SIT211 - DÉVELOPPEMENT DIRIGÉ PAR LES TESTS

OBJECTIFS

Il s'agit de découvrir et d'appliquer l'approche de *développement piloté/dirigé par les tests* ou, en anglais, « Test Driven Development ». On en profitera pour utiliser les *Rules* de JUnit.

UTILISATION DES *RULES* DE JUNIT

Par défaut, JUnit considère chaque méthode de test comme un seul et unique test. Si celle-ci enchaîne plusieurs `assert`, il suffit que l'un d'eux échoue pour que JUnit considère aussitôt que le test a échoué. L'éventuel message d'erreur associé est alors affiché, *mais les* `assert` *suivants de cette méthode ne sont pas exécutés !* Cela empêche malheureusement la mise en évidence rapide de tous les bugs détectables par le code de test, ainsi que le calcul de la couverture de test.

C'est pourquoi le mécanisme des règles a été ajouté à JUnit : cela permet d'exécuter *tous* les cas de test prévus et d'en faire un compte rendu détaillé. Ainsi, une méthode de test peut signaler directement plusieurs erreurs, contrairement au comportement par défaut décrit précédemment. Le principe général est de définir un collecteur d'erreurs (cf `ErrorCollector`) et d'y stocker tous les résultats de test (succès ou échec) au fur et à mesure (cf `checkThat`, qui remplace les `assert`) et JUnit fait un compte-rendu détaillé ensuite. Voici l'application à l'exemple qui vous a été fourni dans le cadre du TP JUnit (ndlr : les ajouts sont en rouge, et les suppressions en caractères barrés) :

```
package poureleves.compteur;
~~import static org.junit.Assert.assertEquals;~~
import org.junit.Test;
import org.junit.Rule;
import org.junit.rules.ErrorCollector;
import static org.hamcrest.CoreMatchers.is;

public class CompteurTest2 {

	@Rule
	public ErrorCollector collector = new ErrorCollector();

	/**
	 * Test method for {@link poureleves.compteur.Compteur#ajouterVal(int)}.
	 *
	 * @throws CompteurInvalide
	 */
	@Test
	public final void testAjouterVal() throws CompteurInvalide {
		Compteur c1;
		c1 = new Compteur(5);
		c1.ajouterVal(4);
		// premiere assertion de test :
		// si la valeur du compteur est differente de 9, provoque un echec du
test :
		~~assertEquals("t1.1", 9, c1.getVal());~~
		collector.checkThat("t1.1",c1.getVal(), is(9));
		//
		c1.ajouterVal(3);
		// deuxieme assertion de test
		~~assertEquals("t1.2", 12, c1.getVal());~~
		collector.checkThat("t1.2",c1.getVal(), is(12));
	}
...
}
```

Essayez cette nouvelle version !

Modifiez les valeurs attendues (9 et 12) pour faire en sorte que les 2 cas de test échouent, et

observez comment cela se traduit lors du test.

DÉVELOPPEMENT DIRIGÉ PAR LES TESTS

L'approche de développement logiciel dirigée par les tests (DDT) consiste à pousser à l'extrême un principe que vous avez déjà expérimenté et qui consiste à écrire les tests *d'abord*, *puis* – et non pas avant ! – écrire le code répondant correctement à ces tests. L'approche DDT consiste à répéter autant de fois que nécessaire ce principe, pour construire progressivement l'application souhaitée : c'est, en général, une méthode itérative[1]. À chaque itération, on enrichit le code de test avec de nouveaux cas *qui font échouer la version courante de l'application*, et celle-ci doit alors être modifiée *a minima* pour passer le nouveau jeu de tests. Comme toutes les méthodes itératives, cette approche permet de converger plus sûrement/sereinement vers une application fonctionnelle, mais nécessite parfois de la réécriture de code (« refactoring »). Comme la DDT impose une mobilisation constante des tests, ceux-ci doivent être entièrement automatisés… JUnit sera votre allié pour cela.

