TP Outil de développement logiciel : les tests #1

Rendre une archive par TP sur Celene ; un rendu par étudiant

Consignes générales

- Tout ce qui est nécessaire au TP est installé dans la VM. Il y a un compte "sti" mot de passe "sti"
- Vous rendez vos rapports pour le contrôle continu sur Celene, avant minuit le jour du TP, sous forme d'archive contenant un rapport en pdf ainsi que tout code utile
- **Par défaut le clavier est en anglais,** $setxkbmapfr$ **résoudra ce problème.**

Prologue au TP

Le but de ce TD est de vous familiariser avec les pratiques et l' écosystème logiciel typiques d'une entreprise dont l'activité principale est le développement.

En entreprise, on ne développe jamais seul : le plus souvent, on fait partie d'une équipe de développement qui produit du code. La structuration d'une équipe et sa manière de travailler peut être très variée ; on retrouve souvent un certain nombre de rôles clefs (architecte, chef de projet, responsable qualité, développeur, expert), les activités des divers acteurs étant coordonnées par l'application d'une méthode de développement (de nos jours celle-ci est souvent *agile*).

Les activitités sont variées : produire, tester le code de manière unitaire, le mettre à disposition, l'intégrer et effectuer des tests d'intégration, pour qu'il puisse *in fine* être déployé dans un environnement de production.

Le test est une partie très importante du développement : il permet de **vérifier** au jour le jour si le code produit est conforme à ses spécifications, ainsi que **valider** si le produit répond bien aux besoins exprimés. Les tests permettent par exemple lors de la livraison du produit de le **valider** : quand on livre, le logiciel développé est soumis a des tests de recette définit dans le cahier des charges. Si les tests ne passent pas, la livraison peut être refusée.

Il existe énormément de type de tests différents : tests fonctionnels (le programme répond-t-il correctement ?), les tests de charge (quand il a beaucoup d'activité, le programme tient-il le coup ?), les tests de stress (résiste-il bien à des pics d'activités ?), de robustesse, etc, etc. De même, il existe des alternatives aux tests nommées vérification formelle, comme le model checking ou la preuve. Dans ce TD nous nous intéresserons uniquement aux tests fonctionnels.

Elements de vocabulaire

- On parle de tests *boîte noire* quand on a pas accès au code et de tests *boîte blanche* quand on y a accès.
- Un *test unitaire* est un test est une procédure permettant de vérifier le bon fonctionnement d'une partie précise d'un logiciel ou d'une portion d'un programme.
- Un *test d'intégration* vérifie lui le comportement de tout le programme par rapport aux spécifications.
- Un *test de validation* établit si le programme est conforme aux besoins exprimés.

Couverture de tests et graphe de contrôle

Quel que soit le type de test, celui-ci repose toujours sur le même principe :

- On fixe les données d'entrées
- On exécute le composant
- On vérifie que le comportement est attendu (ce qui est souvent fait par quelque chose qu'on appelle *l'oracle*).

Un test va donc permettre de tester pour un jeu de données d'entrée le comportement du composant. Cela signifie donc que pour tester le composant totalement, il faudrait tester l'intégralité des jeux de données possibles ! Heureusement, quand on fait des tests "boîte blanche" on a accès au code et on peut vérifier si l'on parcours tous les chemins possibles d'exécution ou si l'on teste bien toutes les fonctionnalités d'un programme.

On appelle *couverture* la portion du programme qui a été testé. Il existe bien sûr plusieurs types de couverture suivant ce que l'on teste : la couverture structurelle qui s'assure que l'on passe bien par tous les chemins d'exécutions possible et la fonctionnelle, qui s'assure que l'on teste bien toutes les fonctionnalités d'un programme.

Un outil habituel pour voir cette couverture est le graphe de flot de contrôle qui permet de voir les différentes branches d'un programme, dont voici un exemple à la figure 1.

