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, à 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 parties. La première section présente les différentes extensions qui ont été réalisées à partir de la version originale de JPacman. La seconde section met en avant les principaux problèmes de conflits rencontrés lors de l'intégration des extensions individuelles et leurs résolutions. Ensuite, la troisième section traite des *refactorings* qui ont été réalisés avant l'analyse de la qualité. La prochaine section présente une comparaison entre la qualité du code source finale avec la qualité du code source original. Ensuite, la section suivante définit une analyse de l'évolution du logiciel en terme de nombres de lignes de code. Pour finir, un complément d'information concernant l'utilisation du logiciel est fourni dans la dernière partie de ce rapport.

2 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 de fonctionnalités liées au 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 Fonctionnalité "Score"

Cette fonctionnalité requiert l'implémentation de deux sous-fonctionnalités :

La première sous-fonctionnalité (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, fantômes 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.

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

La deuxième sous-fonctionnalité (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 indépendantes. Le joueur doit avoir la possibilité de visualiser les hauts faits qu'il a accompli avant de commencer à jouer, à la suite de quoi le jeu proposera de 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 fantômes). Ces valeurs sont localisées dans les classes *LevelFactory* (gommes et fruits) et *Ghost* (fantômes).

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 *resources\HoF.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 *resources\DefaultHoF.txt*.

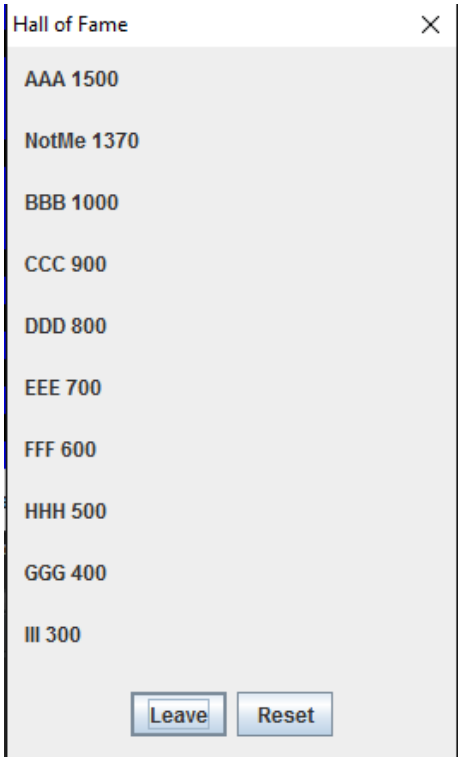


FIGURE 1 – Le Hall of Fame après que le joueur "NotMe" a terminé une partie

Enfin, un système de gestion des hauts faits a été réalisé. Pour cela, il a été nécessaire de mettre en œuvre un système d'authentification du joueur. Ce système repose sur les éléments que 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 fantôme, 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 fantôme 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'ils sont accomplis, une description définissant comment les obtenir et le haut fait à recommencer après leur obtention.

2.2 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#getPossibleSquares()* 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).

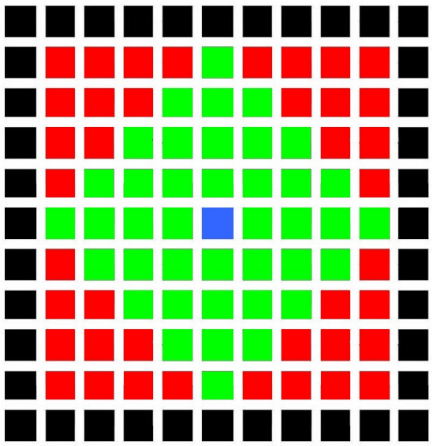


FIGURE 2 – Cases à une distance Manhattan de 4

(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) .

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

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.

Convertisseur de cartes 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.

