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 le Professeur Tom Mens durant l'année académique 2015-2016, a pour but de mettre en pratique les concepts d'évolution logicielle 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 initiale 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 (réalisée par Damien Legay)
- L'implémentation d'une série de labyrinthes (réalisée par Adrien Coppens)
- L'implémentation d'une intelligence artificielle pour pacman (réalisée par Nicolas Leemans)

2.1.1 Fonctionnalité "Score"

Cette fontionalité requièrt l'implémentation de de deux sous-fonctionalités :

La première fonctionalité (Hall of Fame) consiste à maintenir une liste de hauts scores. Si un joueur dépasse l'un des dix meilleurs scores, son nom doit être inséré dans le Hall of Fame. Ceci peut être réalisé de deux manières différentes : si le joueur est identifié, son nom de joueur sera automatiquement entré, si ce n'est pas le cas, il lui sera demandé de fournir un nom à la fin de la partie. Le score du joueur est affecté par le nombre de gommes, phantômes et fruits consommés par Pacman, ainsi que par les hauts faits réalisés (voir ci-bas).

Lorsque la partie est terminée, le Hall of Fame est affiché et le joueur a l'option de le réinitialiser à ses valeurs par défaut. Pour éviter que celle-ci ne se fasse par mégarde, une confirmation est demandée.

La deuxième fonctionalité (Achievements) implique d'implémenter un système de gestion de hauts faits. Pour ce faire, un système de profils doit être mis en place, permettant au joueur la possibilité de s'identifier et ainsi réaliser des hauts faits qui sont mémorisés. Ces hauts faits peuvent être accomplis sur plusieurs parties indépendentes. Le joueur doit avoir la possibilité de visualiser les hauts faits qu'il a accompli avant de commençer à jouer, à la suite de quoi le jeu proposera de

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

nouveaux haut faits à réaliser, au maximum de trois. Lorsque le joueur obtient un haut fait, il reçoit un bonus de score.

Pour réaliser ces fonctionnalités, il a d'abord convenu d'associer à chaque élément pouvant être consommé par Pacman une valeur numérique à ajouter au score, même pour les éléments que Pacman ne peut pas manger à ce stade puisque dépendants de fonctionnalités non implémentées (fruits, supergomme pour manger les phantômes). Ces valeurs sont localisées dans les classes *LevelFactory* (gommes et fruits) et *Ghost* (phantômes).

Ensuite, un système de Hall of Fame a été mis en place à l'aide de la classe *HallOfFame* qui offre les services suivants :

1. La méthode *HallOfFame#handleHoF()* se charge de mettre à jour le Hall of Fame si nécessaire, c'est à dire si le score du joueur excède l'un des scores sauvegardés dans le fichier *resourcesHoF.txt*, et de l'afficher qu'il ait été mis à jour ou non.
2. La méthode *HallOfFame#resetHoF()*, quant à elle, a pour rôle de réinitialiser le Hall of Fame aux valeurs par défaut telles que définies dans le fichier *resourcesDefaultHoF.txt*.

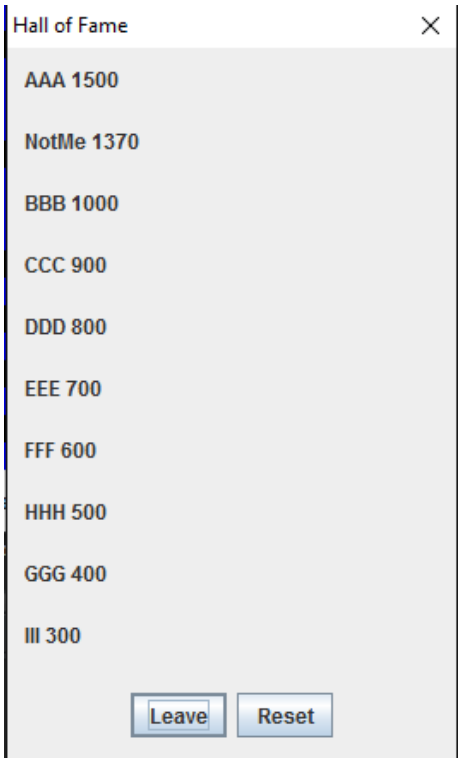

FIGURE 1 – Le Hall of Fame après que le joueur "NotMe" a terminé une partie

Enfin, un système de gestion des hauts faits a été réalisé. Pour cela, il a été nécessaire de mettre un oeuvre un système d'authentification du joueur. Ce système se repose sur les éléments qui voici :

1. Le fichier *resourceslogin.txt*, dans lequel sont stockés les identifiants et mots de passes (non en clair) des joueurs ;
2. Le répertoire *resourcesprofiles*, qui contient les profils des joueurs ayant créé un compte. Le format des fichiers .prf contenus dans ce répertoire est tel que suit :
   — La première ligne contient des informations diverses : score maximal atteint par le joueur, nombre fois qu'il a été tué par tel phantôme, niveau maximum que le joueur a complété,...
   — Les lignes suivantes contiennent le nom d'un haut fait accompli par le joueur.

