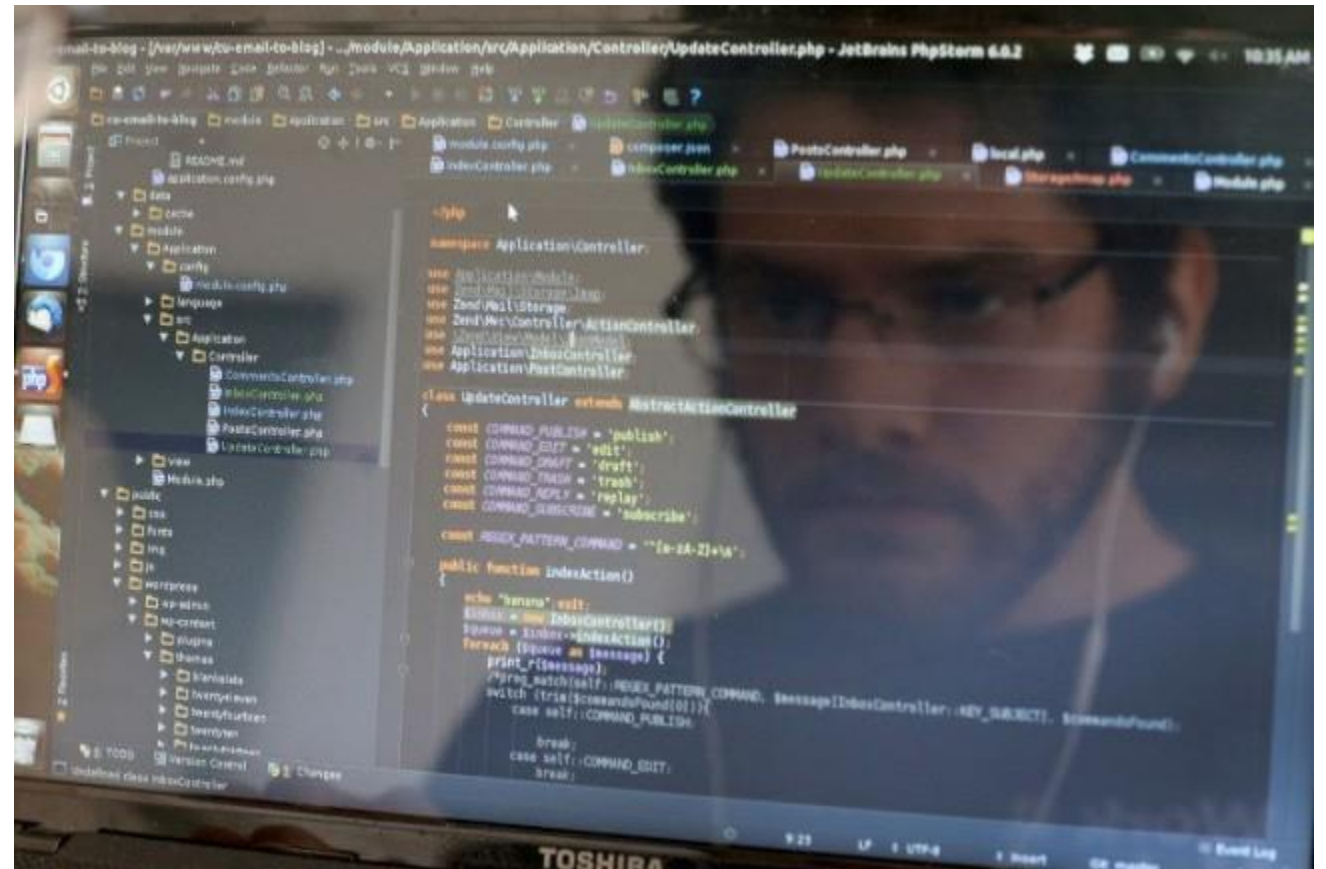
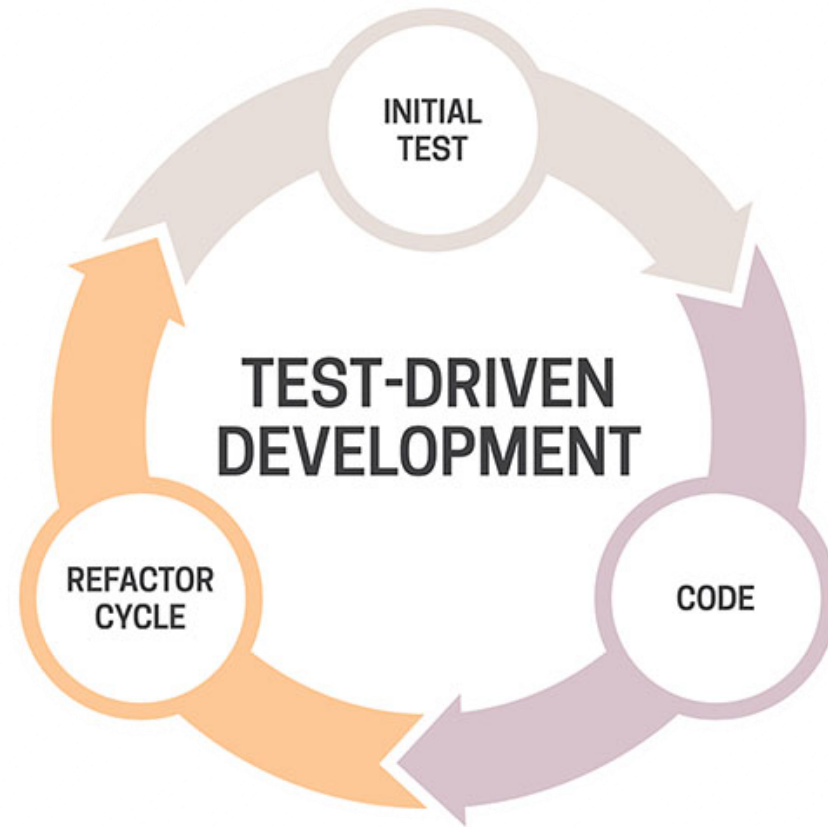


années 90 que sont  
apparus des outils  
permettant aux  
développeurs de  
créer leurs propres  
tests



**années 2000 de pilotage  
par les tests (« Test Driven »).**



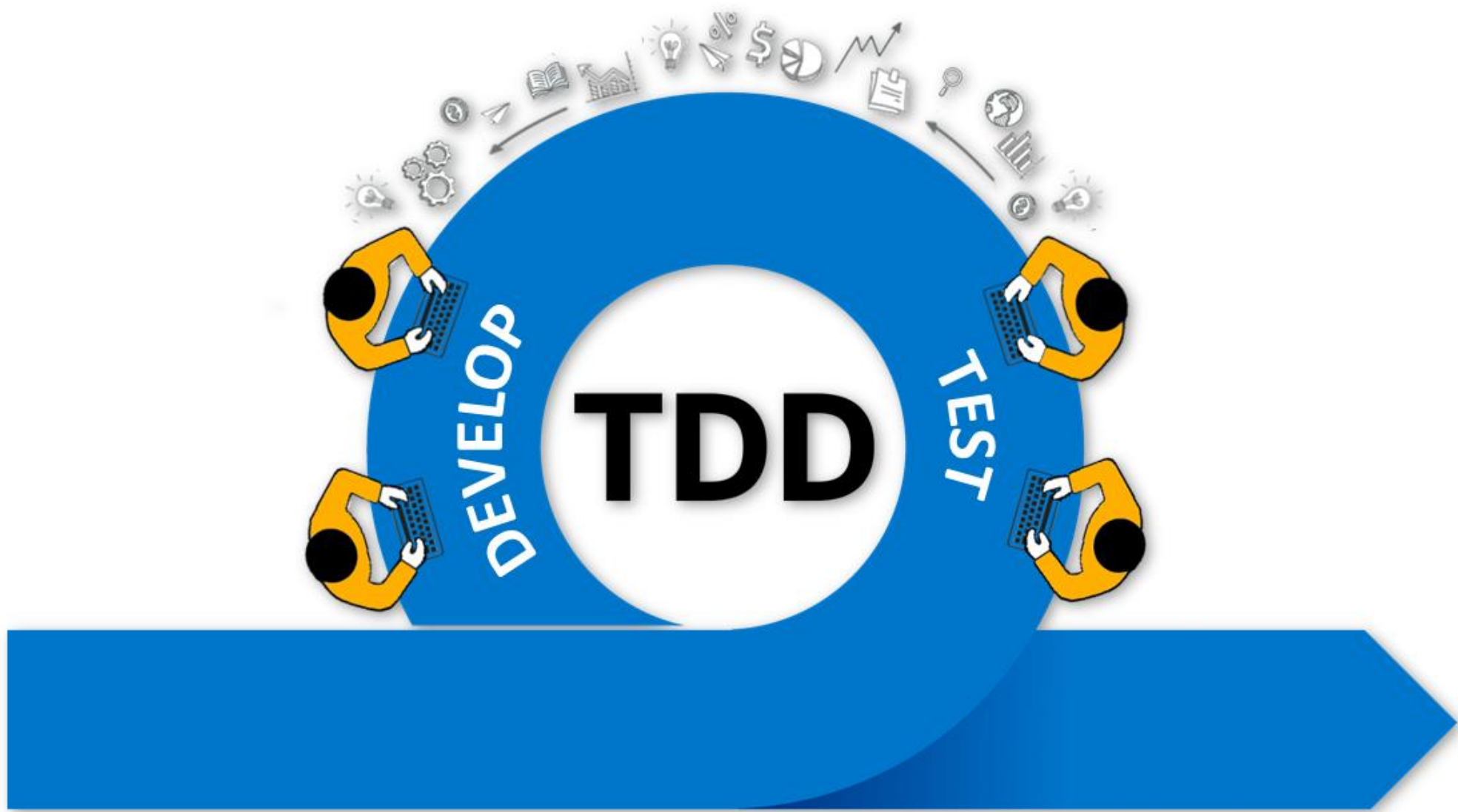
le **TDD** (pour « Test Driven Development »), ou Développement Piloté par les Tests, est réellement codifié en 2003 par Kent Beck



# Test Driven Development (TDD) - codifié en 2003

- ▶ 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 (JUnit, JUnit, RUnit, etc.)