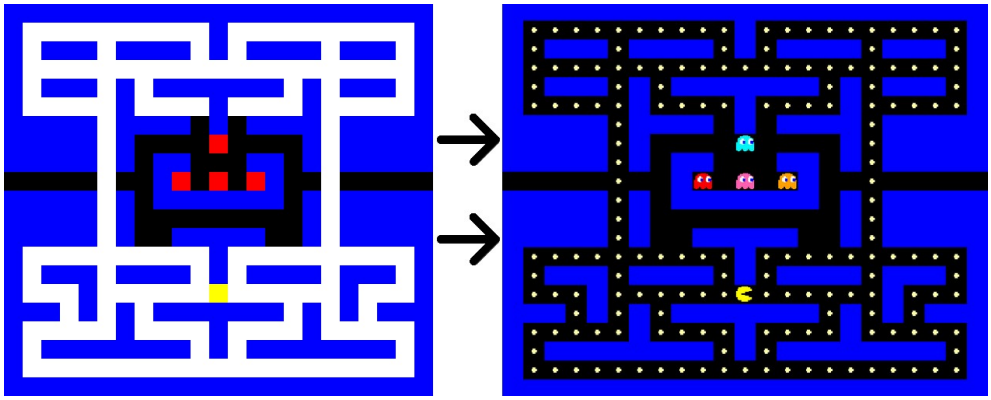


FIGURE 3 – Génération d'un niveau à partir d'une image

2.3 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 section 2.2).

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

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

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 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 déplacées dans une nouvelle classe *AILevel* (voir section 4). 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 Conflits et solutions lors de l'intégration

Cette section met en avant les problèmes de conflits rencontrés lors de l'intégration des extensions individuelles (voir section 2) ainsi que les solutions qui ont été apportées à ces problèmes.

La liste non exhaustive suivante présente les principaux conflits qui ont été rencontrés :

- Incohérence au niveau de la documentation de certaines méthodes causée par l'assemblage de plusieurs documentations différentes lors de l'intégration.
- Problèmes de conflits liés à l'ajout ou la suppression d'accolades et d'importations dans le code.
- Conflits entre deux extensions ayant des comportements différents dans une même situation. Par exemple, une extension présentait un jeu où le joueur avait une seule vie tandis que l'autre présentait un jeu où le joueur avec trois vies, ce qui a conduit à des problèmes dans le cas où le joueur mourait.
- Problèmes de dysfonctionnement au niveau de certaines fonctionnalités causés par un conflit avec une autre extension. Par exemple, des problèmes ont été rencontrées entre l'intelligence artificielle et le systèmes de plusieurs niveaux.
- Incohérence au niveau des paramètres et du corps de certaines méthodes causée par l'assemblage de plusieurs méthodes différentes ayant le même nom dans le code.
- Dysfonctionnement de certains tests unitaires qui devaient être adaptés aux comportements des autres extensions.
- Problèmes de gestion de fichiers (création et écriture) lié au fait que certaines extensions ont été développé sur des systèmes d'exploitation différents.
- ...

Avant de présenter les solutions apportées aux différents problèmes de conflits rencontrés, il faut savoir que la plupart des conflits ont été résolus par l'outil de merge *Kdiff3*[4]. Ce logiciel permet de fusionner automatiquement de nombreuses différences entre diverses versions d'un code source qui posent conflits lorsque merge avec git, ainsi que de manuellement résoudre les conflits non gérés automatiquement.

La liste non exhaustive suivante présente les principales solutions qui ont été apportées :

- Adaptation de certains paramètres de méthodes pour pouvoir intégrer les fonctionnalités ensemble.
- Adaptation du processus de victoire et de défaite lié aux niveaux et aux vies.
- Adaptation du système de gestion de fichier pour qu'il puisse fonctionner sur tout système d'exploitation.
- Adaptation de certains tests unitaires.
- Adaptation du comportement de l'IA pour un jeu contenant plusieurs niveaux.
- ...

Il est également important de signaler que l'étape d'intégration a été réalisée assez facilement grâce à d'importantes discussions, au préalable, concernant les choix d'implémentations qui ont permises de faciliter l'intégration en minimisant les conflits.

4. http ://kdiff3.sourceforge.net/

4 Refactorings avant analyse

Cette section présente les *refactorings* qui ont 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 ».

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.

Déplacement de la gestion des IA de la classe *Level* vers *AILevel* : Ce déplacement permet de regrouper toutes les méthodes liées aux IA (joueurs et fantômes) au sein d'une unique classe. Cette classe héritant de la classe *Level* permet d'avoir toujours accès aux informations du niveau courant dont les IA ont besoin pour fonctionner. Ce *refactoring* a été appliqué dans le but de réduire la classe *Level* qui devenait trop complexe et de localiser la gestion des IA au sein d'une classe pour faciliter l'évolution du logicielle et sa maintenance.

5 Analyse de la qualité du code source

Dans cette section, 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 différentes techniques. Pour faire cela, nous ferons appel à plusieurs outils d'analyse de qualité que nous détaillerons par la suite. Cette phase se déroulera en deux étapes : une analyse statique et une analyse dynamique du code. Cette analyse doit permettre de comprendre la structure du logiciel et de situer les parties du code pouvant causer problème.

