Les tests unitaires automatisés avec JUnit

Un test automatisé est un programme qui se découpe en trois phases dites AAA pour *Arrange*, *Act*, *Assert*.

- Arrange

  La mise en place de l'environnement : création et initialisation des objets nécessaires à l'exécution du test.

- Act

  Le test proprement dit.

- Assert

  La vérification des résultats obtenus par le test.

Le sous-système (l'ensemble des objets) éprouvé par le test est parfois appelé **SUT** (*System Under Test*).

On distingue différentes catégories de tests :

- Tests unitaires : testent une partie (une unité) d'un système afin de s'assurer qu'il fonctionne correctement (*build the system right*)
- Tests d'acceptation : testent le système afin de s'assurer qu'il est conforme aux besoins (*build the right system*)
- Tests d'intégration : testent le système sur une plate-forme proche de la plate-forme cible
- Tests de sécurité : testent que l'application ne contient pas de failles de sécurité connues (injection de code, attaque XSS, ...)
- Tests de robustesse : testent le comportement de l'application au limite des ressources disponibles (mémoire, CPU, ...) sur la plate-forme

JUnit

JUnit est le framework de tests unitaires le plus utilisé en Java.

Structure d'une classe de test JUnit

Comme Java est un langage orienté Objet, les tests JUnit sont regroupés dans des classes de test. Généralement, on groupe dans une classe les tests ayant la même classe comme point d'entrée et on nomme la classe de test à partir du nom de la classe testée préfixé ou suffixé par `Test` en la plaçant dans le même package. Par exemple, pour tester la classe `DateFormatter`, on créera une classe `TestDateFormatter` ou `DateFormatterTest`.

Suffixer ou post-fixer par `Test` le nom de la classe de test est juste une convention. Néanmoins, il est très fortement conseillé de la respecter car les IDE et Maven utilisent également cette convention pour découvrir les classes de test à exécuter.

Exécution des tests JUnit

JUnit est pris en charge par les IDE Java et par les outils de build comme Maven.

Dans Eclipse, il suffit de faire un clic droit dans l'explorateur de projet sur un fichier source, une classe de Test ou un package et de choisir **« Run as... > JUnit Test »**. On peut également presser la touche F11 dans l'éditeur de code source de la classe de test.

Les méthodes de test

Une classe de test est simplement une classe déclarant des méthodes publiques sans paramètre et sans valeur de retour et qui sont annotées par @Test.

Une méthode de test contient :

- un ensemble d'instructions correspondant à la phase *arrange* (si nécessaire),
- un ensemble d'instructions correspondant à la phase *act* (qui se limite généralement à l'appel de la méthode à tester),
- un ensemble d'instructions correspondant à la phase *assert*.

L'exemple ci-dessous teste la méthode String.toUpperCase :

Exemple d'une classe de test

```
package dev.formation;

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

public class StringTest {

  @Test
  public void upperCaseProduitUneChaineEnMajuscules() throws Exception {
    // Bloc arrange
    String s = "Bonjour le monde";

    // Bloc act
    String maj = s.toUpperCase();

    // Bloc assert
    assertEquals("BONJOUR LE MONDE", maj);
  }

}
```

Une méthode de test ne devrait pas contenir d'instruction `if` ou `switch` puisqu'un test traduit un cas d'utilisation simple sans choix possible. De même, une méthode de test ne devrait contenir qu'exceptionnellement des boucles `for` ou `while`.

Les assertions

La classe Assert est une classe outil contenant des méthodes statiques pour déclarer des assertions. Ces méthodes permettent de vérifier la valeur d'un paramètre ou de comparer deux valeurs passées en paramètres. Si l'assertion est fausse, ces méthodes produisent une exception de type `AssertionError` qui fait échouer le test.

Parmi les méthodes d'assertion, on trouve :

Méthode	Utilisation
assertTrue(boolean condition)	Vérifie que la condition passée en paramètre est vraie.
assertFalse(boolean condition)	Vérifie que la condition passée en paramètre est fausse.
assertEquals(Object expected, Object actual)	Compare les deux paramètres pour vérifier qu'ils sont égaux.
assertNotEquals(Object expected, Object actual)	Compare les deux paramètres pour vérifier qu'ils ne sont pas égaux.
assertSame(Object expected, Object actual)	Vérifie que les deux objets passés en paramètre sont en fait le même objet (en utilisant l'opérateur `==`).
assertNotSame(Object expected, Object actual)	Vérifie que les deux objets passés en paramètre ne sont pas les mêmes objets (en utilisant l'opérateur `!=`).
assertNull(Object actual)	Vérifie que l'expression passée en paramètre s'évalue à `null`.
assertNotNull(Object actual)	Vérifie que l'expression passée en paramètre ne s'évalue pas à `null`.
fail()	Il ne s'agit pas vraiment d'une assertion puisqu'un appel à cette méthode fait échouer le test immédiatement. Nous verrons plus bas que cette méthode est utile pour tester les exceptions.

