Développement dirigé par modèles

Travaux Pratiques

Gradle

Tests unitaires & Mock Testing

Année Académique 2023-2024

Enseignant: Stéphane **Dupont**
Assistant: Guillaume **Cardoen**

Département d'Informatique
Faculté des Sciences
Université de Mons

30 novembre 2023

Table des matières

1 Gradle

1.1 Introduction à Gradle

Gradle est un outil permettant d'automatiser certaines tâches lors du développement de vos applications. En effet, lorsque vous souhaitez réaliser un *.jar* exécutable, il est souvent nécessaire de d'abord compiler votre projet Java et ensuite de créer le fichier *.jar*. Grâce à Gradle, il est possible d'automatiser ces deux processus derrière une seule commande à exécuter. Il est également possible d'inclure des commandes dans ces tâches qui permettent d'ajouter de nombreuses fonctionnalités, tel que déplacer des ressources de votre application dans les bons répertoires pour la mise en production, générer la documentation Java de votre application, etc. Gradle permet également d'importer des librairies externes dans votre application et de gérer les éventuelles dépendances pour le bon fonctionnement de celles-ci.

Nous utiliserons Gradle dans ce TP pour exécuter les applications et les tests unitaires, ainsi que pour importer JavaFX dans le projet.

1.2 Installer Gradle

Pour installer Gradle, commencez par installer le *Java Development Kit* (JDK). Pour ce faire, installez le JDK correspondant à votre système. Une fois le JDK installé, testez votre installation en ouvrant un terminal et en tapant les commandes java –version, puis javac –version. Les deux commandes doivent vous donner une version de Java.

Suivez ensuite les instructions sur le site officiel pour installer Gradle (`https://gradle.org/install/`). Entrez alors la commande gradle -v. Gradle vous affichera alors un message de bienvenue ainsi que différentes versions pour différents langage.

1.3 Création d'un projet Gradle

Ouvrez un terminal et utilisez la commande cd pour vous déplacer dans le dossier où vous souhaitez créer le projet. Créer un dossier à cet emplacement (via votre explorateur de fichiers ou avec la commande mkdir) et placez votre terminal à l'intérieur de celui-ci. Dans ce protocole, nous supposons que le projet se trouvera dans le répertoire `D:\\myproject`.

Utilisez alors la commande gradle init pour créer votre projet Gradle. Choisissez le type application et gardez les autres options par défaut.

Ouvrez ensuite le fichier build.gradle situé dans le dossier app et ajoutez entre les sections plugins et repositories les lignes de code suivantes :

```
apply plugin: 'java';
apply plugin: 'eclipse'
apply plugin: 'idea'
```

Le code source du projet se trouve dans le dossier `app/src/main/java` et le code des tests unitaires est dans `app/src/test/java`.

1.4 Utiliser Gradle

1.4.1 En ligne de commande

Pour exécuter votre application en ligne de commande, utilisez la commande gradlew run. Un message indiquant BUILD SUCCESSFUL devrait s'afficher si tout est correct. De même, pour exécuter les tests unitaires, utilisez la commande gradlew test.

Notez qu'on utilise ici la commande gradlew et non plus gradle ! gradlew est un wrapper généré par Gradle. Il sera introduit dans la Section 1.7.

1.5 Rapport de testing

Lorsque vous exécutez les tests unitaires, un rapport au format HTML est généré par Gradle. Ce rapport se trouve dans `app\\build\\reports\\tests\\test`. Ouvrez le fichier index.html dans votre navigateur pour visualiser les informations concernant les tests qui ont réussi ou échoué.

1.6 Ajouter Mockito à Gradle

Si vous souhaitez développer des test unitaires qui utilisent des doublures d'objets avec Mockito, vous pouvez lire cette section. Si pas, passez à la section suivante.

Modifiez la section dependencies de votre *build.gradle* pour y ajouter le support de Mockito :

```
dependencies {
    // Some other things

    // Use JUnit Jupiter for testing.
    testImplementation('org.junit.jupiter:junit-jupiter:5.9.3')

    // Add mockito to the dependencies
    // You can change the version to a newer one
    implementation('org.mockito:mockito-core:5.7.0')
}
```

1.7 Le wrapper : Gradle sans installation

Si le projet doit être compilé sur une machine sans Gradle ou sur laquelle la version disponible est trop vieille, Gradle permet de générer un *wrapper*. Il s'agit d'un script se comportant comme Gradle mais qui, lors de la première utilisation, va télécharger la version de Gradle à utiliser.