5.1 Analyse statique

5.1.1 Métriques

Dans cette section, nous allons analyser la qualité du logiciel initial ainsi que la qualité du logiciel après intégration des extensions, grâce à des métriques, afin de comparer les deux versions. Un ensemble de métriques logiciels peut être calculé pour mesurer la qualité interne du logiciel. Ces métriques peuvent ainsi être utilisées pour identifier les problèmes liés à la qualité des systèmes logiciels existant. Il existe un grand nombre de catégories de métriques telles que les métriques de taille, de cohésion et de couplage, de dépendance, de complexité ou encore de similarité. Pour ce projet, l'ensemble des métriques est calculé par le biais d'un plugin de l'IDE Intellij Idea qui s'appelle *Metrics Reloaded* (version 1.6.1). Ce plugin permet d'obtenir des informations concernant un grand nombre de métriques sur un projet.

1) Métrique de taille

Les métriques de taille sont les métriques statiques les plus simples à définir, comprendre et calculer. Elles donnent une idée de la taille du logiciel, en calculant par exemple le nombre de lignes de code source dans les classes, le nombre de classes dans les paquets,... Généralement les métriques de taille sont utilisées pour comparer la taille de différents logiciels, pour prévoir les efforts de développement et de maintenance.

a) Nombre de classes

Cette métrique fournit des informations concernant la distribution des classes au sein du projet en donnant le nombre total de classes (en prenant en compte les interfaces) par package.

Logiciel initial

Les résultats obtenus sont reportés dans les figures suivantes (Figures 4 et 5) où la colonne *C* représente le nombre de classes et d'interfaces dans chaque package et la colonne *C(rec)* représente le nombre de classes et d'interfaces dans chaque package en incluant les sous-classes. Les autres colonnes concernent la distribution des classes au sein du code produit et du code de test.

package	C	C(rec)	Cp	Cp(rec)	Ct	TC(rec)
nl.tudelft.jpacman.ui	13	13	13	13	0	0
nl.tudelft.jpacman.sprite	7	7	6	6	1	1
nl.tudelft.jpacman.npc.ghost	10	10	9	9	1	1
nl.tudelft.jpacman.npc	1	11	1	10	0	1
nl.tudelft.jpacman.level	18	18	17	17	1	1
nl.tudelft.jpacman.game	3	3	3	3	0	0
nl.tudelft.jpacman.board	13	13	7	7	6	6
nl.tudelft.jpacman	7	72	6	62	1	10
nl.tudelft		72		62		10
nl		72		62		10
		72		62		10
Total	**72**		**62**		**10**	
Average	9,00		7,75		1,25	

FIGURE 4 – Nombre de classes et d'interfaces dans chaque package

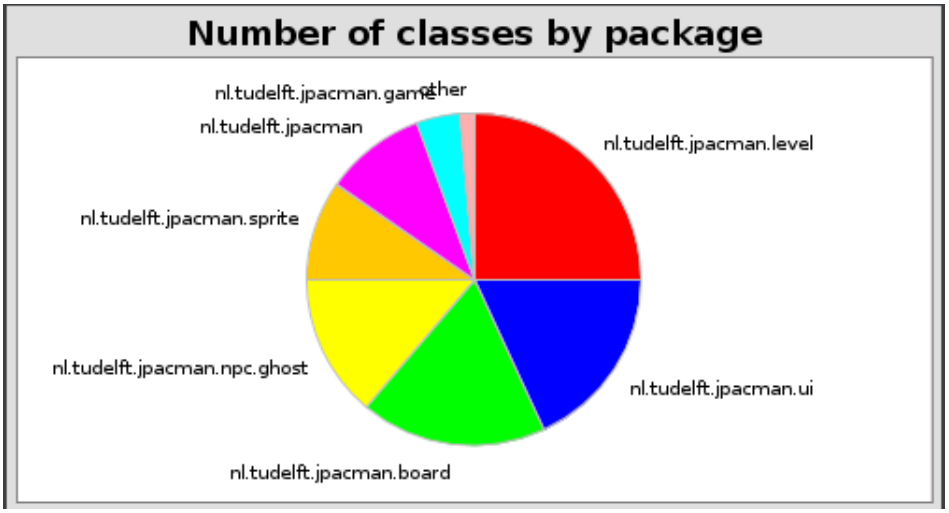


FIGURE 5 – Répartition graphique des classes au sein des packages

Logiciel final

Effectuons cette analyse sur le logiciel final, c'est-à-dire le logiciel qui contient la version après intégration des fonctionnalités.

