TP Test unitaire de programmes Java

Objectif du TP :

- Manipuler JUnit v4 dans l'IDE Eclipse pour faire du test unitaire de programmes Java
- Utiliser le plugin Emma pour mesurer la couverture réalisée par les tests
- Comprendre comment utiliser EasyMock pour simuler un bouchon à partir d'une interface
- Expérimenter, sur 2 classes Java, une approche de développement dirigé par les tests ("Test Driven development", TDD)
- Concevoir des données de test selon différents critères structurels : test des instructions, test des branches, test des chemins

Fichiers à récupérer :

Récupérer l'archive zip contenant :
- le fichier entreetp.txt et le répertoire MesStats à utiliser dans la première partie du TP (TDD),
- le répertoire Exo2 à utiliser pour la seconde partie du TP (conception des tests, mocks).

Plugin Emma (2.1.4) à installer :

- Help - Install New Software - Add.
- Donner un nom (EclEmma) et une adresse (http ://update.eclemma.org/).
- Puis faire Ok et installer le plugin.
- Accepter la licence puis redémarrer Eclipse.
- Le bouton Emma apparait à coté du bouton debug.

1 Développement dirigé par les tests

Pour cette partie, charger dans Eclipse le projet MesStats.

Le répertoire MesStats contient un répertoire des sources de l'application à étudier, et un répertoire pour les tests associés. Le projet est composé des sources de 7 classes Java (Main.java, Pluviometrie.java, Temperatures.Java, Vents.java, Releve.java, TraitementData.java, RecupData.java), ainsi que deux classes de test (l'une pour tester la classe Releve, l'autre pour tester la classe TraitementData) et une classe pour une suite de test (regroupant les 2 classes de test précédentes).

Ce projet utilise le fichier entreetp.txt comme entrée dans le main (c'est un exemple de fichier de données). Vous devez donc modifier dans Main.java le chemin d'accès à entreetp.txt afin qu'il soit conforme à votre configuration personnelle.

Par la suite, dans ce projet, seuls les sources de Releve.java et TraitementData.java sont incorrects. Vous ne devez donc travailler et modifier que ces 2 classes.

Dans l'approche de développement dirigé par les tests, on commence par concevoir les tests avant le code de l'application (ce qui est fait ici par les 3 classes ReleveTest.java, TraitementDataTest.java et AllTests.java). Le code correspondant doit être le plus simple possible, son seul objectif au départ étant de passer à la compilation et non pas d'implanter les spécifications (cf les versions proposées pour Releve.java et TraitementData.java). Il est donc normal (et souhaitable) que les tests échouent sur ces versions initiales.

Travail à faire :

1. Lancer l'application en cliquant sur Main.java - RUN as java application. Que pensez-vous des résultats obtenus ?
2. Test et couverture de Releve.java
   - Tester la classe Releve en un clic sur ReleveTest.java - Run as junit test. Qu'observez-vous ?
   - Lancer la mesure de couverture en cliquant sur le logo Emma. Observez la coloration syntaxique des codes source et de test (vert = couvert, rouge = non couvert, jaune = partiellement couvert) et l'affichage des résultats (en nombre et pourcentage d'instructions couvertes) dans la fenêtre Coverage.
   - Modifier la configuration d'Emma pour que la coloration et les mesures de couverture ne concernent que les sources de l'application : clic droit sur logo Emma - coverage configurations - onglet coverage - déselectionner le répertoire test, apply puis relancer coverage.
   - Dans l'explorer, clic droit sur Releve.java - properties - coverage : observez le rapport de couverture en termes d'instructions, de lignes, de méthodes, de types, et mesure de complexité.
3. Test et couverture globaux de l'application
   - Clic sur AllTests.java - Run as junit test. Qu'observez-vous ?
   - Lancer la mesure de couverture en cliquant sur le logo Emma (fait le run en meme temps).
   - Modifier la configuration d'Emma pour que la coloration et les mesures de couverture ne concernent que les sources de l'application : clic droit sur logo Emma - coverage configurations - onglet coverage - déselectionner le répertoire test, apply puis relancer coverage. Observez l'affichage des résultats dans la fenêtre Coverage.
   - Dans l'explorer, clic droit sur package messtats - properties - coverage : observez le rapport de couverture.
4. Développement dirigé par les tests
   - Modifier itérativement la classe Releve.java de façon à fixer un test à chaque étape. On arrête lorsque les tests de ReleveTest.java passent tous avec succès.
   - Modifier itérativement la classe TraitementData.java de façon à fixer un test à chaque étape. On arrête lorsque les tests de TraitementDataTest.java passent tous avec succès.
   - Vérifier que la suite de test AllTests.java passe avec succès et observer la couverture réalisée.
   - Dans l'explorer, clic droit sur Releve.java - properties - coverage : analyser le rapport de couverture.
   - Dans l'explorer, clic droit sur TraitementData.java - properties - coverage : analyser le rapport de couverture.
5. Pour finir, exécuter l'application. Que pensez-vous des résultats obtenus ?