# TDD

- Le Test

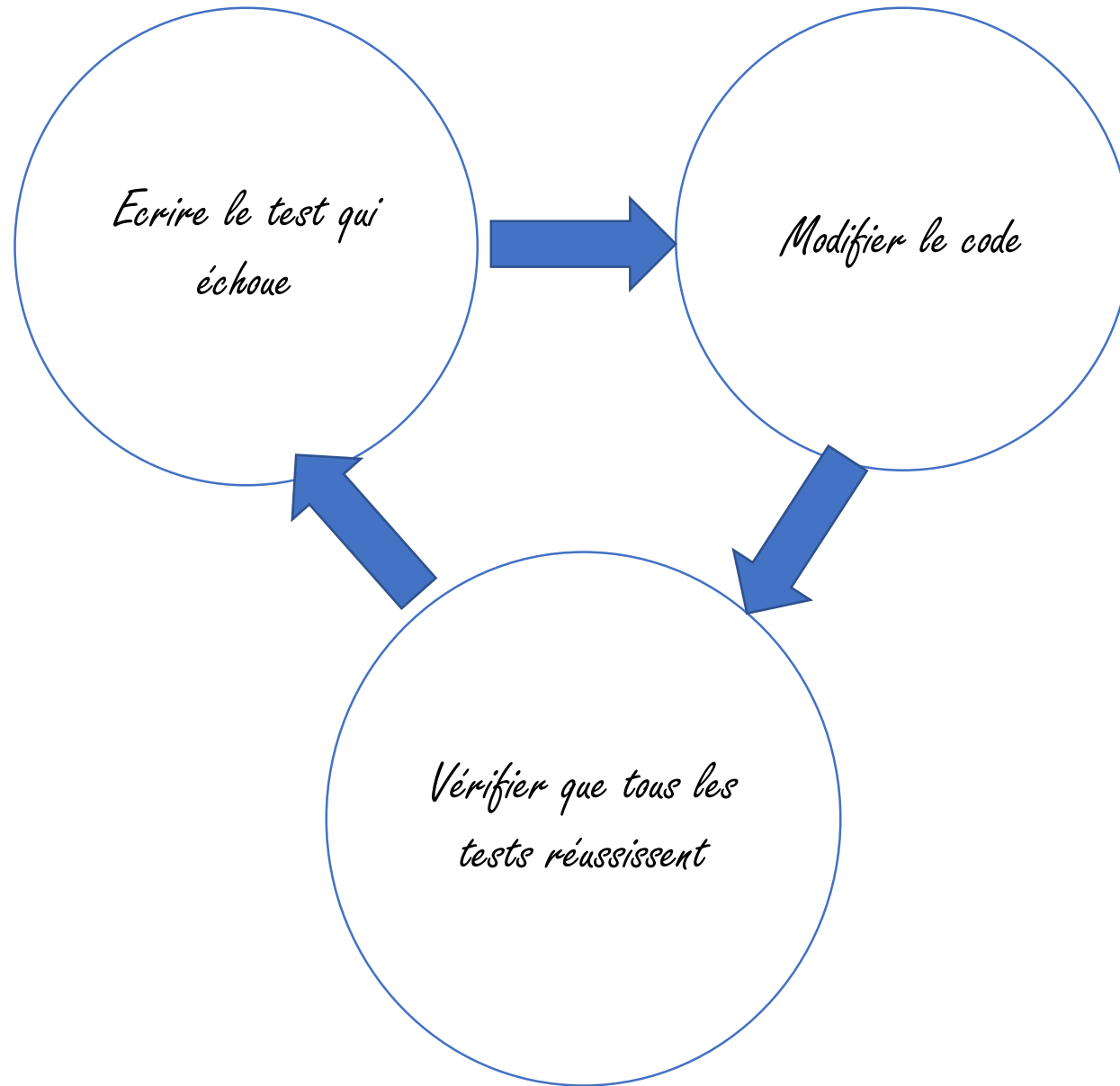
- Il vérifie une nouvelle fonctionnalité ou qui a été changé.
- Il est indépendant des autres tests.
- Il vérifie un seul aspect ou comportement et documente le comportement attendu
- Il ne doit pas vérifier trop de fonctionnalité

# TDD

- Les 3 lois du TDD

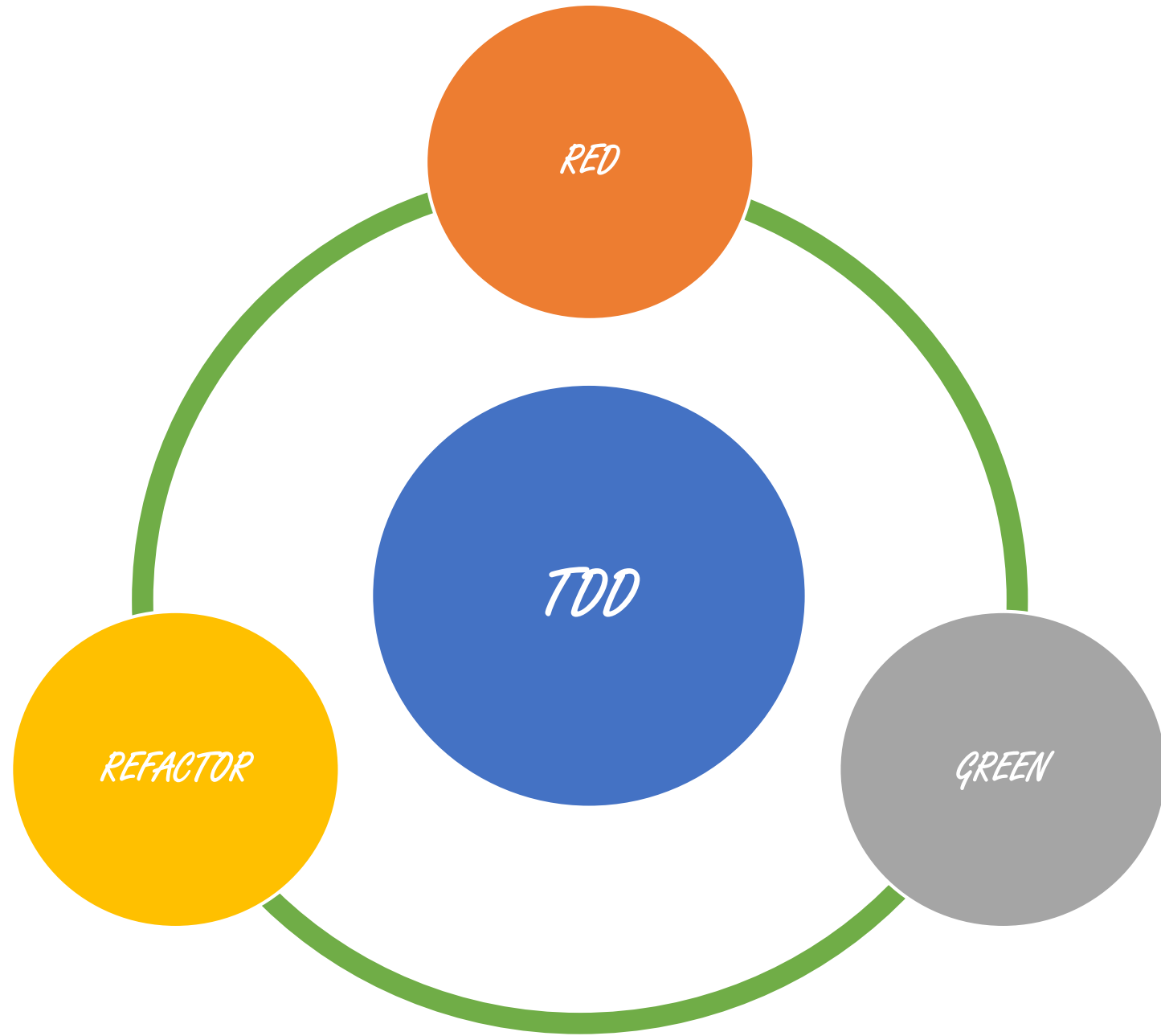
- 1. Vous ne devez pas commencer à écrire de code tant que vous n'avez pas écrit un test unitaire qui échoue
- 2. Vous devez écrire le test suffisant pour échouer.
- 3. Vous ne devez pas écrire plus de code que nécessaire pour la réussite du test qui est en cours