Il existe une surcharge de méthode pour chacune des méthodes précédentes qui accepte une chaîne de caractères comme premier paramètre. Il s'agit du message d'erreur produit par le test si l'assertion échoue.

Note

Pour les méthodes d'assertion qui attendent deux valeurs pour les comparer, notez que la première valeur correspond à la valeur attendue pour ce test et la deuxième valeur correspond à la valeur produite au moment du test.

Exemple d'utilisation des assertions

```
package dev.formation;

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

public class StringTest {

  @Test
  public void testString() throws Exception {
    String s = "Bonjour le monde";
```

```
    assertEquals("Bonjour le monde", s);
    assertNotEquals("Bonsoir le monde", s);
    assertFalse(s.isBlank());
  }

}
```

Exercice - Tests unitaires de java.util.Math#abs(int)

Écrire les tests unitaires pour la méthode java.util.Math#abs(int).

Modèle de projet Maven

Vous pouvez télécharger le modèle de projet Maven :

> exotest.zip

Note

L'exercice précédent propose de tester une méthode sans effet de bord (ce que l'on appelle également une fonction pure). Les tests sur ce type de méthodes sont faciles à écrire. Ils restent cependant l'exception lorsqu'on utilise la programmation orientée objet. En effet, l'appel d'une méthode sur un objet modifie le plus souvent son état et provoque généralement des effets de bord en sollicitant d'autres objets avec lesquels l'objet entretient des dépendances.

Les fixtures

Pour réaliser un test, il est parfois nécessaire un grand nombre d'objets et de préparer le SUT (*System Under Test*). Plutôt que d'écrire le code nécessaire au début d'un test (au risque de le rendre moins lisible), on préfère écrire ce code dans une classe à part ou une méthode à part. Dans ce cas, on qualifie ce nouvel objet ou cette nouvelle méthode de **fixtures**.

Avec JUnit, il est possible d'exécuter des méthodes avant et après chaque test pour allouer et désallouer des ressources nécessaires à l'exécution des tests. On déclare pour cela des méthodes publiques sans paramètre annotées avec @Before ou @After.

Exemple d'une classe de test avec @Before et @After

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class UneClasseTest {

  @Before
  public void initTestEnvironment() {
    // cette méthode est exécutée avant chaque test
  }

  @After
  public void destroyTestEnvironment() {
    // cette méthode est exécutée après chaque test
  }

  @Test
```

```
  public void methodeDeTest() throws Exception {
    // la méthode de test
  }

}
```

Note

Il est également possible de déclarer des méthodes **static** annotées avec @BeforeClass ou @AfterClass. Ces méthodes ne sont appelées qu'une seule fois respectivement avant ou après l'ensemble des méthodes de test de la classe.

Tester les exceptions

Grâce aux tests unitaires, il est également plus facile de tester les cas non nominaux qui se traduisent la plupart du temps par la production d'une exception en Java. Si on désire tester un cas d'erreur par exemple, cela signifie que le test sera ok si une exception précise est produite lors de la phase *act*.

Avec JUnit, il existe plusieurs façons d'écrire un test pour une telle situation. Pour les exemples suivants, nous testerons la méthode Integer#parseInt qui produit une exception de type NumberFormatException lorsqu'une chaîne de caractères qui ne correspond pas à un nombre est passée en paramètre d'appel.

Tester une exception (façon 1)

```
package dev.formation;

import static org.junit.Assert.fail;
import org.junit.Test;

public class IntegerTest {

  @Test
  public void parseIntThrowsExceptionWhenNotANumber() throws Exception {
    try {
     Integer.parseInt("not a number");
     fail("NumberFormatException expected");
    } catch (NumberFormatException e) {
    }
  }

