Autour du test

Attention : Version d'Eclipse à utiliser : Eclipse JEE. Cette version d'eclipse contient déjà gradle installé, donc inutile d'installer le greffon gradle.

Exercice 1 sdkman! est un outil qui permet de gérer les multiples versions de kit de développement logiciel (Software Development Kit). Par exemple, il permet d'installer plusieurs version du SDK Java et de passer de l'une à l'autre.

Nous allons l'utiliser pour installer gradle.

1.1. *Installation de sdkman.* Il suffit de faire dans un terminal :

```
export SDKMAN_DIR="$HOME/nosave/sdkman" && curl -s "https://get.sdkman.io" | bash
```

Dans un nouveau terminal, on aura alors accès à la commande sdk. On peut vérifier sa bonne installation en faisant `sdk version`

1.2. *Installation de gradle.* Il suffit de faire :

```
sdk install gradle 6.8.3
```

On peut alors vérifier la bonne installation en faisant :

```
gradle --version
```

Exercice 2 : Gradle

Gradle est un outil fonctionnant sur la plateforme Java. Son objectif est d'automatiser la construction de projets en gérant les activités associées comme l'exécution des tests unitaires, la production de la documentation, la production d'archives, etc.

En particulier, il permet grâce à un DSL écrit en groovy (ou en kotlin) de définir dans une syntaxe légère (par rapport à Ant ou Maven qui s'appuient sur les fichiers de configuration en XML) la configuration du projet, y compris les bibliothèques requises qui sont alors téléchargées automatiquement depuis des serveurs d'archives...

2.1. *Créer un projet avec gradle.* On veut créer un projet Gradle pour une *application java*, en utilisant le langage groovy comme langage de configuration et junit comme canevas de test. Plutôt que de rentrer interactivement toutes ces informations, faire un copier/coller des lignes suivantes :

```
mkdir TPtest
cd TPtest
gradle init --project-name TPtest --package fr.n7.gls.test \
        --type java-application --dsl groovy --test-framework junit
```

2.2. Pour vérifier que projet est correctement créé, faire :

```
./gradlew run
```

2.3. Récupérer l'archive du TP : TPtest.tar et faire :

```
cd ..
tar xvf TPtest.tar
cd TPtest
./gradlew test
```

Exercice 3 : Intégration de Gradle dans Eclipse

Voyons comment importer un projet Gradle existant sous Eclipse.

3.1. *Installer le greffon pour Gradle.*

Cette question n'est pas à faire car `gradle` est déjà installé sur l'eclipse JEE des stations ubuntu de l'ENSEEIHT.

Sous Eclipse, faire *Help* > *Install New Software...* [1]. Dans la boîte de dialogue, renseigner le champ « *Work with :* » avec le dépôt d'Eclipse (il est de la forme

```
Oxygen - http://download.eclipse.org/releases/oxygen/
```

Oxygen qui dépend de la version d'Eclipse utilisée). La liste des greffons disponibles apparait.

Dans la zone de saisie juste en dessous (*type filter text*), taper *gradle* et le greffon *Buildship : Eclipse Plug-ins for Gradle* apparait seul. Le cocher et faire *Next*, Eclipse calcule les dépendances et affiche la liste des greffons à installer. Faire *Next* pour accéder à l'écran suivant. Lire la licence et l'accepter (ou la refuser, l'installation sera alors annulée).

3.2. *Importer un projet existant.* Il suffit de faire *Import...*, sélectionner *Existing Gradle Project* et suivre les instructions. En particulier, le *Project root directory* doit être le dossier racine du projet Gradle à importer.

3.3. *Ouvrir la vue des tâches Gradle.* Faire *Window* > *Show View* > *Others...* et sélectionner *Gradle Tasks*. Cette vue présente sous forme arborescentes les tâches disponibles sous Gradle. Elle peuvent être lancées avec un double-click ou un click droit.

Exercice 4 : Utilisation d'un outil de couverture de code par les tests

Un outil de couverture de code par les tests peut être utilisé pour identifier des « morceaux » de code qui n'ont pas été testés. Nous allons utiliser la bibliothèque JaCoCo.

Nous prenons comme exemple la classe `Statistiques` qui contient une méthode `statistiques`. Cette dernière prend en paramètre un « itérable » pour lequel elle calcule la plus petite valeur, la plus grande, le nombre d'occurrences de la plus petite valeur et le nombre d'occurrences de la plus grande valeur. Lire la documentation de cette méthode pour avoir plus de détails. On peut engendrer la documentation en lançant la tâche gradle `javadoc`. La documentation est engendrée dans `build/docs/javadoc`.

La méthode `statistiques` a été testée avec `StatistiquesTest`, classe de test bien incomplète !

1. on aurait pu aussi passer par le *Eclipse Marketplace* toujours dans le menu *Help*.