# TDD



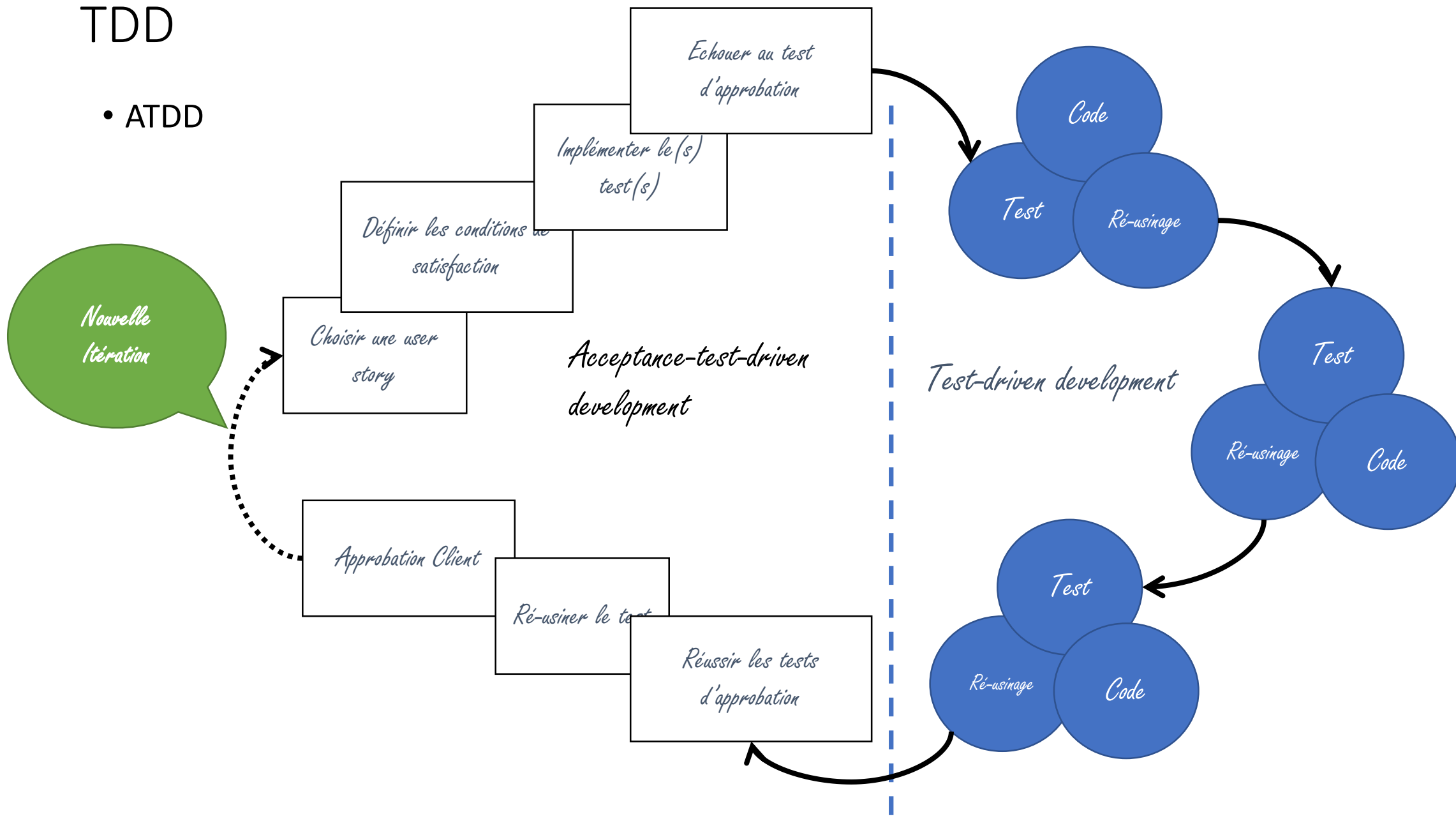


# TDD



# TDD

- ATDD



Java SE

## Les tests avec JUnit V4

# Objectifs

- Présentation des annotations
- Présentation du Framework JUnit V4
- Mise en place dans un projet sous Eclipse
- Valider la couverture des tests avec Eclemma
- Automatisation des tests à l'aide d'un script Ant



# Les tests avec JUnit V4

Présentation des annotations

# Contenu du module

- Définition des annotations
- Utilisation d'une annotation
- Création d'une annotation

# Définition d'une annotation

- Une annotation permet de "marquer" certains éléments du langage Java afin de leur ajouter une propriété particulière.
- Ces annotations peuvent ensuite être utilisées à la compilation ou à l'exécution (grâce à l'API de réflexion `java.lang.reflect`) pour automatiser certaines tâches...
- Une annotation peut être utilisée sur plusieurs type d'éléments
  - package, class, interface, enum, annotation,
  - constructeur, méthode, paramètre, champs d'une classe ou variable locale
- Plusieurs annotations différentes peuvent être utilisées sur un même élément mais on ne peut pas utiliser deux fois la même annotation.

# Utilisation d'une annotation 1/2

- L'API standard de Java 5.0 propose seulement trois annotations. Elle permettent d'interagir avec le compilateur Java.
  - **@Deprecated**
    - L'annotation **@Deprecated** vient remplacer le tag javadoc **@deprecated** afin de signaler au compilateur que l'élément marqué est déprécié et ne devrait plus être utilisé.  
Le compilateur affichera un warning si l'élément est utilisé dans du code non-déprécié (ou une 'note', selon la configuration du compilateur).
  - **@Override**
    - L'annotation **@Override** ne peut être utilisée que sur une méthode afin de préciser au compilateur que cette méthode est redéfinie et doit donc 'cacher' une méthode héritée d'une classe parent. Si ce n'est pas le cas (par exemple si une faute de frappe s'est glissée dans le nom de la méthode), le compilateur doit afficher un erreur (et donc faire échouer la compilation).
  - **@SuppressWarnings**
    - L'annotation **@SuppressWarnings** indique au compilateur de ne pas afficher certains warnings. Le principal intérêt de cette annotation est de cacher des warnings sur des parties de code plus anciennes sans pour autant les cacher sur toute l'application. Elle reste toutefois à utiliser avec parcimonie.



## Utilisation d'une annotation 2/2

- En général l'annotation est placée devant la déclaration de l'élément qu'elle marque

```
@Deprecated
public String getValeur()
{
    return valeur;
}
@SuppressWarnings("deprecation")
public void uneMethode()
{
    String v = getValeur();
    //...
    System.out.println("v="+v);
}
@Override
public String toString()
{
    return super.toString();
}
```

# Création d'une annotation 1/2

- Annotation « TODO »                      Menu new -> Annotation

```
import java.lang.annotation.Documented;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;

@Documented
@Retention(RetentionPolicy.SOURCE)
public @interface TODO {
    /** Message décrivant la tâche à effectuer. */
    String value();
}
```

- Un attribut value permet de définir la tâche à réaliser
- Des méta-annotations spécifient le comportement des annotations
  - `@Documented` : Le contenu de l'annotation TODO sera écrite dans la javadoc
  - `@Retention(SOURCE)`: L'annotation TODO ne sera utilisée que dans les sources. Elles ne sera pas recopier dans le fichier .class

# Création d'une annotation 2/2

- Les méta-annotations
  - **@Documented**: c'est pour savoir si l'annotation doit être documentée par Javadoc
  - **@Inherit**: c'est pour savoir si l'annotation est héritée par les classes filles. Cela sert pour les annotations déposées au niveau d'une classe
  - **@Retention(CLASS, RUNTIME ou SOURCE)**: C'est pour savoir à quel endroit doit être maintenu l'annotation
  - **@Target**: c'est pour savoir si on doit restreindre l'utilisation de l'annotation à certains éléments (class, méthodes, attributs...)
- Utilisation de notre annotation

```
@TODO(value="Prévoir le changement de l'appel à getValeur")
@SuppressWarnings("deprecation")
public void uneMethode()
{
    String v = getValeur();
    //...
    System.out.println("v="+v);
}
```

# Et JUnit dans tout ça...

- JUnit utilise des annotations personnalisées pour diriger l'exécution des tests.
- Voici une liste non exhaustive des annotations utilisées:
  - **@Test**: permet de définir les méthodes de test
  - **@RunWith** et **@SuiteClasses**: permettent de définir une campagne de test
  - **@BeforeClass**, **@AfterClass**, **@Before**, **@After**, **@Ignore**: permettent de contrôler le contexte d'exécution des tests.
- Toutes ces annotations ont une méta-annotation **@Retention(value=RUNTIME)**

org.junit

## Annotation Type Test

---

```
@Retention(value=RUNTIME)
@Target(value=METHOD)
public @interface Test
```

- L'API de réflexion va permettre au moteur d'exécution de JUnit de définir le rôle de chaque méthode et de les exécuter au bon moment
- <http://junit.org/junit/javadoc/4.5/index.html>

