# Le test de logiciel

CM1- Introduction au test du logiciel

**CM2- Le test unitaire des logiciels à objets**

CM3- L’analyse de mutation

CM4- Le test d’intégration des logiciels à objets

CM5- Le test système des logiciels à objets

# Plan

1. Introduction au test unitaire
2. JUnit, une bibliothèque Java de test unitaire
3. Test-driven development

# Plan

1. **Introduction au test unitaire**
2. JUnit, une bibliothèque Java de test unitaire
3. Test-driven development

# Test unitaire OO

- **Tester une unité isolée du reste du système**
- **L'unité est la classe**
  - Test unitaire = test d'une classe
- **Test du point de vue client**
  - les cas de tests appellent les méthodes depuis l'extérieur
  - on ne peut tester que ce qui est public
  - Le test d'une classe se fait à partir d'une classe extérieure
- **Au moins un cas de test par méthode publique**
- **Il faut choisir un ordre pour le test**
  - quelles méthodes sont interdépendantes?

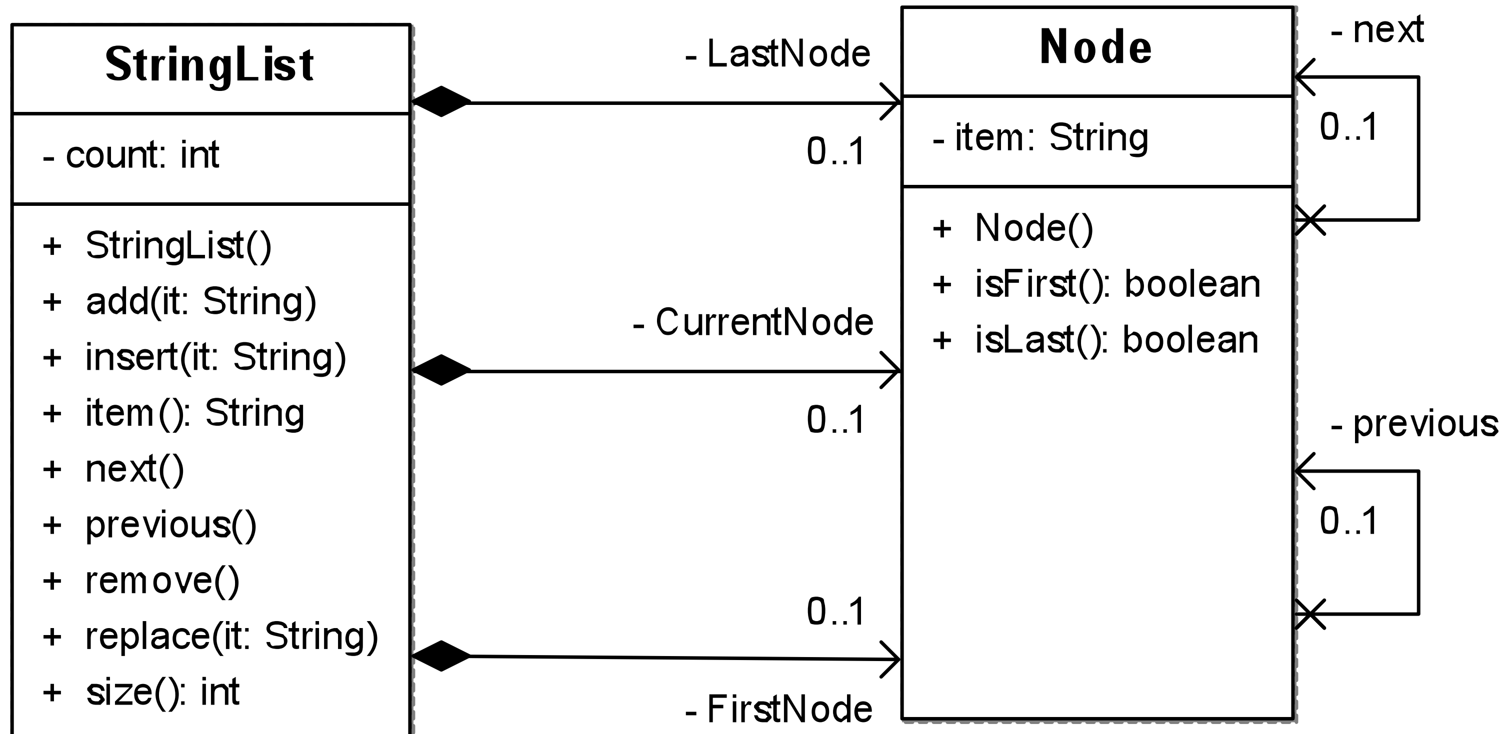
# Test unitaire OO

- **Problème pour l'oracle :**
  - Encapsulation : les attributs sont souvent privés
  - Difficile de récupérer l'état d'un objet
- **Penser au test au moment du développement (« testabilité »)**
  - prévoir des accesseurs en lecture sur les attributs privés
  - des méthodes pour accéder à l'état de l'objet

# Cas de test unitaire

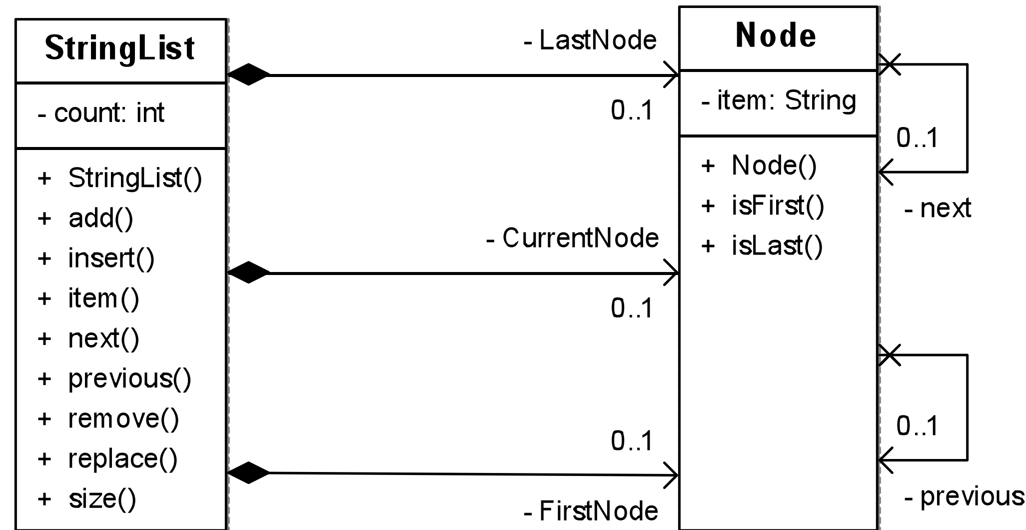