1.7.1 Générer le wrapper

Pour générer le wrapper, vous devez ouvrir le dossier du projet dans l'invite de commande et utiliser la commande ci-dessous.

```
gradle wrapper --gradle-version 7.4.1
```

Pour une autre version de Gradle, remplacez le `7.4.1` par la version souhaitée. Cette commande va générer les deux fichiers `gradlew` et `gradlew.bat`. Le premier est le wrapper à utiliser sous GNU/Linux et MacOS, le second est à utliser sous Windows.

1.7.2 Utiliser le wrapper

Pour utiliser le wrapper Gradle, il suffit de remplacer la commande `gradle` par `./gradlew` sous MacOS ou GNU/Linux et par `gradlew.bat` sous Windows comme illustré ci-dessous.

```
// Remplacer...
gradle run
// par...
```

```
./gradlew run      //sous GNU/Linux et MacOS
gradlew.bat run    //sous Windows
```

Dans un IDE, le wrapper Gradle est normalement utilisé par défaut.

1.7.3 Erreur : permission non-accordée

Sous MacOS ou GNU/Linux, il est possible que vous ayez une erreur de permission lorsque vous essayez d'exécuter le wrapper. Cette erreur vient simplement du fait que le fichier gradlew n'a pas la permission de s'exécuter sur votre machine. Pour résoudre ce problème, il suffit d'exécuter la commande ci-dessous en invite de commande.

```
chmod +x gradlew
```

2 Les tests unitaires en Java

2.1 Introduction

L'objectif de ce TP est de vous aider à prendre la main avec les tests unitaires. Vous serez amenés, au cours de trois exercices différents, à comprendre des tests unitaires, ainsi que prédire leurs résultats. Il vous sera également demandé d'implémenter des tests pour des applications déjà codées.

Tous les codes utiles à la séance se trouvent sur Moodle. Toutefois, veuillez prendre en compte les consignes suivantes :

— Lorsque l'on vous demande de prédire les résultats des tests, faites l'exercice **avant** d'exécuter les tests.

— Lorsque l'on vous demande de modifier un test ou d'en créer un nouveau, celui-ci doit être fonctionnel **sans toucher au code de l'application testée** lorsque cela n'est pas demandé par l'énoncé.

Nous vous encourageons, pour votre culture et comme exercice supplémentaire, à lire la Section 1 afin de savoir comment créer et configurer votre projet Gradle personnel. Cependant, afin d'éviter à certains d'entre-vous de perdre du temps dans l'installation de Gradle, un projet `tp1` pré-configuré est disponible sur Moodle. Celui-ci ne nécessitent pas l'installation de Gradle sur votre machine. Nous vous encourageons à utiliser les lignes de commande (Section 1.4.1) pour exécuter Gradle afin de vous familiariser avec celles-ci. Cependant, la plupart des IDE permettent d'abstraire ces étapes.

2.2 Exercices

Exercice 2.1. [Question examen janvier 2010]

Soit l'équation du second degré suivante : $ax^2+bx+c=0$. On crée une classe `SolveEquation` chargée de determiner les valeurs réelles de x telles que l'équation est vérifiée, en fonction des paramètres a, b et c. Les valeurs complexes représentant une solution de l'équation ne sont pas considérées. En fonction des paramètres a, b et c, il peut y avoir 0, 1, 2, ou une infinité de solutions pour x. La méthode `getNbSolutions` retourne ce nombre de solutions. Le cas $a = b = c = 0$ pour lequel il existe une infinité de solutions pour x n'est pas traité par cette méthode, qui lève une exception de type `ArithmeticException` lorsque cette situation se présente.

Les tests unitaires de la classe SolveEquationTest servent à tester la classe `SolveEquation`.

1. Prédisez le résultat (succès, échec, erreur) pour les trois tests `test1()`, `test2()` et `test3()`. Pour chacun des résultats, expliquez pourquoi le résultat se produit.
2. Corrigez ces tests afin qu'aucun d'entre-eux ne soient en échec ou en erreur.
3. Écrivez un test unitaire `test4()` qui vérifie que l'équation $3x^2 = 0$ possède une et une seule solution.
4. Proposez une nouvelle version de `test1()` qui évite d'appeler la méthode `getSolutions()` plusieurs fois, en utilisant la méthode `Pair.equals(Object)`.

Exercice 2.2. Nous souhaitons implémenter une méthode `sqrt` en Java pour calculer la racine carrée d'une valeur double. La méthode de Newton consiste à partir d'une valeur de départ et de l'améliorer itérativement. Supposons vouloir calculer $x = \sqrt{a}$ et être à l'étape n, la valeur améliorée sera alors

$$x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n})$$

Nous continuons d'améliorer la valeur tant que $|a - x_{n+1}^2| > \epsilon$ où ϵ est une valeur assez petite (10^{-8} par exemple).

