ENSA Oujda

Module Usine Logiciel

M. MELLAH Youssef

ENSAO 2021-2022

Rappel du Plan générale Usine Logiciel

- Framework de Test Unitaires: JUnit3/JUnit4
- Outil de build : Maven
- Système de gestion de versions : Git
- Analyse de qualité de code source : SonarQube
- Intégration continue : Jenkins
- Outil de virtualisation : Docker

Junit FrameWork & Tests

M. MELLAH Youssef

ENSAO 2021-2022

Plan

- Rôles des tests dans un projet
- Différents types de tests
- Modes des tests
- Présentation de Junit
- Junit en Pratique
- Bonnes pratiques pour Junit
- TP

Rôles des tests dans un projet

- Objectif d'un test:
  - Trouver des anomalies qui n'ont pas été détectées
  - Vérifier la conformité d'un logiciel avec ses spécifications
  - Assurer l'avancement du projet

Différents types de tests

- Les tests unitaires :
  - Tests réalisés sur des fichiers ou des programmes isolés de toutes relations avec d'autres programmes
  - Valider la qualité du code et les performances des différents modules

Différents types de tests

- Les tests d'intégration:
  - Utiliser pour révéler les problèmes d'interface entre les différents programmes
  - Valider l’intégration des modules entre eux et dans leur environnement d’exploitation définitif

Différents types de tests

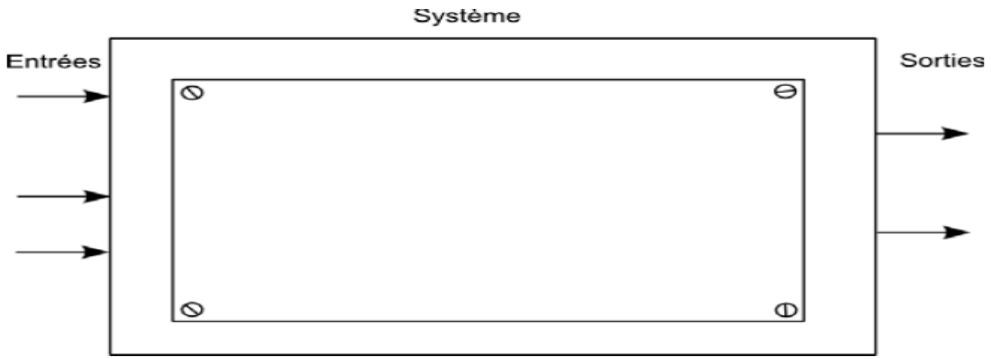
- Les tests fonctionnels:
  - Vérifier qu'il n'y a pas d'anomalies dans les fonctions réalisées par l'application
  - Valider les spécifications techniques et les exigences fonctionnelles

Différents types de tests

- Les tests de non-régression
  - Vérifier que les modifications apportées n'ont pas perturbé l'application

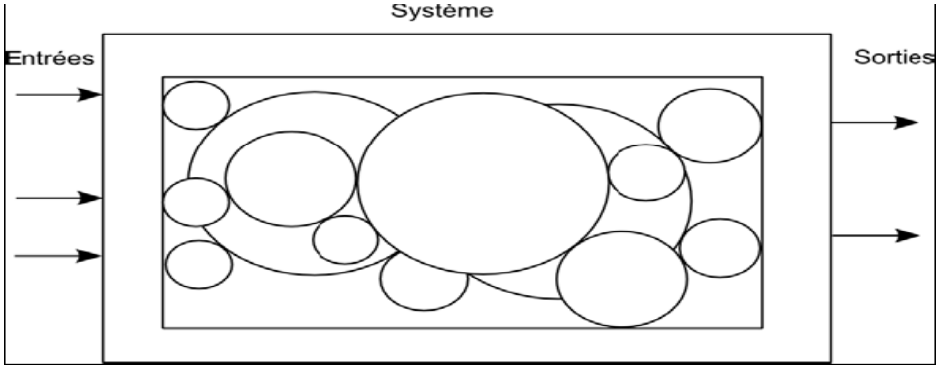
Modes de tests

- Les tests en boîte noire:
  - Les tests consistent alors à soumettre le système à différentes séries d'entrées et à en étudier le comportement en sortie.
  - Dans le domaine logiciel, cela revient à valider le comportement d'une classe en n'exploitant que ses méthodes publiques et sans connaissance préalable de son implémentation.