3. La méthode *IdentifiedPlayer#createNewPlayer()*, qui permet à un joueur de créér un nouveau profil joueur, et, si besoin est, le répertoire ci-dessus ;
4. La méthode *IdentifiedPlayer#authenticate()*, qui vérifie que l'identifiant et le mot de passe entrés par le joueur correspondent bien à des données situées dans *resourceslogin.txt* et, si le joueur le désire, affiche les hauts faits qu'il a déjà obtenus ;
5. La méthode *IdentifiedPlayer#addAchievement()*, servant à ajouter un haut fait que le joueur a obtenu dans son fichier profil, si celui-ci ne s' y trouve déjà ;
6. La méthode *IdentifiedPlayer#saveScore()*, qui met à jour le score maximal du joueur et lui accorde les hauts faits liés à celui-ci ;
7. La méthode *IdentifiedPlayer#killedBy()*, mettant à jour les informations liées au fait de s'être fait tuer par un phantôme et affectant les hauts faits correspondants ;
8. La méthode *IdentifiedPlayer#levelCompleted()* en fait de même pour le niveau maximal atteint ;
9. La méthode *IdentifiedPlayer#displayProfileStats()*, qui affiche les informations concernant le joueur, s'il est identifié, et lui propose d'en accomplir d'autres, s'il n'a pas accompli tous les hauts faits disponibles ;
10. La classe utilitaire *FileChecker* qui effectue certaines opérations de vérification sur les fichiers susnommés ;
11. L'énumération *Achievement* dans laquelle sont stockés les hauts faits, le score qu'ils accordent lorsqu'accomplis, une description définissant comment les obtenir et le haut fait à recommender après leur obtention.

2.1.2 Fonctionnalité "Série de labyrinthes"

Les ajouts à effectuer pour cette extension sont :

1. Système de vies pour Pac-man, au nombre de 3 initialement,
2. Gain d'une vie supplémentaire tous les 10000 points,
3. Téléportation de Pac-man lorsqu'il est tué par un fantôme (et qu'il lui reste au moins une vie),
4. Ajouts d'autres niveaux en préservant la vie et le score de Pac-man lors du passage au niveau suivant (ce qui se produit lorsque toutes les gommes ont été ramassées),
5. Sauvegarde du « meilleur » niveau atteint pour pouvoir, par la suite, débuter directement à un niveau précédemment accédé.

(1) a simplement été réalisé par l'ajout d'un champ *lives* dans la classe *Player* et en faisant en sorte que, lors de la collision avec un fantôme, Pac-man perde une vie plutôt que de mourir (appel à *Player#loseLife* plutôt qu'à *Player#setAlive(false)*).

De la même manière, la réalisation de (2) a pu se faire rapidement, en ajoutant une méthode *Player#checkNewLifeThreshold(int)*, appelée à chaque fois que le joueur reçoit des points. Cette méthode se contente de vérifier si le seuil a été dépassé et ajoute une vie au joueur le cas échéant.

L'ajout (3) a demandé plus de réflexion quant à la manière de l'implémenter. En effet, pour permettre une téléportation « safe », il est nécessaire d'avoir connaissance du niveau (le *Board*), alors que le joueur est, en l'état, le seul à être au courant de sa mort.

Il a été choisi ici d'implémenter un design pattern *Observer*, via une interface *PlayerListener*, qui ne contient qu'une méthode *onPlayerLoseLife(Player)* mais qui pourrait être facilement étendue (pour par exemple permettre l'affichage d'une notification lorsque le joueur reçoit une vie).

Via l'implémentation de *PlayerListener*, *Level* est capable de réagir à la mort du joueur et, puisqu'il possède une référence vers le *Board*, de le téléporter. L'énoncé demande une téléportation *aléatoire* à une distance de 4 cases de tout fantôme, il sera fait utilisation de la distance Manhattan [2].

Lorsque Pac-man meurt, une liste des cases « possibles » est récupérée via la méthode *Level#getPossibleSqua* et le joueur est effectivement transporté aléatoirement sur l'une de celles-ci. Afin de déterminer si une case est « possible », on s'assure qu'elle soit accessible (qu'elle ne corresponde pas à un mur par exemple) et « safe », c'est-à-dire qu'aucun fantôme ne soit à portée.

Pour ce faire, pour toute case accessible, on vérifie les occupants des cases « voisines » en s'assurant qu'elles ne contiennent pas de fantôme. En réalité, puisque les cases situées à une distance Manhattan d'une case donnée forment une sorte de losange, on itère sur le rectangle des cases situées à une distance horizontale et verticale inférieure à 4, en filtrant ensuite les cases trop éloignées en distance Manhattan.

Cette situation est représentée à la figure 2, dans laquelle on s'intéresse à une distance Manhattan de 4 cases à partir de la case bleue. Les cases à une telle distance sont en vert alors que les cases filtrées sont en rouge (les cases noires ne sont pas du tout prises en compte).

