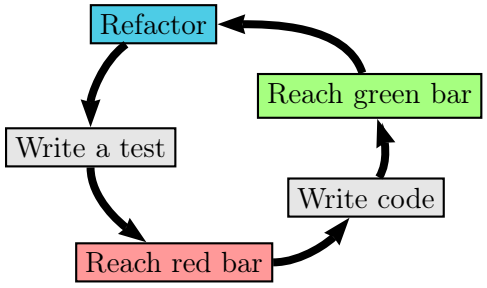
ENSEIRB
Année 2012-2013

3ème année
IT306

TD n°4 - Développement dirigé par les tests

Résumé : Cette séance décrit une méthode de développment nommée *développement dirigé par les tests*, qui s'appuie sur le principe d'écrire les tests avant le code. Cette technique est fortement défendue par K. Beck, l'un des développeurs principaux de JUnit [Bec03, Lin03].

La technique de développement dirigé par les tests (*test-driven development* ou TDD) est basée sur le cycle de travail (*workflow*) suivant :



Au long de cette feuille, nous allons appliquer pas à pas cette méthode. Tout l'intérêt du TDD consiste à suivre le cycle précédent dans l'ordre. Donc, toute déviation par rapport à l'ordre des questions dans cette feuille réduit l'intérêt de ce travail.

⚠ Pour les besoins de l'exercice, on utilisera Eclipse. De plus, il est nécessaire d'ajouter à l'espace de travail les fichiers externes jar externes suivants : JUnit, Hamcrest, Mockito (Menu *File* → *Properties* → *Java Build Path*) disponibles sur la page du cours.

Au cours de cette feuille, 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.

Le sujet de cette feuille 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 tâches adaptée à ce sujet.

Exercice 1: 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 :

```
1 @Test public void testDictionaryName(){
2   Dictionary dict = new Dictionary("Example");
3   assertThat(dict.getName(), is(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.

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".

3. 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.

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.

5. 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

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.

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

Exercice 2: 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).

```
1 @Test public void testOneTranslation() throws NotFoundException {
2   dict.addTranslation("contre", "against");
3   assertThat(dict.getTranslation("contre"), is(equalTo("against")));
4 }
```

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.

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.

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

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.

5. Ajouter à la classe Dictionary une table de hachage Hashmap<String,String>translations.
6. Rendre le code de addTranslation et de getTranslation correct.

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 :

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

Exercice 3: Un peu de JUnit : les *fixtures*

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 :

```
1 @Before public void initialize () {
2   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.

Exercice 4: 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.
3. Proposer une implémentation *simple*, *rapide* et *qui passe les tests existants* afin d'atteindre la green bar.

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. Modifier addTranslation pour prendre en compte la nouvelle table de hachage, et écrire une nouvelle méthode getMultipleTranslations qui renvoie une liste de traductions.
5. Adapter les tests utilisant getTranslation pour qu'ils utilisent getMultipleTranslations.
6. Supprimer la méthode getTranslation et l'ancienne table de hachage, et utiliser les outils de refactoring d'Eclipse pour renommer la méthode getMultipleTranslations en getTranslation (*Right-click* → *Refactor* → *Rename*).

Exercice 5: 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.

Exercice 6: 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.

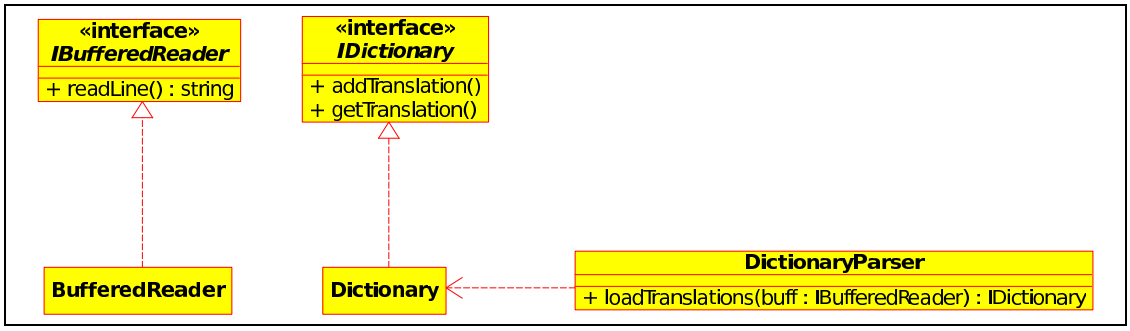
```
Example
contre = against
contre = versus
```

Assez naturellement, il va s'agir d'écrire un parseur pour ce format. Pour des raisons de simplicité, le parseur prendra en argument un objet implémentant l'interface IBufferedReader qui contient la fonction readLine().

1. Quelles types de solution (de conception logicielle) pouvez-vous envisager pour intégrer le parseur dans l'architecture existante ?

   Donner au moins deux solutions différentes de la solution proposée à la question suivante. Pour chaque solution, discuter ses avantages et inconvénients.

Pour la suite de l'exercice, on se propose d'utiliser l'architecture suivante :



2. Doit-on tester l'interaction entre DictionaryParser et Dictionary ?

Exercice 7: Encore un peu de JUnit : les *test suites*

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* :

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({
  DictionaryTest.class,
})

public class AllTests {  // Empty class (introspection)
}
```

A l'aide de méthodes d'introspection, JUnit repère à l'aide des noms des classes de tests indiquées par le mot-clé SuiteClasses l'ensemble des tests qu'il doit exécuter.

▷ Créer une *test suite* pour JUnit nommée AllTests dans Eclipse (Menu *File* → *New* → *Other*, puis sélectionner JUnit Test Suite).
Remplacer si nécessaire le code existant par les quelques lignes ci-dessus.

Exercice 8: 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* vue précédemment, en écrivant les tests en premier.

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 BufferedReader.

2. Quel sera le type du mock object et pour quelles raisons ?
3. Prévoir une Mockery pour générer les mock objects, et une Sequence pour contenir les séquences d'appels (utiliser une *fixture* pour initialiser vos tests).

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 :

4. le cas d'un fichier vide ;
5. le cas d'un fichier avec seulement un nom ;
6. le cas d'un fichier contenant une traduction ;
7. le cas d'un fichier erroné (choix libre).

Exercice 9: Conclusion

1. L'outil Eclemma permet 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. Discuter des avantages et des inconvénients de la technique du TDD par rapport à vos techniques de développement usuelles.

Références

[Bec03] K. Beck. *Test-driven development by example.* Addison-Wesley, 2003.

[Lin03] J. Link. *Tests unitaires en Java : les tests au coeur du développement.* Dunod, 2003.