Modes de tests

- Les tests en boîte blanche :
  - Par opposition aux tests en boîte noire s'est développée la notion de tests en boîte blanche. Ceux-ci présupposent une connaissance du fonctionnement interne et viennent valider certains points précis en utilisant des moyens non publics ou en testant les rouages internes de façon individuelle.



Test Driven Development

- Le Test Driven Development (TDD) ou en français développement piloté par les tests est une technique de développement de logiciel qui préconise d'écrire les tests unitaires avant d'écrire le code source d'un logiciel.
- **Le cycle de TDD**

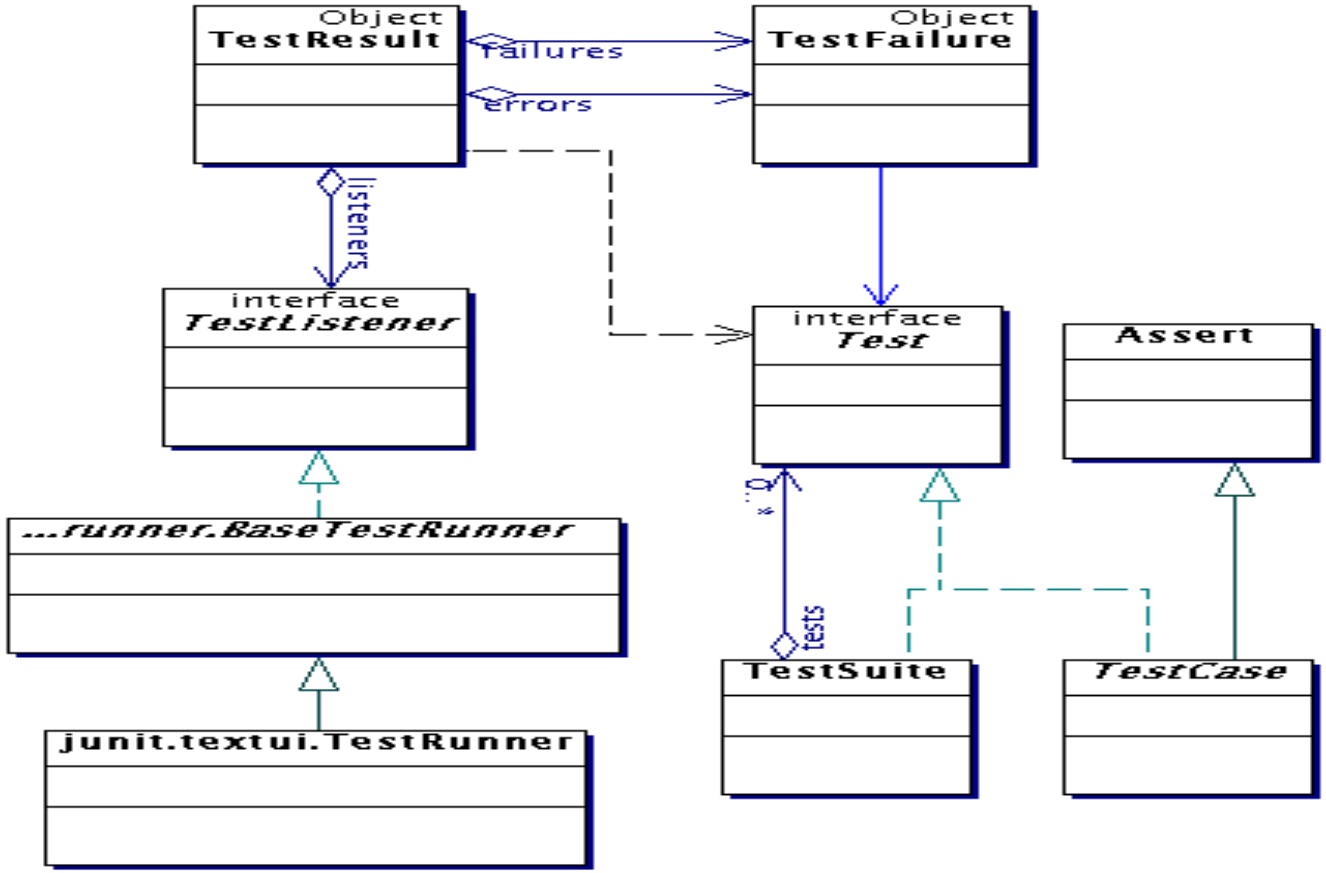
Le cycle préconisé par TDD comporte cinq étapes :

1. écrire 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. écrire juste le code suffisant pour passer le test ;
4. vérifier que le test passe ;
5. puis refactoriser le code, c'est-à-dire l'améliorer tout en gardant les mêmes fonctionnalités.

