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

Chapitre 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 pour ensuite l'évoluer à l'aide de *"refactorings"*. 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, ...

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

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"

Chapitre 3

Refactorings avant analyse

3.0.0.1 Fusion des tests sur « Player »

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

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

Chapitre 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.

4.1 Analyse statique

4.1.1 Code dupliqué

Puisque nous utilisons tous les 3 IntelliJ IDEA, l'outil intégré a été utilisé dans un premier temps. La figure 4.1 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).
```

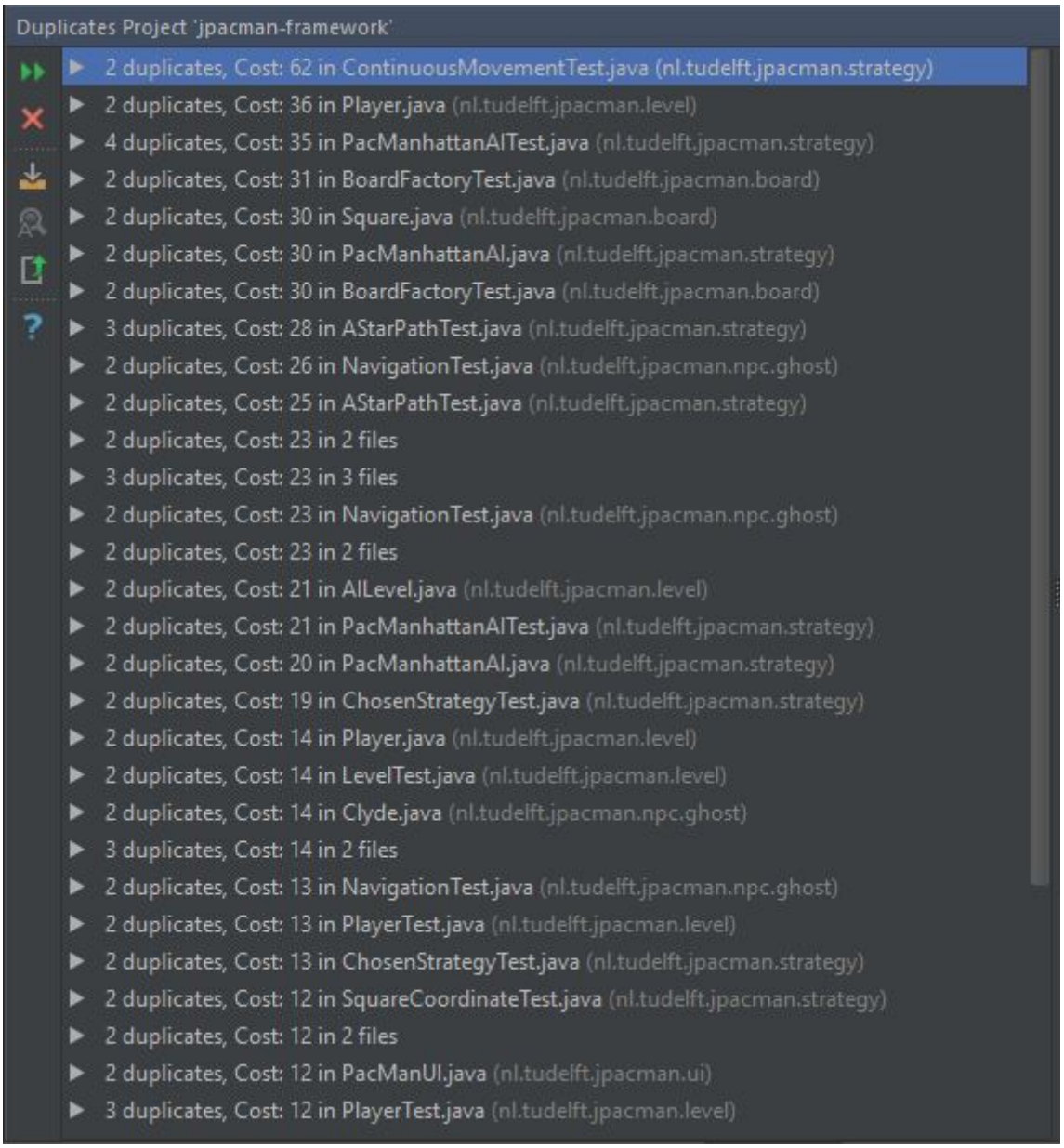

FIGURE 4.1 – Recherche de code dupliqué via l'outil intégré à *IntelliJ IDEA*

4.1.2 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. 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, sauf pour les tests où un tel refactoring nous semblait inutile.
— 104 violations de type *LineLength*

4.1.3 IntelliJ Code inspection

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

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

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

4.1.4 PMD

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

4.1.4.2 Problème détecté par PMD : Level == godclass

4.2 Analyse dynamique

4.3 Mesure de la qualité du logiciel

4.4 Conclusion de l'analyse