Développement dirigé par les tests (TDD) (version IntelliJ)

Cet énoncé s'étend sur 5 pages.

L'objectif de ce TD est de développer une simple application *Pierre-Feuille-Ciseaux*, en utilisant le principe de TDD, avec le framework de test TestNG.

Une version en ligne de cette application très populaire est ligne sur le site du New York Times : http://www.nytimes.com/interactive/science/rock-paper-scissors.html

Situations du vainqueur (vainqueur à gauche) :

— **Pierre** vs. Ciseaux
— **Ciseaux** vs. Feuille
— **Feuille** vs. Pierre

1. Préparation de l'architecture

Les instructions de configuration spécifique indiquées dans cet énoncé ont été réalisées avec l'IDE IntelliJ IDEA 12 (édition Ultimate).

Créez un projet de type Java Module. Créez un package `fr.p10.miage.rps.model` dans lequel vous placerez les déclarations suivantes.

— Déclarez une énumération `RPSEnum`, qui définit les 3 mouvements possibles : ROCK, PAPER, SCISSORS
— Déclarez une énumération `Result`, qui définit les 3 résultats possibles : WIN, LOST, TIE
— Déclarez une classe `RockPaperScissors`, pour le moment sans constructeur, ou avec le constructeur par défaut vide. Dans cette classe, déclarez une méthode Result play(RPSEnum p1, RPSEnum p2), de laquelle vous retournerez un seul résultat : LOST.

Le framework de test que nous allons utiliser est `TestNG`, c'est une alternative à `JUnit` avec plus de fonctionnalités. Vérifiez dans la liste des plugins (IntelliJ IDEA > Preferences ...) qu'il est installé. Cherchez `TestNG` dans l'aide de l'IDE. Allez sur le site de `TestNG` (http://testng.org/) et notez l'existence des pages concernant la documentation, l'utilisation de l'IDE IntelliJ IDEA, et Maven (cela vous servira pour la suite des TD).

Concernant la couverture du code nous utiliserons `Coverage` et `JaCoCo`. Vérifiez dans la liste des plugins que `Coverage` est installé. Notez qu'`Emma` est une alternative que nous n'utiliserons pas (`JaCoCo` remplace `Emma`).

Lancer une application avec tests et couverture n'est pas bien différent de la façon habituelle dont vous utilisez votre IDE préféré. Pour lancer les tests uniquement il suffira de faire un click-droit sur le projet puis Run > All Tests (TestNG). Pour lancer les tests et la couverture du code par ces derniers faire un click-droit puis Run with Coverage > All Tests (TestNG). Il est aussi possible de procéder avec plus de précision en créant et éditant des configurations d'éxécution (ce que nous ferons aussi dans la suite).

Nous allons maintenant mettre en œuvre TDD sur cette application, pas à pas.

2. Classe TestRockPapersScissors

Crééz un répertoire `tests` au même niveau que `src` et signalez qu'il s'agit d'un répertoire de tests. Créez un package `fr.p10.miage.rps.tests` dans ce répertoire.

Convention. C'est une bonne pratique de nommer une classe de test d'après la classe qu'elle teste. Par exemple, si la classe à tester s'appelle `Foo`, alors la classe de test s'appellera `TestFoo` ou `FooTest`. Il en de même pour les méthodes à tester.

Créez une classe de test, `RockPaperScissorsTest`, en utilisant les actions d'intentions d'IntelliJ (alt-enter sur le nom de la classe à tester). Il est aussi possible bien sûr de créer les classes de tests manuellement : création d'une classe dans le répertoire des tests. Dans le cas présent, spécifiez que notre framework de test est `TestNG`, que le package pour les tests est `fr.p10.miage.rps.tests` et générez les méthodes de mise en place (`setUp`) et de finalisation (`tearDown`) des tests.