Téléchargez l'archive `MesStats.tar.gz` sur Moodle, et importez ce projet dans Eclipse : *`File → import → general/existing projects into workspace → archive file → MesStats.tar.gz`*

L'application comprend 7 classes : `Main`, `Pluviometrie`, `RecupData`, `Releve`, `Temperature`, `TraitementData` et `Vents`. Il y a par ailleurs 2 classes de test (pour tester respectivement les classes `Releve` et `TraitementData`), et une suite de tests pour enchaîner ces 2 classes. Tout ceci est réparti dans 2 répertoires de code[2], respectivement `src` et `test`. C'est bien pratique pour ranger ainsi d'un côté le code de l'application, et de l'autre côté tout ce qui sert à tester.

Voyez comme la structure des packages est la même dans les deux répertoires `src` et `test`.

Le projet comporte par ailleurs un répertoire « simple », `data`, qui contient le fichier `entreeTP.txt` : c'est un exemple de fichier de données que `Main` devra traiter.

Vous pouvez d'ores et déjà lancer l'application `Main`. Comme elle n'est pas finalisée, les informations affichées ne sont pas correctes.

Selon l'approche DDT, les tests sont écrits en premier, *avant* le code de l'application. Ce travail a été entamé : les classes de tests `ReleveTest` et `TraitementDataTest` comportent en effet d'ores et déjà tous les cas de test prévus.

Exécutez les tests existants (via `AllTests`). Inspectez les résultats détaillés affichés par `JUnit`.

Que pensez-vous de la couverture de test ? (ndlr : utilisez `Emma` !)

Dans la suite de **ce travail, seules les classes `Releve` et `TraitementData` seront modifiées.**

Modifiez progressivement la classe `Releve` de façon à fixer un test à chaque étape, jusqu'à ce que tous les tests de `ReleveTest` soient OK.

Modifiez progressivement la classe `TraitementData` de façon à fixer un test à chaque étape, jusqu'à ce que tous les tests de `TraitementDataTest` soient OK.

Vérifiez que la suite `AllTests` passe avec succès, et observez la couverture obtenue.

DDT – DÉMARCHE COMPLÈTE

Vous allez cette fois-ci travailler sur un autre projet, dont les tests n'ont pas été encore écrits. Ce sera justement votre travail que de les écrire !

1 Pour comprendre la différence entre « itératif » et « incrémental » : http://romy.tetue.net/mona-lisa-agile
2 Un répertoire de code est un répertoire référencé dans les variables d'environnement Java comme contenant du code.

Téléchargez l'archive `Voitures.tar.gz` sur Moodle, et importez ce projet dans Eclipse : *`File → import → general/existing projects into workspace → archive file → Voitures.tar.gz`*

Comme précédemment, il y a un répertoire pour l'application (`src`) et un répertoire pour les tests associés (`test`). Les classes mises à disposition (que ce soit le code de l'application ou celui de test) ne sont encore à ce stade que des squelettes.

Création des jeux de test

`AutoTest.java` prévoit déjà une méthode de test (vide) pour chaque fonction de la classe `Auto`.

Complétez le code de `AutoTest`, en vous appuyant sur les spécifications fournies dans la classe `Auto` (cf Javadoc).

De la même manière, élaborez le code de test `GarageTest`.

NB :Vous pouvez vous répartir les fonctions à tester au sein de votre binôme.

Exécutez les tests, via la classe `AllTests`, et vérifiez si le code testé est bien couvert par vos tests.

Implémentation de l'application

Il est maintenant temps de s'occuper du code de l'application.

Élaborez chacun les codes des classes `Auto`, puis `Garage`, permettant de répondre correctement à vos tests. Si vous vous êtes répartis l'écriture des codes de test, vous écrirez les fonctions concernées par les tests écrits par votre équipier.

La couverture de test est-elle toujours maximale ? Faut-il rajouter de nouveaux cas de tests ?