Chaque bloc d'instruction y est représenté comme un noeud, chaque branchement conditionnel comme un arc (on parle de branche). La couverture peut être calculée par instruction, par branche, par chemins avec itérations, par flux de données. Par défaut dans ce document, nous nous intéresserons à la couverture par branche.

Pour visualiser le parcours d'un programme, il existe aussi des outils, comme le plugin eclEmma sous Eclipse.

1 Premier pas : partage de code et tests boîte noire

Dans une entreprise, il est essentiel d'utiliser des gestionnaires de versions comme SVN ou git pour ne pas perdre le travail et le partager avec les autres développeurs.

Un de vos collègues (John) a écrit une librairie de mathématique et vous a donné le jar correspondant pour que vous le testiez. John a mis sur Celene la javadoc du jar et le jar.

La librairie contient :

— ppcm : plus petit commun multiple (voir la page Wikipédia) en trois versions : la correcte, l'optimisée et celle qui ne fonctionne pas ...

Vous allez devoir tester chacune des fonctions disponibles en utilisant JUnit. JUnit est un outil qui permet de mettre en place des tests unitaires. Il est par défaut fourni avec Eclipse.

1.1 Comment créer un test unitaire avec JUnit

Tout d'abord, il vous faut créer un nouveau projet, et ajouter johnArithmetics.jar aux archives pour y avoir accès. Pour ajouter correctement johnArithmetics.jar aux librairies du projet, il faut d'abord créer un projet Java puis faire un cilc-droit sur le projet -> build path -> libraries -> Add external jar ... puis choisir johnArithmetics.jar. Ensuite, il va vous falloir créer des tests. C'est très simple :

— File > New > JUnit Test Case

— Gardez les valeurs par défaut, le plus important est de renseigner la classe que vous allez tester, ici johnArithmetics.MathUtils ...

— Selectionnez toutes les méthodes de la classe pour pouvoir générer des tests pour chacune d'elles.

— Cliquez jusqu'au finish.

Eclipse vous a généré une classe avec une méthode par test. Chaque méthode contient un

```
fail("Not yet implemented");
```

```
int main(int argc, char *argv[])
{
   int i = 0;
   int j = 0;

   i = 2;
   while (i+j <= 20) {
      if (i >= 10) {
         i += 4;
      } else if (j <= 4) {
         i += 2;
         j++;
      }
   }

   assert(j-i != 0)

   return EXIT_SUCESS;
}
```

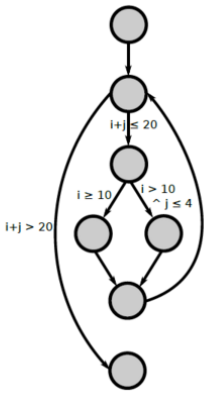

FIGURE 1 – Exemple de flot de contrôle (copyright Attribution-Share Alike 3.0 Unported ENSIMAG

C'est dans chacune de ces méthodes que vous devez initialiser vos données et configurer les différents paramètres pour pouvoir tester ce que vous voulez. Ici, il n'y a que des méthodes statiques. Il suffit donc de tester que les résultats sont ceux attendus. Cela se fait par exemple par la méthode assertEquals() :

```
public void testJohnBetterPPCM() {
assertEquals(0,MathsUtils.johnBetterPPCM(0,0));

}
```

assertEquals fait appel à la méthode equals pour vérifier si le premier argument (le résultat attendu) est égal au résultat du test. Quand vous lancez les tests, vous allez obtenir un petit rapport avec les failure et les erreurs.

QUESTION Implémentez pour chacune des méthodes pour différents cas à tester :

— lancer ppcm de (0,0) doit donner 0.
— lancer ppcm de (0,X) doit donner 0.
— lancer ppcm de (X,0) doit donner 0.
— lancer ppcm de (X,X) doit donner X.
— lancer ppcm de (60,168) doit donner 840.

Sauvegardez votre code dans un coin pour le rendre à la fin des TP. Observez les résultats :

— Les différents codes fonctionnent-ils correctement ?
— D'après vous, en terme de couverture, vos tests sont-ils exhaustifs ?