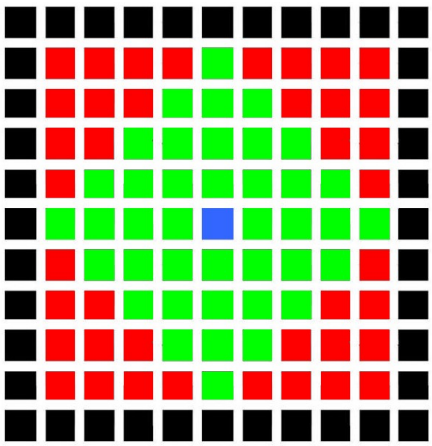

FIGURE 2 – Cases à une distance Manhattan de 4

(4) était également problématique puisque *Level* et *Game* étaient les seuls à être au courant de la réussite d'un niveau, alors que les méthodes permettant le chargement d'un niveau (via *MapParser*) se trouvaient dans *Launcher*.

La solution logique semblait être de déplacer les méthodes appelant *MapParser* de *Launcher* vers *Game* mais il a été décidé d'attendre la fusion des différentes extensions individuelles pour réaliser ce changement. Nous voulions en réalité éviter les conflits potentiels lors de la fusion puisque ce « transfert »de méthodes demande un nombre important de modifications dans le code d'origine. En attendant la fusion, on a donc simplement fait en sorte que *Game* aie la référence vers une instance de *Launcher*.

(5) posait lui aussi quelques difficultés, non pas sur sa réalisation en soi mais bien sur la redondance potentielle et la future intégration avec le système de profil de l'extension « Score » (section 2.1.1) .

Pour cette fonctionnalité, il a été choisi de simuler l'authentification en respectant des conventions de nommage communes. On fournira ici les méthodes permettant, une fois qu'a été récupéré le niveau maximum atteint par un joueur, d'ajouter des boutons de sélection/chargement de niveau tout en réinitialisant le score et les vies du joueur.

2. https://en.wiktionary.org/wiki/Manhattan_distance

Pour les ajouts (4) et (5), il a été nécessaire d'ajouter des fichiers de niveaux supplémentaires. Puisque la démarche était fastidieuse, une fonctionnalité « bonus » a été implémentée : la génération de fichiers de niveaux au format Pac-man à partir d'une image.

En effet, en attribuant une couleur à chaque type d'élément du jeu, une image peut être convertie en un fichier texte équivalent et « lisible » par *MapParser*. Cela permet de créer des niveaux rapidement avec n'importe quel logiciel d'édition d'image puisqu'ils permettent d'utiliser des fonctionnalités comme le « remplissage »d'une zone, qui évite donc d'entrer le même caractère un grand nombre de fois (on obtient également un rendu visuel plus clair que le fichier texte équivalent).

La figure 3 montre un exemple de résultat de conversion d'une image en son niveau équivalent dans JPacman.

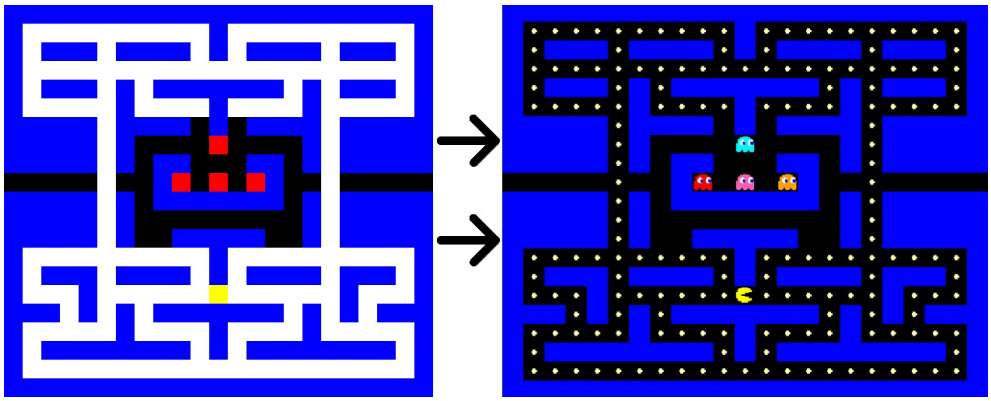

FIGURE 3 – Génération d'un niveau à partir d'une image

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.