Si les annotations (@BeforeMethod et @AfterMethod) ne sont pas reconnues, utilisez les actions d'intention pour corriger le problème. Manuellement : ajout de testng.jar aux librairies du projet et import de `org.testng.annotations`. Comme nous voulons faire la mise en place et la finalisation pour l'ensemble des tests remplacez les annotations @BeforeMethod/@AfterMethod par @BeforeClass/@AfterClass puis :

1. définissez un attribut `rps` de type `RockPaperScissors` (ressource utilisée par les tests) dans la classe de test et initialisez le dans `setUp` ;
2. libérez les ressources (mettez `rps` à `null`) dans `tearDown` ;
3. créez une méthode de test `void testWinPlay(String p1, String p2)` avec l'annotation @Test. Elle va nous servir à tester un cas de victoire (`Resut.WIN`). Dans cette méthode, invoquez la méthode `play`. Utilisez `assertEquals` dans le code pour vérifier le résultat attendu :
   `assertEquals(rps.play(RPSEnum.valueOf(p1), RPSEnum.valueOf(p2)),Result.WIN)` Vous pouvez utiliser l'action d'intention pour importer ce qui manque ou plus simplement rajouter `import static org.testng.Assert.*;` au code.

   TestNG semble plus amical avec les String comme paramètres des méthodes de test qu'avec des objets utilisateur, qui ne sont acceptés que dans le cas des `Data Providers` (prochaine section de cet exercice). Il faut donc fournir à la méthode de test les labels des littéraux de l'énumération RPSEnum, puis les transformer en leur type avec `RPSEnum.valueOf(p1)` (idem pour p2) à l'invocation de la méthode `play`. La signature de cette méthode de test avec les paramètres devient donc :

   ```
   @Parameters({"paper","rock"})
   @Test
   void testWinPlay(String p1, String p2)
   ```

   Exécution des tests. Si vous exécutez les tests vous avez un message d'erreur. Normal, « paper » et « rock » ne sont pas définis.

   La version simple est d'éditer la configuration d'exécution et d'indiquer la valeur des paramètres : (paper, PAPER), (rock, ROCK), (scissors, SCISSORS). Faites-le puis relancez les tests. Cette fois ci les tests s'exécutent mais échouent (c'est normal). Pourquoi ?

   Une alternative (plus compliquée mais plus générale en termes d'utilisation de `TestNG` est de jouer sur le fichier de paramétrage de ce dernier, `testng.xml`.

En effet, TestNG supporte la configuration de l'exécution des tests par un fichier XML. Vous pouvez donc créer un fichier XML pour configurer l'exécution des tests, à la racine de votre projet. On suppose ce fichier nommé `testng.xml`, avec le contenu suivant (configuration donnée pour la méthode `testWinPlay`). **Attention** : le copier-coller du XML suivant pourrait vous poser ensuite des problèmes de formattage dans votre IDE. Vérifiez dans tous les cas que le XML est bien formé dans votre IDE (copier-coller ou réécrit par vous-même).

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd" >

<suite name="RPS" verbose="1">
        <parameter name="paper" value="PAPER" />
        <parameter name="rock" value="ROCK" />
        <parameter name="rock" value="SCISSORS" />

