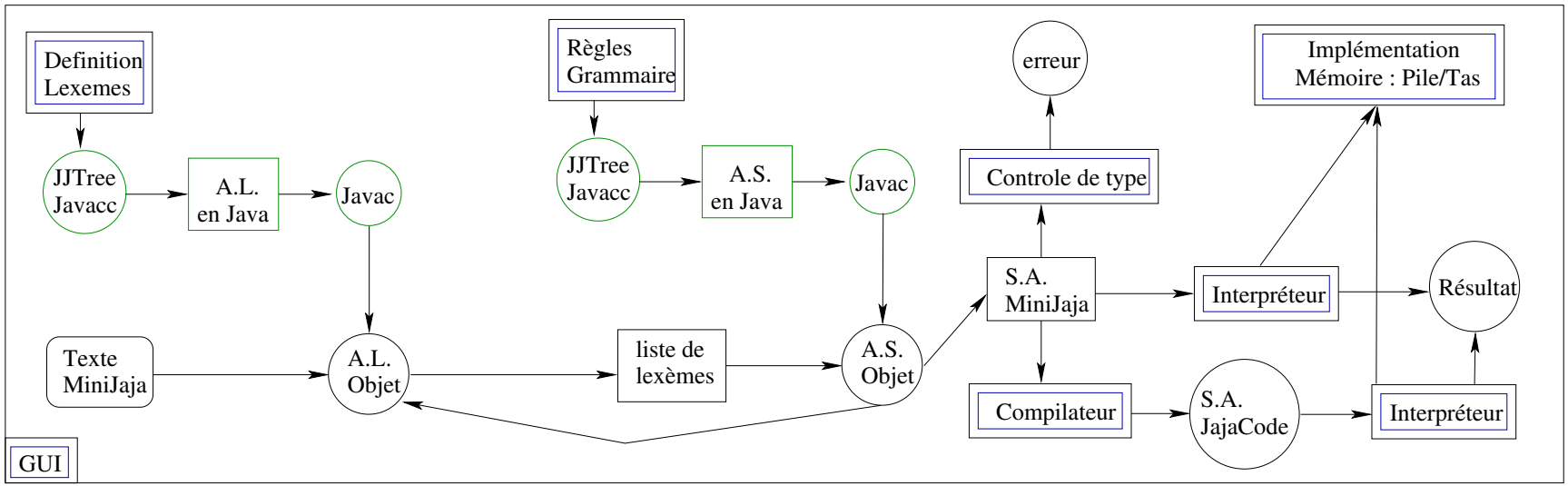
Compilation et Interprétations - TU



F. Bouquet

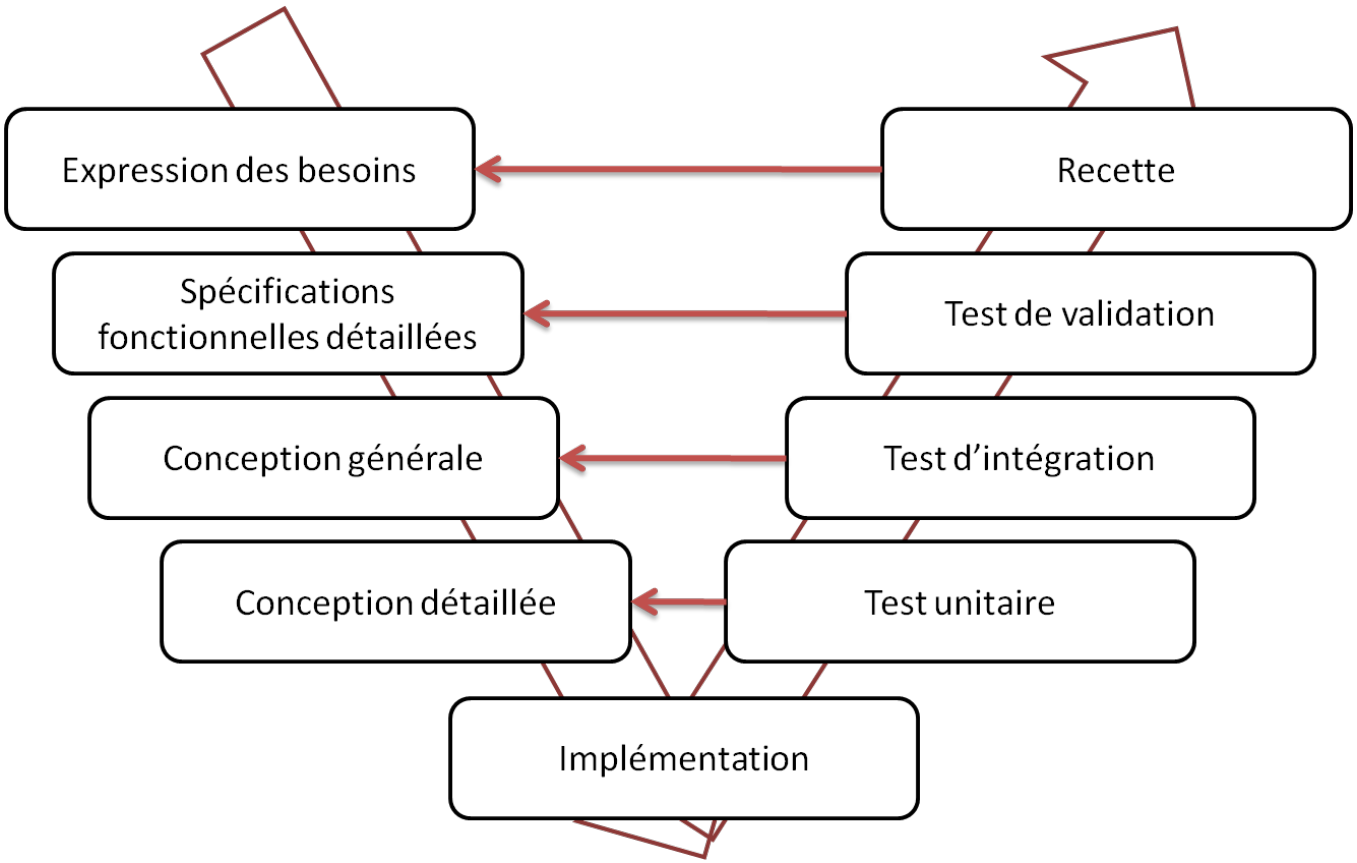
Master S&T - Mention Informatique

2015 - 2016

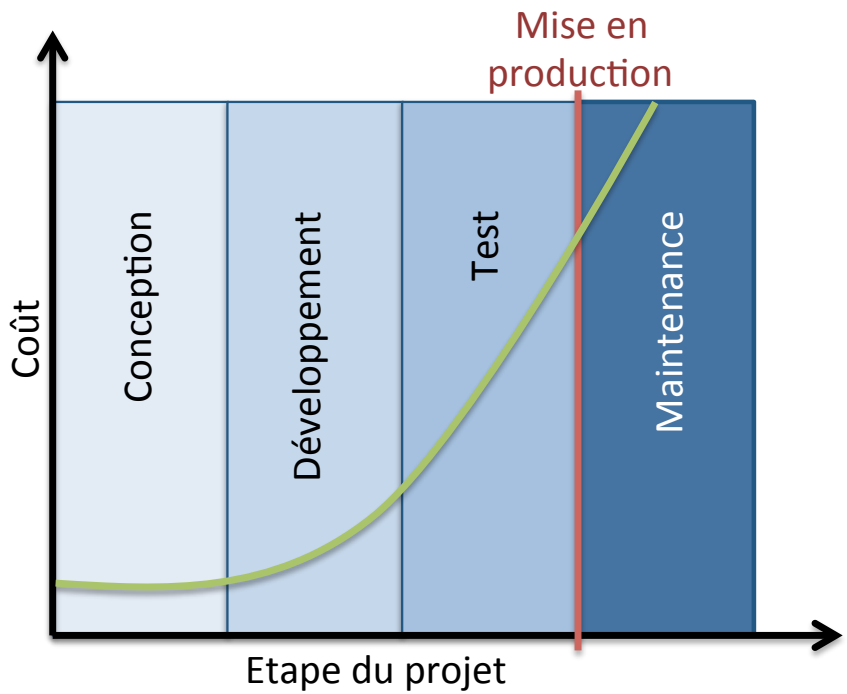
Université de Franche-Comté

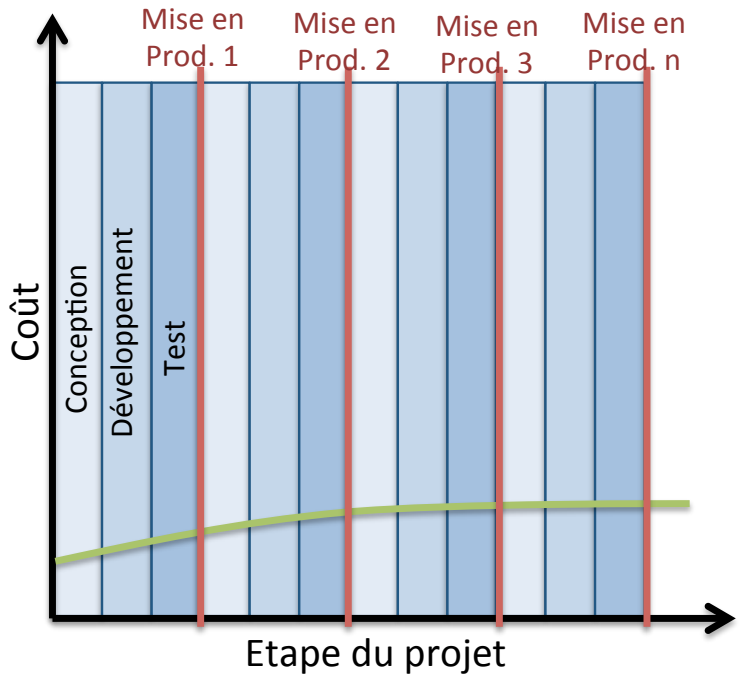
Plan

Cycle de développement en V

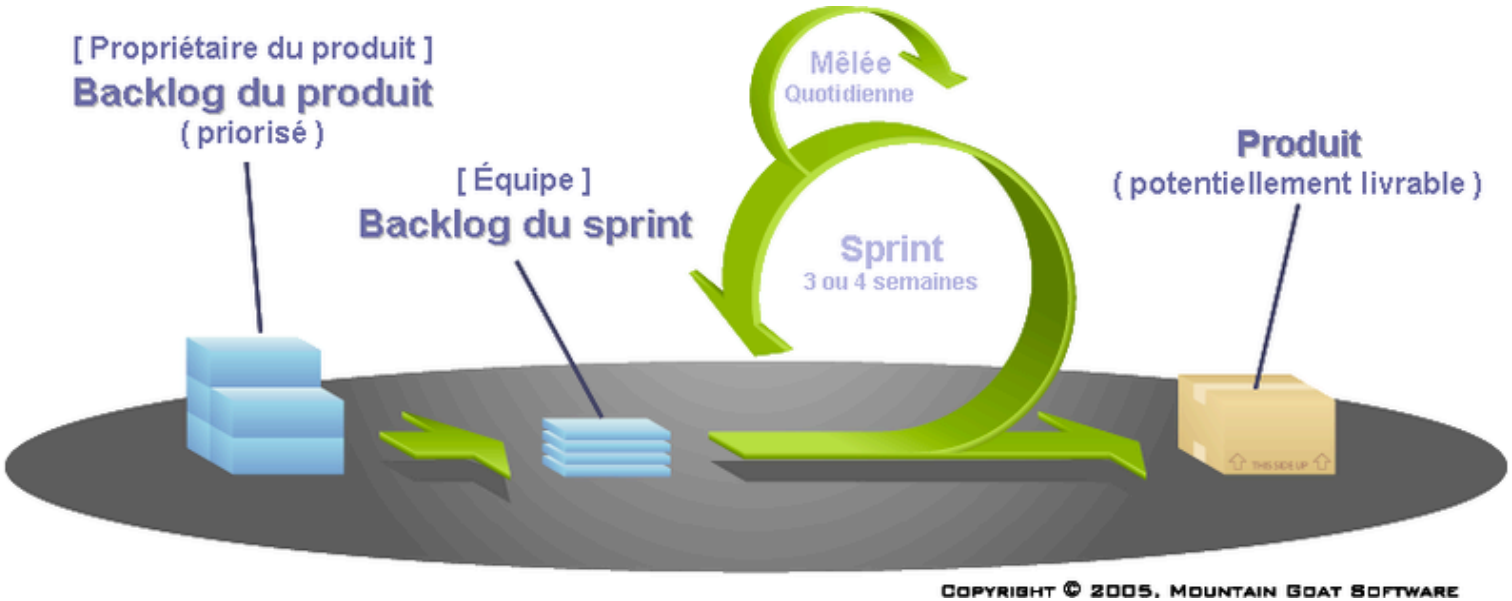


Coût


Mise en production
Coût
Conception
Développement
Test
Maintenance
Etape du projet


Mise en Prod. 1