# Les tests avec JUnit V4

Présentation du Framework Junit V4

# Présentation

- JUnit est un framework de tests unitaires pour Java
- Un test unitaire se déroule en 4 étapes:
  - Setup: initialiser des objets ou des ressources
  - Call: exécuter le code à vérifier
  - Verify: vérifier des données issues du traitement
  - TearDown: permettre de faire le ménage ou de libérer des ressources
- Quelques principes à suivre:
  - Un test doit être le plus petit et le plus simple possible
  - Chaque test doit être isolé et ne pas dépendre d'un autre test
  - Les tests doivent être exécutés régulièrement (donc doivent être automatisés)

# Les quatre étapes

- Les 4 étapes correspondent à l'exécution des méthodes:
  - Setup: exécution des méthodes préfixées par l'annotation **@BeforeClass** et/ou **@Before**
    - **@BeforeClass**: ces méthodes ne sont exécutées qu'une seule fois pour l'ensemble des tests d'une classe
    - **@Before**: ces méthodes sont exécutées avant chaque méthode de test
  - Call: exécution des méthodes préfixées par l'annotation **@Test**
  - Verify: cette étape est réalisée dans l'étape Call et correspond à l'exécution de méthodes de vérification disponibles dans la classe **junit.framework.Assert**
  - TearDown: exécution des méthodes préfixées par l'annotation **@AfterClass** et/ou **@After**
    - **@AfterClass**: ces méthodes ne sont exécutées qu'une seule fois pour l'ensemble des tests d'une classe
    - **@After**: ces méthodes sont exécutées après chaque méthode de test

# Comportement à l'exécution

```
package fr.eni_ecole.jse;
import static org.junit.Assert.*;

public class DesTests {
    @BeforeClass
    public static void setupBeforeClass()
    {
        System.out.println("Exécution BeforeClass");
    }
    @Before
    public void setupBefore()
    {
        System.out.println("Exécution Before");
    }
    @Test
    public void test1() {
        System.out.println("Exécution Test 1");
    }
    @Test
    public void test2() {
        System.out.println("Exécution Test 2");
    }
    @AfterClass
    public static void tearDownAfterClass()
    {
        System.out.println("Exécution AfterClass");
    }
    @After
    public void tearDownAfter()
    {
        System.out.println("Exécution After");
    }
}
```



Exécution BeforeClass  
Exécution Before  
Exécution Test 1  
Exécution After  
Exécution Before  
Exécution Test 2  
Exécution After  
Exécution AfterClass



# Campagne de tests

- Les tests sont souvent situés dans différentes classes
- Le regroupement de ces classes pour l'exécution des tests écrits à l'intérieur s'appelle une campagne de test
- JUnit offre la possibilité de regrouper un certain nombre de classe de test dans une campagne de test
- Il faut utiliser pour cela les annotations **@RunWith** et **@SuiteClasses**

```
package fr.eni_ecole.jse;

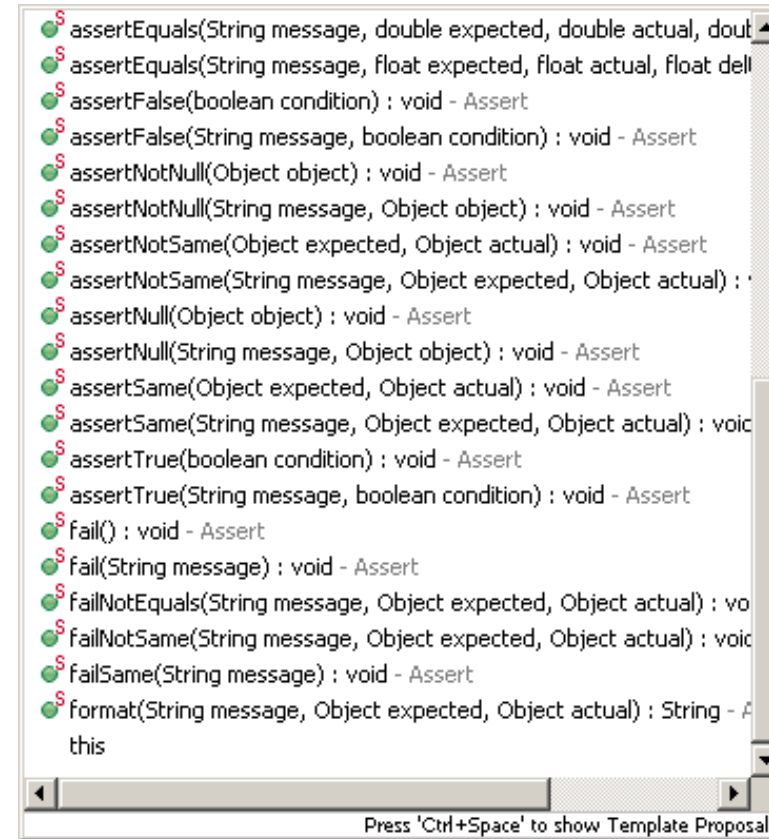
import org.junit.runner.RunWith;

@RunWith(Suite.class)
@SuiteClasses({ DAutresTests.class,
                DesTests.class })
public class AllTests {

}
```

# Les méthodes de vérification

- La classe `junit.framework.Assert` propose un ensemble de méthodes permettant de réaliser des vérifications



- Pour tester les exceptions attendues, il faut utiliser l'attribut **expected** de l'annotation **@Test**

```
@Test(expected=java.lang.IndexOutOfBoundsException.class)
```

# Les tests avec JUnit V4

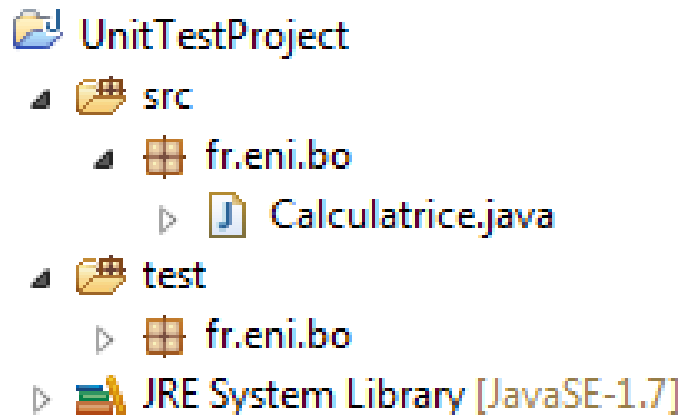
Mise en place des tests unitaires sous Eclipse

# Présentation

- Eclipse Luna possède nativement un plugin pour JUnit

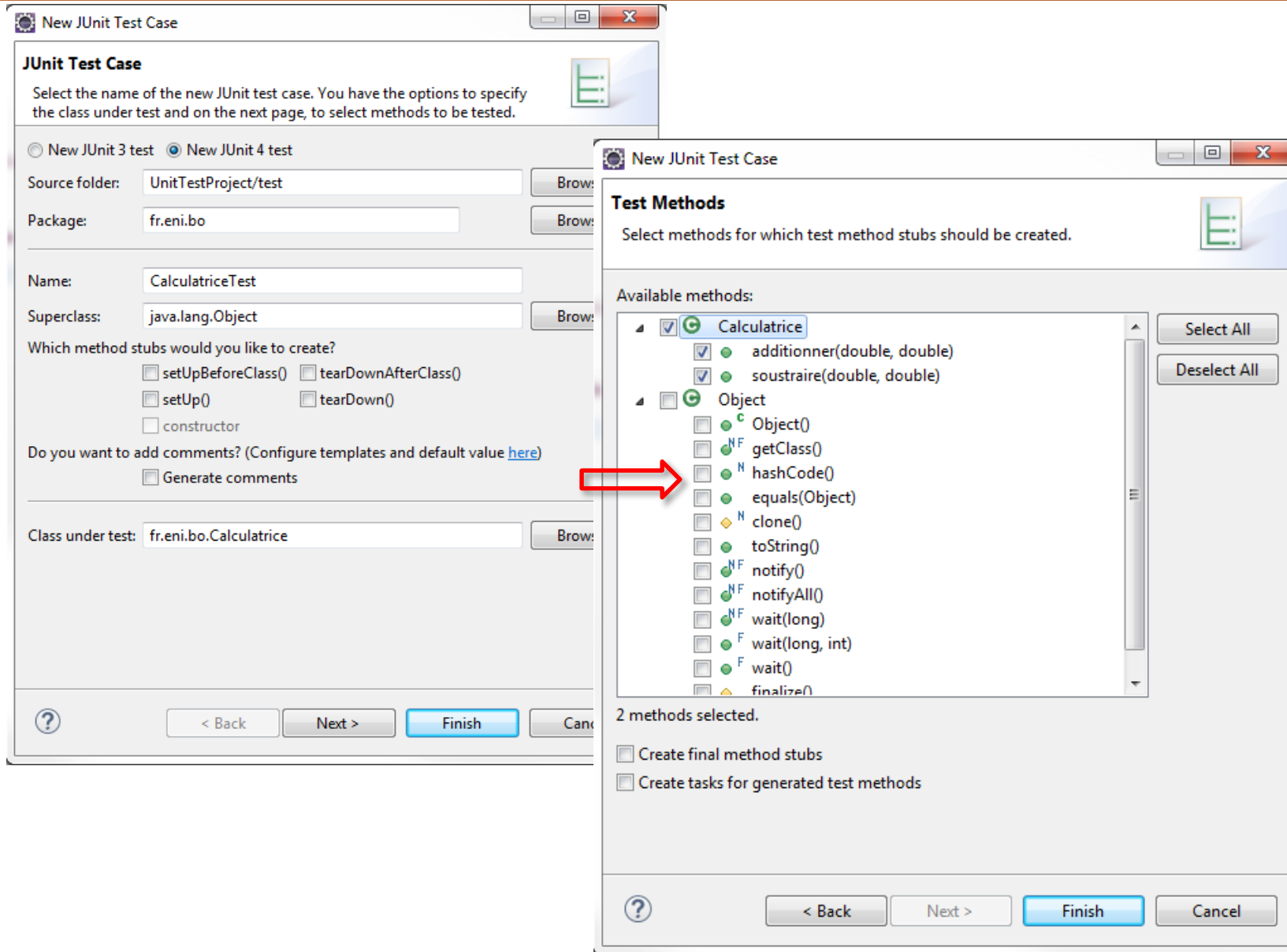
# Création de la classe de test 1/3

```
public class Calculatrice {  
  
    public static double additionner(double a,double b){  
        return (a + b);  
    }  
  
    public static double soustraire(double a,double b){  
        return (a - b);  
    }  
  
}
```



