M2104 - TD n°1 – Révisions : Mise en place de bonnes pratiques de développement autour du refactoring du Kata Tennis

Il vous est conseillé de faire ce TD en pair-programming 😉

Exercice 1 : Revue de code (à la recherche de *code smell*)

Dans le cadre de ce TD, cet exercice est à faire sur papier.
Les ordinateurs doivent être fermés pour le moment !!!

Prenez connaissance du code suivant …

```
public interface TennisGame {
    void wonPoint(String playerName);
    String getScore();
}
```

```
public class TennisGame1 implements TennisGame {

    private int m_score1 = 0;
    private int m_score2 = 0;
    private String player1Name;
    private String player2Name;

    public TennisGame1(String player1Name, String player2Name) {
        this.player1Name = player1Name;
        this.player2Name = player2Name;
    }


     public void wonPoint(String playerName) {
        if (playerName == "player1")
            m_score1 += 1;
        else
            m_score2 += 1;
    }

    public String getScore() {
        String score = "";
        int tempScore=0;
        if (m_score1==m_score2)
        {
            switch (m_score1)
            {
                case 0:
                        score = "Love-All";
                    break;
                case 1:
                        score = "Fifteen-All";
                    break;
                case 2:
                        score = "Thirty-All";
                    break;
                default:
                        score = "Deuce";
                    break;
            }
        }
        else if (m_score1>=4 || m_score2>=4)
        {
            int minusResult = m_score1-m_score2;
            if (minusResult==1) score ="Advantage player1";
            else if (minusResult ==-1) score ="Advantage player2";
            else if (minusResult>=2) score = "Win for player1";
            else score ="Win for player2";
        }
        else
        {
            for (int i=1; i<3; i++)
            {
                if (i==1) tempScore = m_score1;
                else { score+="-"; tempScore = m_score2;}
                switch(tempScore)
                {
                    case 0:
                        score+="Love";
                        break;
                    case 1:
                        score+="Fifteen";
                        break;
                    case 2:
                        score+="Thirty";
                        break;
                    case 3:
                        score+="Forty";
                        break;
                }
            }
        }
        return score;
    }
}
```

1.1 Que fait ce code ?

Ce code pourrait être du code existant que vous venez d'hériter d'un collègue développeur et qui aurait été écrit à partir du cahier des charges suivants :
(code et énoncé en anglais - à lire ci-dessous- extrait de : https://github.com/emilybache/Tennis-Refactoring-Kata)

Imagine you work for a consultancy company, and one of your colleagues has been doing some work for the Tennis Society. The contract is for 10 hours billable work, and your colleague has spent 8.5 hours working on it. Unfortunately he has now fallen ill. He says he has completed the work, and the tests all pass. Your boss has asked you to take over from him. She wants you to spend an hour or so on the code so she can bill the client for the full 10 hours. She instructs you to tidy up the code a little and perhaps make some notes so you can give your colleague some feedback on his chosen design. You should also prepare to talk to your boss about the value of this refactoring work, over and above the extra billable hours.

Tennis has a rather quirky scoring system, and to newcomers it can be a little difficult to keep track of. The tennis society has contracted you to build a scoreboard to display the current score during tennis games.
Your task is to write a ***`TennisGame`*** *class containing the logic which* ***outputs the correct score as a `String` for display*** *on the scoreboard.*
When a player scores a point, it triggers a method to be called on your class letting you know who scored the point. Later, you will get a call ***`score()`*** *from the scoreboard asking what it should display.*
This method should ***return a `String` with the current score****.*

You can read more about Tennis scores here which is summarized below:

1. *A game is won by the first player to have won at least four points in total and at least two points more than the opponent.*
2. *The running score of each game is described in a manner peculiar to tennis: scores from zero to three points are described as "Love", "Fifteen", "Thirty", and "Forty" respectively.*
3. *If at least three points have been scored by each player, and the scores are equal, the score is "Deuce".*
4. *If at least three points have been scored by each side and a player has one more point than his opponent, the score of the game is "Advantage" for the player in the lead.*

You need only report the score for the current game. Sets and Matches are out of scope.

1.2 Votre travail consiste maintenant à réaliser une revue de code **c-a-d** relire, essayer de comprendre le code précédent et identifier les mauvaises pratiques de conception (***code smells*** : mauvaises odeurs dans le code) que vous détectez en examinant ce code.
A la manière d'une correction, annotez directement sur les pages 1 et 2 de cet énoncé, les mauvaises pratiques qui vous sautent aux yeux lors de la lecture du code…

Exercice 2 : Eviter la non-régression par la mise en place d'un harnais de tests
(garantir le comportement par les tests)

Rappelons la définition du refactoring :

Un refactoring (remaniement) consiste à changer la structure interne d'un logiciel sans en changer son comportement observable *(M. Fowler)*

Pour garantir le comportement du système, il est donc indispensable, avant de procéder à un quelconque refactoring, de disposer d'un ensemble de tests automatisés. Ces tests serviront de *filet de sécurité* et permettront de remanier le code en toute sécurité : ce sont des tests de régression qui, pouvant être exécutés à tout moment, préviennent la non-régression du logiciel.