Les ajouts à effectuer pour cette extension sont les suivantes :

1. Déplacement continu de Pacman,
2. Intelligence artificielle contrôlant Pacman et qui optimise son score,
3. Différents comportement pour Pacman,
4. Choix du mode de jeu en début de partie (contrôle du Pacman ou spectateur passif)
5. Extension possible et facile de nouvelles stratégies en bannissant tout type de triche

L'ajout de la fonctionnalité (1) a été réalisé en ajoutant une méthode *Game#continousMovement()*, appelée à chaque fois que le joueur appuie sur une touche directionnelle, qui va manipuler un unique *thread* qui va appeler, de façon périodique, la méthode *Game#move()*. A chaque fois que le joueur appuiera sur une touche directionnelle, la méthode *Game#continousMovement()* va tester si la case voulant être accédée par le joueur est accessible à Pacman. Si c'est le cas, la tâche courante du *thread*

est mis à jour et Pacman avance alors dans une nouvelle direction. Par contre, si ce n'est pas le cas, l'instruction est simplement ignorée et Pacman conserve alors son ancienne direction.

Remarque : Pour cette fonctionnalité, il est supposé qu'il suffit d'appuyer une fois sur la touche pour conduire le Pacman dans la direction désirée ce qui veut dire que l'appui abusif d'une même touche directionnelle provoquera un ralentissement voir un arrêt du Pacman.

L'ajout de la fonctionnalité (2) concerne l'implémentation d'une intelligence artificielle manipulant Pacman et qui optimise son score. Le comportement de l'IA est défini dans la classe *PacManhatanAI#nextMove()*. Ce comportement est calculé grâce à une recherche en largeur qui permet de déterminer la gomme la plus proche (*PacManhatanAI#bfsNearestSafetyPelletSquare()*). Une fois la case déterminée, il a été choisi d'utiliser un algorithme de type A^* (classe *AStarPath*) afin de déterminer le chemin le plus court et le plus sûr en appliquant un système de poids aux cases de la carte en fonction de la présence d'une gomme, d'un fantôme ou de la proximité d'un fantôme. L'algorithme A^* a été implémenté grâce à une classe template open-source fournie par Giuseppe Scrivano [3] (classe *AStar*). Cet algorithme permet de se focaliser sur la direction du chemin en privilégiant toute direction se rapprochant de la destination par opposition à l'algorithme Dijkstra qui calcule tous les chemins possibles, ce qui est en général moins performant que l'algorithme A^* car celui-ci évite autant que possible les chemins s'éloignant de la destination. Une fois un chemin déterminé, il est stocké dans une collection *ArrayDeque* sous forme d'un ensemble de directions qui sera suivie par Pacman si aucun danger ne se présente (*PacManhatanAI#convertPathToDirection()*). Avant de choisir le prochain mouvement, l'IA détermine si Pacman est dans une zone sûre ou non (*PacManhatanAI#bfsNearestSafetySquare()*). S'il est trop proche d'un fantôme, alors l'IA choisit de prendre une nouvelle destination qui est la case sûre la plus proche. Le seuil initial pour déterminer si Pacman est proche d'un fantôme est de 14 cases, dans l'implémentation, ce qui permet de jouer un début de partie de façon très sûre. Ce seuil diminuera au cours de la partie en fonction des gommes restantes pour se focaliser sur les gommes en fin de partie (*PacManhatanAI#updatePacmanbehaviour()*). Le fait d'avoir un seuil dynamique offre de bons résultats en pratique. Une fois que Pacman est dans une zone sûre, il recommence à chasser les gommes. Pour finir, pour avoir la certitude que l'IA renvoie une direction à suivre peu importe où le Pacman se trouve sur le plateau de jeu, une méthode *PacManhatanAI#hurryMove()* a été implémentée pour déterminer une direction imposée qui mène à une case accessible dans le cas où aucune autre direction n'a pu être choisie. Bien que ce cas de figure n'apparaît jamais en pratique, il a été pensé en vue de l'intégration d'autres fonctionnalités comme la téléportation (voir 2.1.2 Fonctionnalité "Série de labyrinthes").

Remarque : On peut facilement constater que l'IA se base sur la carte existante et ne passe pas par la construction d'un graphe, ce qui évite de devoir reconstruire le graphe à chaque action, ce qui pourrait être plus gourmand en performances. De plus, ce choix de comportement a été effectué en prenant en considération qu'un autre membre du projet travaillait sur un système de niveaux différents et donc les stratégies définies sur certains sites pour le jeu Pacman ne pouvaient pas être appliquées sur d'autres types de cartes. Ne pouvant pas implémenter une stratégie gagnante, il a été préféré d'appliquer une stratégie qui joue de façon la plus sûre possible tout en essayant de finir le niveau.

L'ajout de la fonctionnalité (3) a été réalisé en utilisant le design pattern "Strategy" qui va permettre d'établir différents comportement pour Pacman, sous forme de stratégie, et de choisir un type de comportement en début de partie. En ce qui concerne son implémentation, une stratégie est définie par une classe abstraite *PacmanStrategy* qui définit certains attributs utiles aux stratégies, une méthode *PacmanStrategy#getTypeStrategy()* qui retourne le type de la stratégie (contrôle du Pacman par le joueur ou par une AI), une méthode *PacmanStrategy#executeStrategy()* qui permet d'attribuer les touches au joueur dans le cas où la stratégie choisie est le contrôle du Pacman par le joueur et

3. http ://a-star.googlecode.com/svn/trunk/java/AStar.java

une méthode *PacmanStrategy#nextMove()* qui retourne la prochaine direction à suivre. Une classe intermédiaire "AIStrategy" est définie pour manipuler plus facilement les stratégies où Pacman est conduit par une intelligence artificielle. Cette classe définit également des attributs plus spécifiques qui sont uniquement utiles aux stratégies de type IA pour déterminer la prochaine direction à suivre lors d'une partie en cours.