package	C	C(rec)	Cp	Cp(rec)	Ct	TC(rec)
nl.tudelft.jpacman.ui	13	13	13	13	0	0
nl.tudelft.jpacman.strategy	16	16	8	8	8	8
nl.tudelft.jpacman.sprite	7	7	6	6	1	1
nl.tudelft.jpacman.npc.ghost	10	10	9	9	1	1
nl.tudelft.jpacman.npc	1	11	1	10	0	1
nl.tudelft.jpacman.level	24	24	21	21	3	3
nl.tudelft.jpacman.game	7	7	6	6	1	1
nl.tudelft.jpacman.board	15	15	8	8	7	7
nl.tudelft.jpacman	5	98	3	75	2	23
nl.tudelft		98		75		23
nl		98		75		23
		98		75		23
Total	**98**		**75**		**23**	
Average	10,89		8,33		2,56	

FIGURE 6 – Nombre de classes et d'interfaces dans chaque package

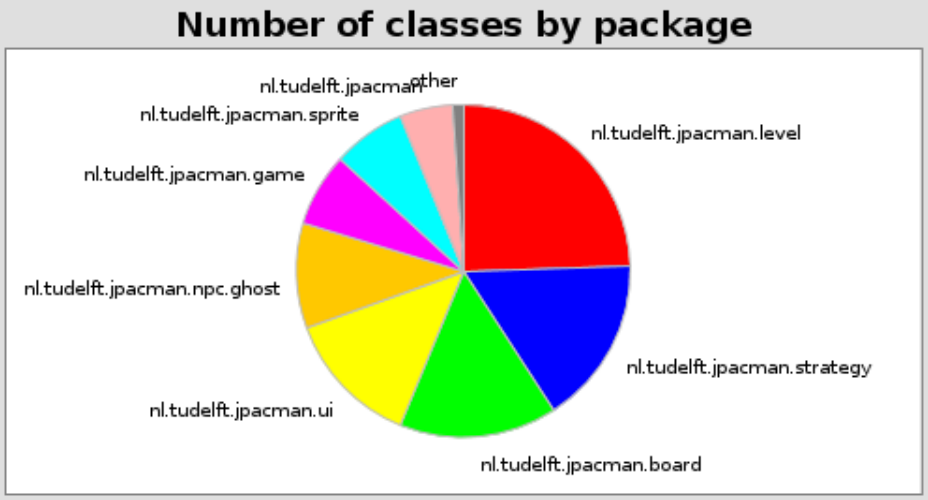


FIGURE 7 – Répartition graphique des classes au sein des packages

Après l'ajout d'un nouveau package (strategy) et de nouvelles classes, on constate que la distribution des classes au sein des packages n'a pas beaucoup changé avec une moyenne de 2 classes (ou interfaces) en plus par package (Figure 6 et 7).

b) Lignes de codes

Cette métrique permet de détecter les classes volumineuses qui peuvent être à l'origine d'une grande complexité. Elle permet également de détecter d'éventuelles "god class".

Logiciel initial

Dans un premier temps, on peut analyser le nombre de lignes de code à l'intérieur de chaque package (Figure 8) en ne considérant pas les classes de tests. On constate ainsi une moyenne de 482 lignes de codes par package. Le package le plus volumineux semble être le package "Level" qui contient 1183 lignes de codes. Ce n'est pas très étonnant car ce package a été analysé comme étant celui qui contient le plus de classes (voir métrique "Nombre de classes"). On peut aussi observer qu'il y a, au total, 3860 lignes de codes si on inclus les lignes de commentaires et 1905 lignes de code java sans les commentaires (Figure 9).