        <test name="testWinPlay">
                <classes>
                        <class name="fr.p10.miage.rps.test.TestRockPapersScissors">
                                <methods>
                                        <include name="testWinPlay"></include>
                                </methods>
                        </class>
                </classes>
        </test>
</suite>
```

La prochaine tâche sera de remanier le code de l'application de telle sorte que le test passe.

1. Remaniez le code de la méthode `play` de la classe `RockPaperScissors` de manière à faire passer le test précédent. En TDD, il faut faire le minimum requis pour que le test passe, donc se réfréner de trop en faire.
2. Exécutez encore le test précédent jusqu'à ce que le test passe. Une fois terminé, vous pouvez passer au prochain test.
3. Écrivez une nouvelle méthode de test, `void testTiePlay(String p1, String p2)` en utilisant la même démarche que plus haut. Fournissez à cette méthode les arguments pour lesquels vous attendez un résultat TIE (paper, ?), via l'annotation @Parameters (piochez les noms dans le fichier de configuration `testng.xml`). Utiliser `assertEquals` dans la méthode de test pour vérifier le résultat attendu.
4. Exécuter le test : il devrait échouer.
5. Remanier encore la méthode `play` (a minima) jusqu'à ce que le test passe.
6. Répétez la même démarche pour une dernière méthode de test (paper, ?), `void testLostPlay(RPSEnum p1, RSPEnum p2)`

Vous venez d'accomplir un cycle de TDD, en écrivant un test qui échoue, puis en remaniant le code jusqu'à ce que le test passe, et ainsi de suite, un test à la fois. Pour visualiser la couverture de code, passons à la section 3.

3. Non régression et couverture de code

Vous avez dû observer qu'au fur et à mesure que vous ajoutez de nouveaux tests, les précédents sont également exécutés avec les nouveaux. Cela est utile au minimum pour vérifier la non régression de code, où l'ajout d'une nouvelle fonctionnalité *casse* une fonctionnalité existante qui marchait bien jusqu'alors.

Cependant, la qualité du code testé est fonction de la qualité des tests et pas de leur nombre. Il est alors intéressant et important d'évaluer la couverture du code de votre application. `Coverage` vous sera très utile pour visualiser quels chemins du flux de contrôle de votre code ont été testés, de manière à vous permettre de déterminer plus facilement quels sont les nouveaux tests pertinents à écrire.

Visualisez la couverture de votre code par vos tests (normalement cela devrait être loin d'être parfait) et générez un rapport dans un répertoire `report` au même niveau que `src` et `tests`. Comparez l'utilisation et les résultats de `Coverage` et `JaCoCo`.

4. Data Providers

Cependant, avez-vous considéré tous les scénarios possibles ? Comment procéder pour tester plusieurs combinaisons possibles de valeurs ? Au lieu d'invoquer chaque test avec juste une paire de valeurs qui sont des chaînes de caractères, nous allons effectuer des tests en rafale avec des ensembles d'objets utilisateur autres que les String.

Nous allons maintenant étudier à cet effet une fonctionnalité intéressante de `TestNG`, les **Data Providers**. Un Data Provider est une méthode de la classe de test, qui retourne un tableau de tableaux d'objets. Cette méthode est annotée avec @DataProvider.

Précédemment, chaque méthode de test évaluait uniquement une configuration spécifique de jeu : Papier vs Pierre, ou Ciseaux vs Papier, etc. Maintenant, remaniez la signature des méthodes de test de telle sorte qu'elles acceptent les actions des joueurs comme paramètres, typés par RPSEnum et non plus par String. Cependant, nous allons conserver les méthodes précédentes et utiliser la surcharge pour accomplir cette nouvelle démarche. Par exemple :