- **Cas de test = une méthode**
- **Corps de la méthode**
  - Configuration initiale
  - Une donnée de test
    - un ou plusieurs paramètres pour appeler la méthode testée
  - Un oracle
    - il faut construire le résultat attendu
    - ou vérifier des propriétés sur le résultat obtenu
- **Une classe de test pour une classe testée**
  - Regroupe les cas de test
  - Il peut y avoir plusieurs classes de test pour une classe testée

# Exemple : test de StringList




- Créer une classe de test qui manipule des instances de la classe StringList
- Au moins 9 cas de test (1 par méthode publique)
- Pas accès aux attributs privés : count, LastNode, CurrentNode, FirstNode

# Exemple : insertion dans une liste




spécification du cas de test

initialisation

appel avec donnée de test

oracle

```
//first test for insert: call insert and see if
//current element is the one that's been inserted
public void testInsert1(){
  StringList list = new StringList();
  list.add("first");
  list.add("second");
  list.insert("third");
  assertTrue(list.size()==3);
  assertTrue(list.item()=="third");
}
```

# Plan

1. Introduction au test unitaire
2. JUnit, une bibliothèque Java de test unitaire
3. Test-driven development

# JUnit

## Origine

- Xtreme Programming (test-first development)
- framework de test écrit en Java par E. Gamma et K. Beck
- open source: www.junit.org

