Chapitre 8

Documentation, tests unitaires et gestion des dépendances

Thibaud Martinez
thibaud.martinez@dauphine.psl.eu

Dauphine | PSL
UNIVERSITÉ PARIS

Documentation

- Explique ce que fait le code, comment l'utiliser et pourquoi il est écrit comme il l'est.
- Incontournable pour la **collaboration** et pour que le projet soit **maintenable**.
- Conserver la documentation proche du code documenté, ou même directement dans le code, facilite les mises à jour.

Javadoc

Le JDK contient un outil très utile, appelé `javadoc`, qui génère une **documentation HTML** à partir des fichiers sources.

En ajoutant au code source des commentaires spéciaux `/**`, on peut facilement générer une documentation avec `javadoc`.

Documenter une classe

```java
/**
* Un objet {@code Carte} représente une carte à jouer, telle que
* "Reine de cœur". Une carte a une couleur (Carreau, Coeur,
* Pique ou Trèfle) et une valeur (1 = As, 2 . . . 10, 11 = Valet,
* 12 = Reine, 13 = Roi)
*/
public class Carte {
    // ...
}
```

Documenter une méthode

```java
/**
* Augmente le salaire d'un employé.
* @param pourcentage le pourcentage d'augmentation du salaire (par exemple, 0.1 signifie 10 %).
* @return le montant de l'augmentation
*/
public double augmenteSalaire(double pourcentage) {
       double augmentation = salaire * pourcentage;
       salaire += augmentation;
       return salaire;
}
```

Documenter des variables

Il n'est nécessaire de documenter **que les membres publics**, ce qui signifie généralement des constantes statiques.

```
/**
* La couleur "Coeur"
*/
public static final int HEARTS = 1;
```

Générer la documentation

En supposant que les classes sont toutes dans le dossier `src/`. La commande suivante génèrera la documentation pour tout le code source et stockera le résultat dans un dossier `doc`.

```
javadoc -d doc src/*
```

Test unitaires

Les **tests unitaires** sont une méthode de test de logiciels par laquelle des **unités individuelles** de code source telles que des méthodes sont testées pour déterminer si elles fonctionnent comme attendues.

→ On teste des unités de code **en isolation**.

→ Il existe d'autres types de tests : tests d'intégration, tests système...

JUnit

JUnit est un framework pour écrire et exécuter des tests unitaires en Java. Il est actuellement à la **version 5**.

Il comprend 3 sous-projets :

- **JUnit Platform** : sert de base au lancement de tests sur la JVM.
- **JUnit Jupiter** : fournit des utilitaires pour écrires des tests unitaires.
- **JUnit Vintage** : pour lancer des tests correspondants à des versions antérieurs de JUnit.

Utilisation de JUnit

Supposons qu'on souhaite tester la classe suivante :

```
package exemple;

public class Calculatrice {
    public static int addition(int a, int b) {
        return a + b;
    }

    public static int division(int a, int b) {
        return a / b;
    }
}
```

On écrit le test unitaire suivant :

```
import static org.junit.jupiter.api.Assertions.assertEquals;
import exemple.Calculator;
import org.junit.jupiter.api.Test;

class CalculatriceTests {
    private final Calculatrice calc = new Calculatrice();

    @Test
    void addition() {
        assertEquals(2, calc.addition(1, 1));
    }
}
```

Définir des tests avec JUnit

- Généralement, une classe de tests unitaires est associée à une classe;
- on ajoute une annotation `@Test` pour indiquer que la méthode est un test;
- le nom de la méthode est libre;
- la méthode doit renvoyer `void` et ne prends pas de paramètres.

Assertions

Les assertions sont des méthodes utilitaires qui permettent de déclarer que des conditions doivent être remplies pour que le test soit réussi.

- assertTrue
- assertFalse
- assertNull
- assertNotNull
- assertEquals
- ...

Faire échouer un test

On peut utiliser l'assertion `fail` pour faire échouer un test.

```java
import static org.junit.jupiter.api.Assertions.fail;
import exemple.Calculatrice;
import org.junit.jupiter.api.Test;

class CalculatriceTests {
    private final Calculatrice calc = new Calculatrice();

    @Test
    void addition() {
        int res = calc.addition(1, 1);
        if (res < 0) {
            fail("Un nombre négatif n'est pas attendu");
        }
    }
}
```

Avant et après les tests

On peut exécuter des instructions avant et après les tests grâce à `@BeforeAll`, `@BeforeEach`, `@AfterEach`, `@AfterAll`.

```java
import org.junit.jupiter.api.*;

class StandardTests {
    @BeforeAll
    static void initAll() {}

    @BeforeEach
    void init() {}

    @AfterEach
    void tearDown() {}

    @AfterAll
    static void tearDownAll() {}
}
```

