MODEL - TD TDD

Objectifs

Dans ce TD nous allons illustrer le principe du TDD sur un exemple simple. Nous allons aussi illustrer l'utilisation de stubs et de couverture du code par les tests. Vous pourrez utiliser l'IDE de votre choix **mais** l'ensemble du travail doit pouvoir fonctionner hors IDE à l'aide du système de build indiqué dans la suite.

Installation

1. créer un projet gradle et le versionner sur GitHub. Remplacer le fichier `build.gradle` par celui fourni en ligne (pensez à modifier ce qui serait spécifique à votre projet).
2. dans ce projet nous allons utiliser JUnit, Hamcrest et Mockito.

   A quoi servent ces librairies ?
3. vérifier que tout marche bien avec `./gradlew cleanTest test`

Sujet

Au cours de ce TD, nous allons procéder de la manière suivante. Nous allons tenir à jour une *task list* contenant une liste de tâches à réaliser vis-à-vis du code. A chaque étape, nous choisirons dans cette liste vers quelle étape procéder, tout en rajoutant éventuellement d'autres tâches sur cette liste. Cette liste de tâche sera un fichier texte organisé comme suit :

```
[X] 1- description de la tâche
[X] 2- description de la tâche
[ ] 3- description de la tâche
...
```

Le sujet de ce TD porte sur la réalisation d'un dictionnaire bilingue. Stricto sensu, il s'agit de pouvoir réaliser des traductions entre deux langues, dans les deux sens. Pour des raisons de portabilité, il sera intéressant de pouvoir sauvegarder et charger des dictionnaires à travers des fichiers externes.

Construire une liste de tâche associée au TD

Fake it !

La question du choix de la première tâche à réaliser est une question importante. Elle doit être suffisamment simple pour permettre de réaliser un premier cycle *red-green-refactor* rapidement.Manifestement, dans notre problème, la classe centrale est la classe `Dictionary`.

Commençons par écrire un test qui ne passe pas pour cette classe.

1. Construire une classe de tests nommée `DictionaryTest`.

2. Ecrire dans cette classe un test créant un objet de type `Dictionary`, lui assignant un nom, et vérifiant que ce nom est correctement stocké dans l'objet.

   Il est fondamental d'écrire un test qui ne passe pas, comme par exemple :

   ```
   @Test public void testDictionaryName() {
   assertThat(dict.getName(), equalTo("Example"));
   }
   ```

   En ce sens, rien n'est imposé par rapport au choix des noms des classes ou des méthodes, puisque la classe `Dictionary` n'existe pas encore.

3. Maintenant que la *red bar* est atteinte, nous allons tâcher de faire passer le test. Pour cela, il existe plusieurs techniques. Celle utilisée ici est nommée *Fake it* (litt. « Fais semblant »). Concrètement, elle consiste à faire le minimum nécessaire pour faire passer le test. Dans notre cas, il suffit d'une méthode `getName()` renvoyant la chaîne de caractères `"Example"`.

   Créer une classe `Dictionary` vide, puis lui ajouter un constructeur vide.

4. Écrire une méthode vide `getName()` renvoyant la chaîne de caractères `"Example"`.

   La notion de « semblant » est ici aussi fondamentale : elle permet de construire le code pas à pas, en utilisant à chaque étape une méthode simple, rapide, et faisant passer les tests existants.

   Cette pratique autorise un grand nombre de dérives dans le style de programmation : variables globales ou publiques, conversions du type (cast) des objets ... dont il faudra tenir compte lors de la phase de refactoring.

5. Lorsque la *green bar* est atteinte arrive la phase la plus complexe du TDD : la phase de *refactoring*. Pour l'instant, nous allons nous limiter à éliminer les duplications de code.

   Quelle duplication existe pour l'instant dans notre code ?

6. Supprimer la duplication du code en introduisant un attribut privé `name`, et adapter le constructeur et la méthode `getName()` de manière à s'assurer que cette variable soit correctement positionnée et renvoyée.

   Qu'englobe la notion de refactoring ? Toute forme de modification du code qui conserve le passage des tests existants, et qui permet d'obtenir une architecture logicielle avec un minimum de défauts. Quelques exemples :

   – supprimer la duplication du code / déplacer du code ;

   – ajuster le caractère privé/public des attributs/méthodes.

7. Le cycle de travail est maintenant bouclé. Il devient alors possible de recommencer ce cycle avec un nouveau test. Les tests pré-existant assurent une certaine confiance dans le code déjà écrit, et permettent d'envisager les modifications futures avec sérénité.

   Utiliser la technique précédente pour écrire un test, puis une méthode permettant de vérifier si un dictionnaire est vide ou

pas (méthode `isEmpty`). En l'absence de méthodes pour ajouter quoi que ce soit au dictionnaire, on se limitera à renvoyer une valeur constante.

8. Comme cette fonctionnalité n'est pas implémentée de manière correcte, rajouter le problème du traitement du dictionnaire vide dans la task list.

