tests
4. Pour aller plus loin

Chapitre I.
Introduction aux Tests Unitaires

1. Introduction
2. Définition
3. Principes

I. Introduction aux Tests Unitaires

1) Introduction

Le test ?

I. Introduction aux Tests Unitaires

1) Introduction

Qu'est-ce que le test ?

- Procédure qui vise à identifier les comportements problématiques d'un programme afin d'en augmenter la qualité
- Vérifier par l'exécution
- Confronter la réalisation aux
  - Spécifications
  - Exigences
- Doit répondre à la question:
  La réalisation fait-elle bien
    - Ce qui devait être implémenté ?
    - Ce que voulait le client ?
- Permet de statuer sur le **succès** ou l'**échec** d'une vérification

I. Introduction aux Tests Unitaires

1) Introduction

Pourquoi tester ?

- Pour gagner du temps
  - Gain du temps passé à debugger
- Pour valider le logiciel
- Pour valider les remaniements du code
- Pour détecter les régressions

I. Introduction aux Tests Unitaires

1) Introduction

Votre expérience du test

- **SOUVENT**: Affichage en console
- **PARFOIS**: Traces des fonctions / programmes avec quelques cas
- **RAREMENT**: Utilisateur d'un debugger

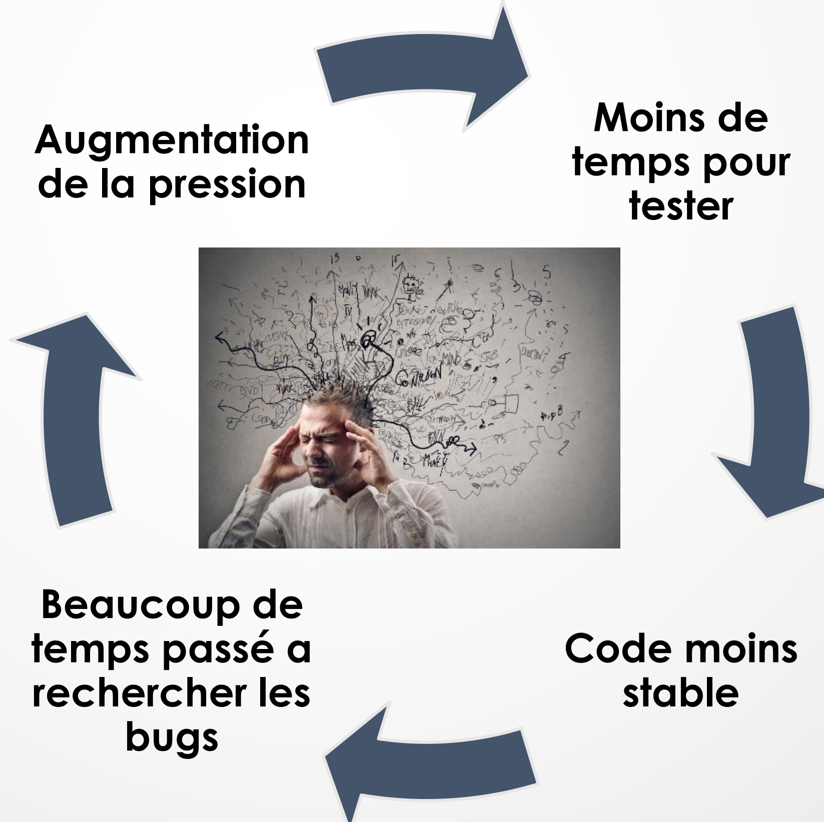
Problèmes

- Difficulté à isoler les éléments que vous voulez tester
- Perte de lisibilité à cause des nombreux affichages
- Plus généralement: manque de recul / hauteur !
  - **Celui qui code est également celui qui teste !**

I. Introduction aux Tests Unitaires

1) Introduction

En pratique: paradoxe du test



I. Introduction aux Tests Unitaires

1) Introduction

4 niveaux de tests

- **Unitaire***
  - Tester les parties élémentaires d'un programme
- **Intégration**
  - Tester la bonne collaboration des classes

En amont

- **Système**
  - Tester la conformité à un système spécifique
- **Acceptation *(user acceptance testing)***
  - Tester la conformité du produit aux exigences