```
@Test(dataProvider = "winData")
void testWinPlay(RPSEnum p1, RPSEnum p2)
```

Fournissez à chaque méthode de test un Data Provider (étudiez l'exemple dans la documentation de `TestNG`) :

— un Data Provider pour les 3 situations de WIN : Object[ ][ ] createWinData()

— un Data Provider pour les 3 situations de TIE : Object[ ][ ] createTieData()

— un Data Provider pour les 3 situations de LOST : Object [ ][ ] createLostData()

1. Exécutez les tests. Très probablement si vous avez bien modifié le code de `play` a minima, plusieurs tests ne devraient pas passer. Comment trouver l'erreur ? (Attention au copier-coller de tests, les tests peuvent être faux aussi ...).
2. corrigez `play`. Tous les tests devraient passer pour chaque méthode de test, avec le Data Provider que vous lui avez assigné.
3. Pour vous assurer que vos tests sont bien conçus, échangez les Data Providers entre les 3 méthodes de tests, de telle sorte que tous les tests échouent. Par exemple, assignez le Data Provider WIN à la méthode `testTiePlay(RPSEnum p1, RSPEnum p2)`, et assurez-vous que les tests exécutés par cette méthode échouent. Et ainsi de suite.
4. vérifiez que la couverture du code est bien satisfaisante et générez un second rapport (vous écraserez le précédent)

5. Session de jeu

Poussons un peu plus loin les tests pour gérer une logique applicative plus élaborée. L'intention est d'écrire une session de jeu dans laquelle deux joueurs effectuent un certain nombre de rounds, puis un score global est calculé pour déterminer le vainqueur.

Par exemple pour un joueur donné J1 :

— 10 rounds joués, 6 gagnés => gagné ;

— 10 rounds joués, 5 gagnés => égalité ;

— sinon, perdu

Créez une session de jeu en utilisant deux `Joueur`, chacun avec une collection prédéterminée de mouvements (choisis au hasard, ou explicitement spécifiés) :

1. Déclarez une classe `Player`, caractérisée par un nom, un score et une collection de mouvements (une liste d'éléments de type RPSEnum).
2. Ajoutez un constructeur prenant en paramètre le nom d'un joueur et une collection de mouvements.
3. Ajoutez un deuxième constructeur prenant juste le nom du joueur en paramètre. Vous générerez la collection de mouvements avec un nombre fixé de mouvements choisis au hasard. Le nombre de mouvements pourrait être au moins 10, ou 20, etc. Pour faciliter la génération, associez un nombre entre 0 et 2 à chaque élement RPSEnum ; puis, choisissez l'élément RPSEnum correspondant au nombre généré.
4. Fournissez des getters à `Joueur` : nom, score, le nombre fixé de mouvements et une méthode `RPSEnum getNextMove()` qui retourne le prochain élément de la collection des mouvements du joueur. Ne fournissez pas de getter pour la collection des mouvements.

5. Fournissez un setter à `Joueur` pour l'attribut `score`.
6. Ajouter une méthode `Result play(Player p1, Player p2)` à la classe `RockPaperScissors`, qui retourne juste LOST pour le moment.

Écrivez progressivement les tests de la session de jeu :

- Écrivez de nouvelles méthodes de test pour vérifier que, étant donné un ensemble de mouvements connus pour chaque joueur, vous obtenez le résultat attendu pour le joueur 1 (win, tie, lose).
  - Initialisez la collection de mouvements pour chaque joueur dans la méthode de configuration `setUpClass()`.
  - Libérez les ressources dans la méthode `tearDownClass()`.
- Exécutez le test : il devrait échouer.
- Maintenant remaniez la méthode `play(Player p1, Player p2)`. Créez une boucle dans laquelle :
  - vous invoquez la méthode `play(RPSEnum p1, RPSEnum p2)` pour chaque paire d'éléments RPSEnum que vous récupérez des joueurs 1 et 2 respectivement à travers leur méthode `getNextMove()` ;
  - vous positionnez leur score (+1 aux deux si TIE, +1 au vainqueur sinon) ;
  - vous retournez le résultat WIN si p1 a gagné, LOST si p1 a perdu, TIE sinon.
- Exécutez le test et remaniez le code autant que nécessaire pour que le test passe.
- Écrivez le test suivant selon la même démarche, et ainsi de suite jusqu'à ce que les trois tests pour win, tie et lost passent.

Félicitations, si vous avez suivi les instructions et développé l'application à travers la démarche exposée dans cet exercice (écrire un test qui échoue, remanier le code jusqu'à ce que le test passe), sans raccourci, alors vous êtes sur la bonne voie pour devenir un développeur TDD.

Désormais, adoptez cette démarche aussi souvent que possible pour vos projets de développement d'application.

6. Spock

En utilisant cette fois-ci le framework de test Spock, réécrivez les tests dans sa syntaxe de Spock.

Ressources en ligne :

- Spock home page
- Spock vs. JUnit
- Spock testing framework versus JUnit
- Spock Primer