Au niveau des stratégies implémentées, la classe *HumanControllerStrategy* représente la stratégie où le Pacman est contrôlé par un joueur humain. Le but de cette stratégie étant seulement de définir les touches de contrôle et de les attribuer au joueur pour qu'il puisse manipuler le Pacman lors d'une partie. La classe *PacManhatanAI* représente la classe définissant le comportement de l'IA expliqué au point (2).

La fonctionnalité (4) a été mise en place pour que le joueur puisse choisir quelle stratégie appliquer, c'est-à-dire quel comportement attribué au Pacman pour une partie complète. Afin de laisser le joueur choisir, une fenêtre de sélection est affichée au lancement du jeu et permet de déterminer la stratégie qui sera appliquée pour toute une partie (la stratégie ne pourra pas changer lors d'un jeu en cours). Une fois que le joueur aura choisi une stratégie, il lui suffit de lancer la partie pour que celle-ci définisse un comportement spécifique à Pacman.

La classe *Level* est utilisée pour implémenter les méthodes liées à la gestion des stratégies durant une partie. En effet, cette classe semblait la plus adéquate car elle correspond au niveau courant qui contient toutes les informations dont les stratégies ont besoin. Il est important de signaler qu'après l'intégration de toutes les fonctionnalités, les méthodes liées à la gestion des intelligences artificielles (joueur et fantômes) ont été déplacé dans une nouvelle classe *AILevel* (voir Refactorings...). Les méthodes *Level#start()* et *Level#stop()* permettent d'exécuter ou d'arrêter l'action spécifique définie selon le type de stratégie choisie. Dans le cas d'une stratégie où le joueur est un humain, l'attribution des touches sera appliquée au joueur. Tandis que dans le cas d'une stratégie d'intelligence artificielle, la méthode *Level#startAIStrategy()* est appelée pour lancer un unique *thread* qui va de façon périodique attribuer une direction à suivre au Pacman et appliquer un mouvement dans cette direction. Une nouvelle direction devant être calculée uniquement lorsque le Pacman se trouve à une intersection, un test a été ajouté pour définir cette situation. Dans le cas où le Pacman ne se trouve pas à une intersection, l'action effectuée correspondra à l'action courante (ancien mouvement). Le comportement du *thread* pour la gestion des intelligences artificielles est défini dans la méthode *Level/PlayerMoveTask#run()*

La fonctionnalité (5) concerne le fait de pouvoir ajouter facilement des stratégies définissant de nouveaux comportements pour Pacman sans pour autant autoriser des stratégies qui trichent. Tout d'abord, l'implémentation des stratégies a été réfléchie pour que l'ajout de nouvelles stratégies soient faciles. En effet, comme expliqué en (3), les classes abstraites *PacmanStrategy* et *AIStrategy* permettent de manipuler plus facilement les différents types de stratégies possibles en définissant les informations et les méthodes nécessaires aux différentes stratégies. L'implémentation d'une nouvelle stratégie se fera donc par la création d'une nouvelle classe, implémentant le comportement de la stratégie, et qui étendra l'une de ces deux classes en fonction de son type. Pour les stratégies d'intelligence artificielle, la nouvelle classe devra implémenter une méthode *nextMove()* qui retournera la prochaine direction à suivre par Pacman. Ensuite, pour interdire tout type de triche, les attributs, se trouvant dans les classes mères *PacmanStrategy* et *AIStrategy*, qui représentent les caractéristiques du niveau en cours lors d'une partie, ont été implémenté avec le mot clé "final" et uniquement des méthodes d'accesseurs. Ainsi, comme les stratégies étendent ces classes, ils auront accès à ces attributs grâce aux méthodes accesseurs mais ne pourront en aucun cas modifier ces valeurs. En ne permettant pas de modifier les données du jeu, les stratégies ne pourront pas tricher dans la manipulation des données lors du calcul de la prochaine direction à suivre.

3 Refactorings avant analyse

Cette section reprend des refactorings ayant été effectués avant que des outils d'analyse n'aient été utilisés. Certaines ont d'ailleurs été planifiées avant la fusion des versions individuelles mais n'ont pas été réalisées dans l'immédiat pour limiter le nombre de conflits avec les autres versions lors de l'intégration des « pull requests ».

Fusion des tests sur *Player* Des tests avaient été effectués par 2 d'entre nous et lors de la fusion, nous avions simplement fusionné ceux-ci en ignorant la redondance de variables d'instances. L'un des 2 tests n'utilisant pas les « mocks », une légère adaptation a été nécessaire pour que le mock de fantôme retourne une valeur lors de l'appel à *Ghost#getIdentity()*.