En aval

I. Introduction aux Tests Unitaires

2) Définition

Test unitaire

- Procédure permettant de vérifier le bon fonctionnement
  - d'une partie précise d'un logiciel
  - d'une portion d'un programme (appelée « **unité** » ou « **module** »)

Wikipédia, 2016

Objectifs

- S'assurer qu'une unité fonctionnelle ne comporte pas d'erreurs
- Vérifier que les classes collaborent bien
- Garantir l'identification des erreurs au fur et à mesure des modifications du code

I. Introduction aux Tests Unitaires

2) Définition

Intérêt des tests

- Segmenter les portions à tester
- Développer efficacement
- Gagner du temps
- Mieux supporter les évolutions du code
- Faciliter le travail en équipe

I. Introduction aux Tests Unitaires

3) Principes

Test unitaire repose sur deux principes simples

1. **SI** ça fonctionne une fois, **ALORS** ça fonctionnera les autres fois
2. **SI** ça fonctionne pour quelques valeurs bien identifiées, **ALORS** ça fonctionnera pour toutes les autres

Exemple

- Soit la méthode *String concatene(String t1, String t2)* qui concatène les chaines de caractères t1 et t2
  - SANS TESTS UNITAIRES
    - Afficher le résultat de la méthode dans le programme lors de l'appel de la fonction
  - AVEC LES TESTS UNITAIRES
    - Créer une classe dédiée qui se chargera de faire un ensemble de tests avec différentes valeurs

Doit-on tester tous les cas possibles ?

I. Introduction aux Tests Unitaires

3) Principes

Schéma de test de classique:

- Un état de départ
- Un état attendu
- Un état d'arrivée
- Un oracle: calcule l'état d'arrivé, et vérifie qu'il correspond à l'état attendu

```
public boolean testConcatene(){
    String t1 = "Bonjour " ;
    String t2 = "Toto";
    String attendu = "Bonjour Toto";
    String arrivee = concatene(t1, t2);
    Return arrivee.equals(attendu);
}
```

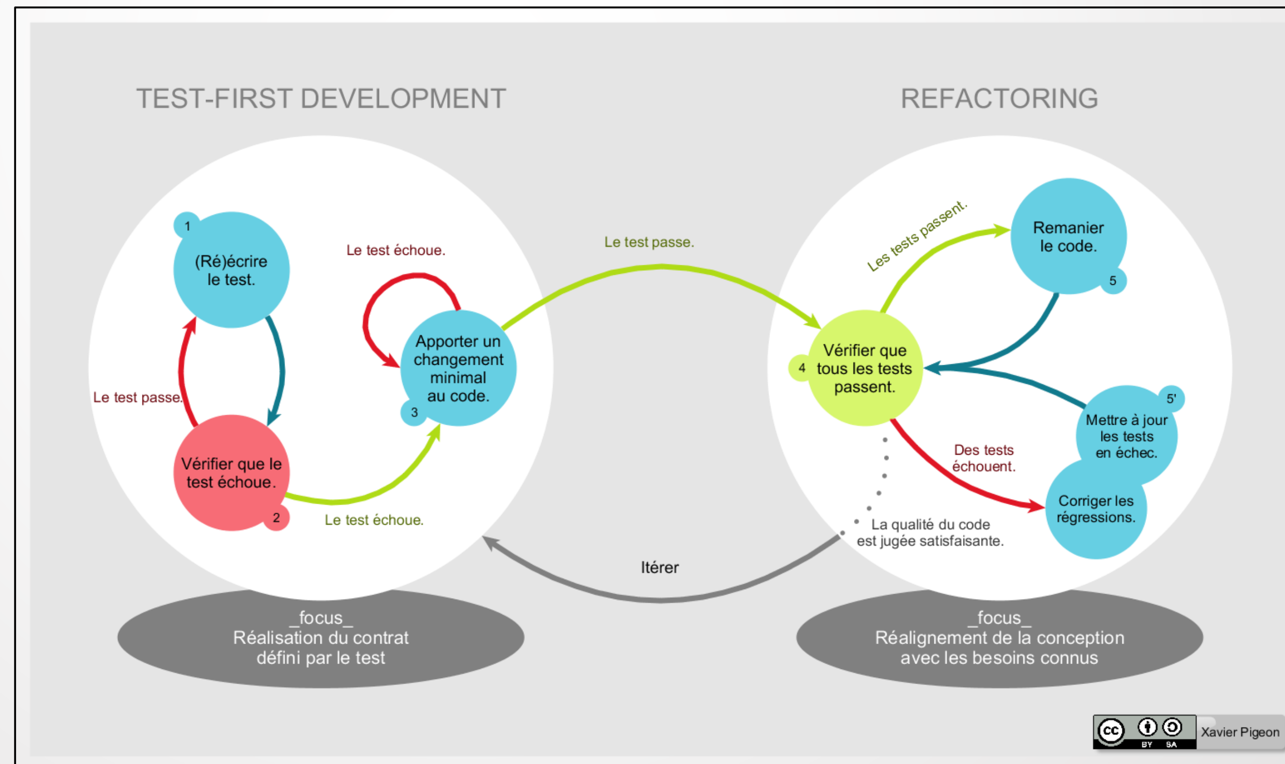
Pour coder ce test, il n'est pas nécessaire de savoir comment est codé la méthode *concatene*