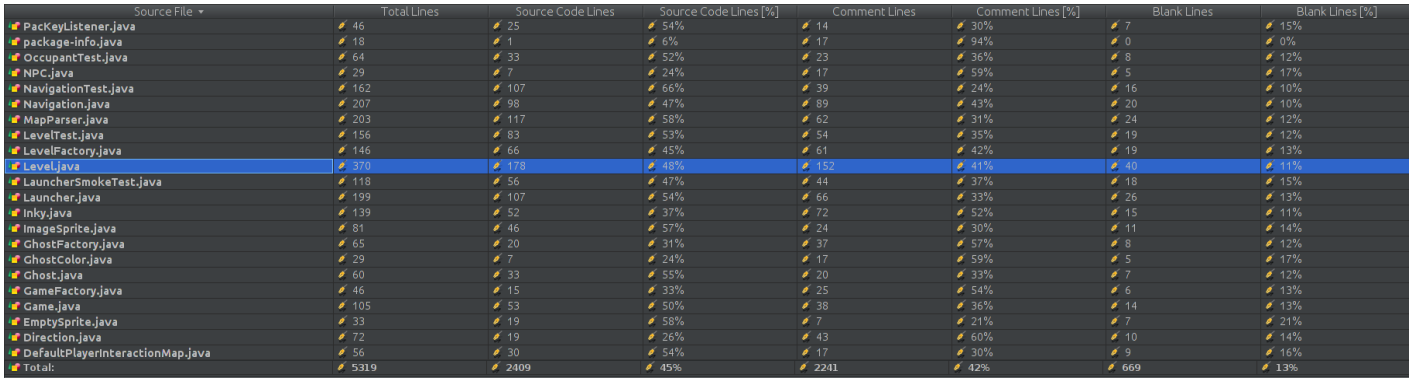
package	LOC	LOC(rec)	LOCp	LOCp(rec)	LOCt	LOCt(rec)
		3.860		3.860		0
nl		3.860		3.860		0
nl.tudelft		3.860		3.860		0
nl.tudelft.jpacman	200	3.860	200	3.860	0	0
nl.tudelft.jpacman.board	473	473	473	473	0	0
nl.tudelft.jpacman.game	174	174	174	174	0	0
nl.tudelft.jpacman.level	1.183	1.183	1.183	1.183	0	0
nl.tudelft.jpacman.npc	24	794	24	794	0	0
nl.tudelft.jpacman.npc.ghost	770	770	770	770	0	0
nl.tudelft.jpacman.sprite	502	502	502	502	0	0
nl.tudelft.jpacman.ui	534	534	534	534	0	0
Total	**3.860**		**3.860**		**0**	
Average	482,50		482,50		0,00	

FIGURE 8 – Nombre de lignes de code dans chaque package

file type	LOC	NCLOC
Java	3.860	1.905

FIGURE 9 – Nombre de lignes de code avec et sans les commentaires

Ensuite, grâce au plugin "Statistic" de l'IDE Intellij Idea, on peut afficher plus en détail les statistiques concernant les lignes de code dans le projet (en prenant en compte les classes de tests). Dans les résultats dans la figure 10, on constate que la classe la plus volumineuse est la classe *Level*.