2.1 Ecrire les premiers **tests unitaires :**

Cet exercice est à faire sur papier.
Les ordinateurs doivent être fermés pour le moment !!!

En respectant le pattern ***AAA*** (***A***rrange ***A***ct ***A***ssert), écrire les premiers tests unitaires qui permettraient de couvrir le comportement de ce code :

→ Mettre en place un premier test pour vérifier le scénario "`Love-All`"
→ Mettre en place un deuxième test pour vérifier le scénario "`Fifteen-Love`"
→ Mettre en place un troisième test pour vérifier le scénario "`Fifteen-All`"
→ Mettre en place un quatrième test pour vérifier le scénario "`Thirty-Fifteen`"
→ …

2.2 **Améliorer la lisibilité des tests !**

Difficile d'écrire des tests sans pouvoir les exécuter et récupérer leur verdict !

Il est donc grand temps de récupérer le code et de travailler sur machine !

2.2.a **Mise en place du projet dans l'IDE** :

Dans votre IDE préféré, créez un projet Maven `katatennis`.

Récupérez dans **src/main/java** les codes source de `TennisGame` et `TennisGame1` disponible dans le répertoire **ressources** `katatennis` de ce dépôt
https://github.com/iblasquez/enseignement-iut-m3105-conception-avancee.

Faites en sorte que ce code compile !

Le formatage de ce code ne convient peut-être pas à certains d'entre vous car il ne correspond pas à la *forme* des programmes que vous avez l'habitude d'écrire avec votre IDE préféré (en terme d'accolade, d'indentation…)…
Pas de problème, commencez donc pas reformater ce code de manière à ce qu'il corresponde mieux à nos standards habituels. Pour cela, Sélectionnez donc tout le code de la classe `TennisGame1`, puis à partir de la barre des menus sous Eclipse sélectionnez **Source->Format.**

Faites en sorte que ce code compile !
(normalement le formatage n'a rien changé et le code doit continuer à compiler !)

2.2.b **Des tests paramétrés pour une meilleure lisibilité !**

Récupérez dans **src/test/java** le code de test `TennisTest` disponible dans le répertoire **Ressources** `katatennis` de ce dépôt :
https://github.com/iblasquez/enseignement-iut-m3105-conception-avancee.

Faites en sorte que ce code compile et exécutez les test !
Vous devez avoir une barre verte !!!

Jetez un petit coup d'œil sur le fichier de tests : `TennisTest`

Pour faciliter l'écriture des tests et en améliorer leur lisibilité, des tests paramétrés ont été écrits pour couvrir le comportement de code (qui sinon aurait nécessité l'écriture d'un ensemble de tests très répétitifs comme l'a montré la question 2.1). Le concept de tests paramétrés est une solution pour réduire la duplication (du code de test) dans le cas d'un comportement répétitif à tester.

Zoom sur la notion de tests paramétrés (mis en œuvre via JUnit4) :

Le framework **Junit** offre la possibilité d'écrire des tests paramétrés **via le runner** `Parameterized` (classe avec laquelle les tests unitaires sont lancés) et des annotations.

→ tout d'abord, il faut annoter la classe de tests à l'aide de l'instruction : `@RunWith(Parameterized.class)` pour pouvoir utiliser ce **runner** sur vos tests.

→ Les **paramètres du test** (*donnée(s) en entrée et résultat(s) attendu(s)*) doivent ensuite être décrits dans une collection d'objets (sorte de tableau) au sein d'une méthode `static` annotée de : `@Parameters` dont la signature est : `public static Collection<Object[]> data() .`

→ Vous avez sans doute également remarqué la présence d'un constructeur. Ce constructeur a une signature composée d'autant de paramètres que ceux à transmettre à chaque test. Ces paramètres doivent être *bien* nommés pour illustrer au mieux le *rôle* qu'il vont jouer dans le test.
Ce constructeur permet d'instancier les attributs : ce qui explique pourquoi il est nécessaire de déclarer autant d'attributs que de paramètres à transmettre aux tests.
Il est à noter que c'est via ce constructeur, que le runner `Parameterized` va pouvoir injecter les valeurs des paramètres (fournies dans la méthode `data` qui dans notre contexte est devenue `getAllScores`) dans des attributs de classe afin de rendre ces dernières accessible aux méthodes de test.