4.1. Quel est l'objectif de la démarche de test (et des programmes de test) ?
4.2. Définir, sur papier, des cas de tests pour la méthode `statistiques`.
4.3. Le fichier de configuration de gradle fait apparaître jacoco dans les greffons (plugins). Quand les tests seront lancées (par exemple `./gradlew test` dans un terminal), l'analyse de couverture sera lancée.

Pour produire un rapport du calcul de couverture, on peut faire `./gradlew jacocoTestReport` qui engendre les résultats dans le dossier `build/reports/jacoco/test/html`. Le fichier `index.html` est le fichier à ouvrir dans un navigateur.

Lancer les tests et consulter le rapport de couverture.
4.4. Compléter les tests pour atteindre un taux de couverture de 100% pour les instructions et les décisions. Attention, il ne faut pas modifier les tests existants mais ajouter de nouveaux tests !
4.5. Que peut-on conclure quand on a atteint un taux de couverture de 100% en instructions et décisions avec des tests qui réussissent ?
4.6. Est-ce que tous les cas de tests identifiés en 4.2 sont couverts par les tests effectués ? Dans la négative, ajouter des tests pour les couvrir.

Exercice 5 : Utilisation de doublures (mock) pour les tests

Certaines méthodes que l'on veut tester peuvent nécessiter d'autres objets qu'il n'est pas facile d'utiliser de manière réelle comme par exemple la connexion à un serveur, une base de données, etc. Dans ces cas, il peut être intéressant d'utiliser des doublures des objets réels (des *mock* en anglais) pour simuler le comportement des objets réels ou vérifier que les objets réels seront correctement utilisés.

Nous allons l'illustrer avec la méthode `somme` de la classe `Somme`. Cette méthode calcule la somme des entiers qui sont contenus dans le `BufferedReader` qui est fourni en paramètre. En cas d'erreur dans le fichier (une ligne qui ne correspond pas à un entier, ou un entier négatif), une erreur est signalée en utilisant l'interface `Erreurs`.

L'objectif est de tester le bon comportement de cette méthode aussi bien en ce qui concerne le traitement des informations du `BufferedReader` que le signalement des erreurs.

Au lieu de s'appuyer sur des vrais fichiers, nous allons utiliser des doublures en utilisant Mockito.

Pour la suite, on utilisera la documentation sur Mockito qui est disponible à l'URL suivante :
`http://static.javadoc.io/org.mockito/mockito-core/2.23.4/org/mockito/Mockito.html#1`
5.1. Lire, comprendre le test `testerSommeAvecUnVraiFichier`.
5.2. Lire, comprendre le test `testerSommeAvecUneDoublure`.
5.3. Exécuter les tests (./gradlew test) et, en cas d'erreur, consulter le rapport sur les tests qui est disponible via `build/reports/tests/test/index.html`
5.4. Ajouter un nouveau test en utilisant une doublure de `BufferedReader` qui produit les informations suivantes : "1", "-5", "2", "xxx", "4" et bien sûr **`null`** qui indique la fin du flot.

Le résultat attendu devrait être 7 car -5 sera ignoré car il est négatif et "xxx" aussi car il est invalide (pas un entier).
5.5. Ajouter un nouveau test avec une doublure qui lève une exception de type `IOException` après avoir fourni les valeurs "1", "-1", "2", "trois" et avant de fournir les valeurs "10", "20" et **`null`**.

Pour réussir, le test doit vérifier que la méthode `somme` laisse se propager l'exception `IOException`.

Serait-il facile de réaliser ce test avec un vrai fichier ?

5.6. Compléter l'avant dernier test (somme vaut 7) pour vérifier que les erreurs sont correctement signalées. On utilisera une doublure de l'interface `Erreurs` !

5.7. Vérifier que les erreurs sont signalées dans le bon ordre.

Exercice 6 : Évaluer la qualité d'une suite de test (avec Pitest)

Pour évaluer la qualité des programmes de test, une stratégie consiste à modifier le programme sous test (par exemple changer un « < » en « > », un « + » en « - », etc.) et à vérifier que les tests détecte l'erreur. Le programme modifié est obtenu par « mutation » du programme sous test et est appelé « mutant ». Si les tests signalent une erreur (et donc échouent) sur le mutant, c'est que ce sont des bons tests. Si les tests réunissent, on dit que le mutant « survit » et on en déduit que les tests sont incomplets.

Pitest est un outil qui réalise ce type de test.

6.1. *Exécuter pitest.* La configuration pour pitest a déjà était faite. Pour exécuter pitest, il suffit donc de faire :

```
gradle pitest
```

6.2. *Consulter les résultats.* Les résultats sont produits dans le dossier `build/reports/pitest/` avec un sous-dossier par lancement de pitest qui correspond à la date du lancement. On peut alors ouvrir le fichier `index.html` dans un navigateur.

6.3. *Compléter les tests.* Compléter les tests pour essayer de tuer les mutants survivants.