2 Couverture de code

2.1 Graphe de contrôle

John a donc des erreurs dans son code. Il le partage avec vous ... Vous le trouverez dans le répertoire Java de votre répertoire `TP_Arithmetics`.

Question : Pour chacune des 3 méthodes, dessinez les graphes de contrôle des méthodes.

2.2 Visualisation de la couverture des tests avec EclEmma

EclEmma est un plugin Eclipse qui met en place un text higlighting basé sur la couverture par les tests : quand un bloc d'instruction n'est pas couvert, il est coloré en rouge.

Il faut dans un premier temps inclure dans son projet le .java de MathUtils, qui remplace donc le .jar. Pour supprimer le .jar de votre projet, il faut repasser par Propriétés -> Build Path -> Librairies puis faire Remove pour celui-ci, puis importer le fichier java. Une fois ceci fait, EclEmma aura à sa disposition le code source de MathUtils, et pourra donc indiquer quelles branches sont couvertes par les tests, et lesquelles ne le sont pas.

On l'utilise de la manière suivante : parmi les icônes vertes de run, il y en a une, tout à gauche, avec une petite barre en dessous. C'est elle qu'il faut lancer pour voir quelles sont les parties du code qui ne sont pas couvertes.

QUESTION : L'intégralité de votre code est-il couvert par les tests ? Pour quelles méthodes des branches ne sont pas couvertes ? Lesquelles, pourquoi ?

QUESTION : Ecrire des tests qui permettent de couvrir tous les branchements de chacune des 3 versions. A cette occasion, séparez les différents assertEquals pour que chacun ait sa propre méthode de test.

2.3 Faisons évoluer le code

En fait, si le code remplit bien les spécifications, celles-ci ne suivent pas la définition du ppcm, qui ne doit pas être défini quand l'un ou l'autre des paramètres sont nuls. Il faut ajouter au code de John une exception, qui doit être levée quand cette condition est remplie. Pour cela, il faut utiliser la syntaxe suivante (JUnit 4 seulement) :

```
@Test(expected=JohnException.class)
public void testJohnBetterPCCM() {
   MathsUtils.johnBetterPCCM(0,0);
}
```

QUESTION :

— Ajoutez l'exception, modifiez le code et les tests en conséquence pour que les méthodes passent correctement les tests.

3 Annexes

```
package johnArithmetics;

/**
 *
 * My vision of maths, by John, Another dev.
 *
 * @author John
 *
 */
public class MathsUtils {
/**
 *
 * John's optimized PPCM version
 *
 * @param a
 *            one value
 * @param b
 *            another value
 * @return ppcm
 */
public static int johnBetterPPCM(int a, int b) {
int sum, modulo, PPCM;

sum = a * b;
modulo = a % b;
while (modulo != 0) {
a = b;
b = modulo;
modulo = a % b;
}
PPCM = sum / b;
return PPCM;
}

/**
 *
 * John's faulty PPCM version. I can't figure out if it's working or not.
 *
 * @param a
 *            one value
 * @param b
 *            another value
 * @return ppcm
 */
public static int johnFaultyPPCM(int a, int b) {
int p, mincm = 0;
p = a * b;
while ((p > a) && (p > b)) {
if ((p % a == 0) && (p % b == 0)) {
```

```
mincm = p;
}
p = p - 1;
}
return mincm;
}

/**
 *
 * John's correct PPCM version. Someone came and add a few line to correct a
 * bug in my code. Not sure if he's right or wrong.
 *
 * @param a
 *            one value
 * @param b
 *            another value
 * @return ppcm
 */
public static int johnCorrectPPCM(int a, int b) {
int p, mincm = 0;
p = a * b;
if (a == b)
return a;

while ((p > a) && (p > b)) {
if ((p % a == 0) && (p % b == 0)) {
mincm = p;
}
p = p - 1;
}
return mincm;
}
}
```