Mise en Prod. 2
Mise en Prod. 3
Mise en Prod. n
Coût
Conception
Développement
Test
Etape du projet

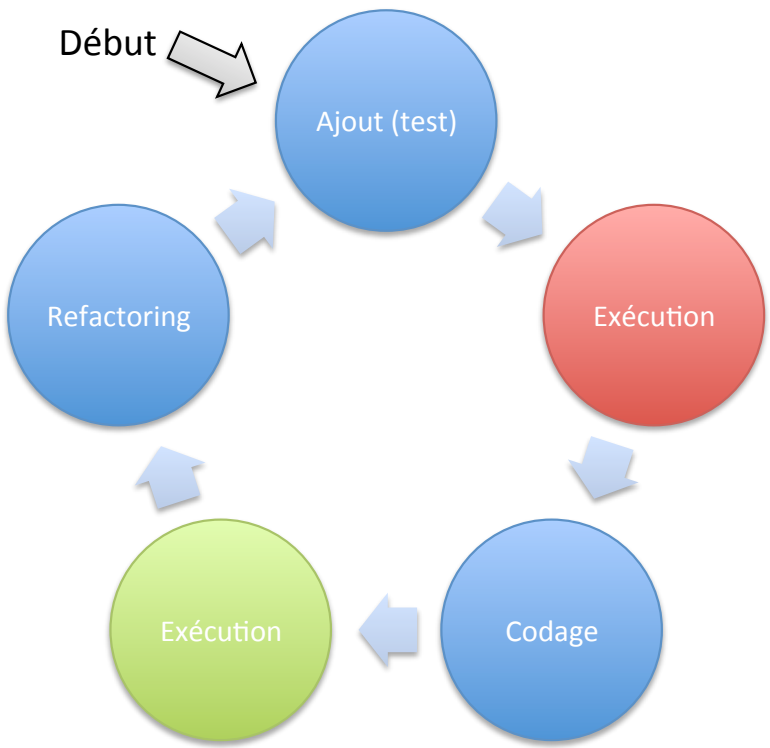
Méthode Scrum



Développement Guidé par les Tests (TDD)

Développement Dirigé par les Tests

- Méthode de développement dans les méthodes agiles
- Préconise l'écriture des tests avant le développement du code



Acteurs du test unitaire

Qui fait quoi ?

De part sa nature et sa proximité du code source, le test unitaire est pleinement associé à l'activité de développement.

Le test unitaire est à la charge du développeur

Test ?

IEEE (Standard Glossary of Software Engineering Terminology)

"Le test est l'exécution ou l'évaluation d'un système ou d'un composant par des moyens automatiques ou manuels, pour vérifier qu'il répond à ses spécifications ou identifier les différences entre les résultats attendus et les résultats obtenus"

G. Myers (The Art of Software testing)

"Tester, c'est exécuter le programme dans l'intention d'y trouver des anomalies ou des défauts "

Edsgar W. Dijkstra. Notes on structured programming. Academic Press, 1972

"Testing can reveal the presence of errors but never their absence ?"

Koidonc (Spécification)

Calculer la moyenne d'une liste de notes avec coefficients. La procédure prend :

1. En entrée une liste de notes entières suivie de son coefficient
2. En sortie la moyenne

Comment savoir si elle fait bien ce qu'il faut ?