Présentation de Junit

- JUnit est un framework de tests unitaires pour le langage Java créé par Kent Beck et ErichGamma.
- JUnit est particulièrement adapté aux développeurs souhaitant faire du développement dirigé par les tests (Tests Driven Development)
- Objectifs
  - test des applications Java
  - faciliter la création des tests
  - tests de non régression
- Deux versions majeurs : 3 et 4

Présentation de Junit



Junit 3 en Pratique (1/6)

- 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
- Lancement des tests
  - le TestRunner

Exemple:une classe Rectangle

```
public class Rectangle {
        private int largeur;
        private int longueur;

        public Rectangle(int largeur, int longueur)
        {
                this.largeur = largeur;
                this.longueur = longueur;
        }

        /* Calcule du périmètre du rectangle */
        public int calculPerimetre(){
                return  longueur + largeur;
        }
        …
}
```

Junit 3 en Pratique (2/6)

- Codage d'un « TestCase »:
  - déclaration de la classe:

```
public class RectangleTest extends TestCase {
        //délaration des instances
        private Rectangle aRectangle;
        ...

        //contructeur
        public RectangleTest(String name){
        super(Name);  }

        //setUp() et tearDown()
        //méthodes de test
        //création d'une suite de test
}
```

Junit 3 en Pratique (3/6)

- la méthode setUp:

```
protected void setUp() {
        //appellée avant chaque méthode de test
         aRectangle = new Rectangle(12, 10);

…}
```

- la méthode tearDown:

```
protected void tearDown {
        //appellée après chaque méthode de test
         aRectangle = null;
        ...
        ...
}
```

Junit 3 en Pratique (4/6)

- Dans les méthodes de test unitaire, les méthodes testées sont appelées et leur résultat est testé à l'aide d'assertions :
- assertEquals(a,b)
  - Teste si a est égal à b (a et b sont soit des valeurs primitives, soit des objets possédant des bonnes méthodes equals)
- assertTrue(a) et assertFalse(a)
  - Testent si a est vrai resp. faux, avec a une expression booléenne.
- assertSame(a,b) et assertNotSame(a,b)
  - Testent si a et b référent au même objet ou non.
- assertNull(a) et assertNotNull(a)
  - Testent si a est null ou non, avec a un objet

Junit 3 en Pratique (4/6)

- les méthodes de test:

```
public void testCalculPerimetre() {
        // {[(12 + 10)*2 = 44]}
        int perimetre = aRectangle.calculPerimetre();

        assertEquals("La methode CalculPerimetre ne
retourne pas la bonne valeur pour largeur = 12 et
longeur  = 10 !!", perimetre, 44);
}
…
```

- caractéristiques:
  - nom préfixé par « test »
  - contient une assertion

```
import junit.framework.TestCase;
import junit.framework.TestSuite;
public class ExempleTest extends TestCase
{
  public ExempleTest(String name) {
    super(name); }
  protected void setUp() {
  exemple = new Exemple(); }
  protected void tearDown() {
  exemple = null; }

public void testMin () {
                exemple.min(3,6)
                assertTrue (exemple.getLastResult() == 3); }
public void testFactoriel() {
                exemple.factoriel(4);
                assertTrue(exemple.getLastResult() == 24); }
private Exemple exemple;
…}
```

```
class Exemple {
protected int lastResult;
public int getLastResult() {
                return lastResult;}
public void factoriel(int n) {
                int Result;
                Result = 1;
                for (int i=1;i<=n;i++)
                                Result = Result * i;
                lastResult = Result;}
public void min(int a, int b) {
                if (a>b) lastResult = b;
                else lastResult = a;}
…}
```

Bonnes pratiques pour JUnit

- Ecrire les test en même temps que le code.
- Tester uniquement ce qui peut vraiment provoquer des erreurs.
- Exécuter ses tests aussi souvent que possible, idéalement après chaque changement de code.
- Ecrire un test pour tout bogue signalé (même s'il est corrigé).
- Ne pas tester plusieurs méthodes dans un même test : JUnit s'arrête à la première erreur.
- Attention, les méthodes privées ne peuvent pas être testées !