2 Conception de tests

Pour cette partie, charger dans Eclipse le projet Exo2.
Le répertoire Exo2 contient le répertoire des sources de l'application à tester.
Il s'agit de 4 classes Java : MonMenu.java, MesFonctions.java, MesVerifs.Java, IGenerateur.java.
IGenerateur.java est une interface pour laquelle nous ne disposons d'aucune implémentation.
Pour tester MonMenu.java, nous allons donc devoir utiliser un bouchon simulant une implémentation de IGenerateur.java. Pour cela, l'outil EasyMock va nous permettre de définir un mock de l'interface Igenerateur.java. Nous pourrons ainsi tester MonMenu.java, MesFonctions.java et MesVerifs.Java.

Installer EasyMock :
- Télécharger EasyMock (3.1) à partir de EasyMock Homepage (http ://easymock.org/)
- Ajouter easymock.jar au classpath (i.e. Exo2 - Clic droit - Properties - Java BuildPath - Libraries - Add External Jars)

Travail à faire :

1. Création des jeux de test pour les classes **MesFonctions** et **MesVerifs** (*)
   - Test de la classe **MesFonctions.java** :
     - Construire un jeu de test (nom par défaut : **MesFonctionsTest.java**) adéquat au critère du test des **instructions** pour la classe **MesFonctions.java**.
     - Le jeu de test proposé est-il également adéquat au test des **branches** ? Si non, construire un nouveau jeu de test qui le soit (appelé **MesFonctionsBranchesTest.java**).
     - Le jeu de test proposé est-il adéquat au test des **chemins** ? Si non, construire un nouveau jeu de test qui le soit (appelé **MesFonctionsCheminsTest.java**).
   - Test de la classe **MesVerifs.java** :
     - Construire un jeu de test (nom par défaut : **MesVerifsTest.java**) adéquat au critère du test des **instructions** pour la classe **MesVerifs.java**.
     - Le jeu de test proposé est-il également adéquat au test des **branches** ? Si non, construire un nouveau jeu de test qui le soit (appelé **MesVerifsBranchesTest.java**).
     - Le jeu de test proposé est-il adéquat au test des **chemins** ? Si non, construire un nouveau jeu de test qui le soit (appelé **MesVerifsCheminsTest.java**).
2. Création des jeux de test pour la classe **MonMenu.java** (en créant un mock pour l'interface IGenerateur (***)) :
   - Construire un jeu de test (nom par défaut : **MonMenuTest.java**) adéquat au critère du test des **instructions** pour la classe **MonMenu.java**.
   - Le jeu de test proposé est-il également adéquat au test des **branches** ? Si non, construire un nouveau jeu de test qui le soit (appelé **MonMenuBranchesTest.java**) .
   - Le jeu de test proposé est-il adéquat au test des **chemins** ? Si non, construire un nouveau jeu de test qui le soit (appelé **MonMenuCheminsTest.java**, cf les deux critères spécifiques aux boucles vus en cours).
3. Construire une suite de test (nom par défaut : **AllTests.java**) globale au projet (i.e. testant à la fois **MesFonctions, MesVerifs** et **MonMenu**) pour chacun des critères énoncés précédemment. (**)

4. Exécution des jeux de test, synthèse de la campagne de test et édition des rapports de couverture
   - Exécuter la suite de tests AllTests
   - Noter les verdicts pass/no pass pour les différents tests
   - Créer le rapport de couverture en html : fenêtre Coverage, clic droit - export session
   - Faire de même pour la suite AllTestsChemins
5. Correction du code : itérer les phases de correction du code en suivant les indications données par les tests et les phases de test jusqu'à ce que tous les tests passent (contrôler aussi les affichages)
6. Bilan final des tests réalisés et des couvertures obtenues.