I. Introduction aux Tests Unitaires

3) Principes

Test Driven Development (TDD)

- Proposé par Kent Beck, 2002
- Ecrire les tests avant de coder l'application (Méthodes Agiles XP, Scrum, etc.)



I. Introduction aux Tests Unitaires

3) Principes

Avantages du TDD

- Ecrire les tests en premier
  - Manipuler le programme avant meme son existence
- Permet de segmenter les taches
- Garanti que l'ensemble du programme est testé
- Facilite le travail en équipe et la collaboration entre les modules
- Augmente la confiance du programmeur lors de la modification du code
- Evite la régression

I. Introduction aux Tests Unitaires

3) Principes

Tests unitaires et langage de programmation

- Framework pour mettre en place des tests unitaires
  - **JAVA: Framework JUnit**
    http://junit.org/junit4/
  - C: Framework CUnit
    http://cunit.sourceforge.net/
  - PHP: Framework PHPUnit
    https://phpunit.de/
  - …

Chapitre II.
Mise en place de Tests avec JUnit

1. Présentation de JUnit
2. Classe et méthodes de tests
3. Instructions de tests
4. Annotations
5. Exécution des Tests
6. Tests et exception

II. Mise en place de Tests avec JUnit

1) Présentation

Framework JUnit

- Framework permettant la mise en place de Test Unitaires en JAVA
- Open Source: www.junit.org
- Créé par
  Kent Beck *(eXtreme Programming)*
  et
  Erich Gamma *(Design patterns)*
- Intégré par défaut dans la plupart des IDE
  - Eclipse
  - NetBeans
  - BlueJ
  - …

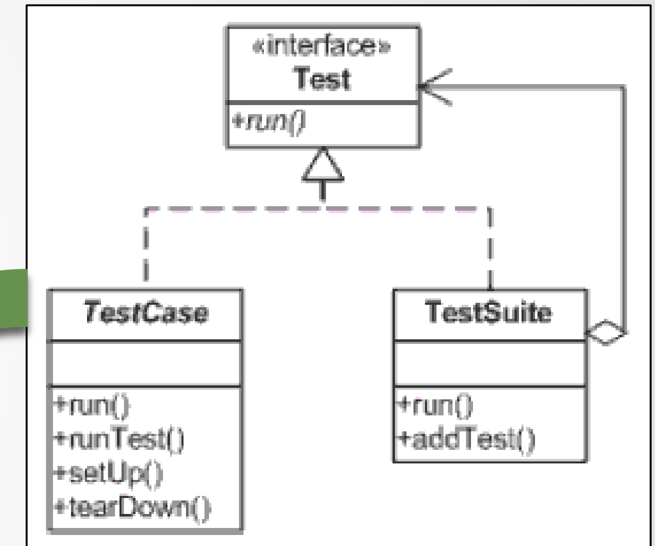
En 2013, une étude menée sur les projets hébergés sur GitHub montre que 31% des projets utilisent JUnit