En se basant sur la spécification mathématique de la fonction racine carrée, vous devriez fournir au moins les tests suivants :

Tests de sortie : la valeur retournée par sqrt doit être $x \geq 0$

Tests de précision : la valeur retournée doit être égale à la valeur réelle de la racine carrée avec une approximation de ϵ (une valeur très petite que vous pouvez définir comme constante dans votre programme Java).

Tests d'entrées valides : $\forall x \in]0, 1[: x < \sqrt{x} < 1$

Tests d'entrées valides : $\forall x \in]1, +\infty[: 1 < \sqrt{x} < x$

Tests de borne : $\sqrt{0} = 0$ et $\sqrt{1} = 1$

Test avec une très grande valeur : par exemple 10^{15}. Il faut s'assurer dans ce cas que (i) on peut trouver un résultat avec une approximation suffisante (ii) le calcul du résultat ne prend pas trop de temps.

Test d'entrée incorrecte : passer une valeur de x négative à l'algorithme donne une erreur du type IllegalArgumentException.

Utilisez le processus de développement dirigé par les tests (test-driven development, "test twice - code once"), en appliquant les actions suivantes dans l'ordre suivant :

1. Implémentez les tests avant de commencer à implémenter le code source Java pour le problème donné.
2. Lancez les tests pour vous assurer qu'ils sont bien définis et qu'ils échouent tous.
3. Implémentez la fonction de racine carrée en utilisant la méthode de Newton (avec une approche itérative ou récursive).
4. Lancez les tests sur votre implémentation de la méthode de Newton, et continuez à corriger cette implémentation jusqu'à ce que tous les tests réussissent. Ajoutez ou corrigez des tests si besoin.

Exercice 2.3. [Question examen août 2014]

La suite de Fibonacci est une suite d'entiers très connue dans laquelle chaque entier est la somme des deux entiers précédents. Voici les dix premiers termes de cette suite :

```
0 1 1 2 3 5 8 13 21 34 ...
```

La méthode itérative de la classe Fibonacci permet de calculer la valeur du $n^{\text{ème}}$ élément de cette liste, en comptant à partir de 0.

Nous disposons des tests unitaires de la classe FibonacciTest pour tester cette méthode itérative.

1. Prédisez le résultat de chacun des tests (succès, échec ou erreur) et expliquez pourquoi ce résultat se produit.
2. Pour les tests qui produisent une erreur ou failure, corrigez-les pour qu'ils réussissent (tout en restant des tests utiles).
3. Écrivez un quatrième test unitaire test4() vérifiant que la méthode fibo est capable de calculer le 1000ème élément de la suite de Fibonacci, en moins d'une seconde.

3 Mock testing

Nous travaillons avec une application permettant de vendre des composants informatiques, que nous appelons des produits (Product). Chaque instance d'un produit est identifiée par un *ID* et est caractérisée par le *nom* du produit acheté ainsi que le *nombre d'exemplaires* du produit souhaité par le client.

L'application ShoppingCart doit pouvoir contenir une liste des produits sélectionnés ainsi qu'une fonction getCartValue qui calcule le montant total du panier de l'utilisateur (sur base des produits ajoutés dans la liste du ShoppingCart).

La société mère de notre magasin met à disposition une API web avec laquelle il est possible d'interagir via une classe Java ProductService. Cette classe dispose d'une fonction getPrice qui prend en paramètre un produit et est supposée retourner le prix de ce produit. Une méthode getDiscount est également présente. Elle prend en paramètre le montant total d'un achat et retourne une éventuelle réduction sous la forme d'un nombre décimal entre 0 et 1. Par exemple, une réduction de 5% sera représentée par 0.05.

Si vous le souhaitez, vous pouvez suivre le protocole Gradle pour créer un nouveau projet Gradle. Toutefois, avant d'utiliser la commande gradlew build ..., modifiez la section dependencies de votre fichier build.gradle pour y ajouter le support de Mockito. Un exemple de modification peut être trouvé au Listing 1.

Listing 1 – Exemple de la section dependencies de Gradle avec Mockito

```
dependencies {
    // Whatever is already there

    // Add mockito as a dependency
    testImplementation('org.mockito:mockito-core:5.7.0')
    // Allow to use mockito with junit jupiter
    testImplementation("org.mockito:mockito-junit-jupiter:5.7.0")
}
```

Exercice 3.1. (a) Proposez une implémentation d'un test unitaire en utilisant JUnit et Mockito pour vous assurer que la fonction getCartValue est correctement implémentée. Dans ce test, vous devez simuler le scénario où le client a acheté 2 RTX 4090 à 1500 euros l'unité (ou un prix réaliste) et 4 GTX 1650 à 290,50 euros l'unité.

(b) Créez ensuite un second test unitaire pour considérer qu'une remise de 10% est appliquée au panier du client. Modifiez la fonction getCartValue pour tenir compte de cette remise.