Triangulation

Le TDD insiste profondément sur la programmation par nécessité. Il faut d'abord écrire le test qui génère un besoin fonctionnel (*test-first*), avant de coder ce besoin. Néanmoins, la méthode *Fake it* vue précédemment montre qu'il est possible de faire passer des tests à un programme sans réellement écrire le code nécessaire. Le problème vient ici du fait que nous n'avons pas suffisamment spécifié les tests permettant de cerner le comportement d'une méthode. Pour raffiner les tests, nous allons appliquer la méthode de triangulation.

1. Écrire un test permettant de vérifier que l'ajout d'une traduction au dictionnaire (`addTranslation`) se passe correctement lors de la vérification (`getTranslation`).

```
@Test public void testOneTranslation() {
    dict.addTranslation("contre", "against");
    assertThat(dict.getTranslation("contre"), equalsTo("against"));
}
```

2. Est-il possible de faire un test qui n'implique l'ajout que d'une seule de ces deux méthodes ?

3. Utiliser *Fake it* pour faire passer le test en faisant renvoyer à `getTranslation` la réponse attendue par le test.

4. Ici, notre test n'est pas suffisamment précis, et l'implémentation obtenue est correcte d'un point de vue des tests. Trianguler consiste ici à raffiner le test pour mieux cibler le comportement du code.

   Ajouter dans le test la vérification d'une seconde traduction qui soit différente de la première.

5. Maintenant, il faut faire un choix : soit se limiter à une implémentation-simulacre, soit ajouter un morceau de code capable d'effectivement gérer les traductions. C'est la deuxième solution que l'on choisit maintenant.

   Ajouter à la classe `Dictionary` une table de hachage `Map<String, String> translations`.

6. Rendre le code de `addTranslation` et de `getTranslation` correct.

7. Vu que l'on dispose à présent d'un moyen correct pour remplir le dictionnaire avec des traductions, il devient possible de s'occuper du cas du test du dictionnaire vide.

   Améliorer le test du vide du dictionnaire en augmentant le test initial.

Les fixtures de JUnit

L'écriture de tests dans la classe `Dictionary` fait souvent preuve de redondance. Il est possible de factoriser l'initialisation des tests que l'on réalise avec JUnit, et cela en utilisant encore les annotations Java :

```
@Before public void initialize () {
    dict = new Dictionary("Example");
}
```

Une méthode annotée avec `@Before` sera exécutée avant chaque test, et une méthode annotée avec `@After` à la fin de chaque test. Ces deux fonctionnalités permettent de mettre en place une installation (fixture) commune à tous les tests.

Prévoir une fixture pour l'ensemble des tests de la classe DictionaryTest.

Traductions multiples

L'une des spécificités d'un dictionnaire consiste à pouvoir manipuler des traductions multiples. Il s'agit d'un cas d'utilisation non prévu initialement dans notre architecture.

1. Que proposez-vous pour pouvoir gérer les traductions multiples ?
2. Ajouter un test permettant de vérifier le fonctionnement d'une traduction ayant deux sens possibles.

   Remarque : Il faut faire attention à ce que votre test ne dépende pas trop de l'implémentation à l'intérieur de la classe `Dictionary`. En particulier, l'ordre des éléments dans les listes ne doit pas être pris en compte lors du test (vous pourrez par exemple utiliser le *matcher* `containsInAnyOrder` de Hamcrest).
3. Proposer une implémentation simple, rapide et qui passe les tests existants afin d'atteindre la *green bar*.
4. Comment allons-nous procéder pour effectuer notre phase de refactoring ? Afin de garder les tests existants au vert, nous allons procéder de la manière suivante :

   4.1. Modifier `addTranslation` pour prendre en compte la nouvelle table de hachage, et écrire une nouvelle méthode `getMultipleTranslations` qui renvoie une liste de traductions.

   4.2. Adapter les tests utilisant `getTranslation` pour qu'ils utilisent `getMultipleTranslations`.

   4.3 Supprimer la méthode `getTranslation` et l'ancienne table de hachage, et refactorer pour renommer la méthode `getMultipleTranslations` en `getTranslation`.

Traduction inverse

Supposons maintenant vouloir prendre en compte les traductions dans les deux sens, comme pour un dictionnaire bilingue.

1. Écrire un test afin de vérifier le fonctionnement de telles traductions.
2. Proposer une implémentation simple, rapide et qui passe les tests existants afin d'atteindre la *green bar*.
3. Que pourrait-on proposer comme implémentation pour résoudre ce problème? Comparer vos propositions avec la solution rapide implémentée à la question précédente.