## Objectifs

- test d'applications en Java
- faciliter la création des tests
- tests de non régression

## JUnit : un framework

*« Un framework est un ensemble de classes et de collaborations entre les instances de ces classes. »*

http://st-www.cs.uiuc.edu/users/johnson/frameworks.html

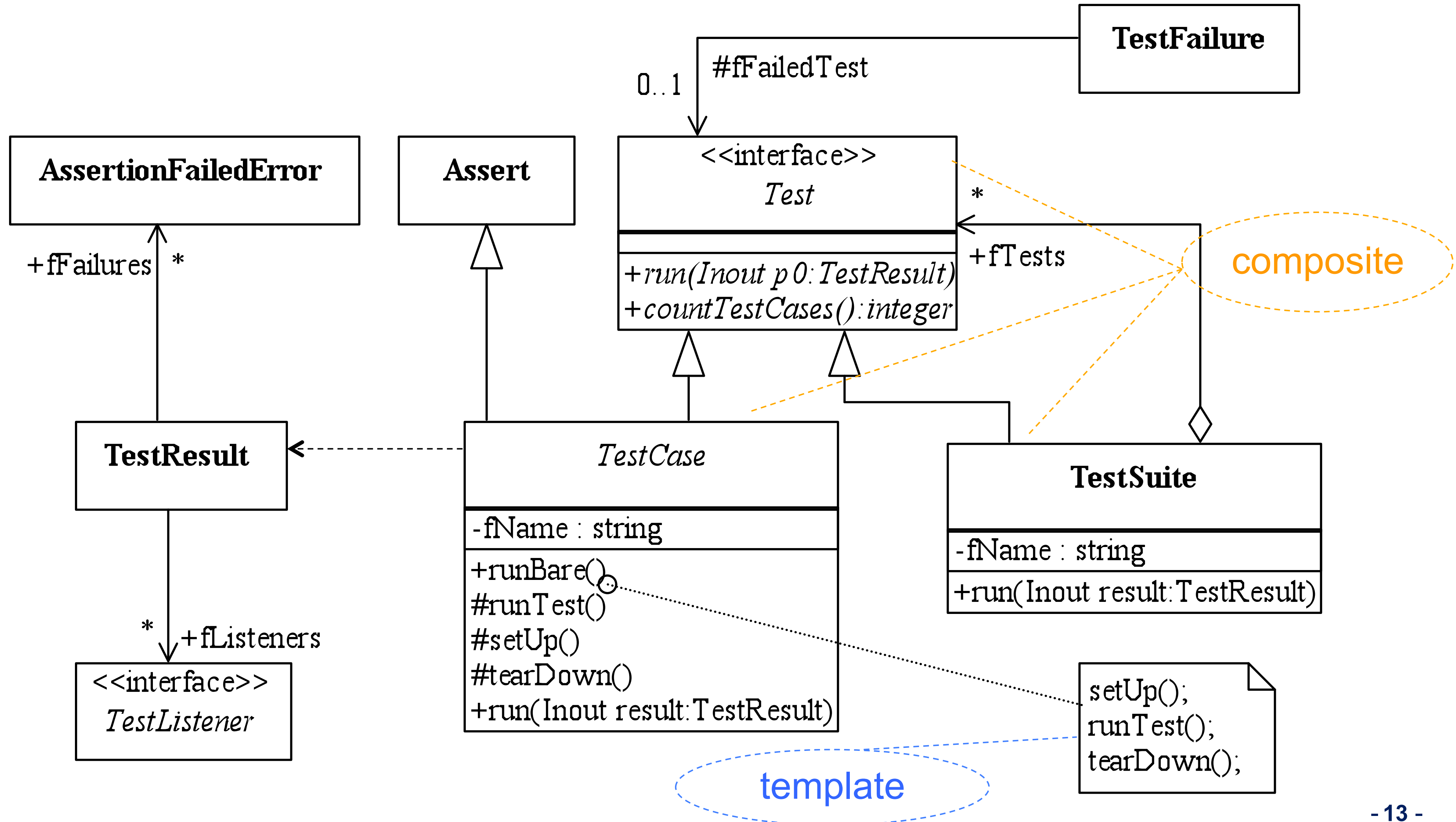
# JUnit : un framework

- **Le source d'un framework est disponible**
- **Ne s'utilise pas directement: il se spécialise**