II. Mise en place de Tests avec JUnit

1) Présentation

Versions de JUnit

- **Version 3.8**
  - Basé sur un ensemble de Classes et d'interfaces à implémenter
  - Mise en place des tests en implémentant des classes respectant le Framework
- **Version 4**
  - Plus souple
  - Basé sur des annotations (Java 5+)
  - Permettent de marquer les méthodes dédiées au test




```
public class monTest {
        @test
        public void test(){
        …
        }
}
```

II. Mise en place de Tests avec JUnit

1) Présentation

Principe JUnit

- A chaque classe, on associe une classe de test
- Une classe de test hérite de la classe ***junit.framework.TestCase*** pour hériter des méthodes de tests
- Depuis la version 4
  - Inutile de préciser la relation d'héritage
  - les méthodes de tests sont identifiés par des ***annotations*** Java

II. Mise en place de Tests avec JUnit

2) Classe et méthodes de tests

Classe de Test

- Nom quelconque
  - En général <nomDeLaClasseTestée>TEST.java
  - Ex. CounteurTest.java
- Importer packages
  - Import org.junit.Test;
  - Import static org.junit.Assert.*;

```
//Classe qui permet de manipuler un compteur
public class Compteur{
        int valeur;
        Compteur(){…}
        Compteur(int init){…}
        int increment(){…}
        int decrement(){…}
        int getValeur(){…}
}
```

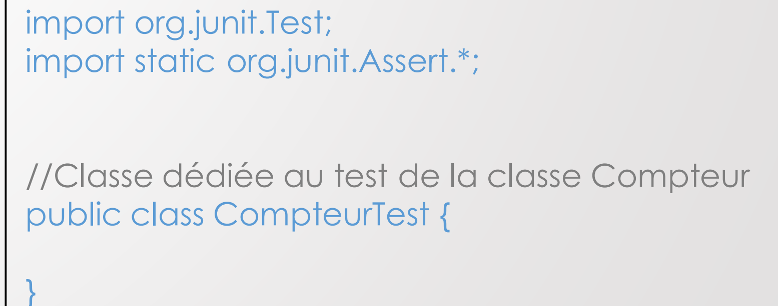
II. Mise en place de Tests avec JUnit

2) Classe et méthodes de tests

Classe de Test

- Nom quelconque
  - En général <nomDeLaClasseTestée>TEST.java
  - Ex. CounteurTest.java
- Importer packages
  - Import org.junit.Test;
  - Import static org.junit.Assert.*;

```
//Classe qui permet de manipuler un compteur
public class Compteur{
        int valeur;
        Compteur(){…}
        Compteur(int init){…}
        int increment(){…}
        int decrement(){…}
        int getValeur(){…}
}
```



II. Mise en place de Tests avec JUnit

2) Classe et méthodes de tests

Méthodes de test

- Nom quelconque
  - Commence par *test*, suivi du nom de la méthode testée
    Ex. *testIncrement()*
- Visibilité *public*
- Type de retour: *void*
- Pas de paramètres
- Peut lever une exception
- Annotée **@Test**
- Utilise **des instructions de test**

II. Mise en place de Tests avec JUnit

2) Classe et méthodes de tests

Méthodes de test

- Nom quelconque
  - Commence par *test*, suivi du nom de la méthode testée
    Ex. *testIncrement()*
- Visibilité *public*
- Type de retour: *void*
- Pas de paramètres
- Peut lever une exception
- Annotée **@Test**
- Utilise **des instructions de test**

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


//Classe dédiée au test de la classe Compteur
public class CompteurTest {
		@Test
		public void testIncrement(){
		//La méthode increment() est testée ici

		}
		@Test
		public void testDecrement(){
		//La méthode decrement() est testée ici

