UMONS
Université de Mons

Projet de Software Evolution - JPacman

Réalisateurs :
Damien Legay
Adrien Coppens
Nicolas Leemans

Enseignant : M. Tom Mens
Date de remise : 9 mai 2016
Année d'étude : Master 1

Année académique 2015 - 2016

Table des matières

1 Introduction

Ce projet, effectué dans le cadre du cours de "Software Evolution" dispensé par Monsieur Tom Mens durant l'année académique 2015-2016, a pour but de mettre en pratique les concepts d'évolution logicielles vus au cours théorique. Il consiste à analyser et à étendre un projet en effectuant un contrôle de la qualité au travers de différentes métriques, le *"refactoring"* du code source, ainsi que l'implémentation de nouvelles fonctionnalités. Le projet concerné s'appelle JPacman [1]. Il s'agit d'une implémentation très basique du jeu Pacman en Java, créé par l'équipe du professeur Arie van Deursen, Delft University of Technology (Pays-Bas). JPacman contient plusieurs simplifications par rapport au jeu Pac-Man original. Le jeu consiste à déplacer Pac-Man, un personnage qui, vu de profil, ressemble à un diagramme circulaire à l'intérieur d'un labyrinthe, afin de lui faire manger toutes les pac-gommes qui s'y trouvent en évitant d'être touché par des fantômes.

Ce rapport s'organise en plusieurs chapitres : dans un premier chapitre, ...

2 Extension du projet et ajout de tests unitaires pour cette extension

2.1 Extension du logiciel

La première partie de ce projet consistait à étendre la version initial de JPacman en ajoutant de nouvelles fonctionnalités et en suivant un processus de développement dirigé par les tests. De nouveaux tests unitaires ont donc été ajoutés pour chaque fonctionnalité afin de vérifier que le comportement initial du logiciel n'a pas été altéré.

Chaque membre du groupe a donc implémenté une des fonctionnalités suivantes :
- L'implémentation d'un score (réalisé par Damien Legay)
- L'implémentation d'une série de labyrinthes (réalisé par Adrien Coppens)
- L'implémentation d'une intelligence artificielle pour pacman (réalisé par Nicolas Leemans)

2.1.1 Fonctionnalité "Score"

2.1.2 Fonctionnalité "Série de labyrinthes"

2.1.3 Fonctionnalité "IA pour Pacman"

L'objectif de cette fonctionnalité est d'intégrer, au code existant, une intelligence artificielle pour Pacman afin qu'il puisse jouer de façon autonome tout en optimisant son score. En début de partie, le joueur doit pouvoir choisir entre contrôler Pacman manuellement ou être un spectateur passif de la partie en choisissant l'intelligence artificielle qui contrôlera Pacman à la place du joueur. Pour mettre en place cela, il a été recommandé d'utiliser un design pattern nommé "Strategy". Le design pattern "Strategy" consiste à définir un comportement (appelée "strategie") qui va permettre de différencier l'utilisation entre l'IA ou le contrôle manuel. Ce design pattern offre également une flexibilité pour modifier la stratégie suivie très facilement.

1. https ://github.com/SERG-Delft/jpacman-framework

3 Refactorings avant analyse

3.1 Fusion des tests sur « Player »

3.2 Déplacement de la variable retenant le niveau courant de « Level »vers « Game »

3.3 Déplacement des méthodes concernant le « map parser »de « Launcher »vers « Game »

4 Analyse de la qualité du code source

Dans ce chapitre, nous allons comparer la qualité du code source qui intègre toutes les extensions individuelles avec la qualité du code source de la version de départ de JPacman. Pour pouvoir effectuer cette comparaison, il va, tout d'abord, falloir effectuer des analyses sur la qualité du code source des deux versions en utilisant différents types d'analyses et de techniques. Pour effectuer cette analyse, nous ferons appel à plusieurs outils d'analyse de qualité que nous détaillerons par la suite. Cette phase d'analyse se déroulera en trois étapes : une analyse statique et dynamique du code ainsi qu'une analyse de la qualité par plusieurs métriques logicielles qui peuvent aider à déceler de mauvaises pratiques.