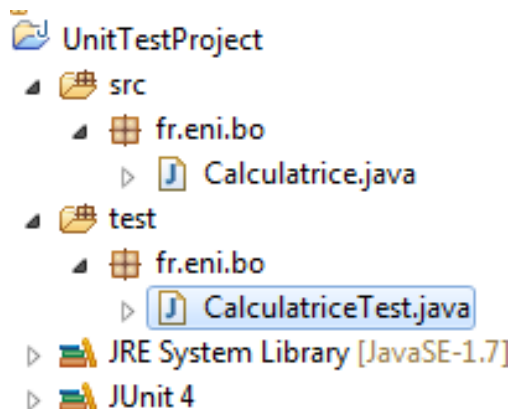
**Clique droit sur la classe Calculatrice**  
**Menu New → JUnit Test Case**

# Création de la classe de test 2/3



# Création de la classe de test 3/3

- Une classe minimaliste de test est générée.



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

import org.junit.Test;

public class CalculatriceTest {

    @Test
    public void testAdditionner() {
        fail("Not yet implemented");
    }

    @Test
    public void testSoustraire() {
        fail("Not yet implemented");
    }
}
```



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

public class CalculatriceTest {

    @Test
    public void testAdditionner() {
        //Arrange (préparer)
        double expected = 30;
        double actual = 0;

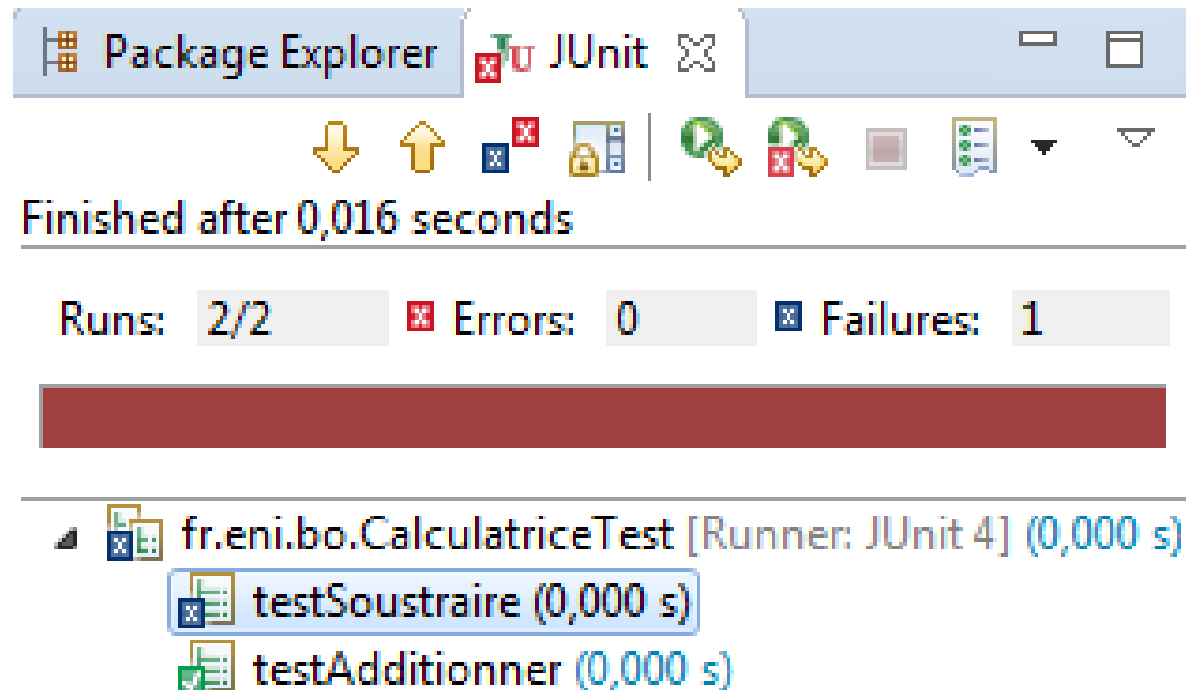
        //Act (effectuer)
        actual = Calculatrice.additionner(10, 20);

        //Accept (valider)
        Assert.assertEquals(expected, actual, 0.00001);
    }

    @Test
    public void testSoustraire() {
        fail("Not yet implemented");
    }
}
```

# Exécution d'une classe de test

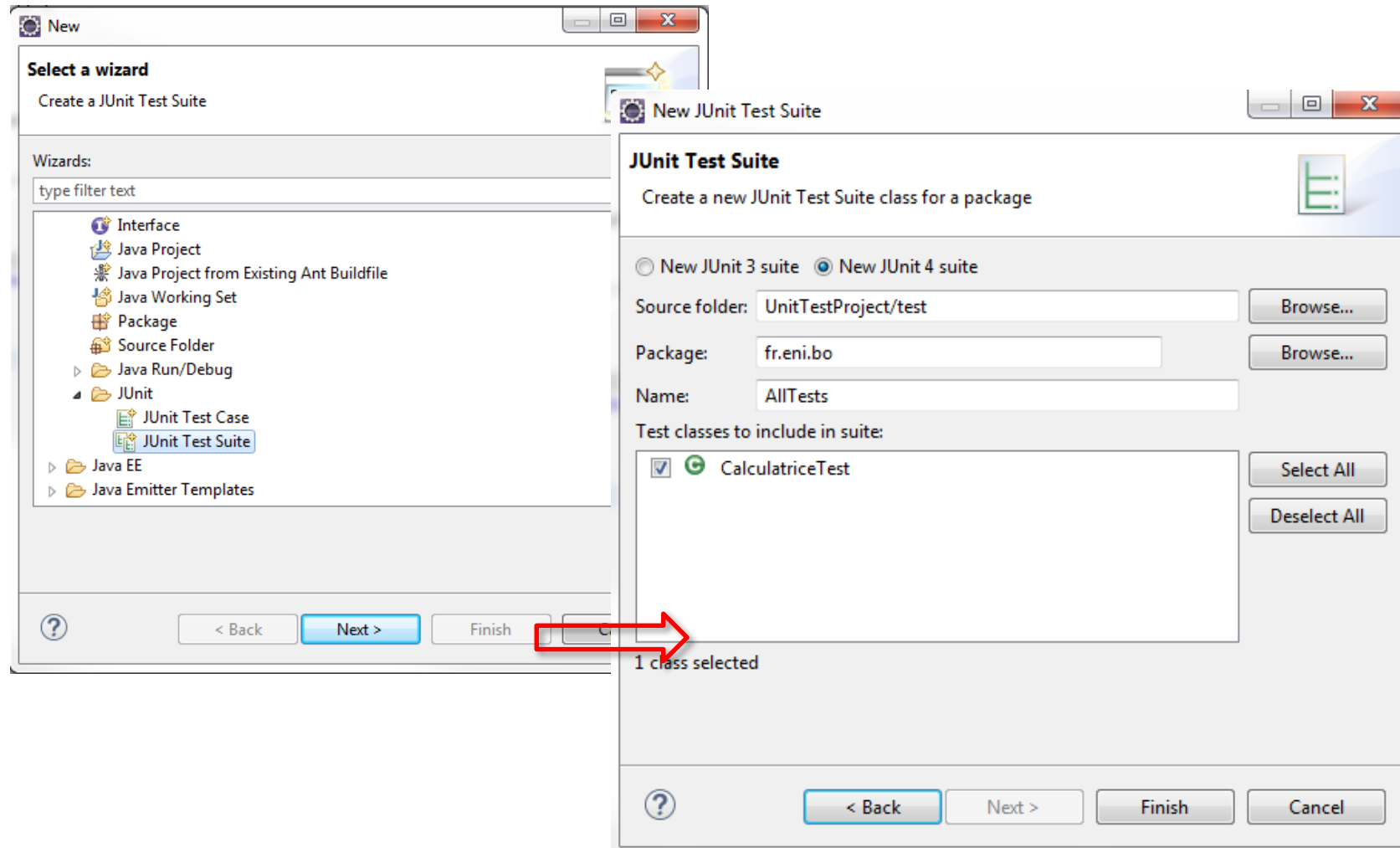
- A partir du projet : Menu Run As -> JUnit Test
- Un écran de résultat apparaît:





# Création d'une campagne de tests 1/2

- Nécessaire lorsqu'on souhaite exécuter les tests contenus dans plusieurs classes de test



# Création d'une campagne de tests 2/2

- Une classe « AllTests » est créée
  - Elle recense les classes de test à exécuter

```
package fr.eni.bo;  
  
import org.junit.runner.RunWith;  
  
@RunWith(Suite.class)  
@SuiteClasses({ CalculatriceTest.class })  
public class AllTests {  
  
}
```

- Cette classe peut être le point d'entrée pour exécuter des tests

# Exécution d'une campagne de tests

- L'exécution se fait de la même manière que pour une classe de test unique
- L'écran des résultats montre le résultat de l'exécution de chacune des classes contenues dans la campagne de tests

# Les tests avec JUnit V4

Valider la couverture des tests à l'aide  
d'Eclemma

# Objectifs

- S'assurer que les tests unitaires valident le code écrit.
- Mesurer la couverture du code par les tests
- Initialiser le processus d'industrialisation des développements et l'intégration continue
- Pour cela, il existe plusieurs utilitaires pour lancer les tests unitaires et visualiser la couverture du code.
- Nous utiliserons pour la suite le plug-in Eclemma

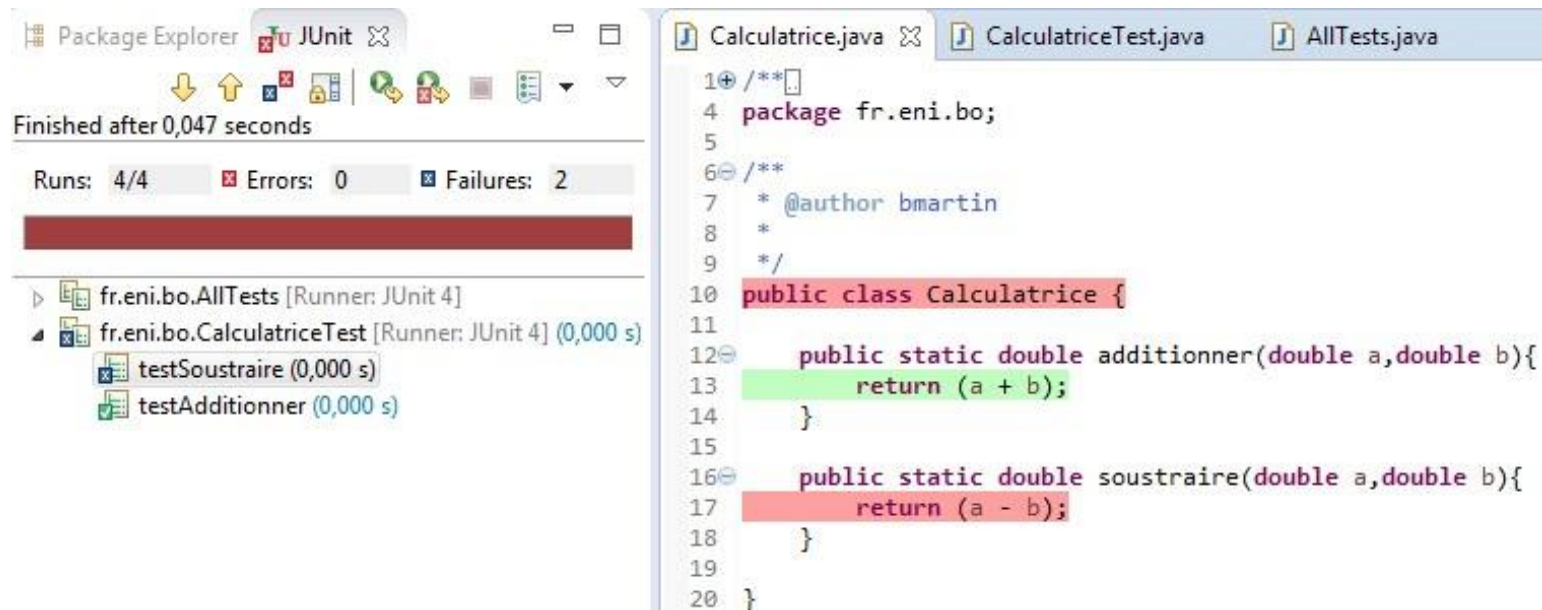
# Eclemma

- Télécharger le plug-in <http://www.eclemma.org/download.html>
- Installation du plug-in <http://www.eclemma.org/installation.html>
  - Menu **Help** -> **Install New Software** cliquer sur **Add**
  - Renseigner les champs :
    - **Name** : Eclemma plug-in for Eclipse
    - **Location** : chemin d'accès à l'archive représentant le plug-in (bouton **Archive**)
  - Sélectionner **Eclemma**
  - Décocher **Contact all update sites...**
  - Redémarrer Eclipse



# Exécuter les tests avec Eclemma

- A partir du projet : Menu Coverage As -> JUnit Test
- Un écran de résultat apparaît:

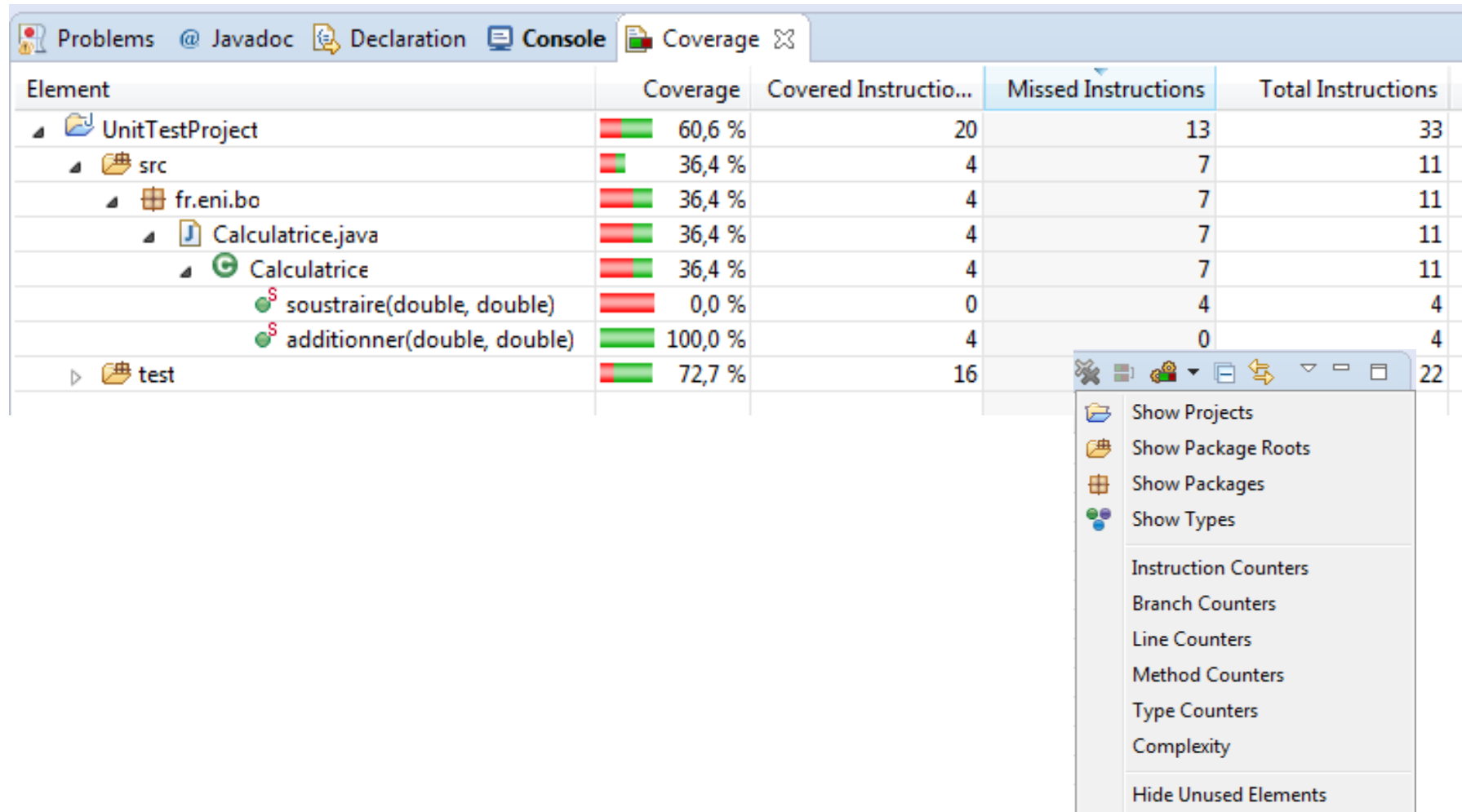










The screenshot shows an IDE with two main panels. The left panel displays the 'JUnit' test runner results. It indicates 'Finished after 0,047 seconds' and shows a summary: 'Runs: 4/4', 'Errors: 0', and 'Failures: 2'. Below this, a tree view shows the test hierarchy: 'fr.eni.bo.AllTests [Runner: JUnit 4]' and 'fr.eni.bo.CalculatriceTest [Runner: JUnit 4] (0,000 s)'. Under 'CalculatriceTest', two tests are listed: 'testSoustraire (0,000 s)' with a failure icon (red X) and 'testAdditionner (0,000 s)' with a success icon (green checkmark). The right panel shows the source code of 'Calculatrice.java'. The code is as follows:

```
1+ /**  
4 package fr.eni.bo;  
5  
6+ /**  
7  * @author bmartin  
8  *  
9  */  
10 public class Calculatrice {  
11  
12+     public static double additionner(double a,double b){  
13         return (a + b);  
14     }  
15  
16+     public static double soustraire(double a,double b){  
17         return (a - b);  
18     }  
19  
20 }
```