Chargement de fichier

Attaquons-nous maintenant au problème du chargement d'un dictionnaire à partir d'un fichier externe. Le format d'entrée des dictionnaires consiste en un fichier au format texte, dans lequel la première ligne représente l'identifiant du dictionnaire, et les lignes suivantes contiennent une chaîne de caractères `"␣=␣"` séparant les deux traductions d'un mot donné.

```
Example
contre = against
contre = versus
```

Assez naturellement, il va s'agir d'écrire un parseur pour ce format. Ce parser utilisera la classe `BufferedReader` qui permet de lire un fichier en mode ligne via la méthode `readLine`. Cependant nous avons besoin d'une interface correspondante et il n'en a pas de disponible dans l'API Java (étrangement ...). Nous nous baserons sur l'architecture suivante.

```
// API Java
public class BufferedReader ... {
    ...
    public String readLine() throws IOException {
        return reader.readLine();
    }
    ...
}

// Interface spécifique (non disponible pour BufferedReader dans l'API Java)
public interface ILineReader {
    public String readLine() throws IOException;
}

// Utilisation d'un adaptateur pour pallier le problème ci-dessus
public class LineReader implements ILineReader {

    private BufferedReader reader;

    public LineReader(Reader in) {
        reader = new BufferedReader(in);
    }

    @Override
    public String readLine() throws IOException {
        return reader.readLine();
    }
}
```

Pour des raisons de simplicité, le parseur prendra en argument un objet implémentant l'interface `ILineReader` (en théorie un `LineReader` mais on verra qu'on s'en passera par la suite).

L'architecture sera complétée comme suit :

```
public interface IDictionary {
    ... méthodes
}

public class Dictionary implements IDictionary {
    ... implantation
}

public class DictionaryParser   {
    public IDictionary loadTranslations(ILineReader reader) {
        ... implantation du parser qui utilise reader
    }
}
```

1. Il est possible d'écrire à partir de cette architecture une version du programme qui fonctionne et combine des objets des classes `Dictionary`, `DictionaryParser` et `LineReader` (ainsi que `BufferedReader` via l'adaptation). Quel type de test réalise-t-on dans ce cas ?

Les test suites de JUnit

A partir de maintenant, nous avons plusieurs classes à gérer. Pour permettre à JUnit de traiter les tests de manière uniforme, il est pratique d'écrire un ensemble de tests, sous la forme d'une test suite :

```
@RunWith(Suite.class)
@Suite.SuiteClasses({
    DictionaryTest.class,
})
public class AllTests { // Empty class ( introspection )
}
```

Pour lancer les tests associés à une suite de tests, utiliser :

```
./gradlew cleanTest test --tests nom.du.package.AllTests
```

Ecrivez cette suite de test et vérifiez que cela fonctionne.

Free wheeling

Dans la dernière partie de cette feuille, on applique la technique du TDD dans le cadre du test de composants. L'idée va consister à utiliser la technique des mock objects.

1. Créer une nouvelle classe `DictionaryParserTest` comme classe de tests JUnit, et la rajouter à `AllTests`.

La lecture de fichiers est un cas typique dans lequel les mock objects permettent de simplifier les tests. Dans notre cas, nous allons simuler la lecture de fichiers par un mock object mimant le résultat d'un lecteur de fichier en mode ligne.

2. Quel sera le type du mock object et pour quelles raisons ?

3. Mettre en place votre mock object (cf la doocumentation de Mockito).

   Vous pouvez vous inspirer du code suivant :

```
// interface correspondant à l'API de la classe à mocker (partie utilisée)
public interface I {
    T doSomething()
}

// classe qu'on va mocker (qu'on peut ne pas avoir)
public class C implements I {
    T doSomething() { ... implémentation ... }
}

// class qu'on teste
public class A {
    public A() { ... }
    public T someMethod(I i) { ... use i to get a T ... }
}

// tests unitaires utilisant des mock objects
public class Test {
    @Test public void testSomeMethod() {
        I mockObject = mock(I.class);
        T someT, someT1, someT2, ...;
        when(mockObject.doSomething()).thenReturn(someT1, someT2, ...);
        A object = new A();
        assertThat(object.someMethod(mockObject), equalTo(someT));
    }
}
```

   A partir de maintenant, il s'agit de construire la fonction `loadTranslations` en pratiquant une série de cycles de TDD, et en faisant passer les tests suivants un par un :

   3.1. le cas d'un fichier vide ;

   3.2. le cas d'un fichier avec seulement un nom ;

   3.3. le cas d'un fichier contenant une traduction ;

   3.4. le cas d'un fichier erroné (choix libre).

Couverture

1. JaCoCo (Java Code Coverage) et l'un des moyens de calculer la couverture du code dans les tests.

   Typiquement, que peut-on espérer concernant le taux de couverture atteint-on en utilisant la technique du TDD ?

2. Lancer le calcul des informations de couverture en rajoutant la tâche `jacocoTestReport` au lancement de `./gradlew`. Observez les résultats (ouvrir la page d'index HTML générée par JaCoCo). **Complétez au besoin les tests de `Dictionary` et `DictionaryParser`.**

Conclusion

Discuter des avantages et des inconvénients de la technique du TDD par rapport à vos techniques de développement usuelles.