5 Analyse statique

5.1 Code dupliqué

Puisque nous utilisons tous les 3 IntelliJ IDEA, l'outil intégré a été utilisé dans un premier temps. La figure 1 montre les résultats obtenus via cette analyse. On peut noter que les détections ayant un « coût » inférieur à ∼20 ne sont pas réellement préoccupantes. Pour exemple, les lignes suivantes, extraites de *SquareCoordinateTest*, sont considérées comme dupliquées par l'outil avec un score de 10 :

```
assertEquals(square.getSquareAt(Direction.WEST).getY(), 15);
assertEquals(square.getSquareAt(Direction.EAST).getY(), 15);
```

Il s'agit en effet de 2 lignes très similaires mais il ne nous a pas semblé intéressant de supprimer ce type de duplicat. Dans un second temps, nous avons analysé le code via *CPD* inclus dans *PMD* et qui était utilisé dans la suite de rapports à générer par *Maven*. Le seul dupliqué signalé par cet outil concerne la classe *AStarPathTest* pour laquelle les méthodes *hTest* et *gTest* contiennent en effet toutes deux ce bloc de code :

```
final AStarPath aStarPath = new AStarPath(game);

assertNotNull(aStarPath);
final Player player = game.getPlayers().get(0);
final Square square = player.getSquare();

assertNotNull(player);
assertNotNull(square);
```

```
final Square origin = player.getSquare();
final Square destination = player.getSquare().getSquareAt(Direction.EAST);
final Square destination2 = player.getSquare().getSquareAt(Direction.EAST).

final Square destination3 = player.getSquare().getSquareAt(Direction.WEST);
final Square destination4 = player.getSquare().getSquareAt(Direction.WEST).
```