Koidonc (Code)

```
public double koidonc(List<Integer> bag) {
    double moy=0.0;
    int nb=0, coef = 0, val = 0;
    for (int t : bag) {
        if (nb == 1) {
            coef += t;
            moy += t*val;
        } else
            val = t;
     nb = 1 - nb;
    }
    return moy / coef ;
}
```

Test JUnit

```
import org.junit.*;

public class TestFoobar {
    @BeforeClass
    public static void setUpClass() throws Exception {
        // Code exécuté avant l'exécution du premier test (et de la
            méthode @Before)
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
        // Code exécuté après l'exécution de tous les tests
    }

    @Before
    public void setUp() throws Exception {
        // Code exécuté avant chaque test
    }

    @After
    public void tearDown() throws Exception {
        // Code exécuté après chaque test
    }

    @Test
    public void test()
    {
        fail("Pas implémenté");
    }
}
```

AssertArrayEquals

assertArrayEquals([java.lang.String message,] Attendu, Obtenu)

- byte[]
- char[]
- short[]
- int[]
- long []
- java.lang.Object[]

Exemple

```
@Test public void test() throws Exception {
    int [] tab = classeSousTest.trieTab(new int[]{3,2,1});
    Assert.assertArrayEquals("Pb Tri tab",new int[]{1,2,3},tab);
}
```

AssertEquals

assertEquals([java.lang.String message,] Attendu, Obtenu)

- double (delta)
- long
- java.lang.Object
- java.lang.Object[]

Exemple

```
@Test public void test() throws Exception {
    double moy = classeSousTest.koidonc(Arrays.asList
        (10,2,10,5,10,7));
    Assert.assertEquals("Pb moyenne",10.0,moy,0.001);
}
```

AssertSame / AssertNotSame

assertSame([java.lang.String message,]
java.lang.Object Attendu, java.lang.Object Obtenu)

assertNotSame([java.lang.String message,]
java.lang.Object Attendu, java.lang.Object Obtenu)

Différence entre **"Same"** et **"Equals"** c'est la comparaison :

- Same, on utilise ==
- Equals, on utilise `.equals`

Exemple

```
@Test public void test() throws Exception {
int [] tabIn = new int[]{3,2,1};
    int [] tab = classeSousTest.trieTab(tabIn);
    Assert.assertSame("Pb Tri tableau",tabIn,tab);
}
```

AssertFalse / AssertTrue

assertFalse([java.lang.String message,] boolean condition)
assertTrue([java.lang.String message,] boolean condition)

Exemple

```
@Test public void test() throws Exception {
    double moy = classeSousTest.koidonc(Arrays.asList
        (10,2,10,5,10,7));
    Assert.assertFalse("Pb moyenne",10.0 != moy);
}
```

Exemple

```
@Test public void test() throws Exception {
    double moy = classeSousTest.koidonc(Arrays.asList
        (10,2,10,5,10,7));
    Assert.assertTrue("Pb moyenne",10.0 == moy);
}
```

AssertNull / AssertNotNull

assertNull([java.lang.String message,] java.lang.Object object)
assertNotNull([java.lang.String message,] java.lang.Object object)

Exemple

```
@Test public void test() throws Exception {
    int [] tab = classeSousTest.trieTab(new int[]{3,2,1});
    Assert.assertNull("Tableau non null",tab);
}
```

Exemple

```
@Test public void test() throws Exception {
    int [] tab = classeSousTest.trieTab(new int[]{3,2,1});
    Assert.assertNotNull("Tableau null",tab);
}
```

Fail

fail([java.lang.String message])

Exemple

```
@Test public void test() throws Exception {
    int v = classeSousTest.AuDessusMoy(new int[]{4,3,2,1});
    if ( v < 3) fail("Calcule Moyenne");
}
```

AsserThat 1/2

assertThat(java.lang.String reason, T actual,
org.hamcrest.Matcher<T> matcher)

- **Corps**
  - `any()` : Correspond à tous
  - `is()` : vérifie si les objets donnés sont égaux.
  - `describedAs()` : Ajoute une description à la correspondance
- **Logique**
  - `allOf()` : Prend un tableau de "matcher" et tous doivent correspondre à l'objet cible.
  - `anyOf()` : Prend un tableau de "matcher", et au moins un doit correspondre à l'objet cible.
  - `not()` : Prend la négation.