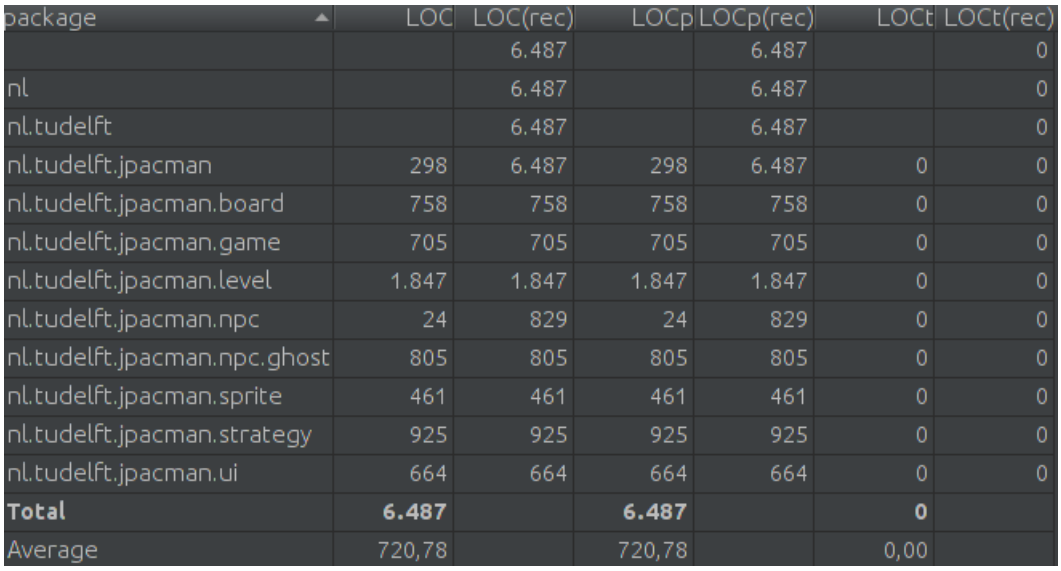


Source File	Total Lines	Source Code Lines	Source Code Lines [%]	Comment Lines	Comment Lines [%]	Blank Lines	Blank Lines [%]
PacKeyListener.java	46	25	54%	14	30%	7	15%
package-info.java	18	1	6%	17	94%	0	0%
OccupantTest.java	64	33	52%	23	36%	8	12%
NPC.java	29	7	24%	17	59%	5	17%
NavigationTest.java	162	107	66%	39	24%	16	10%
Navigation.java	207	98	47%	89	43%	20	10%
MapParser.java	203	117	58%	62	31%	24	12%
LevelTest.java	156	83	53%	54	35%	19	12%
LevelFactory.java	146	66	45%	61	42%	19	13%
Level.java	370	178	48%	152	41%	40	11%
LauncherSmokeTest.java	118	56	47%	44	37%	18	15%
Launcher.java	199	107	54%	66	33%	26	13%
Inky.java	139	52	37%	72	52%	15	11%
ImageSprite.java	81	46	57%	24	30%	11	14%
GhostFactory.java	65	20	31%	37	57%	8	12%
GhostColor.java	29	7	24%	17	59%	5	17%
Ghost.java	60	33	55%	20	33%	7	12%
GameFactory.java	46	15	33%	25	54%	6	13%
Game.java	105	53	50%	38	36%	14	13%
EmptySprite.java	33	19	58%	7	21%	7	21%
Direction.java	72	19	26%	43	60%	10	14%
DefaultPlayerInteractionMap.java	56	30	54%	17	30%	9	16%
Total:	5319	2409	45%	2241	42%	669	13%

FIGURE 10 – Statistiques sur les lignes de codes

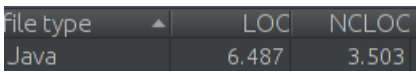
Logiciel final

Concernant la version finale, les résultats de la distribution des lignes de codes dans les packages sont reportés dans les figures 11 et 12. Ces résultats montrent que le package "Level" reste le plus volumineux de tous et le nombre moyen de lignes de code par package a augmenté de 482 à 720. Quant au nombre de lignes de code total, il est passé de 3860 à 6487 et de 1905 à 3503 si on ne compte pas les commentaires.



package	LOC	LOC(rec)	LOCp	LOCp(rec)	LOCt	LOCt(rec)
		6.487		6.487		0
nl		6.487		6.487		0
nl.tudelft		6.487		6.487		0
nl.tudelft.jpacman	298	6.487	298	6.487	0	0
nl.tudelft.jpacman.board	758	758	758	758	0	0
nl.tudelft.jpacman.game	705	705	705	705	0	0
nl.tudelft.jpacman.level	1.847	1.847	1.847	1.847	0	0
nl.tudelft.jpacman.npc	24	829	24	829	0	0
nl.tudelft.jpacman.npc.ghost	805	805	805	805	0	0
nl.tudelft.jpacman.sprite	461	461	461	461	0	0
nl.tudelft.jpacman.strategy	925	925	925	925	0	0
nl.tudelft.jpacman.ui	664	664	664	664	0	0
Total	**6.487**		**6.487**		**0**	
Average	720,78		720,78		0,00	

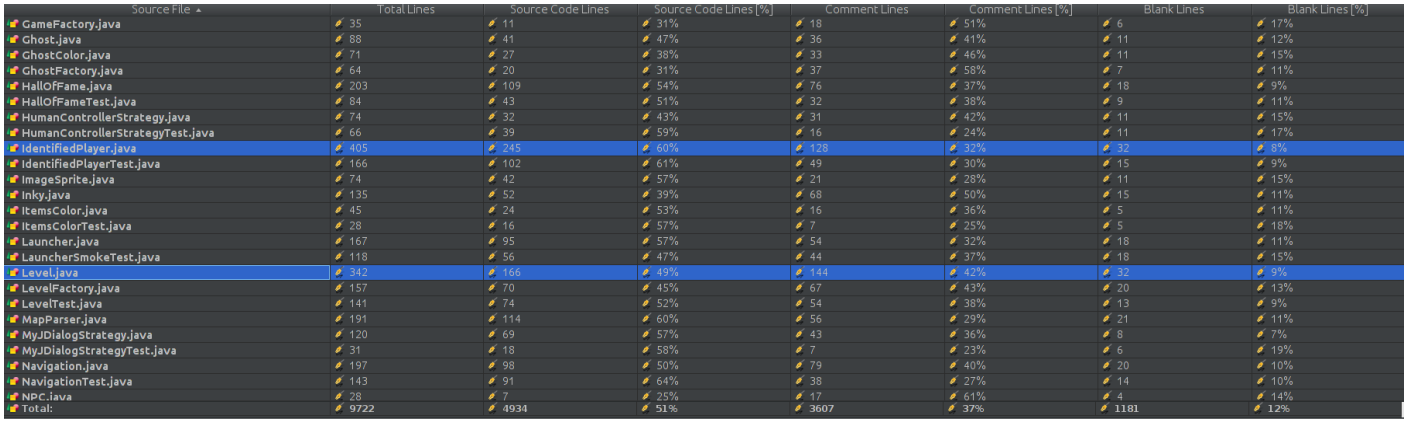
FIGURE 11 – Nombre de lignes de code dans chaque package