Déplacement de la variable retenant le niveau courant de *Level* vers *Game* Ce placement semble plus logique et évite un problème de « feature envy »de méthodes de *Game* y accédant.

Déplacement des méthodes concernant le « map parser » de *Launcher* vers *Game* Cette nouvelle organisation semble plus logique et évite que *Game* ne doive avoir une référence vers le *Launcher* qui l'a créé (pour pouvoir charger le niveau suivant en cas de victoire). Cela a cependant nécessité de passer des méthodes/variables en « static » : une partie des « accesseurs » aux « Factories » de *Launcher*, puisque *MapParser* en a besoin. Cette modification reste malgré tout logique puisque ces méthodes/variables sont en effet uniques au runtime.

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 4 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).
```

5.2 Audit de code

5.2.1 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.

5.2.2 IntelliJ Code inspection

Détection d'une partie de condition impliquant QUICK_WIN comme « pointless » Puisque la variable **QUICK_WIN** est mise à « true » ou « false » dans le code et pas par un paramètre pou-

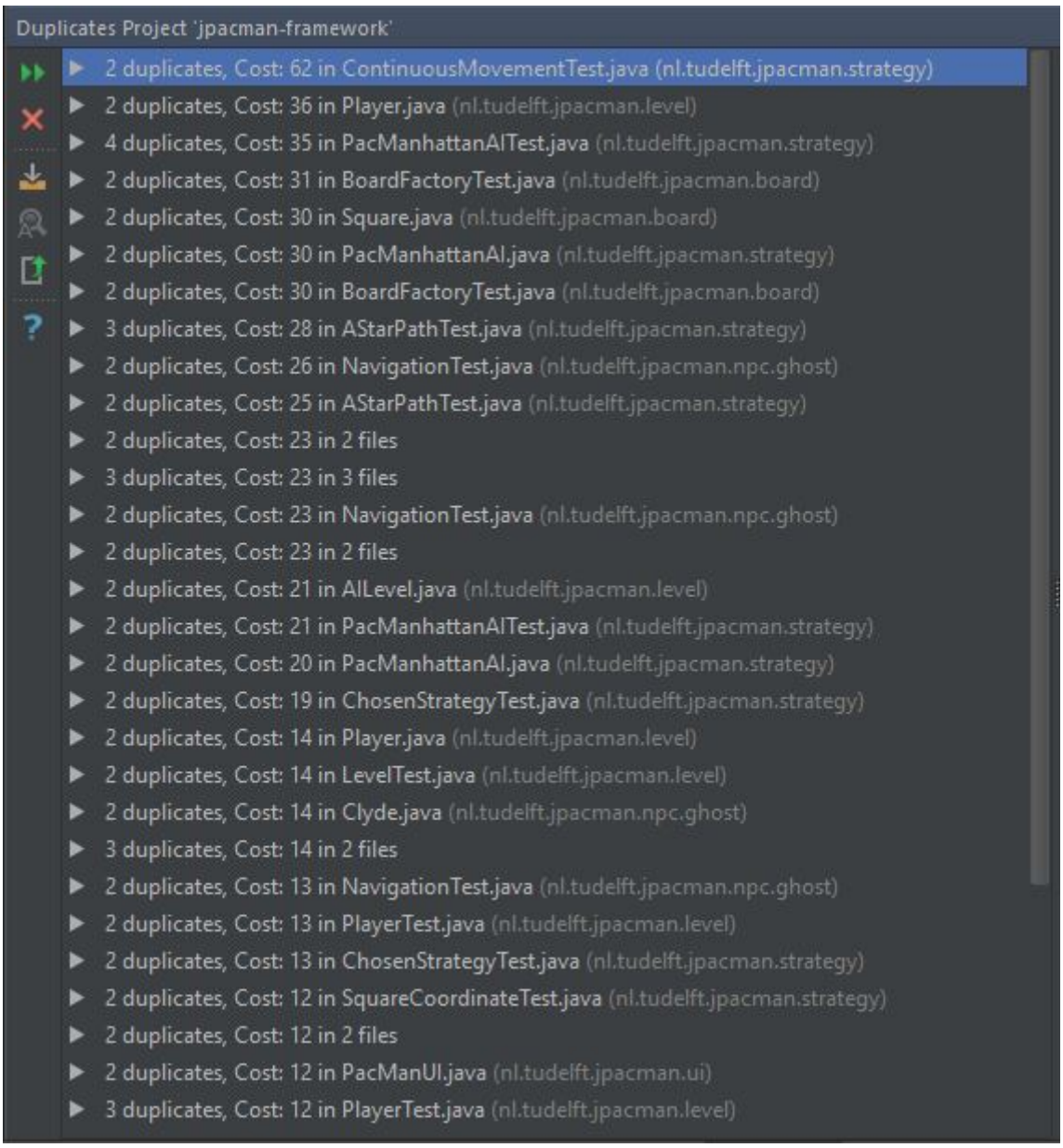

FIGURE 4 – Recherche de code dupliqué via l'outil intégré à *IntelliJ IDEA*

vant varier au runtime, la version compilée contient effectivement une (partie de) condition qui n'a pas vraiment de sens (toujours vraie ou fausse). Néanmoins, bien qu'il eut été possible de faire varier sa valeur « dynamiquement » (par un paramètre optionnel au lancement du programme par exemple), nous n'avons pas jugé utile d'implémenter une telle fonctionnalité. En effet, il s'agit surtout d'une option de développement permettant de tester une suite de niveau, pas d'une « feature » destinée aux utilisateurs (sa présence dans le code au stade de la release ayant pour but de faciliter le travail des « évaluateurs »).

Mineures altérations diverses

- La variable de classe « Level » dans *SinglePlayerTest* fut convertie en variable locale,
- 4 booléens n'avaient aucun intérêt et furent donc ôtés,
- 6 paramètres avaient toujours la même valeur et furent donc remplaçés par celle-ci,
- 14 accès furent rendus plus restrictifs,
- La valeur de retour de la méthode *Level/PlayerMoveTask#run()* n'était pas utilisée et cette méthode a donc été convertie en « void »,

— 8 variables ont été déclarées « final »,
— 11 éléments étaient reportés inutilisés, ceux qui n'avaient pas d'usage ont été enlevés (les autres sont suite à la gestion du score de fonctionnalités non implémentées),
— Les 5 constantes dans *SinglePlayerTest#constantTest()* étaient accédées par référence d' instance, elles sont maintenant accédées par référence de classe,
— 7 types anonymes ont été remplaçés par des expressions lambda,
— 3 déclarations de type explicites furent remplaçés par « <> » dans le contexte de listes,
— 2 boucles de type « for each » furent remplaçées par des flux,
— Intellij a soulévé 3 erreurs dans la Javadoc, qui furent corrigées,
— 6 « asserts » étaient toujours vrais, ils ne le sont plus.

« Warnings » sur des « problèmes de modernité » Ces signalements, surtout liés au code dans sa version de base (avant nos changements individuels), se présentaient par exemple aux endroits où un « for indexé » pouvait être remplacé par un « foreach » ou bien lorsqu'une classe anonyme « classique » pouvait disparaitre au profit d'une expression lambda. Les « modernisations » ont donc été effectuées (pour la plupart grâce à des « refactorings » automatiques d'*IntelliJ IDEA*).

5.2.3 PMD

« Feature envy » potentiel lié à la récupération des cases « safe » Lorsque le joueur doit être téléporté vers une case à l'écart des fantômes, il est nécessaire de regarder le contenu du *Board*. Il est donc logique que ce soit celui-ci qui se charge de récupérer la liste des cases « safe ». De fait, pour éviter un problème potentiel de « feature envy », cette récupération a été déplacée de *Level* vers *Board*.