  Ex: pour créer un cas de test on hérite de la classe TestCase

➔ Un framework peut être vu comme un programme à « trous » qui offre la partie commune des traitements et chaque utilisateur le spécialise pour son cas particulier.

# JUnit : un framework



# JUnit v3 : codage (1/5)

## Organisation du code des tests

- cas de Test: TestCase
  - setUp() et tearDown()
  - les méthodes de test
- suite de Test: TestSuite
  - Méthodes de test
  - Cas de test
  - Suite de Test

## Exemple : une classe StringList

```
//Une classe qui défini une liste de chaines de caractères

public class StringList {
    private int count;
    private Node lastNode;
    private Node firstNode;

    public StringList(){
        count=0;
    }

    public String item(){...}
    public int size(){...}
    public void add (String it){...}
    public void insert (String it){...}
...
}
```

# JUnit v3 : codage (2/5)

## Codage d'un « TestCase »:

- déclaration de la classe
- Hérite de `JUnit.framework.TestCase`

```
public class TestStringList extends TestCase {
    //déclaration des instances
    private StringList list;

    //setUp()
    //tearDown()
    //méthodes de test
    //main()
}
```

# JUnit v3 : codage (3/5)

- **la méthode setUp:**

```
//appelée avant chaque cas de test
//permet de factoriser la construction de « l'état du monde »

protected void setUp() throws Exception {
      list = new StringList();
}
```

- **la méthode tearDown:**

```
//appelée après chaque cas de test
//permet de défaire « l'état du monde »

protected void tearDown() throws Exception {
      super.tearDown();
}
```

# JUnit v3 : codage (4/5)

- **les méthodes de test:**

```
//test add two elements
public void testAdd2(){
    list.add("first");
    list.add("second");
    assertTrue(list.size()==2);
    assertTrue(list.item()=="second");
}
```

- **caractéristiques:**
  - nom préfixé par « test »
  - publique, type de retour `void`
  - contient des assertions (définie l’oracle)

# Les assertions de JUnit

- assertTrue(...), assertFalse(...)
- assertEquals(Object, Object)
- assertSame(Object, Object)
- assertNull(...)
- assertEquals(double expected, double actual, double delta)

⇨ Voir la liste des méthodes static de la classe Assert

# JUnit v3 : codage (5/5)

- **regroupement des méthodes de test:**

```
public static Test suite() {
   TestSuite suite = new TestSuite();
   suite.addTest(newTestedClass("testAdd2()");
   suite.addTest(TestStringList.suite());
   suite.addTestSuite(TestStringList.class)
   return suite;
}
```

# JUnit v3 : lancement des tests

- **On utilise un TestRunner graphique ou textuel.**

  - graphique : « *keep the bar green to keep the code clean* »

    ```
    java junit.swingui.TestRunner
    ```

  - textuel : affichage des résultats sur la console

    ```
    junit.textui.TestRunner.run(<ma suite de test>);
    // main
    junit.textui.TestRunner <ma suite de Test>
    // ligne de commande
    ```

  ➔ java -classpath $CLASSPATH:~/<monchemin>junit.jar <TestClass>

# JUnit v3 : détail d’implémentation

- **Pour exécuter une suite de tests, JUnit utilise l’introspection**

```
public TestSuite (final Class theClass){
    …
    Method[] = theClass.getDeclaredMethods
    …
}
private boolean isTestMethod(Method m) {
    String name= m.getName();
    Class[] parameters= m.getParameterTypes();
    Class returnType= m.getReturnType();
    return parameters.length == 0 && name.startsWith("test") &&
returnType.equals(Void.TYPE);
    }
```