# Une vue d'ensemble de la couverture

- La vue Coverage



| Element                     | Coverage  | Covered Instructio... | Missed Instructions | Total Instructions |
|-----------------------------|---|-----------------------|---------------------|--------------------|
| UnitTestProject             |  60,6 %  | 20                    | 13                  | 33                 |
| src                         |  36,4 %  | 4                     | 7                   | 11                 |
| fr.eni.bo                   |  36,4 %  | 4                     | 7                   | 11                 |
| Calculatrice.java           |  36,4 %  | 4                     | 7                   | 11                 |
| Calculatrice                |  36,4 %  | 4                     | 7                   | 11                 |
| soustraire(double, double)  |  0,0 %   | 0                     | 4                   | 4                  |
| additionner(double, double) |  100,0 % | 4                     | 0                   | 4                  |
| test                        |  72,7 %  | 16                    |                     | 22                 |

- Show Projects
- Show Package Roots
- Show Packages
- Show Types
- Instruction Counters
- Branch Counters
- Line Counters
- Method Counters
- Type Counters
- Complexity
- Hide Unused Elements



# Analyse du code

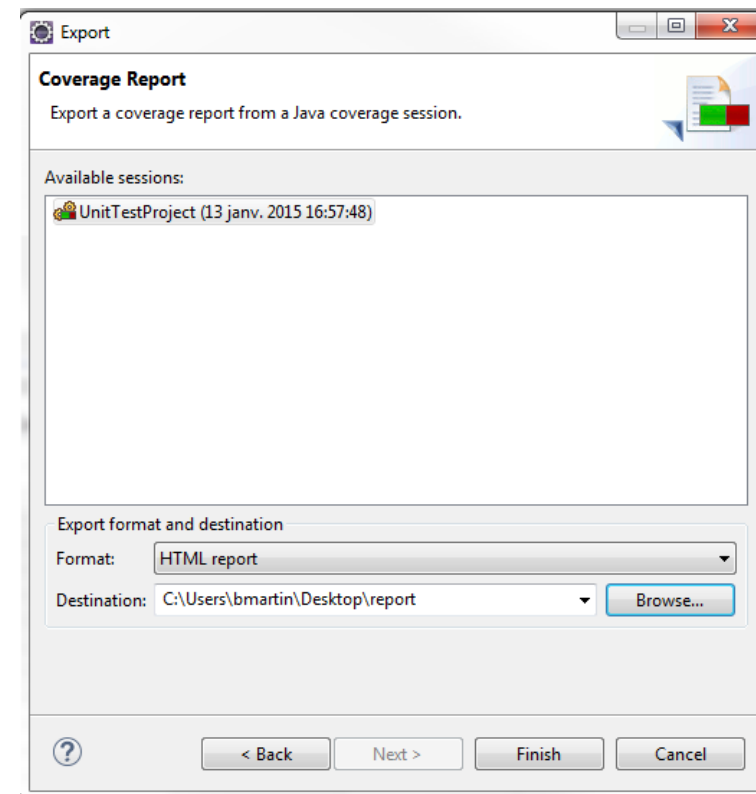
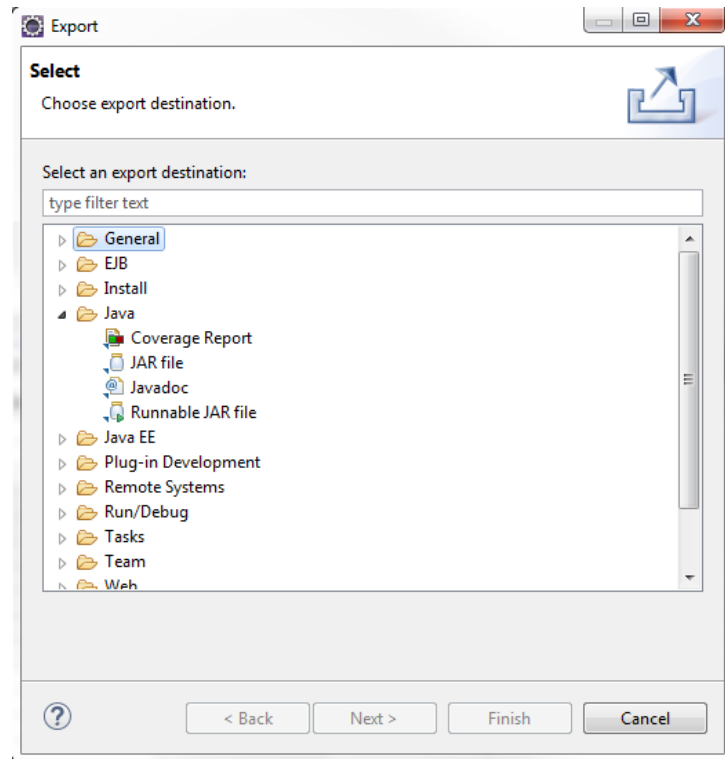
- Le résultat d'une session d'assurance est également directement visible dans le code source java à travers un marquage selon un code couleur. Les couleurs utilisées par défaut sont :
  - La couleur verte pour les lignes totalement couverte,
  - La couleur jaune pour afficher le code que les tests unitaires couvrent partiellement.
  - la couleur rouge est utilisée pour afficher le code que les tests unitaires ne couvrent pas

**Menu windows->preferences->General->Editors->Text Editors ->Annotations**

**Options : no coverage, partial coverage, full coverage**

# Exportation de rapport

- Formats XML, HTML, Texte
  - Cliquez droit vue **Coverage** **Export Session**





# Exportation de rapport

UnitTestProject (13 janv. 2015 16:57:48)

 [Sessions](#)

## UnitTestProject (13 janv. 2015 16:57:48)

| Element   | Missed Instructions   | Cov. | Missed Branches | Cov. | Missed | Cxty | Missed | Lines | Missed | Methods | Missed | Classes |
|---|---|------|-----------------|------|--------|------|--------|-------|--------|---------|--------|---------|
|  <a href="#">UnitTestProject</a> |  | 61 % |                 | n/a  | 4      | 7    | 5      | 12    | 4      | 7       | 1      | 3       |
| Total   | 13 of 33  | 61 % | 0 of 0          | n/a  | 4      | 7    | 5      | 12    | 4      | 7       | 1      | 3       |

UnitTestProject (13 janv. 2015 16:57:48)

Created with [JaCoCo](#) 0.7.2.201409121644

# Les tests avec JUnit V4

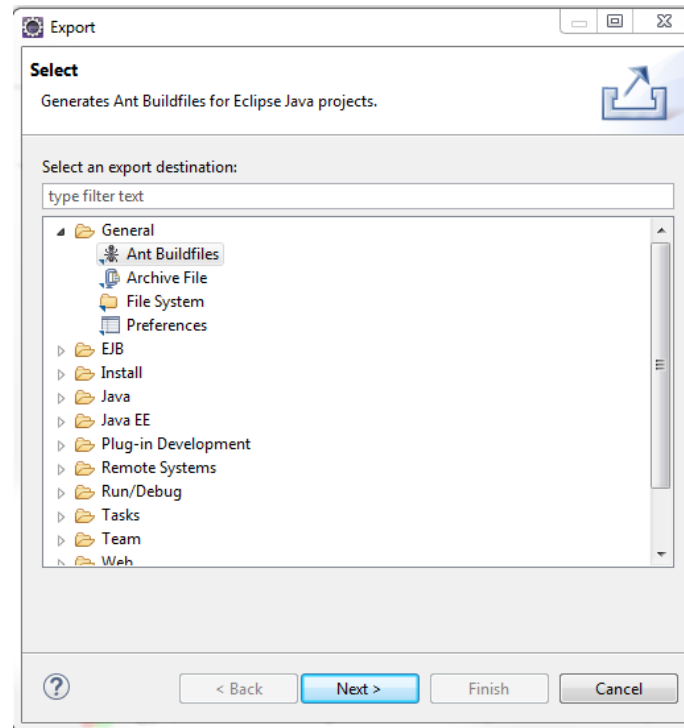
Automatisation des tests à l'aide d'un script  
Ant