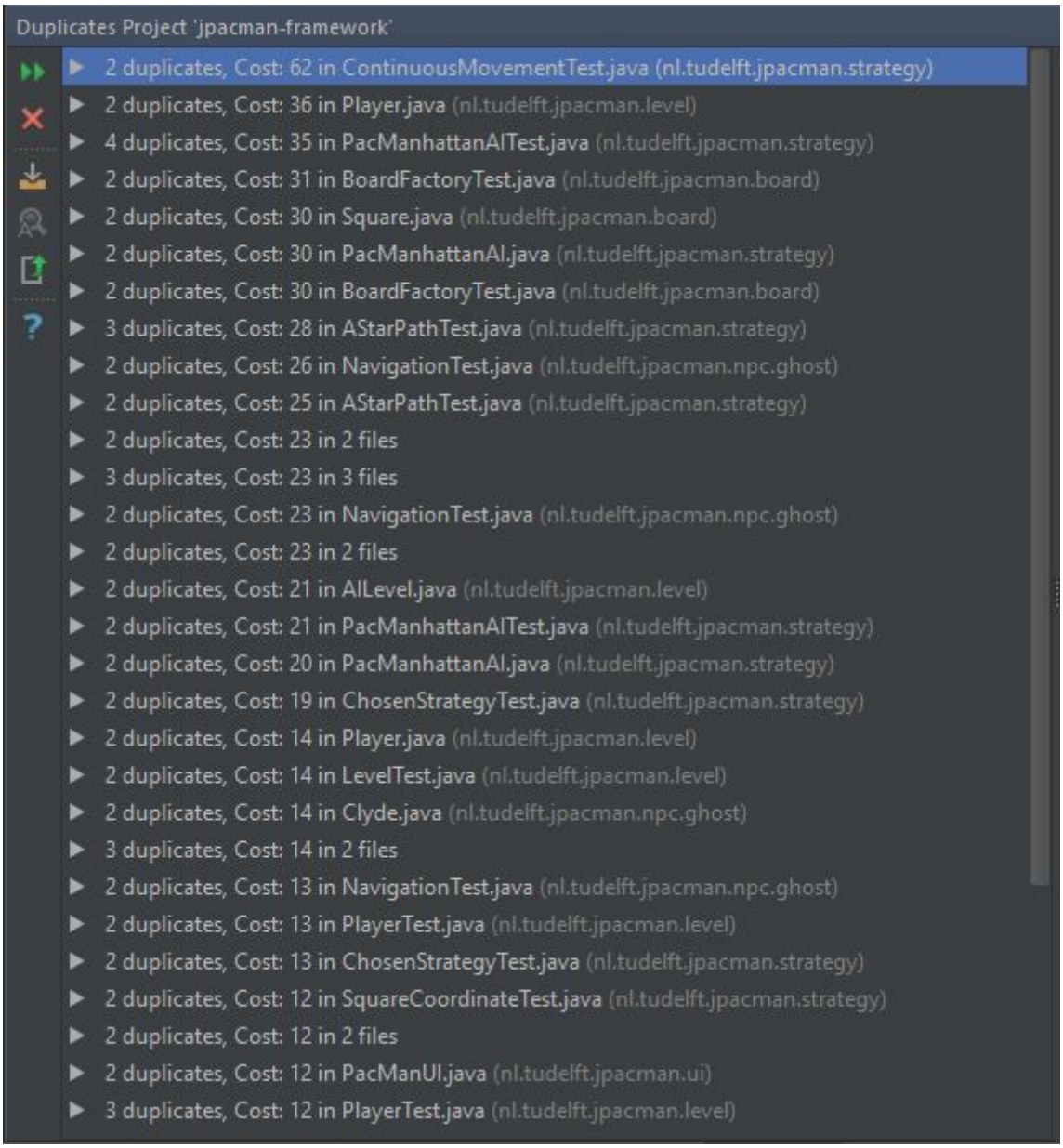

FIGURE 1 – Recherche de code dupliqué via l'outil intégré à *IntelliJ IDEA*

5.2 CheckStyle

Egalement intégré dans la suite d'analyses à effectuer via *Maven*, *CheckStyle* a été utilisé avec le « ruleset » présent dans la version du code d'origine. Aucune erreur n'a été détectée mais de nombreux « warnings » sont cependant présents (plus de 500). Par ordre de nombre de « violations » :

— 148 violations de type *JavadocStyle* : en réalité toutes des « First sentence should end with a period. »→ réglé.

— 110 violations de type *MagicNumber* → constantes extraites là où cela avait du sens, sauf pour les tests où un tel refactoring nous semblait inutile, nous avons donc supprimé les « warnings » pour ceux-ci.
— 104 violations de type *LineLength* → retours à la ligne là où c'était nécessaire.
— 63 violations de type *NeedBraces* → bien que nous ne soyons pas tous d'accord sur la valeur ajoutée d'une telle convention, nous avons ajouté les crochets là où *CheckStyle* le demandait.
— 35 violations de type *AvoidStarImport* → encore une fois désactivés car nous utilisons la fonction « optimize imports » d'*IntelliJ IDEA* qui regroupe parfois des imports en un unique via cette notation.
— Des violations relatives à des éléments de javadoc manquants → ajoutés.
— Des violations relatives à des tableaux déclarés à la « mode C » plutôt qu' à la « mode Java »→ modifiés.
— D'autres violations plus « isolées » non reprises ici.

5.3 IntelliJ Code inspection

5.3.1 IntelliJ dit que la condition impliquant QUICK_WIN est « pointless »

5.3.2 Changements mineurs (« scope »de méthodes/variables, variables qui peuvent être « final »)

5.3.3 Dans le code de base, beaucoup de « warnings »sur des « problèmes de modernité »

5.4 PMD

5.4.1 Déplacement des méthodes liées à la récupération des cases « safe »pour téléporter le joueur de Level vers Board

5.4.2 Problème détecté par PMD : Level == godclass et PacManhatanAI aussi

6 Analyse dynamique

7 Mesure de la qualité du logiciel

8 Notes diverses et guide d'utilisation

Le projet utilisant Maven [2], il est très facile de compiler le code, de l'analyser avec les outils prévus ou d'exécuter la suite de tests. En considérant que cet outil est correctement installé sur votre machine et en plaçant un terminal (une invite de commande ou autre terme décrivant une « console » sur votre système d'exploitation) à la racine du projet (l'endroit où se trouve le fichier "pom.xml"), vous pouvez :

— Exécuter les tests via :

```
$ mvn test
```

— Compiler le projet en produisant une archive "jar" exécutable (inclus l'exécution des tests) via :

2. https://maven.apache.org/

```
$ mvn package
```

— Générer le site Maven (inclus les étapes précédentes, la javadoc et des rapports d'analyse ; le site généré étant accessible en ouvrant `target/site/index.html`) via :

```
$ mvn site
```

— D'autres commandes sont évidemment disponibles pour, par exemple, générer la javadoc séparément mais ne sont pas reprises ici.

9 Conclusion de l'analyse