## JUnit version 4

- Fonctionne avec Java 5
- Utilisation intensive des annotations
- Plus de runner graphique (laissé au soin des IDEs)
- Paquetage `org.junit`
- Nouvelle architecture
- Tests paramétrés, timeouts, etc

# JUnit v4 : classe et méthode de test

## Classe de test :

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

## Méthodes de test :

- Nom de méthode quelconque
- Annotation `@Test`
- Publique, type de retour `void`
- Pas de paramètre, peut lever une exception
- Annotation `@Test(expected = Class)` pour indiquer l'exception attendue
- Annotation `@Ignore` pour ignorer un test

# JUnit v4 : classe et méthode de test

- **Méthodes avec annotations `@Before` ou `@After`**
  - `import org.junit.Before;, etc`
  - Publiques (et plus protected)
  - Exécutées avant/après chaque méthode de test
  - Possibilité d'annoter plusieurs méthodes (ordre d'exécution indéterminé)
- **Méthodes avec annotations `@BeforeClass` et `@AfterClass`**
  - Publiques et statiques
  - Exécutées avant (resp. après) la première (resp. dernière) méthode de test
  - Une seule méthode pour chaque annotation

## JUnit v4 : suite de test

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

```
@RunWith(Suite.class)
@SuiteClasses({TestSuitePartie1.class, TestSuitePartie2.class})
public class TestSuite {
}
```

# JUnit v4 : test paramétrés

## Pour factoriser les données de test :

- runner `org.junit.runners.Parameterized`
- Données fournies par une méthode publique statique qui retourne une collection et est annotée par `@Parameters`
- constructeur public pour la classe de test qui prend en paramètre les données et le résultat attendu
- produit en croix des données de test et des méthodes de test

```java
@RunWith(Parameterized.class)
public class TestParametre {

    @Parameters
    public static List<Object[]> getParametres() {
        return Arrays.asList(new Object[][] {
                { 10L, 0L, new long[] { 10L, 0L } },
                { 30L, 300L, new long[] { 10L, 20L } },
                { 7936L, 1125899906842624L, new long[] { 256L, 512L, 1024L, 2048L, 4096L } },
        });
    }

    private long resultatAddition;
    private long resultatMultiplication;
    private long[] nombres;

    public TestParametre(long pResultatAddition, long pResultatMultiplication, long[] pNombres) {
        resultatAddition = pResultatAddition;
        resultatMultiplication = pResultatMultiplication;
        nombres = pNombres;
    }

    @Test
    public void addition() {
        System.out.println("addition :       - attendu : " + resultatAddition +
                           "\n               - nombres : " + Arrays.toString(nombres));
        final long lAddition = Operations.additionner(nombres);
        Assert.assertEquals(resultatAddition, lAddition);
    }

    @Test
    public void mutiplication() {
        System.out.println("mutiplication : - attendu : " + resultatMultiplication +
                           "\n               - nombres : " + Arrays.toString(nombres));
        final long lMultiplication = Operations.multiplier(nombres);
        Assert.assertEquals(resultatMultiplication, lMultiplication);
    }
}
```

# JUnit v4 : lancement des tests

- **Le runner par défaut exécute tous les tests (non ignorés) de la classe**

  ➔ java org.junit.runner.JUnitCore <TestClass>

# Intégration dans Eclipse

Java - testStringList.java - Eclipse Platform

File Edit Source Refactor Navigate Search Project Run JML Window Help

Package Explorer | Hierarchy | JUnit

Finished after 0,07 seconds

Runs: 18/18 Errors: 0 Failures: 4

Failures | Hierarchy

- testRemove1 - testStringList
- testRemove2 - testStringList
- testRemove3 - testStringList
- testRemove4 - testStringList

Failure Trace

- junit.framework.AssertionFailedError
- at testStringList.testRemove1(testStringList.java:164)
- at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Met
- at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Sc
- at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknow

*list.ucd | *testClass.ucd

**testStringList**
+ testCreate()
+ testAdd1()
+ testAdd2()
+ testPrevious1()
+ testPrevious2()
+ testNext1()
+ testPrevious3()
+ testInsert1()
+ testInsert2()
+ testInsert3()
+ testInsert4()
+ testInsert5()
+ testReplace1()
+ testReplace2()
+ testRemove1()
+ testRemove2()
+ testRemove3()
+ testRemove4()

**StringList**
- count: int
+ StringList()
+ item(): String
+ next()
+ previous()
+ size(): int
+ add(it: String)
+ replace(it: String)
+ insert(it: String)
+ remove()

**Node**
- item: String
+ Node()
+ isFirst(): boolean
+ isLast(): boolean
+ setItem(item: String)

- LastNode 0..1 / - CurrentNode 0..1 / - FirstNode 0..1 / 0..1 - previous / 0..1 - next

StringList.java | testStringList.java

```java
//test remove an element in the middle of a list
public void testRemove1(){
    list.add("first");
    list.add("second");
    list.add("third");
    list.previous();
    list.remove();
    assertTrue(list.size()==2);
}

//test remove last element
public void testRemove2(){
    list.add("first");
    list.add("second");
    list.remove();
    assertTrue(list.size()==1);
}

//test remove first element
public void testRemove3(){
    list.add("first");
    list.add("second");
    list.previous();
```

Problems | Javadoc | Declaration | Console | Properties

Writable | Smart Insert | 2 : 1

démarrer | C:\Documents and Se... | IntegrationTest.ppt | IntroTestDIIC3.ppt | Adobe Acrobat Profe... | Java - testStringList.j... | UnitTest2.ppt | FR 15:48

# JUnit v4 : Intégration dans Eclipse

- **Si les classes de test ne sont pas reconnues comme telles :**
  - compatibilité arrière avec JUnit 3.8
  - adaptateur junit.framework.JUnit4TestAdapter
  - permet de créer une suite de test 3.8
  - ajouter dans chaque classe de test

```
public static junit.framework.Test suite() {
    return new JUnit4TestAdapter(TestParamSum.class);
}
```

# JUnit

- **Permet de structurer les cas de test**
  - cas de test / suite de test
- **Permet de sauvegarder les cas de test**
  - important pour la non régression
  - quand une classe évolue on ré-exécute les cas de test

# JUnit

- **www.junit.org**
- **Avantages**
  - gratuit
  - simple
  - intégré à Eclipse
- **Inconvénients**
  - exploitation des résultats (pas d'historique...)
- **Généralisation des concepts (XUnit)**
  - www.testdriven.com