Détection de « God classes » Signifie en principe que :

1. la classe a une « cohésion interne »faible
2. la classe accède à des attributs appartenant à trop de classes étrangères
3. la classe est trop complexe par rapport à sa taille

En regardant dans le code de PMD [4], il semblerait que la valeur de complexité (cyclomatique apparemment) de la classe soit bien calculée mais qu'à aucun moment une pondération par rapport à la taille de celle-ci (nombre de méthodes par exemple) ne soit faite, ce qui ne respecte pas la définition d'origine. Les classes qui contiennent beaucoup de méthodes sont donc naturellement sujettes à un signalement de type « god class ».

Bien que, comme expliqué ci-dessus, la détection ne soit pas parfaite, nous avons choisi de réagir pour éviter qu'une classe n'aie trop de méthodes. Les changements effectués ont effectivement permis de supprimer les avertissements en question tout en permettant probablement à notre code d'être plus « maintenable » puisque les classes sont globalement moins volumineuses.

Taille de code Deux classes (*Player* et *Launcher*) avaient trop de méthodes, donc certaines furent déplaçées ou intégrées à une méthode appelante lorsque possible.

La classe (*PacManhattanAI* et *PacManhattanAI*) et la méthode *PacManhattanAI#BFSNearestPelletSquare()*) avaient une complexité cyclomatique trop élevée, elles ont donc été l'object de « refactorings »() afin de réduire celle-ci.

4. https://pmd.github.io/pmd-5.4.1/pmd-java/xref/net/sourceforge/pmd/lang/java/rule/design/GodClassRule.html

Loi de Demeter PMD détecten nombreuses violations de celle loi, mais beaucoup d'entre elles viennent du code originel et la vaste majorité relèvent d'appels à des librairies standard Java, il nous a donc paru que la détection de PMD était inadéquate à ce niveau. Pour ce qui est des autres violations, le code existant force largement à les commettre.

Avertissements mineurs

- Deux « if » imbriqués dans *PlayerMoveTask#run()* furent intégrés en un seul,
- 9 Arraylists furent convertis en List et une Hasmap en Map,
- 4 « varargs » furent introduits là où c'était possible,
- 5 déclarations d'attributs de classe furent déplaçés à la tête de leur classe,
- 15 « assert » furent modifiés pour éviter la surdépendance à « assertTrue() »,
- 251 messages d'erreur furent ajoutés aux « assert » qui en étaient privés,
- Certaines variables au nom trop court furent renommées, d' autres non (variable à l'étendue fort restreinte ou correspondant à des coordonnées cartésiennes),
- Des méthodes, variables, classes et constantes ont été renommées pour correspondre aux conventions du langage,
- De nombreuses variables et paramètres furent déclarés « final »,
- L'initialisation superflue de *PlayerMoveTask#finished* a été ôtée.

5.2.4 FindBugs

5.2.5 Métriques

5.3 Analyse de dépendances

6 Analyse dynamique

6.1 Couverture du code

6.2 Couverture des tests

6.3 Profilage

7 Evolution du code au cours du temps

8 Mesure de la qualité du logiciel

9 Notes diverses et guide d'utilisation

9.1 Maven

Le projet utilisant Maven [5], il est très facile de compiler le code, de l'analyser avec les outils prévus ou d'exécuter la suite de tests. On peut noter que la configuration originale du projet ne permettait pas l'exécution de l'archive jar produite, cela a été corrigé pour notre version (en reproduisant simplement les dépendances externes dans l'archive générée, via un plugin).

5. https://maven.apache.org/

En considérant que Maven 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 (c'est-à-dire l'endroit où se trouve le fichier `pom.xml`), vous pouvez :

— Exécuter les tests via :