file type	LOC	NCLOC
Java	6.487	3.503

FIGURE 12 – Nombre de lignes de code avec et sans les commentaires

Quant aux statistiques (Figure 13), on constate que les deux classes les plus volumineuses sont *IdentifiedPlayer* et *Level*. On constate également que la plupart des valeurs totales des statistiques ont été doublées par rapport à la version originale.



Source File	Total Lines	Source Code Lines	Source Code Lines [%]	Comment Lines	Comment Lines [%]	Blank Lines	Blank Lines [%]
GameFactory.java	35	11	31%	18	51%	6	17%
Ghost.java	88	41	47%	36	41%	11	12%
GhostColor.java	71	27	38%	33	46%	11	15%
GhostFactory.java	64	20	31%	37	58%	7	11%
HallOfFame.java	203	109	54%	76	37%	18	9%
HallOfFameTest.java	84	43	51%	32	38%	9	11%
HumanControllerStrategy.java	74	32	43%	31	42%	11	15%
HumanControllerStrategyTest.java	66	39	59%	16	24%	11	17%
IdentifiedPlayer.java	405	245	60%	128	32%	32	8%
IdentifiedPlayerTest.java	166	102	61%	49	30%	15	9%
ImageSprite.java	74	42	57%	21	28%	11	15%
Inky.java	135	52	39%	68	50%	15	11%
ItemsColor.java	45	24	53%	16	36%	5	11%
ItemsColorTest.java	28	16	57%	7	25%	5	18%
Launcher.java	167	95	57%	54	32%	18	11%
LauncherSmokeTest.java	118	56	47%	44	37%	18	15%
Level.java	342	166	49%	144	42%	32	9%
LevelFactory.java	157	70	45%	67	43%	20	13%
LevelTest.java	141	74	52%	54	38%	13	9%
MapParser.java	191	114	60%	56	29%	21	11%
MyJDialogStrategy.java	120	69	57%	43	36%	8	7%
MyJDialogStrategyTest.java	31	18	58%	7	23%	6	19%
Navigation.java	197	98	50%	79	40%	20	10%
NavigationTest.java	143	91	64%	38	27%	14	10%
NPC.java	28	7	25%	17	61%	4	14%
Total:	9722	4934	51%	3607	37%	1181	12%

FIGURE 13 – Statistiques sur les lignes de codes

2) Métrique de complexité

Ces métriques sont utilisées pour mesurer la complexité d'une partie du logiciel. Elles sont souvent corrélées avec les métriques de taille parce qu'un très grand système a de grandes chances d'être également complexe.

a) Complexité

Cette métrique permet de calculer la complexité cyclomatique des différentes classes du projet. La complexité cyclomatique d'une fonction peut être définie comme le nombre de chemins d'exécutions possible dans cette fonction, ce qui est fortement influencé par la présence de boucles ou de conditions dans celle-ci. La complexité cyclomatique d'une méthode vaut au minimum 1, puisqu'il y a toujours au moins un chemin. Au niveau de l'interprétation, la complexité cyclomatique d'une méthode augmente proportionnellement au nombre de points de décision, ce qui signifie qu'une méthode avec une haute complexité est plus difficile à comprendre et à maintenir. En général, une complexité cyclomatique trop élevée indique qu'il faut refactoriser la méthode. Par contre, si une méthode a une complexité convenable (en dessous d'un certain seuil), elle peut être acceptable si elle est suffisamment testée. Cette métrique est liée à la métrique de couverture du code par les tests. En effet, une méthode devrait avoir un nombre de tests unitaires égal à sa complexité cyclomatique pour avoir un "code coverage" de 100%, ainsi chaque chemin de la méthode est testé.

