INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

Java

benjamin.nguyen@insa-cvl.fr

d’après J.-M. Doudoux

CM2 - Tests unitaires

But du cours (compétences)

- Savoir coder une application complexe en Java (avec des tests unitaires)
- Savoir débugger une application complexe en Java
- Savoir gérer les fichiers en Java
- Savoir gérer la persistence en Java avec la sérialisation
- Savoir utiliser les threads en Java
- Savoir créer une interface graphique simple en Java (approche modèle MVC)

Si le temps le permet …

- Connaître les bibliothèques cryptographiques en Java
- Comprendre et avoir utilisé le mécanisme d'introspection en Java
- Savoir gérer la persistence en Java avec une base de données

Comment vérifier qu'on a bien codé ?

- Compilation (évite un certain nombre d'erreurs)
- Lecture de code, lecture croisée de code
- Création d'un programme "main" de test, exécution, sortie de traces, "print"

→difficulté de maintenabilité du programme de test ad-hoc

→Utilisation d'un framework de tests !

Test logiciel kezako ?

Le test logiciel a de nombreux buts.

- s'assurer du bon fonctionnement du logiciel réalisé
- s'assurer qu'il n'y a pas d'erreurs dans le code qui peuvent conduire à des fautes puis défaillances

Toutes les erreurs logicielles ne sont pas forcément dûes à des erreurs de programmation : certains viennent d'un problème de design !

→ Le test logiciel, en particulier les tests en *white box* cherchent à vérifier l'adéquation entre la spécification et un « oracle »

Niveaux de test

- Code source
- Code machine
- Fonctionnalités
- Application/interface
- …

On considère 3 niveaux

1. Tests unitaires
2. Tests d’intégration
3. Tests de système

Niveaux de test

- Code source
- Code machine
- Fonctionnalités
- Application/interface
- …

On considère 3 niveaux

1. Tests unitaires
2. Tests d’intégration
3. Tests de système

Test unitaire

Définition :

Un test *unitaire* est un programme *indépendant et autonome* qui va tester une fonctionnalité du logiciel (unité) par rapport à son comportement / sorties attendus.

Architecture du test :

- Initialisation (setup) : mise en place de l'environnement nécessaire à la construction et l'exécution des tests.
- Exercice : on exécute le programme à tester
- Vérification (assert) : on compare les résultats de l'exécution du programme à ce qui était attendu
- Désactivation : on se remet dans l'état initial du système (chaque test doit être autonome)

Comment écrire les tests ?

- Un test = une fonctionnalité du programme (et non un scénario complexe)
- Peut (et doit !) être défini avant de coder !

Test Driven Development :

1. Création d'un test
2. Ecriture du code pour passer le test
3. Rajout de tests
4. Modification / refactorisation du code pour passer tous les tests
5. GOTO 3

Intérêts du TDD

Robert C. Martin, *The Cycles of TDD*, 2008 (traduction Wikipedia)

- TDD permet d'éviter des modifications de code sans lien avec le but recherché, car on se focalise à chaque cycle sur la satisfaction d'un besoin précis, en conservant le cap du problème d'ensemble à résoudre.
- TDD permet d'éviter les accidents de parcours, où des tests échouent sans qu'on puisse identifier le changement responsable, ce qui aurait pour effet d'allonger la durée d'un cycle de développement.
- TDD permet de maîtriser le coût des évolutions logicielles au fil du temps, grâce à une conception du code perméable au changement.
- TDD permet de s'approprier plus facilement n'importe quelle partie du code en vue de le faire évoluer, car chaque test ajouté dans la construction du logiciel explique et documente le comportement du logiciel en distillant l'intention des auteurs.
- TDD permet de livrer une nouvelle version d’un logiciel avec un haut niveau de confiance dans la qualité des livrables, confiance justifiée par la couverture et la pertinence des tests à sa construction.

Type de tests : “white box” et “black box”

- Test “white box” si on connait le fonctionnement interne du programme
- Test “black box” si on connait ce que doit retourner un programme mais qu’on ne sait pas comment c’est implémenté.

JUnit avec Eclipse

Junit est un framework open source, utilisé pour écrire et exécuter des tests.

Il propose :

- des annotations (@Test) pour identifier les méthodes à tester
- des *assertions* pour tester les résultats attendus
- des *test runners* pour lancer les tests
- peut être structuré de manière hiérarchique

Les résultats sont retournés de manière simple et synthétique.

Cas de test

- Une entrée connue
- Une sortie attendue
- Bien réfléchir au nombre de tests à exécuter pour chaque fonctionnalité (au moins 2 : un test positif et un test négatif)

Le HelloWorld du JUnit

```java
package cm;
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class HelloJUnit {
   @Test
   public void testAdd() {
      String str = "Junit is working fine";
      assertEquals("Junit is working fine",str);
   }
}
```

On peut directement tester avec Eclipse !

JUnit : le Runner

```
package cm;
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestUnitRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(HelloJUnit.class);
      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }
      System.out.println(result.wasSuccessful());
   }
}
```

Test ou suite de tests

Classes de JUnit

- Assert
- TestCase
- TestResult
- TestSuite

Classe Assert

- **void assertTrue(boolean condition)**
- **void assertFalse(boolean condition)**
- **void assertEquals(boolean expectedArray, boolean returnedArray)**
- **void assertNotNull(Object object)**
- **void assertNull(Object object)**
- **void assertSame(object1, object2)**
- **void assertNotSame (object1, object2)**
- **void fail() → fait échouer le test.**