→ enfin, il ne reste plus qu'à créer la méthode de test (annoté de `@Test`) en utilisant les attributs de classe dans le code du test pour le *paramétrer* selon vos besoins.

Pour information, la documentation sur les tests paramétrés est disponible ici : https://github.com/junit-team/junit4/wiki/Parameterized-tests

2.2.c Vérifier la **Couverture de code par les tests** via un outil genre EclEmma :

Avoir des tests c'est bien, connaître la couverture de code par ces tests peut parfois s'avérer interessant, pour savoir quels comportements sont bien couverts par les tests.

Pour connaître la couverture de code (Java Code Coverage) sous Eclipse, le plug-in EclEmma peut être utilisé (normalement vous l'avez déjà installé l'année dernière, sinon vous pouvez le ré installer plus tard via l'Eclipse Market : https://marketplace.eclipse.org)

Lancez le calcul de couverture (Coverage As -> JUnit Test sous Eclipse).
Quelle est le pourcentage obtenu ?

… Vous couvrez donc tout le comportement de votre code…
Vous pouvez donc refactorer en toute sécurité 😉

Exercice 3 : Mise en place d'un gestionnaire de version sur le projet

Pour avoir une démarche professionnelle,
tous vos projets/exercices devront dorénavant être versionnés !
(et ceci sans que cela soit forcément rappelé dans les consignes 😉)

Le gestionnaire de version à utiliser pour ce module est **Git.**

Mettre en place en local une gestion de version via Git sur ce projet :
- ➔ Soit directement en ligne de commande
- ➔ Soit via egit, en reprenant par exemple le mémo suivant : https://github.com/iblasquez/tuto_git/blob/master/egit/git_egit_memo.md

Committez une première fois, avec par exemple comme message :
« **Premier commit : code à refactorer avec tests** »

Exercice 4 : *Inspecter* la qualité de code rapidement dans votre IDE à l'aide de SonarLint

Pour compléter les *code smell* vous avez identifié dans le premier exercice, nous allons utiliser maintenant un outil d'inspection de code (**SonarLint**) qui pourra vous aider à refactorer et à améliorer la qualité de votre code.

Le site de **SonarLint** est : http://www.sonarlint.org/.
SonarLint est un plug-in de SonarSource à installer dans votre IDE préféré (Eclipse, IntelliJ, Visual Studio, Visual Studio Code, Atom).
Il permet de réaliser *en continu* une analyse syntaxique de votre code en vérifiant le respect de certaines règles de programmation (par défaut règles d'analyse SonarQube qui peuvent bien sûr être personnalisées).

Si **SonarLint** n'est pas déjà installé dans votre IDE, c'est le moment !
Pour Eclipse direction l'Eclipse Market (https://marketplace.eclipse.org) pour installer SonarLint dans votre IDE.

Une fois installé, lancez **SonarLint** (placez-vous dans la vue **Package Explorer** sur le projet : **katatennis**) à l'aide d'un clic droit, sélectionnez **SonarLint** puis **Analyze**

SonarLint ouvre la vue **SonarLint Report** : elle contient une liste de *code smell* qui ont été détectés lors de l'analyse du projet via SonarLint. Ces *code smell* sous Sonar sont aussi appelés **Issues**.
Certaines issues sont vertes : ce sont des issues mineures.
D'autres issues sont rouges : ce sont des issues majeures.

Pour vous rendre immédiatement dans votre projet à la ligne de code concerné, double-cliquez sur l'issue. Par exemple, double cliquez sur :
« **Rename this field "m_score1" to match the regular expression '^[a-z][a-zA-Z0-9]*$'.** »
et vous vous retrouverez sur l'instruction suivante : `private int m_score1 = 0;`
Vous noterez au passage que `m_score1` est souligner de vaguelettes bleues : ainsi même sans consulter la vue **SonarLint Report**, on peut détecter à la lecture du code d'éventuels code smells grâce à ce marqueur bleu 😉

Vous avez surement remarqué qu'une autre vue (**SonarLint Rule Description**) s'est ouvert juste à côté. Si vous cliquez sur l'onglet correspondant vous obtiendrez des exemples qui vous expliqueront pourquoi votre code ne respecte pas la *bonne* règle de codage ayant donné lieu à l'issue. Cela vous aidera à corriger vos erreurs 😉

La ***Compliant Solution*** de la ***Field names should comply with a naming convention*** (squid:S00116) montre que le CamelCase doit être préféré aux underscores (_) pour le nommage des variables.