AsserThat 2/2

assertThat(java.lang.String reason, T actual,
org.hamcrest.Matcher<T> matcher)

- **Objet**
  - `equalTo()` : vérifie si les objets donnés sont égaux.
  - `instanceOf()` : vérifie que l'objet cible est bien du type X ou compatible avec le type X
  - `notNullValue()` / `nullValue()` : vérifie si l'objet est null ou non null.
  - `sameInstance()` : vérifie si l'objet est exactement la même instance qu'un autre.

Exemple

```
@Test
public void testWithMatchers() {

    Assert.assertThat("Zéro égale un", 0, is(not(1)));

}
```

Gestion d'exception

Problème

On veut vérifier que dans un cas précis, une méthode renverra bien une exception donnée.

Principe

Spécifier cette exception en paramètre de l'annotation @Test et renvoie une exception si le test ne renvoie pas l'exception de ce type.

Exemple

```
@Test(expected=NullPointerException.class)
public void test() throws Exception {
    double moy = classeSousTest.koidonc(null);
}
```

Gestion du temps

Problème

On veut vérifier qu'une méthode s'exécute dans un délai donné.

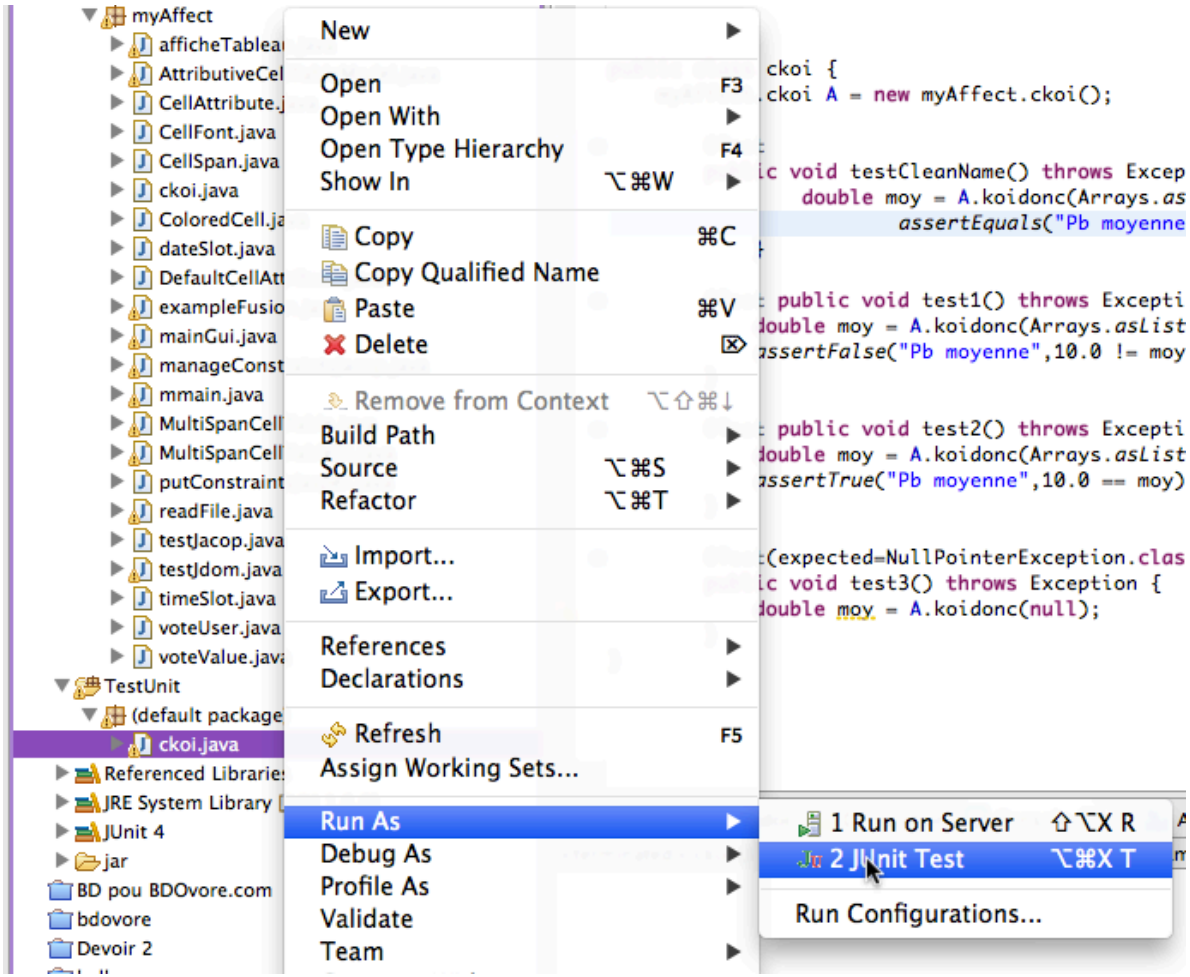
Principe

Spécifier ce délai (en millisecondes) en paramètre de l'annotation @Test et renvoie une exception si le test prend plus de temps.

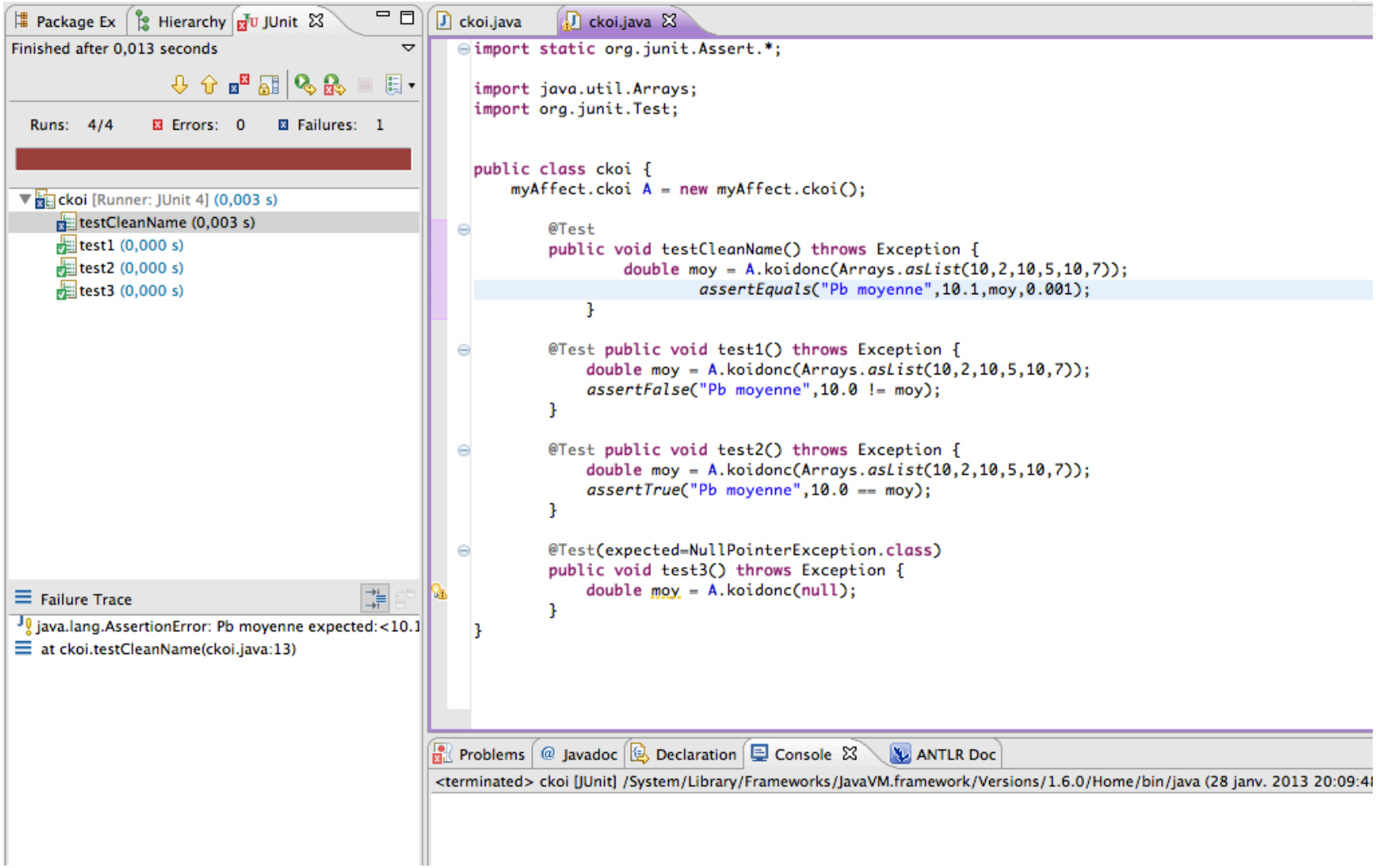
Exemple

```
@Test(timeout=100)
public void test() throws Exception {
    double moy = classeSousTest.koidonc(Arrays.asList
        (10,2,10,5,10,7))
}
```