Tester les exceptions

```java
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.assertThrows;
import exemple.Calculatrice;
import org.junit.jupiter.api.Test;

class CalculatriceTests {
    private final Calculatrice calc = new Calculatrice();

    @Test
    void testException() {
        Exception exception = assertThrows(ArithmeticException.class, () ->
            calc.division(1, 0));
        assertEquals("/ by zero", exception.getMessage());
    }
}
```

Exécuter des tests JUnit

On utilise le *Console Launcher* pour lancer les tests JUnit depuis la console.

La dernière version peut être obtenue depuis Maven Central.

En supposant qu'on travaille avec l'arborescence de fichiers suivante :

```
projet
└── exemple
    ├── Calculatrice.java
    └── CalculatriceTests.java
```

On peut lancer les tests avec :

```
javac -cp '.:junit-platform-console-standalone-1.9.0.jar' exemple/*.java
java -jar junit-platform-console-standalone-1.9.0.jar -cp . --scan-classpath
```

Distribuer et récupérer des logiciels en Java

Archives JAR

Les logiciels Java sont très souvent distribués sous forme d'**archives JAR** (*Java Archive*).

→ Un fichier JAR peut contenir à la fois des fichiers de classe et d'autres types de fichiers tels que des fichiers image et son. De plus, les fichiers JAR sont compressés, en utilisant le format de compression ZIP bien connu.

Créer des archives JAR

On utilise l'outil `jar` pour créer des fichiers JAR.

```
jar cvf <jarFileName> <file1> <file2> . . .
```

Par exemple :

```
jar cvf ClassesCalculatrice.jar *.class icon.gif
```

Créer des archives JAR exécutables

On peut utiliser l'option `e` de la commande `jar` pour spécifier le point d'entrée du programme, c'est-à-dire la classe qu'on spécifie normalement lorsqu'on appelle le lanceur de programme java.

```
jar cvfe MonProgramme.jar monpackage.MonProgramme <fichiers à ajouter>
```

On peut ensuite démarrer le programme :

```
java –jar MonProgramme.jar
```

Utiliser des bibliothèques externes sous forme de JAR

On peut vouloir utiliser une bibliothèque distribuée en JAR. Pour pouvoir utiliser les classes présentes dans l'archive, il faut l'ajouter au *classpath*.

ℹ️ Le *classpath* est le chemin à partir duquel l'environnement d'exécution Java recherche les classes et autres fichiers de ressources.

```
javac -cp '.:org.exemple.jar' MaClasse.java
java -cp '.:org.exemple.jar' MaClasse
```

Maven

Maven est un outil pour construire et gérer des projets en Java.

Il permet notamment de **gérer les dépendances du projets**, c'est-à-dire d'importer et d'utiliser des bibliothèques externes. Il simplifie aussi l'**exécution de tests JUnit**.

✨ Maven propose une structure de répertoires particulière. Il est suggéré de respecter cette structure pour organiser vos projets.

Créer un projet Maven

Une fois Maven installé, on peut créer un nouveau projet :

```
mvn archetype:generate \
    -DgroupId=com.mycompany.app \
    -DartifactId=my-app \
    -DarchetypeArtifactId=maven-archetype-quickstart \
    -DarchetypeVersion=1.4 \
    -DinteractiveMode=false
```

Le fichier `pom.xml` (pour *Project Object Model*) contient les informations sur le projet et les détails de configuration utilisés par Maven pour construire le projet.

Ajouter des dépendances

Pour ajouter des bibliothèques à un projet, on ajoutera les informations sur celles-ci entre les balises `<dependencies></dependencies>` dans `pom.xml`.

Par exemple, pour ajouter JUnit 5:

```
<dependencies>
<dependency>
     <groupId>org.junit.jupiter</groupId>
     <artifactId>junit-jupiter-engine</artifactId>
     <version>5.9.0</version>
     <scope>test</scope>
</dependency>
<dependency>
     <groupId>org.junit.platform</groupId>
     <artifactId>junit-platform-runner</artifactId>
     <version> 1.9.0</version>
     <scope>test</scope>
</dependencies>
```

ℹ️ Les autres bibliothèques disponibles sont listées sur Maven Central Repository.

Compiler le code

```
mvn compile
```

Le résultat de la compilation se trouve dans le répertoire `target/`.

Exécuter les tests

```
mvn test
```