		}
}
```

II. Mise en place de Tests avec JUnit

3) Instructions de tests

Principales instructions de tests

Instructions	Description
fail() **fail(string)**	**Provoque l'échec de la méthode de test**
assertTrue(boolean) assertTrue(String, Boolean)	Fait échouer la méthode si le paramètre est faux
assertsEquals(expected, actual) assertsEquals(String, expected, actual)	Fait échouer la méthode si les paramètres ne sont pas égaux Se base sur la méthode **equals()**
assertSame(expected, actual) assertSame(String, expected, actual)	Fait échouer la méthode si les paramètres ne référence pas le même objet se base sur le résultat du **==**
assertNull(Object) assertNull(String, Object)	Fait échouer la méthode si le paramètre est différent de **null**

- **Bcp d'autres instructions:** assertNotNull, assertNotEquals, assertArrayEquals, …
  http://junit.sourceforge.net/javadoc/

II. Mise en place de Tests avec JUnit

3) Instructions de tests

Principales instructions de tests

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

//Classe dédiée au test de la classe Compteur
public class CompteurTest {
		@Test
		public void testIncrement(){
				int attendue;

				attendue = 1;
				Compteur c1 = new Compteur();
				c1.incrementer();
				assertTrue(c1.getValeur() == attendue);

				attendue = 11;
				Compteur c2 = new Compteur(10);
				c2.incrementer();
				assertEquals("Echec test 2",
c2.getValeur(), attendue);
		}
}
```

Valeur attendue

Création des objets pour le test

Appel de la méthode à tester

Vérification que le résultat correspond bien au résultat attendu.

Si ce n'est pas le cas:

1. Lance une *AssertionFailedError* et la méthode de test s'arrête
2. L'exécuteur de Test JUnit attrape cet objet et indique que la méthode a échoué
3. La méthode de test suivante est exécutée

II. Mise en place de Tests avec JUnit

3) Instructions de tests

La méthode *fail()*

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

//Classe dédiée au test de la classe Compteur
```

fail() est l'instruction la plus importante !
Les autres ne sont que des raccourcis d'écriture

```
            attendue = 11;
            Compteur c2 = new Compteur(10);
            c2.incrementer();
            assertEquals("Echec test 2",
c2.getValeur(), attendue);
      }
}
```

II. Mise en place de Tests avec JUnit

3) Instructions de tests

La méthode *fail()*

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

//Classe dédiée au test de la classe Compteur
public class CompteurTest {
		@Test
		public void testIncrement(){
			int attendue;

			attendue = 1;
			Compteur c1 = new Compteur();
			c1.incrementer();
			assertTrue(c1.getValeur() == attendue);

			attendue = 11;
			Compteur c2 = new Compteur(10);
			c2.incrementer();
			assertEquals("Echec test 2",
c2.getValeur(), attendue);
		}
}
```

```
attendue = 1;
Compteur c1 = new Compteur();
c1.incrementer();
if(c1.getValeur() != attendue){
		fail("ECHEC DU TEST");
}
```

II. Mise en place de Tests avec JUnit

4) Annotations

Plusieurs méthodes de tests

```
import org.junit.Test;
import static org.junit.Assert.*;
public class CompteurTest {
		@Test
		public void testIncrement(){
				Compteur c = new Compteur(10);
				c.incrementer();
				int attendu = 11;
				assertTrue(c.getValeur() == attendu);

		}
		@Test
		public void testDecrement(){
				Compteur c = new Compteur(10);
				c.decrementer();
				int attendu = 9;
				assertTrue(c.getValeur() == attendu);
		}
}
```

II. Mise en place de Tests avec JUnit

4) Annotations

Plusieurs méthodes de tests

```
import org.junit.Test;
import static org.junit.Assert.*;
public class CompteurTest {
        @Test
        public void testIncrement(){
                Compteur c = new Compteur(10);
                c.incrementer();
                int attendu = 11;
                assertTrue(c.getValeur() == attendu);

        }
        @Test
        public void testDecrement(){
                Compteur c = new Compteur(10);
                c.decrementer();
                int attendu = 9;
                assertTrue(c.getValeur() == attendu);
        }
}
```

Partie commune à chaque test !

II. Mise en place de Tests avec JUnit

4) Annotations

Plusieurs méthodes de tests

```
import org.junit.Test;
import static org.junit.Assert.*;
public class CompteurTest {
        @Test
        public void testIncrement(){
                Compteur c = new Compteur(10);
                c.incrementer();
                int attendu = 11;
                assertTrue(c.getValeur() == attendu);

        }
        @Test
        public void testDecrement(){
                Compteur c = new Compteur(10);
                c.decrementer();
                int attendu = 9;
                assertTrue(c.getValeur() == attendu);
        }
}
```

```
//Attributs
private Compteur c;
```

```
//Initialiser le compteur avant chaque test
@Before
public void setUp(){
        super.setUp();
        c = new Compteur(10);
}
```