```
$ mvn test
```

— Compiler le projet (inclus l'exécution des tests) via :

```
$ mvn compile
```

— Lancer le jeu via (après avoir compilé) via :

```
$ mvn exec:java -Dexec.mainClass=
    "nl.tudelft.jpacman.Launcher"
```

— Compiler le projet et produire une archive "jar" exécutable via :

```
$ mvn package
```

L'archive est ensuite exécutable via :

```
$ java -jar target/jpacman-framework-7.3.0.jar
```

— Générer le site Maven (inclus les étapes précédentes, la javadoc et des rapports d'analyse) via :

```
$ mvn site
```

(le site généré étant accessible en ouvrant `target/site/index.html`)

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

9.2 Travis

Puisque le projet JPacman dans son état « initial » (avant modifications de notre part) intégrait déjà le fichier de configuration Travis [6], nous l'avons utilisé sur nos différents « forks ».

Les historiques des « builds » (avec leurs logs) de la version d'intégration et de la version individuelle d'Adrien Coppens sont disponibles à l'adresse : https://travis-ci.org/qdrien/jpacman-framework/builds.

9.3 Organisation des dépots Git(Hub)

Les versions correspondant à la phase individuelle sont disponibles aux adresses suivantes :

— Adrien Coppens (extension « Série de labyrinthes ») :
https://github.com/qdrien/jpacman-framework
Tag de la release : `individual-coppens-series-of-mazes-v7.0.0`
(sur la branche `series-of-maze`)

— Damien Legay (extension « Score ») :
https://github.com/DamienLegay/jpacman-framework
Tag de la release :

— Nicolas Leemans (extension « IA pour Pacman ») :
https://github.com/NicolasLeemans7/jpacman-framework
Tag de la release : `individual-leemans-AI-for-Pacman-v7.0.0` (sur la branche `Master`)

6. https://travis-ci.org/

Le dépot commun pour la phase d'intégration est celui d'Adrien Coppens, la release finale étant taggée .

9.4 Spécificités des ajouts individuels

9.4.1 Fonctionnalité "Série de labyrinthes"

- Afin de pouvoir tester différents niveaux plus aisément, un booléen **QUICK_WIN** est configurable dans la classe *Level*. Si on lui affecte la valeur *true*, récupérer treize gommes est suffisant pour réussir un niveau (et passer au suivant).
- Les fichiers correspondants aux niveaux disponibles en jeu se trouvent dans le répertoire `target/classes` à condition d'avoir au moins « compilé »la solution via la commande :

  ```
  $ mvn compile
  ```

  (Toute commande permettant l'exécution d'une étape postérieure du cycle Maven [7] est également recevable)
  Avant compilation, ces mêmes fichiers se trouvent dans `src/main/resources`.
- Un niveau numéroté n est considéré comme partie intégrante du jeu si le fichier "`board`n`.txt`" est présent et si les fichiers des niveaux $1 \ldots n-1$ le sont également.
- Au lancement du jeu, si le fichier "`board`n`.txt`" est manquant mais qu'il existe un fichier "n`.png`", le convertisseur mentionné à la section 2.1.2 en page 6 crée le fichier "`board`n`.txt`" correspondant.

10 Conclusion de l'analyse

7. https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html