}
```

Dans l'exemple ci-dessus, on utilise une structure `try ... catch` pour attraper l'exception qui est attendue. Dans le bloc `try`, l'appel à la méthode `Assert.fail` pour faire échouer le test si jamais la phase *act* (c'est-à-dire l'appel à Integer#parseInt) n'a pas produit d'exception. Cette façon d'écrire le test est simple mais rend le test parfois difficile à lire à cause de la présence des blocs `try ... catch` et de l'absence d'une phase *assert* remplacée par l'appel à `Assert.fail`.

Tester une exception (façon 2)

```java
package dev.formation;

import org.junit.Test;

public class IntegerTest {

  @Test(expected = NumberFormatException.class)
  public void parseIntThrowsExceptionWhenNotANumber() throws Exception {
    Integer.parseInt("not a number");
  }

}
```

Dans l'exemple ci-dessus, on utilise l'attribut `expected` de l'annotation @Test qui permet d'indiquer que l'on s'attend à ce que le test échoue à cause de la propagation d'une exception (si ce n'est pas le cas, le test sera considéré en échec). Cette façon d'écrire le test est plus simple que précédemment mais elle peut être difficile à comprendre car la phase *assert* n'est pas explicite dans la méthode.

Tester une exception (façon 3)

```java
package dev.formation;

import org.junit.Rule;
import org.junit.Test;
import org.junit.rules.ExpectedException;

public class IntegerTest {

  @Rule
  public ExpectedException expectedException = ExpectedException.none();

  @Test
  public void parseIntThrowsExceptionWhenNotANumber() throws Exception {
    expectedException.expect(NumberFormatException.class);

    Integer.parseInt("not a number");
  }

}
```

Dans l'exemple ci-dessus, on utilise une rule JUnit pour signaler avant la phase *act* que l'on attend une exception. Comme précédemment, cette façon d'écrire peut être difficile à comprendre car la phase *assert* est remplacée par un attendu lors de la phase *arrange*.

Note

Cette approche est dépréciée à partir de JUnit 4.13.

Tester une exception (façon 4)

```java
package dev.formation;

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

```java
import org.junit.Test;

public class IntegerTest {

  @Test
  public void parseIntThrowsExceptionWhenNotANumber() throws Exception {
    assertThrows(NumberFormatException.class, () -> {
      Integer.parseInt("not a number");
    });
  }

}
```

Avec l'introduction des lambdas depuis Java 8, il est plus direct d'encapsuler un appel d'un code produisant une exception dans une fonction anonyme. On utilise pour cela `Assert.assertThrows` en précisant le type de l'exception attendue. Si cette approche est élégante, elle mélange tout de même les codes de la phase *act* et de la phase *assert*.

Utilisation de doublure

Parfois, il est utile de contrôler l'environnement de test d'un objet ou d'une collaboration d'objets. Pour cela, on peut faire appel à des doublures qui vont se substituer lors des tests aux objets réellement utilisés lors de l'exécution de l'application dans un environnement de production.

- Simulateur

  Un simulateur fournit une implémentation alternative d'un sous-système. Un simulateur remplace un sous-système qui n'est pas disponible pour l'environnement de test. Par exemple, on peut remplacer un système de base de données par une implémentation simplifiée en mémoire.

- Fake object

  Un fake object permet de remplacer un sous-système dont il est difficile de garantir le comportement. Le comportement du fake object est défini par le test et est donc déterministe. Par exemple, si un objet dépend des informations retournées par un service Web, il est souhaitable de remplacer pour les tests l'implémentation du client par une implémentation qui retournera une réponse déterminée par le test lui-même.

- Mock object

  Un objet mock est proche d'un fake object sauf qu'un objet mock est également capable de faire des assertions sur les méthodes qui sont appelées et les paramètres qui sont transmis à ces méthodes.

Implémentation d'objet mock avec Mockito

Mockito est un *framework* Java pour faciliter la création d'objets mocks à partir d'une classe ou d'une interface.

La méthode statique `Mockito.mock(Class<?>)` permet de créer une instance d'un mock à partir d'une classe ou d'une interface. L'instance d'objet retournée par cette méthode est instrumentalisée par Mockito. Il est possible d'enregistrer sur ce mock des comportements lors de la phase *arrange* grâce, notamment, à la méthode `Mockito.when(Object)`. Lors de la phase *assert*, il est possible de vérifier que les appels de méthodes programmés sur le mock ont bien été réalisés grâce à la méthode `Mockito.verify(Object)`.