II. Mise en place de Tests avec JUnit

4) Annotations

D'autres annotations peuvent être utilisées

Annotation	Description
@Test	Méthode de test
@Before	Méthode exécutée avant chaque test *(méthode **setUp()** avec Junit 3.8)*
@After	Méthode exécutée après chaque test *(méthode **tearDown()** avec Junit 3.8)*
@BeforeClass	Méthode exécutée avant le premier test doit être ***static*** *(méthode **setUpBeforeClass()** avec Junit 3.8)*
@AfterClass	Méthode exécutée après le dernier test doit être ***static*** *(méthode **tearDownAfterClass()** avec Junit 3.8)*

II. Mise en place de Tests avec JUnit

4) Annotations

Ordre d'exécution

1. La méthode annotée **@BeforeClass**
2. Pour chaque méthode annotée **@Test** (ordre indéterminé)
   1. Les méthodes annotées **@Before**
   2. La méthode annotée **@Test**
   3. Les méthodes annotées **@After** (ordre indéterminé)
3. La méthode annotée **@AfterClass**

II. Mise en place de Tests avec JUnit

4) Annotations

Plusieurs méthodes de tests

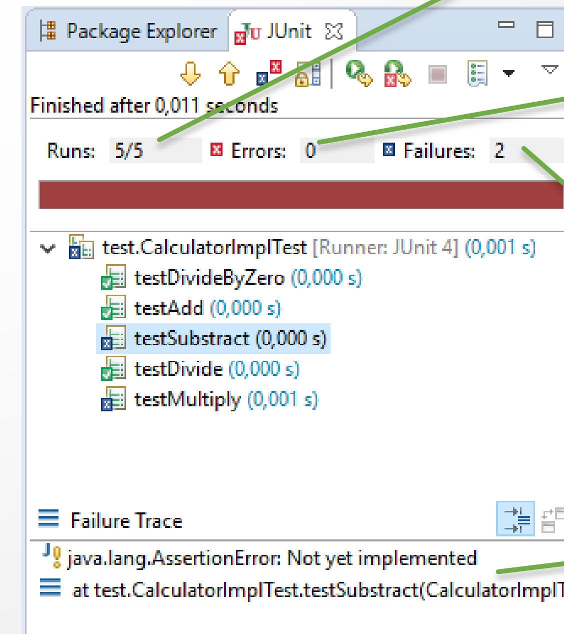
```
public class CompteurTest {
        private Compteur c;

        @Before
        public void setUp(){
                super.setUp();
                c = new Compteur(10);
        }

        @Test
        public void testIncrement(){
                        c.incrementer();
                        int attendu = 11;
                        assertTrue(c.getValeur() == attendu);
        }
        @Test
        public void testDecrement(){
                        c.decrementer();
                        int attendu = 9;
                        assertTrue(c.getValeur() == attendu);
        }
}
```

II. Mise en place de Tests avec JUnit

5) Exécution des tests

Comment lancer les tests?

- En ligne de commande
  > java org.junit.runner.JUnitCore <maClasseTest.java>
- Depuis un code Java
  junit.textui.TestRunner.run(<maClasseTest>);
  - L'affiche se fait sur la console
- **Depuis Eclipse**
  Run > Run As > JUnit Test



Nombre de méthodes exécutées

Nombre d'exception survenues

Nombre d'échecs

Détail des échecs rencontrés

II. Mise en place de Tests avec JUnit

6) Tests et Exceptions

Comment vérifier qu'une méthode lève bien une exception?

```
public class CompteurTest {
		private Compteur c;

		//La méthode décrémenter doit lever une CompteurZeroException si le compteur est à zéro

		@Test
		public void testDecrementAPartirdeZero(){
				c = new Compteur(0);
				c.decrementer();
		}
}
```

II. Mise en place de Tests avec JUnit

6) Tests et Exceptions

Comment vérifier qu'une méthode lève bien une exception?

```
public class CompteurTest {
        private Compteur c;

        //La méthode décrémenter doit lever une CompteurZeroException si le compteur est à zéro

        @Test
        public void testDecrementAPartirdeZero(){
                c = new Compteur(0);
                try{
                        c.decrementer();
                        fail("ECHEC décrémentation d'un compteur à zero !");
                }
                catch(CompteurZeroException  e){
                        //OK L'exception est correctement lancée
                }
        }
}
```