Qu'à cela ne tienne, revenez dans le code. Sélectionnez `m_score1`, faites un clic droit **Refactor -> Rename** ou (directement Alt+Shift+R), puis changez le nom de cette variable en `mScore1`

Est-ce que la vue SonarLint Report a été remise à jour ? Non …
Cette vue n'est remise à jour qu'à chaque fois que l'on lance Analyze…

…Mais il existe une autre vue **Sonar Lint On the Fly**, qui elle est remise à jour à la volée…
Pour ouvrir cette vue, allez dans **Window -> Show view -> Other** … et cliquez **sur Sonar Lint On the Fly** et constatez que l'issue que nous venons de corriger a disparu dans cette vue de la liste des issues. Mieux vaut donc certainement que vous restiez sur cette vue pour que vous suivez l'évolution de vos « issues en temps réel »

Remarque : *Si vous ne souhaitez pas que **SonarLint** inspecte votre code en permanence, il est possible de lui demander d'arrêter de faire cela en se rendant dans les **Properties** du projet propre à **SonarLint** et en décochant : **Run SonarLint automatically**… mais nous ne le ferons pas cela pour le moment car nous souhaitons garder cette option cochée pour assurer une **inspection continue de notre code** !!!*

ATTENTION !!!
En inspectant le code Sonar a relevé la violation d'une règle de codage : ***Field names should comply with a naming convention***
En remplaçant `m_score1` par `mScore1`, l'issue a été levée…
Par contre pensez-vous que `mScore1` montre bien l'intention du code ? **Sonar peut soulever des erreurs par rapports à des règles pré-programmées** (c-a-d des standards de codage - à savoir que vous pouvez personnalisé vos propres règles de codage), **mais il ne va pas vous détecter tous les code smells !!!** Une revue manuelle est bien sûr toujours nécessaire !!!
N'oubliez pas lors du refactoring à venir de renommer cette variable de manière à ce qu'elle montre bien son intention dans le code !!!!

Travail à faire :
Prenez connaissance des différentes issues soulevées par SonarLint : elles pourront vous guider dans votre refactoring à venir…
N'essayez pas de corriger ces issues pour le moment….
La dernière issue majeure souligne une **complexité cyclomatique importante** : pour l'instant vous ne pouvez que constater ce fait : c'est ce que vous mettrez en œuvre au cours de votre refactoring qui vous permettra de faire disparaître cette issue 😉

Exercice 5 : Vous êtes enfin prêt … Refactoring Let's go !

Refactorez le code de `TennisGame1` pas à pas en essayant de corriger les code smells que vous avez identifiés et en vous aidant des issues soulevées par Sonar au fur et à mesure de votre refactoring…

N'oubliez pas de sauvegarder et de relancer les tests à chaque petite modification de code…
Vous êtes en train de refactorer, assurez-vous que le comportement est toujours garanti !!!
Veillez à **commiter fréquemment avec des messages explicites** pour suivre l'évolution de votre refactoring !!!

Remarque : Quelques pistes pour vous aider à refactorer …
- ➔ La méthode `getScore` est trop longue : essayez de commencer par extraire du comportement !
- ➔ Renommez pour rendre le code explicite
- ➔ Intéressez-vous ensuite à la simplification des conditions
- ➔ Sans oublier les code smells !

… mais gardez à l'esprit qu'un refactoring est propre à chaque développeur et donc peut varier d'un binôme à l'autre, le but étant pour chacun d'aller vers un code plus lisible, plus facile à maintenir et à étendre 😉

Exercice 6 : En fin de séance, pousser le projet sur un dépôt distant …

Sur votre compte **Github** (ou Gitlab), créer un nouveau **dépôt public** que vous appellerez **kata-tennis**

Faites en sorte de publier votre projet sur ce dépôt distant public :
- ➔ Soit directement en ligne de commande
- ➔ Soit via **egit** : en reprenant par exemple le mémo suivant : https://github.com/iblasquez/tuto_git/blob/master/egit/git_egit_memo.md

Les bonnes pratiques rappelées au travers de ce TD sont :

- **pair-programming** 😉
- **revue de code**
- Eviter la non-régression par la mise en place d'un harnais de **tests** *(garantir le comportement par une bonne couverture de code par les tests)*
- Soigner la **lisibilité des tests** *(pattern AAA, tests paramétrés, meilleure expressivité des assertions à l'aide de framework tel que AssertJ ou …)*
- **Versionner** votre projet
- **Qualité du code** : respecter les règles de codage et consulter quelques métriques

Pour les plus rapides…

Le code sur lequel vous venez de travailler est un extrait du **kata de refactoring Tennis** proposé par Emily Bache sur son dépôt Github : https://github.com/emilybache/Tennis-Refactoring-Kata
Dans ce kata, plusieurs exemples de code de production sont proposés, dans l'idée que ces codes auraient pu être écrits par des développeurs de niveaux différents (`TennisGame1` par un développeur débutant et `TennisGame3` par un développeur expérimenté).
Vous venez de refactorer `TennisGame1`, vous pouvez maintenant essayer de refactorer `TennisGame2` et `TennisGame3` 😉