Exemple d'une classe de test utilisant Mockito

```
import static org.junit.Assert.*;
import javax.servlet.http.HttpServletRequest;

import org.junit.Test;
import org.mockito.Mockito;

public class TestWithMockito {

  @Test
  public void testDemoMockito() throws Exception {
    HttpServletRequest mockedRequest = Mockito.mock(HttpServletRequest.class);
    Mockito.when(mockedRequest.getParameter("login")).thenReturn("monlogin");

    String parameterValue = mockedRequest.getParameter("login");

    assertEquals("monlogin", parameterValue);
    Mockito.verify(mockedRequest).getParameter("login");
  }

}
```

Utilisation de Mockito avec import static

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

import javax.servlet.http.HttpServletRequest;

import org.junit.Test;

public class TestWithMockito {

  @Test
  public void testDemoMockito() throws Exception {
    HttpServletRequest mockedRequest = mock(HttpServletRequest.class);
    when(mockedRequest.getParameter("login")).thenReturn("monlogin");

    String parameterValue = mockedRequest.getParameter("login");

    assertEquals("monlogin", parameterValue);
    verify(mockedRequest).getParameter("login");
  }

}
```

Les exemples de code ci-dessus ne sont bien évidemment pas de vrais tests JUnit puisqu'ils se contentent de tester le bon fonctionnement de Mockito.

La documentation de Mockito est accessible ici.

TDD : Test Driven Development

Qu'est-ce qu'un test ?

Un test est simplement un moyen de nous rassurer : est-ce que cela marche ?

Structure d'un test : AAA

- Arrange
- Act
- Assert

Un test sert à maîtriser la complexité : ainsi un test ne doit pas être trop long (à écrire, à lire, à exécuter), il ne doit pas être imprécis (pas de if, de boucle, de « ou »…). Un test ne doit pas être fragile (pas de faux positifs).

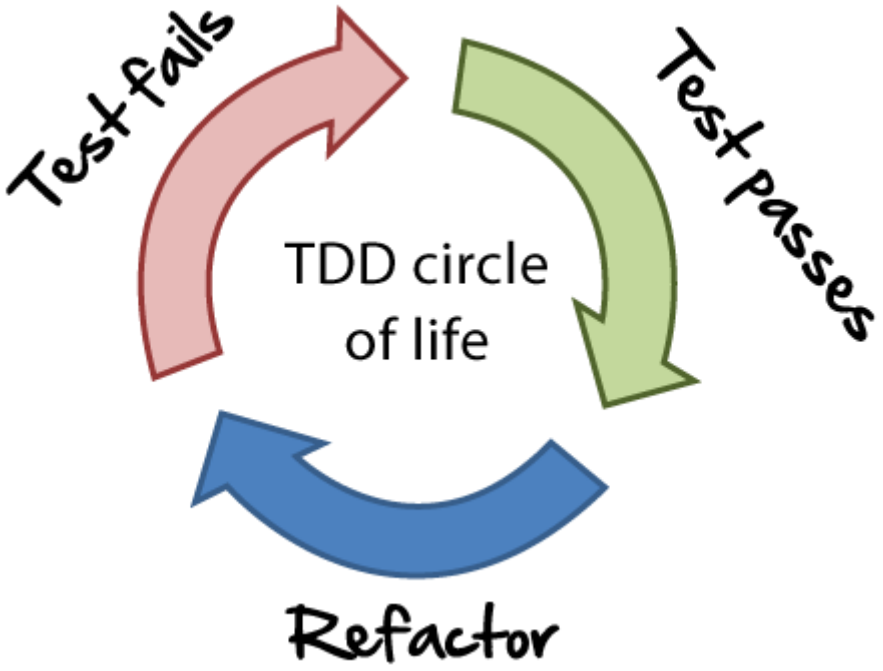
Test Driven Development

Le TDD est une méthode de conception basée sur les tests proposée par Kent Beck.

Parfois, le TDD est une pratique auréolée d'un certain respect, mystère ou, au contraire, qui est complètement dénigrée. En fait, le TDD est une façon particulière d'articuler les pratiques de conception logicielle et d'automatisation des tests.

Les étapes du TDD

1. Écrire un test rouge
2. Écrire le code le plus simple pour que le test soit vert
3. Supprimer la duplication (code/test) et améliorer la lisibilité. C'est la phase du *refactoring*.



Pour la phase 2, tous les coups sont permis. Le plus important est de faire passer le test au vert.

Plus généralement, le TDD est une approche avec laquelle on cherche à cadrer la production de code par petits incréments qui marchent (c'est-à-dire qui sont testés).

Qu’est-ce que concevoir ?

La conception logicielle est un domaine souvent envisagé comme autonome. Pour certains, elle relève d’une expertise particulière (architecture logicielle). Pour d’autres, elle relève de l’esthétique : beau code, application de patterns…

Les règles de la conception simple de Kent Beck :

1. Ça marche (les tests sont verts)
2. Il n’y a pas de duplication de code
3. Le code est expressif
4. Le code contient un nombre minimum de classes et de méthodes

Dans cette vision de la conception, le TDD est bien une pratique de conception pilotée par les tests. Cependant, la conception émerge vraiment à la dernière phase d’un cycle de test au moment du *refactoring*.

Exercices

Exercice à rendre - Le Kata calculette

Réalisez le Kata String calculator.

Astuce