II. Mise en place de Tests avec JUnit

6) Tests et Exceptions

Une autre solution: utilisation des annotations

```
public class CompteurTest {
		private Compteur c;

		//La méthode décrémenter doit lever une CompteurZeroException si le compteur est à zéro

		@Test (expected=CompteurZeroException .class)
		public void testDecrementAPartirdeZero(){
				c = new Compteur(0);
				c.decrementer();
		}
}
```

II. Mise en place de Tests avec JUnit

6) Tests et Exceptions

Des annotions pour tester le temps d'exécution (en milllisecondes)

```
public class CompteurTest {
		private Compteur c;

		//La méthode doit échouer si le temps d'exécution dépasse un seuil

		@Test (timeout=10000)
		public void testIncrementMultiple(){
				c = new Compteur(0);
				c.incrementer(1000000);
		}
}
```

Chapitre III. Couverture des Tests

1. Plugin EclEmma
2. Exemple de couverture

III. Couverture des Tests

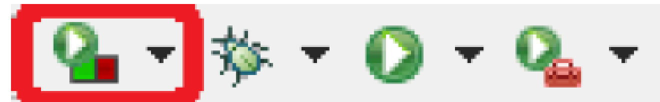
1) Plugin EclEmma

EclEmma

- Outil pour Eclipse qui permet d’évaluer la couverture du code
- «*EclEmma is a free Java code coverage tool for Eclipse, available under the Eclipse Public License. It brings code coverage analysis directly into the Eclipse* »

Installation

- Dans Eclipse: Install New Software
- http://update.eclemma.org/



III. Couverture des Tests

2) Exemple de couverture

Résultat couverture

- Pourcentage de code testé
- Vert: Portion de code vérifié par les tests
- Rouge: Portion de code non testé
- Jaune: Partiellement testé

```
135        a = 5;
136        b  = 0;
137        res = a + b;
138        if (calc.add(a, b)  != res) {
139            fail("b nul");
140        }
141
142        a = 0;
143        b  = 0;
144        res = a + b;
145        if (calc.add(a, b)  != res) {
146            fail("a et b nuls");
147        }
148
149        a = -5;
150        b  = 5;
151        res = a + b;
```

Problems | Javadoc | Declaration | Console | Coverage

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
TestJunit	82,4 %	337	72	409

Chapitre IV.
Pour aller plus loin

1. Limite des tests
2. Tests en aveugle
3. Règles de bonnes conduites

IV. Pour aller plus loin

1) Limite des tests

Problème des tests

- Le test peut également être victime d'une erreur de conception
  - Un état de départ
    - Erreur dans l'initiation (n'hésite une phase d'initialisation complexe: lecture fichier, BD, etc.)
    - Cas limites mal identifiés (nombreux et pas évident)
  - Un état attendu
    - Pas toujours évident d'identifier le résultat théorique attendu
    - La recherche des couples **états de départ / résultats attendus** peut s'avérer difficile
  - Un oracle
    - Simple quand il s'agit de comparer des types primitifs
    - Peut s'avérer complexe si la comparaison porte sur des objets

IV. Limite des Tests

3) Tests en aveugle

Deux techniques de tests:

1. **Boite noire**
   - Le testeur ne connaisse le contenu de la méthode qu'il va tester
   - On teste vraiment ce que devrait faire la méthode
2. **Boite blanche**
   - Le testeur connait le contenu de la méthode testée
   - Le risque est alors de tester le fonctionnement et d'oublier le but final de la méthode.
   - En contre-partie, les tests sont plus précis.

Attention quand la même personne développe à la fois la classe et le test !

IV. Limite des Tests

3) Règles de bonnes conduites

Quelques règles de bonnes conduites

- Ecrire les tests en même temps que le code
- Une classe -> une classe de test
- Le test est implémenté par une autre programmeur
- Effectuer des tests qui couvrent toutes les situations
- Tester les valeurs limites et les cas particuliers
- Lancer les tests après chaque modification du code
- Ne pas tester plusieurs méthodes dans le même test