Test Logiciel – TP1

I3 GL

5 octobre 2023

Environnement de développement et prise en main de JUnit

Objectifs

- Mettre en place un environnement de développement basique adapté pour le reste du module
- Être en mesure d'écrire une application Java et de la compiler, d'écrire des tests unitaires pour cette application et de les exécuter

Note : les instructions présentées ici supposent l'utilisation de l'environnement de développement IntelliJ IDEA Community Edition,[1] qu'il est encouragé d'utiliser. Si vous souhaitez utiliser un autre IDE (comme Eclipse), vous pouvez adapter les procédures pour la création d'un projet Java et l'ajout de dépendances. Au moindre blocage, n'hésitez pas à demander de l'aide.

1 Prérequis

On suppose la présence d'un kit de développement Java (JDK) sur les machines. Assurez-vous que la commande `java -version` s'exécute correctement et retourne une valeur $\geqslant 11$. Notez également le chemin vers l'installation du JDK sur votre machine, par exemple en utilisant la commande `which` sous Linux :

```
$ java -version
openjdk version "17.0.8.1" 2023-08-24
$ java -version
java version "11.0.11" 2021-04-20 LTS
$ which java
/usr/jvm/java-11-openjdk/bin/java # le chemin vers le JDK est /usr/jvm/java-11-openjdk
```

2 Une première application Java

Notre première application est une calculatrice simple en Java dans un projet que nous nommerons `tp1-calculator`. Dans IntelliJ IDEA, naviguez à `File -> New -> Project` et sélectionnez Java comme langage et Maven comme système de *build* dans la fenêtre qui s'affiche. Assurez-vous qu'une version du JDK $\geqslant 11$ est sélectionnée. Si ce n'est pas le cas, pointez vers le JDK dont le chemin a été identifié à l'étape précédente.

Une fois validé, un nouveau projet `tp1-calculator` est créé avec l'arborescence suivante :

```
tp1-calculator/
   ↳ src/
     ↳ main/java/
     ↳ test/java
```

1. https://www.jetbrains.com/idea/download/other.html

Par convention, le code de l'application est placé dans le répertoire `src/main/java` et le code des tests associés dans le répertoire `src/test/java`. Dans le répertoire `tp1-calculator/src/main/java`, créez un *package* (sous-répertoire) `tp1` puis une première classe à l'intérieur (`Clic droit -> New -> Java Class`) que vous nommerez `Calculator` et dont le code source est donné en Listing 1. Cette classe offre un unique service à travers sa méthode `add(int, int)` : l'addition de deux entiers. Confirmez que cette classe compile sans encombre dans l'IDE (`Build -> Build Project`), ainsi qu'en ligne de commande (via la commande `mvn compile`).

Listing 1 – Une calculatrice simple

```
package tp1;

public class Calculator {
  public int add(int a, int b) {
    return a + b;
  }
}
```

3 Un premier test JUnit

JUnit [2] est un *framework* pour le test unitaire en Java créé par Kent Beck et Eric Gamma, qui appartient à la famille des *frameworks* xUnit qui existent pour de nombreux langages de programmation (CUnit pour C, PHPUnit pour PHP, ainsi de suite). JUnit est un *framework* très populaire et est parfaitement intégré avec les systèmes de *build* (Apache Maven, Gradle, Ant) ainsi qu'avec les environnements de développement (IntelliJ IDEA, Eclipse, Visual Studio, etc.) de l'écosystème Java.

Afin de pouvoir l'utiliser, on devra d'abord l'inclure en dépendance de notre projet. Pour cela, on ouvrira le fichier `tp1-calculator/pom.xml` afin d'y include une dépendance vers JUnit, puis on rafraîchira notre projet : `Clic droit sur pom.xml -> Maven -> Reload project`.

Listing 2 – Une dépendance Maven vers JUnit

```
<?xml version="1.0" encoding="UTF-8"?>
<project>
    [...]
    <dependencies>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter</artifactId>
            <version>5.6.0</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

Un cas de test en JUnit est défini par une méthode décorée de l'annotation `@Test`. Les différents cas de test pour une unité particulière (par exemple, une classe) sont généralement regroupés dans une même classe de test dédiée. Par convention, notre classe `tp1.Calculator` aura ainsi ses cas de tests définis dans une autre classe `tp1.CalculatorTest`. Par convention, cette dernière ainsi que tous les autres cas de tests seront situés dans le répertoire `src/test/java`.

Notre premier cas de test pour la classe `Calculator` est donné en Listing 3. Pour le moment, nous supposerons qu'un cas de test suit toujours une structure bien définie en trois étapes :

— Dans une première étape, *Setup*, on initialise le système ou l'unité sous test (ici, notre classe `Calculator`) ainsi que les données d'entrée ;

2. https://junit.org

— Dans une seconde étape, *Exercise*, on interagit avec l'élément que l'on souhaite tester (ici, notre méthode `add(int, int)`;
— Dans une troisième étape, *Verify*, on valide que les résultats obtenus sont conformes aux attentes. Pour cela, on écrit une ou plusieurs *assertions* qui vérifient les résultats obtenus (ici, que la valeur 3 est bien retournée lorsque l'on additionne les entiers 1 et 2). Pour cela, on s'appuie sur le *framework* JUnit qui propose un ensemble d'assertions standards. [3] Si au moins l'une des assertions échoue, alors le cas de test échoue.

Listing 3 – Un premier test JUnit

```java
package tp1;

import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;

public class CalculatorTest {
  @Test
  void add_two_simple_integers() {
    // Setup
    Calculator calc = new Calculator();
    int x = 1;
    int y = 2;

    // Exercise
    int result = calc.add(x, y);

    // Verify
    Assertions.assertEquals(3, result);
  }
}
```

JUnit dispose d'un harnais de test qui permet d'exécuter automatiquement les tests d'un projet. Chaque cas de test est exécuté independamment des autres et leurs résultats sont collectés. Afin d'exécuter notre cas de test, cliquez droit sur la racine du projet `tp1-calculator` puis `Run 'All Tests'`. Une nouvelle vue s'affiche en bas de l'environnement et les cas de test s'exécutent. Pour chaque test, l'interface indique son nom, le temps d'exécution associé, ainsi que son statut (*succès*, *échoué*, ou *erreur*). En cliquant sur un cas de test, on accède aux *logs* produits lors de son exécution dans la console. Confirmez que le cas de test passe avec succès, puis modifiez les valeurs d'entrée pour produire un résultat incorrect. Confirmez que le cas de test échoue avec ces nouvelles valeurs. En ligne de commande, utilisez la commande `mvn test` et confirmez que les résultats sont cohérents avec ceux obtenus dans l'environnement de développement.

4 Poursuivre le test de la calculatrice

Continuez d'ajouter des fonctions de calcul (`subtract(int, int)` et `multiply(int, int)`, par exemple) et écrivez les cas de test correspondants. Vous pourrez vous attaquer à la méthode `divide(int, int)` qui présente des comportements intéressants à tester. Lesquels ? Comment les tests de cette méthode peuvent-ils les mettre en valeur ? L'assertion `assertThrows` pourra se révéler utile.

3. https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html