Classe TestCase

- **int countTestCases() :** compte le nombre de tests
- **void setName(String name)** : donne un nom au test
- **String getName() :** récupère le nom du test
- **void setUp() :** Mise en place de l'environnement du test
- **void tearDown() :** à lancer à la fin du test
- **TestResult createResult()** : Factory pour un objet TestResult
- **TestResult run() :** exécute le test
- **void run(TestResult result) :** exécute le test et récupère les résultats dans result
- **String toString() :** sérialisation en String du TestCase

Classe TestResult

- **void addError(Test test, Throwable t)** : ajout d'une erreur (non prévu)
- **void addFailure(Test test, AssertionFailedError t) :** ajout d'un échec (prévu)
- **void endTest(Test test) :** informe que le test a bien terminé
- **int errorCount()** : compte les erreurs
- **int failureCount()** : compte les échecs
- **Enumeration<TestFailure> errors() :** liste les échecs
- **void startTest(Test test) :** informe qu'on débute le test
- **void stop() :** informe qu'on arrête le test
- **void run(TestCase test) :** exécute le test *test*
- **int runCount() :** donne le nombre de tests lancés

Classe TestSuite

- **void addTest(Test test) :** ajouter un test à la test suite
- **void addTestSuite(Class<? extends TestCase> testClass)** : ajouter tous les tests de la classe testClass à la testSuite
- **int countTestCases()** : compte le nombre de tests
- **int testCount() :** compte le nombre de tests dans la suite
- **void setName() :** renomme la test suite
- **String getName()**: donne le nom de la test suite
- **void run(TestResult result) :** exécute la suite de tests et stocke le résultat dans result
- **Test testAt(int index) :** retourne le Test d'indice index
- **static Test warning(String message) :** retourne un test qui échoue et donne un message d'erreur

Paramètres de @Test

- expected = NullPointerException.class (extends java.lang.Throwable)
- timeout = 100 (en ms)

Les annotations JUnit

- @Test : dit que la méthode public void qui suit peut être lancée comme un cas de test
- @Before : dit que la méthode public void qui suit doit être lancée avant chaque méthode @Test (JUnit4, devenu @BeforeEach en JUnit5)
- @After : dit que la méthode doit être lancée après chaque @Test
- @BeforeClass : pour une méthode public static void (@BeforeAll)
- @AfterClass : pour une méthode public static void
- @Ignore : dit d'ignorer le test (méthode) qui suit

Junit : Suite de test

Une suite de tests (*Test Suite*) regroupe ensemble un certain nombre de tests unitaires et les exécute ensemble.

On utilise les annotations @RunWith et @Suite pour exécuter cette suite de tests

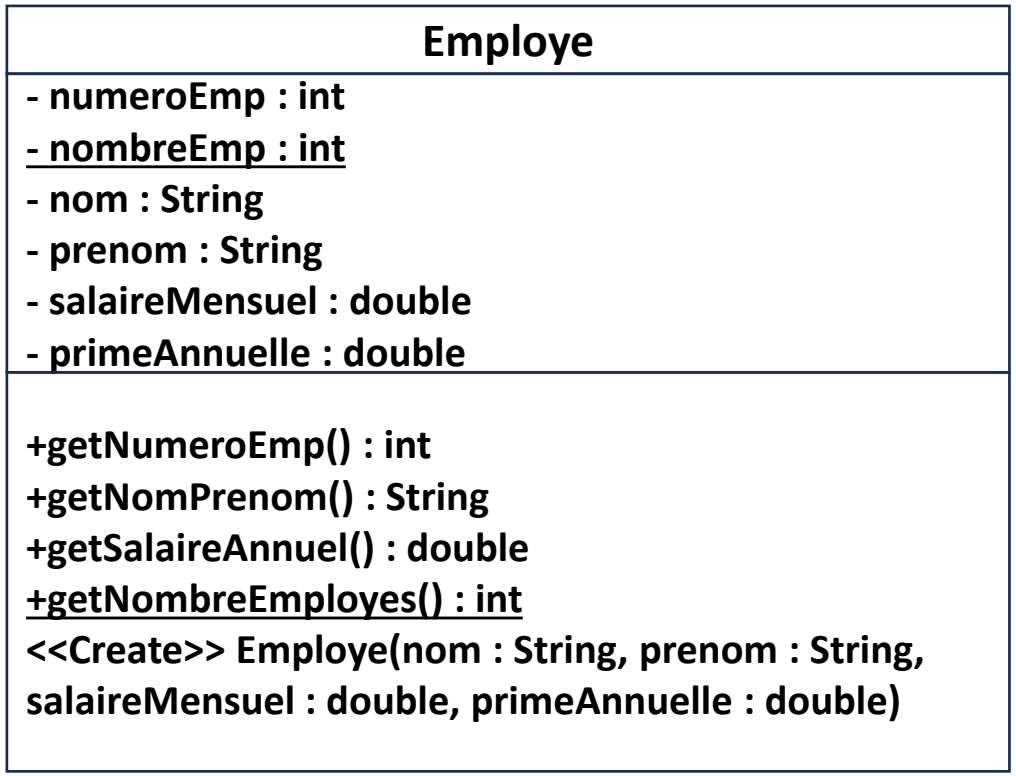
Voir exemples : JUnitTest1, JUnitTest2 et JUnitTestSuiteExemple ainsi que le Runner : TestUnitRunner

Tests paramétrés

- @RunWith(Parametrized.class)

Exemples : TestNombrePremier, JUnitParametrized, JUnitTestNombrePremier

TD : Exemple de classe



- Indications fonctionnelles :
  - Les numéros d'employé partent de 0 et s'incrémente de 1 en 1
  - Le salaire annuel correspond à 12*salaireMensuel + primeAnnuelle
  - La méthode getNomPrenom doit retourner dans cet ordre la chaîne "Nom, Prenom"
- On ne supprime pas d'employés pour le moment

Voir Code