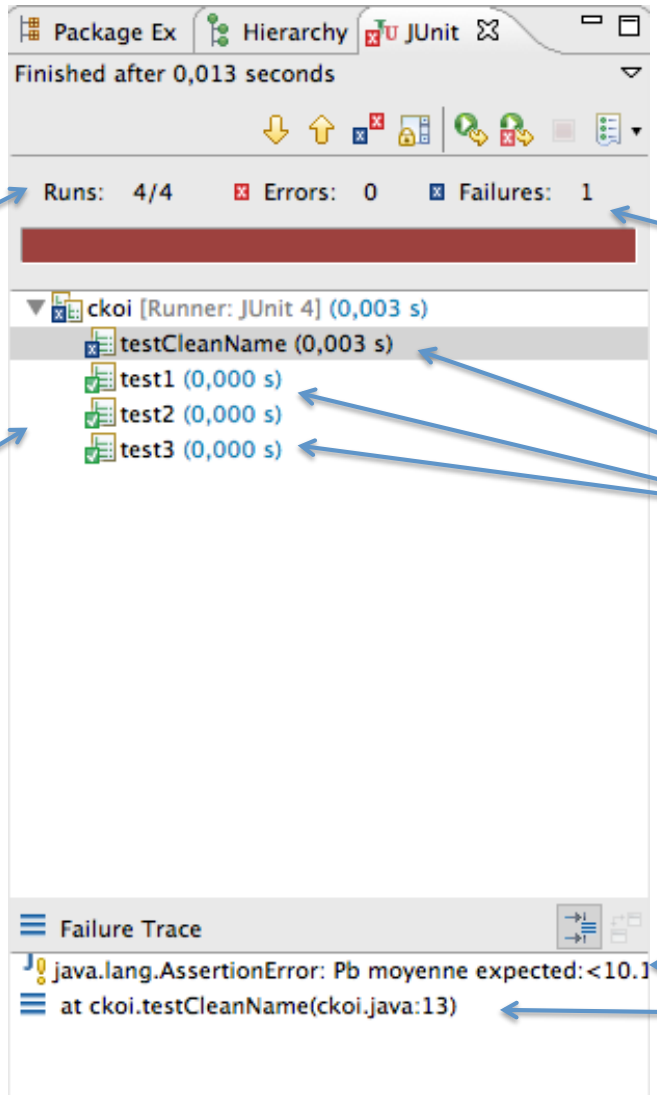
Environnement Eclipse



Environnement Eclipse



Environnement Eclipse



Nombre de tests effectués

Nombre de tests en échec

Liste des tests

Test en échec

Tests réussis

Description de l'erreur

Ligne de l'erreur

Bilan

Bonne Pratique

- Une classe de code / Une classe de tests
- Test et classe dans le même *"package"*
- Les cas nominaux
- Les cas tordus / catastrophes

Mauvaise Pratique

- Répertoire(s) mélangeant les sources et les tests
- Tests triviaux
- Effets de bord (ordre des tests, états dégradés)

Bilan

En résumé, écrire des tests permet :

- Analyse fine ;
- Réduction des bogues ;
- Non-régression du code (refactoring) ;
- Documentation efficace de votre code ;
- Sérénisation du développement
- Efficacité.

$\Longrightarrow$ **Si les tests sont écrits au fur et à mesure**