# Plan

1. Introduction au test unitaire

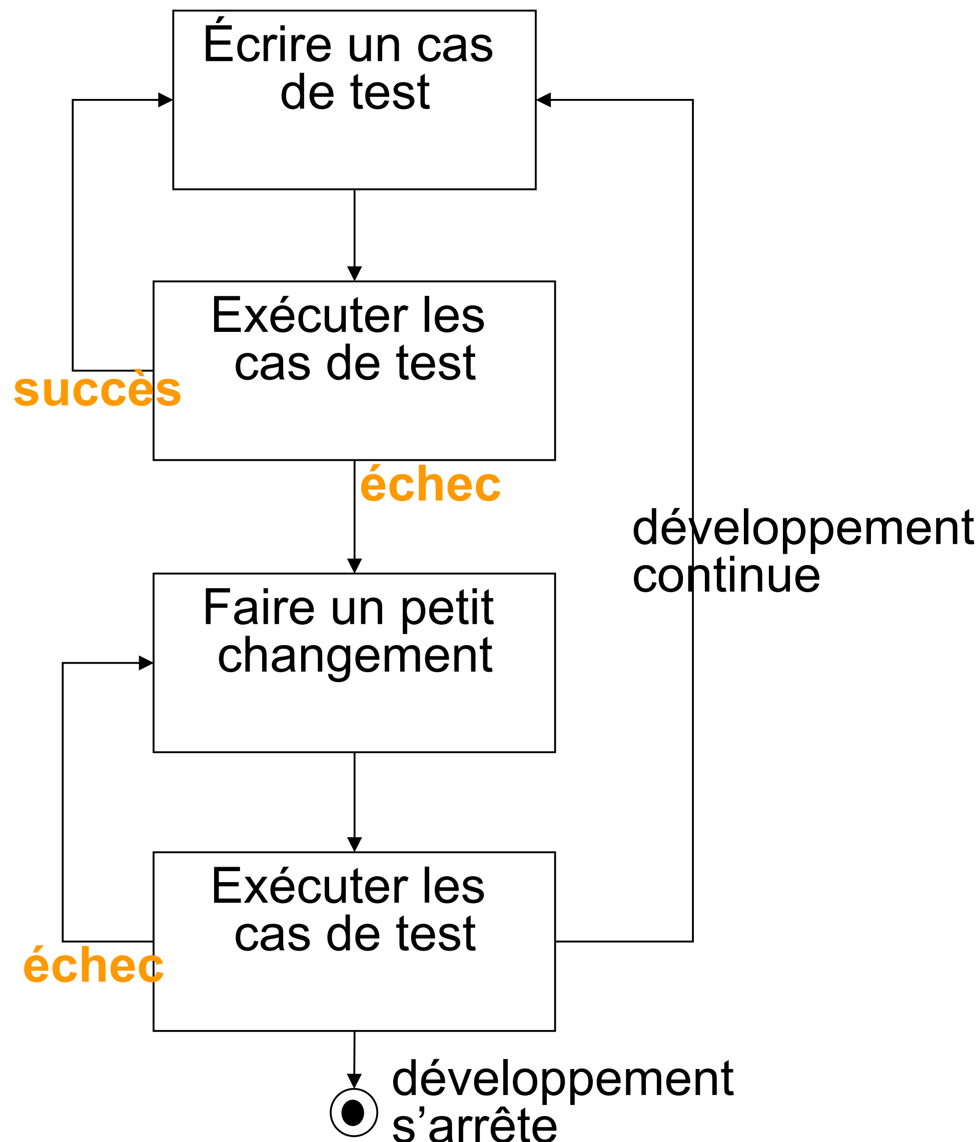
2. JUnit, une bibliothèque Java de test unitaire

3. **Test-driven development**

## Test-first development

- **Xtreme programming**
- **Écrire les cas de test avant le programme**
  - les cas de test décrivent ce que le programme doit faire
  - avant d'écrire le code, les cas de test échouent
  - quand on développe, les cas de test doivent passer
- **Ensuite on développe la partie du programme qui fait passer le cas de test**

# Test-first development




Exemple : ajout dans une liste chainée

```
public void testAdd(){
	list.add("first");
	assertTrue(list.size()==1);
}

public void add (String it){}
	Node node = new Node();
	node.setItem(it);
	node.setNext(null);
	if (firstNode == null) {
		node.setPrevious(null);
		this.setFirstNode(node);
	}
	lastNode = node;
	this.setCurrentNode(node);
}
```

# Test-first development

- **Les cas de test servent de support à la documentation**
  - des exemples d'utilisation du code
- **Tester avec une intention précise**
  - qu'est-ce qu'on teste?
  - pourquoi on le teste?
  - quand est-ce assez testé?
- **Retours rapides sur la qualité du code**
  - itérations courtes dans le cycle de développement
  - on exécute le code tout de suite (avant même de l'avoir écrit)
  - On ne code que quand un test a échoué

# Test-first development

- **Les cas de test spécifient ce que le programme doit faire mais pas comment**
  - il faut associer TDD à des refactorings fréquents
    - revoir la structure du code
    - ne pas oublier la conception
  - Grande importance du test de non-régression
    - quand on refactore les cas de test qui passaient doivent continuer à passer

# Test-first development

- **Importance d'un environnement pour l'exécution automatique des tests**
  - pouvoir exprimer facilement des test unitaires
  - pouvoir les exécuter rapidement
  - pouvoir réexécuter les cas de test
- **JUnit + d'autres outils pour automatiser le test et permettre TDD**