N’essayez pas d’anticiper le développement. Implémentez les différents points les un après les autres. Le but est de réaliser une conception incrémentale en vous basant sur les tests.

Voici la feuille de route (non restrictive) pour ce Kata :

Créez une simple calculatrice avec une méthode :

int add(String numbers)

1. Cette méthode prend en paramètre une chaîne de caractères qui représente 0, 1 ou 2 nombres. Elle retournera leur somme (pour une chaîne de caractères vide, elle retournera 0). Par exemple : `""`, `"1"` ou `"1,2"`
2. Autoriser la méthode `add` à prendre en compte un nombre quelconque de nombres
3. Autoriser la méthode `add` à prendre en compte les retours à la ligne en plus des virgules comme séparateur. Par exemple : `"1\n2,3"` donne 6
4. Par contre `"1,\n2"` n’est pas ok (pas deux séparateurs à la suite)
5. Autoriser la méthode `add` à prendre en compte des séparateurs de un caractère. Pour changer de séparateur, la chaînes de caractères peut commencer par la ligne : `"//[séparateur]\n"`. La première ligne est optionnelle et tous les scénarios précédents restent valides. Par exemple : `"//;\n1;2"` vaut 3
6. Les nombres supérieurs à 1000 sont ignorés. Par exemple `"2,1001"` vaut 2
7. Appeler `add` avec un nombre négatif dans la chaîne entraîne une exception avec le message `"negatives not allowed"` suivi du nombre négatif passé dans la chaîne de caractères. S’il y a plusieurs nombre négatifs, il faut tous les ajouter dans le message de l’exception
8. Le délimiteur peut avoir n’importe quelle longueur. Par exemple : `"//:::\n1:::2:::3"` vaut 6

9. Autoriser plusieurs délimiteurs : `"//*%\n1*2%3"` vaut 6

Modèle de projet Maven

Vous devez télécharger le modèle de projet Maven :

> `tdd.zip`

Exercice de refactoring - Battlecode

Battlecode est un exemple de code Java pour un entraînement au refactoring.

Il va vous falloir faire évoluer le code d'une application. Malheureusement, le code n'est pas toujours aussi clair et *propre* que l'on aimerait.

Heureusement, la couverture de code par les tests est de 100% !

Le code principal tourne autour de la classe `action.Fight` et de sa méthode `execute`

Modèle de projet Maven

Vous pouvez récupérer le projet battlecode comme projet Maven :

> `battlecode.zip`

Exercice à rendre - Jeu *Attrape-moi si tu peux*

Attrape-moi si tu peux est un jeu à deux joueurs. Les joueurs se déplacent sur une grille de taille 10 sur 10.

Les deux joueurs font une action chacun à leur tour. À son tour, un joueur peut faire une des actions suivantes :

- pivoter à gauche de 90° (tourner à gauche)
- pivoter à droite de 90° (tourner à droite)
- avancer d'une ou deux cases

Si un joueur avance en dehors du plateau, son déplacement est annulé.

Une fois que les deux joueurs ont fait leur action, le jeu indique si un joueur voit un autre joueur et à quelle distance il se situe (le nombre de cases). Un joueur ne voit que ce qui se trouve sur la ligne ou la colonne sur laquelle il est orienté. Ensuite, une nouvelle phase de jeu commence et les joueurs peuvent à nouveau choisir une des trois actions.

Pour gagner, un joueur doit se déplacer sur la même case que l'autre joueur.

Essayez de réaliser ce jeu en TDD. Il ne faut pas intégrer des interfaces graphiques ou des sorties sur la console. Contentez-vous de coder le cœur du jeu : les classes qui permettent de gérer une partie et le déplacement des joueurs.

Modèle de projet Maven

Vous devez télécharger le modèle de projet Maven :

> `tdd.zip`