(*) Comment créer un test avec Junit ?

Avec JUnit, un jeu de test (pour une classe à tester) est décrit dans une classe de test. Cette classe de test contient des méthodes pour assurer les préconditions et postconditions des tests, ainsi que des méthodes de test qui encodent chacune un jeu de test différent (pour tester les différentes méthodes de la classe, ou pour tester de plusieurs manières une même méthode).

Pour un projet proprement organisé, les jeux de test et suites de test sont stockés dans un répertoire de fichiers sources de test, habituellement nommé "test".

- Créer en premier le répertoire des tests : clic droit sur le projet, puis New, puis Source Folder, puis donner le nom "test".

- Pour créer une classe de test : clic droit sur la classe à tester, puis New, puis JUnit Test Case, puis changer le nom du Source Folder par "test", puis cocher la classe à tester (pour créer des squelettes de méthodes de test pour chacune des méthodes de la classe à tester).

Vient ensuite la phase de programmation des jeux de test.

- Pour cela, il faut en général déclarer en attribut dans la classe de test au-moins un objet de la classe à tester (sauf dans le cas de méthodes statiques). Cet objet est instancié dans une des méthodes de test, souvent dans la méthode **setUp()**.

- On programme alors le jeu de test d'une méthode en listant dans la méthode de test associée tous les cas de test souhaités. JUnit propose plusieurs primitives pour définir un cas de test, la plus commune étant **assertEquals(résultat_attendu, résultat_obtenu)**.

Le **résultat_obtenu** est le résultat de l'appel de la méthode en cours de test sur une donnée de test, i.e. une des instances (crées par exemple dans la méthode **setUp()**) avec les paramètres nécessaires : cela représente **ce que fait réellement le programme**.

Le **résultat_attendu** est l'oracle, i.e. le résultat donné par la **spécification** sur la même donnée de test.

On peut également utiliser **assertTrue(résultat_obtenu)** ou **assertFalse(résultat_obtenu)** si la méthode testée retourne un booléen et qu'on souhaite un résultat attendu vrai ou faux respectivement.

- Pour faciliter le traçage et l'interprétation des tests, on conseille d'insérer dans les classes de test des "print" permettant d'afficher des informations sur le type de test réalisé, par exemple, "test de la classe Relevé selon le critère du test des instructions" au début d'une méthode **setUpClass()**, ou encore au début de chaque méthode de test un message comme "test de la méthode GetJour0() - cas spécifique bla bla...").

(**) Comment créer une suite de test avec Junit ?

Une suite de test permet de regrouper les tests de différentes classes, par exemple au niveau d'un package. Elle se décrit également sous Junit avec une classe de test.

Pour créer une classe de suite de test : clic droit sur le package à tester, puis New, puis Junit Test Suite, puis changer le nom du Source Folder par "test" et cocher les classes de test à inclure dans la suite.

(***) Comment simuler une interface avec EasyMock ?

- Il faut commencer par la création d'un objet mock pour la classe désirée :

```
MaClasseASimuler MonObjetSimulateur ;
MonObjetSimulateur = EasyMock.createMock(MaClasseASimuler.class) ;
```

- Puis, on doit spécifier les valeurs attendues lors des appels successifs à la(les) méthode(s) simulée(s) MaMethode de la classe MaClasseASimuler :

```
EasyMock.expect(MonObjetSimulateur.MaMethode(ListeArgs)).andReturn(OutputAttendue) ;
```

- Quand la spécification est terminée, il faut activer le mock :

```
EasyMock.replay(MonObjetSimulateur) ;
```

- On peut ensuite écrire les tests Junit comme d'habitude. A la fin des tests, il est intéressant de valider que l'utilisation du mock dans les tests est conforme aux spécifications faites :

```
EasyMock.verify(MonObjetSimulateur) ;
```