# Présentation

- **Ant est un** projet open source de la fondation Apache écrit en Java qui vise le développement d'un logiciel d'automatisation des opérations répétitives tout au long du cycle de développement logiciel.
- Ant est principalement utilisé pour automatiser la construction de projets en langage Java, mais il peut être utilisé pour tout autre type d'automatisation dans n'importe quel langage.
- Parmi les tâches les plus courantes, citons : la compilation, la génération de pages HTML de document (Javadoc), la génération de rapports, l'exécution d'outils annexes, l'archivage sous forme distribuable (JAR etc.)
- Il permet aussi l'exécution de tests JUnit et la sortie des résultats au format HTML
- <http://ant.apache.org/manual/index.html>

# Création du script Ant (fichier build.xml)

- Sur le menu contextuel du projet, cliquez sur Exporter
- Sélectionnez Ant BuildFiles



- Un fichier build.xml est créé

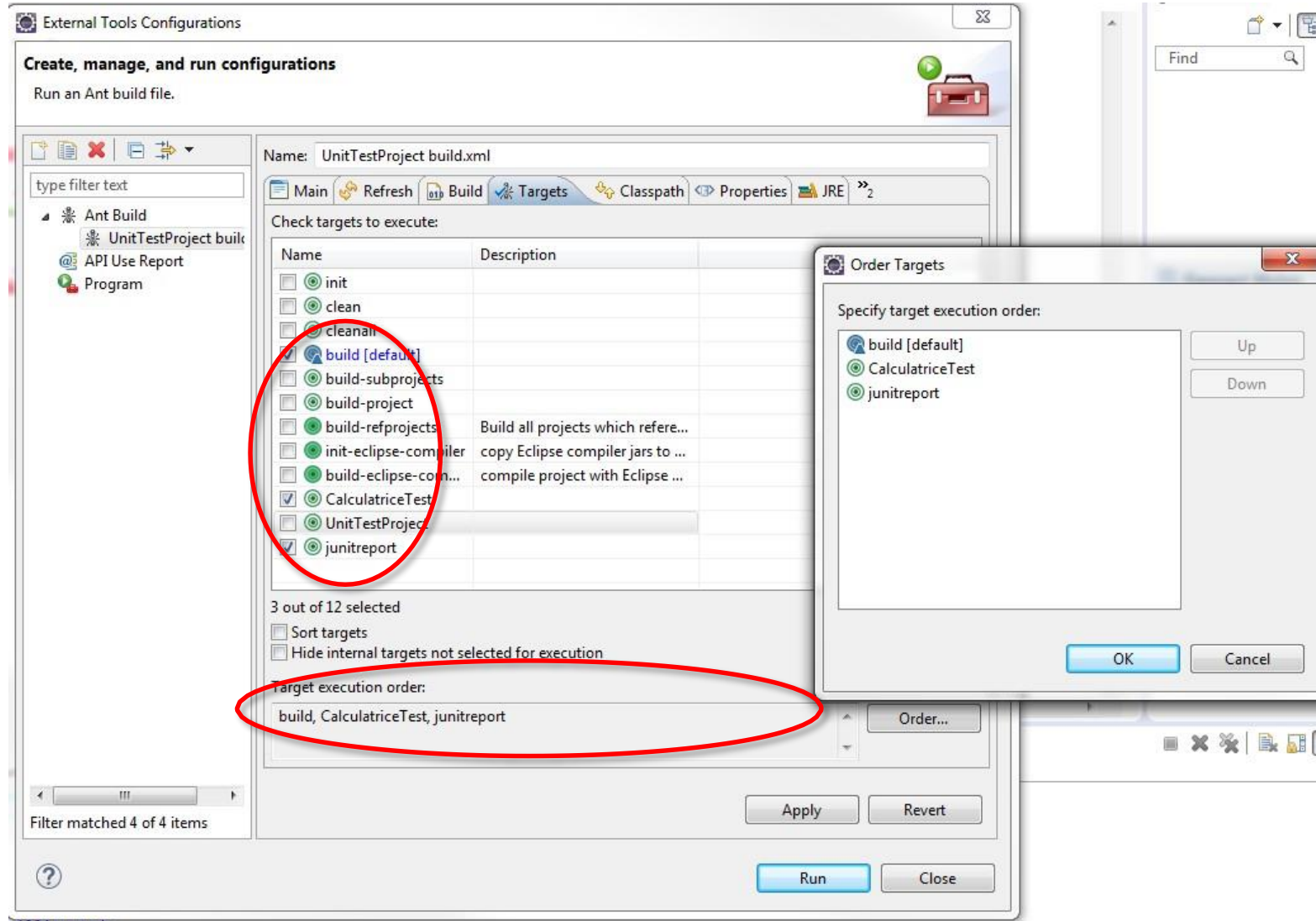
# Le fichier build.xml

- Ce fichier contient un certain nombre d'informations notamment sur les classes de test et sur le rapport de test

```
<target name="CalculatriceTest">
  <mkdir dir="${junit.output.dir}"/>
  <junit fork="yes" printsummary="withOutAndErr">
    <formatter type="xml"/>
    <test name="fr.eni_ecole.jse.CalculatriceTest" todir="${junit.output.dir}"/>
    <classpath refid="Debut.classpath"/>
  </junit>
</target>
<target name="AllTests">
  <mkdir dir="${junit.output.dir}"/>
  <junit fork="yes" printsummary="withOutAndErr">
    <formatter type="xml"/>
    <test name="fr.eni_ecole.jse.AllTests" todir="${junit.output.dir}"/>
    <classpath refid="Debut.classpath"/>
  </junit>
</target>
<target name="junitreport">
  <junitreport todir="${junit.output.dir}">
    <fileset dir="${junit.output.dir}">
      <include name="TEST-*.xml"/>
    </fileset>
    <report format="frames" todir="${junit.output.dir}"/>
  </junitreport>
</target>
```

# Personnalisation de l'exécution

- Cliquez droit sur Build.xml      Run As -> External Tools Configurations



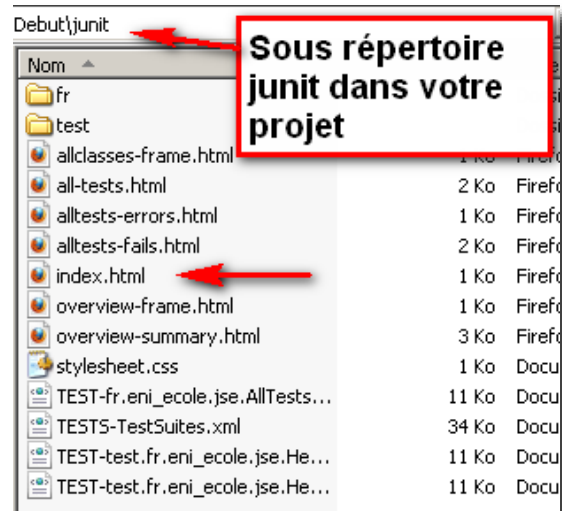


- Cliquez droit sur Build.xml Run As -> 1 Ant Build



```
<terminated> UnitTestProject build.xml [Ant Build] C:\Program Files\Java\jre7\bin\javaw.exe (13 janv. 2015 17:50:41)
Buildfile: C:\Users\bmartin\Documents\Eclipse\Workspace_Luna\UnitTestProject\build.xml
build-subprojects:
init:
build-project:
    [echo] UnitTestProject: C:\Users\bmartin\Documents\Eclipse\Workspace_Luna\UnitTestProject\build.xml
build:
CalculatriceTest:
    [mkdir] Created dir: C:\Users\bmartin\Documents\Eclipse\Workspace_Luna\UnitTestProject\junit
    [junit] Running fr.eni.bo.CalculatriceTest
    [junit] Tests run: 2, Failures: 1, Errors: 0, Skipped: 0, Time elapsed: 0,158 sec
    [junit] Test fr.eni.bo.CalculatriceTest FAILED
junitreport:
[junitreport] Processing C:\Users\bmartin\Documents\Eclipse\Workspace_Luna\UnitTestProject\junit\TESTS-Te
[junitreport] Loading stylesheet jar:file:/C:/Program%20Files/Java/Eclipse_Luna/plugins/org.apache.ant_1.
[junitreport] Transform time: 1804ms
[junitreport] Deleting: C:\Users\bmartin\AppData\Local\Temp\null947295388
BUILD SUCCESSFUL
Total time: 3 seconds
```

# Génération du rapport JUnit



[Home](#)

Packages

[fr.eni\\_ecole.jse](#)

Classes

[AllTests](#)

## Unit Test Results.

Designed for use with [JUnit](#) and [Ant](#).

### Summary

| Tests             | Failures          | Errors            | Success rate | Time  |
|-------------------|-------------------|-------------------|--------------|-------|
| <a href="#">3</a> | <a href="#">1</a> | <a href="#">0</a> | 66.67%       | 0.090 |

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

### Packages

| Name                             | Tests | Errors | Failures | Time(s) | Time Stamp          | Host   |
|----------------------------------|-------|--------|----------|---------|---------------------|--------|
| <a href="#">fr.eni_ecole.jse</a> | 3     | 0      | 1        | 0.090   | 2012-02-13T14:09:15 | 36-254 |