Logiciel initial

Les figure 14 et 15 mettent en évidence les classes et les méthodes possédant une complexité au-dessus d'un certain seuil (les classes ou méthodes en dessous du seuil ne sont pas représentées par souci de lisibilité). Cette analyse nous permet de mettre en évidence que certaines classes ou méthodes sont potentiellement complexes par rapport aux autres au sein du projet. Elles doivent donc être surveillées de près dans les analyses du logiciel final.

method	ev(G)	iv(G)	v(G)
nl.tudelft.jpacman.board.Board.invariant()	4	1	4
nl.tudelft.jpacman.level.CollisionInteractionMap.collide(C1,C2)	4	1	4
nl.tudelft.jpacman.level.LevelFactory.createGhost()	6	6	6
nl.tudelft.jpacman.level.MapParser.checkMapFormat(List<String>)	6	2	6
nl.tudelft.jpacman.npc.ghost.Clyde.nextMove()	4	5	5
nl.tudelft.jpacman.npc.ghost.Inky.nextMove()	5	8	8
nl.tudelft.jpacman.npc.ghost.Navigation.shortestPath(Square,Square,Unit)	4	3	4
Total	**33**	**26**	**37**
Average	4,71	3,71	5,29

FIGURE 14 – Métrique de complexité pour les méthodes

class	OCavg	WMC
nl.tudelft.jpacman.level.Level	2,36	33
nl.tudelft.jpacman.npc.ghost.Navigation	3,20	16
Total		**49**
Average	2,78	24,50

FIGURE 15 – Métrique de complexité pour les classes

Logiciel final

Les figure 16 et 17 mettent en évidence ces métriques pour la version finale du logiciel. En observant ces valeurs, on peut en déduire que la complexité de ces méthodes reste acceptable car chaque méthode ajoutée lors des extensions a fait l'objet de tests unitaires. En observant ces résultats, on constate que très peu de méthodes, par rapport au nombre de méthodes qui ont été rajoutés avec les trois extensions, ont une complexité trop élevée. De plus, la plupart de ces méthodes interviennent dans la gestion de l'intelligence artificielle manipulant Pacman. Vu que ces méthodes utilisent beaucoup d'information et un grand nombre de possibilités différentes, il n'est pas étonnant que leur complexité soit plus élevée. Malgré tout, ces méthodes ont fait l'objet de tests unitaires permettant de valider leurs comportements.

method	ev(G)	iv(G)	v(G)
nl.tudelft.jpacman.board.Board.invariant()	4	1	4
nl.tudelft.jpacman.board.Board.isSafe(int,int)	5	3	5
nl.tudelft.jpacman.game.Achievement.offerAchievements(IdentifiedPlayer)	1	10	10
nl.tudelft.jpacman.level.CollisionInteractionMap.collide(C1,C2)	4	1	4
nl.tudelft.jpacman.level.LevelFactory.createGhost()	6	6	6
nl.tudelft.jpacman.level.MapParser.checkMapFormat(List<String>)	6	2	6
nl.tudelft.jpacman.npc.ghost.Clyde.nextMove()	4	5	5
nl.tudelft.jpacman.npc.ghost.Inky.nextMove()	5	8	8
nl.tudelft.jpacman.npc.ghost.Navigation.shortestPath(Square,Square,Unit)	4	3	4
nl.tudelft.jpacman.strategy.AStar.compute(T)	4	4	5
nl.tudelft.jpacman.strategy.AStarPath.g(Square,Square)	8	6	9
nl.tudelft.jpacman.strategy.PacManhattanAI.nextMove()	7	8	8
Total	**58**	**57**	**74**
Average	4,83	4,75	6,17

FIGURE 16 – Métrique de complexité pour les méthodes

class	OCavg	WMC
nl.tudelft.jpacman.level.IdentifiedPlayer	2,24	47
nl.tudelft.jpacman.level.Level	2,12	34
nl.tudelft.jpacman.npc.ghost.Navigation	3,20	16
nl.tudelft.jpacman.strategy.PacManhattan	3,08	40
Total		**137**
Average	2,66	34,25

FIGURE 17 – Métrique de complexité pour les classes

3) Métrique de couplage et de cohésion

Les métriques de couplage décrivent les dépendances entre les composants du système. Plus le couplage entre deux composants est fort, plus le système est compliqué à comprendre, modifier ou corriger.

Les métriques de cohésion décrivent le taux de connectivité au sein d'un composant du logiciel. Celle-ci mesure donc le taux d'utilisation des attributs d'une classe par les méthodes de cette même classe et mesure donc l'application des principes d'encapsulation des données. C'est un facteur important pour la portabilité, la fiabilité et donc pour la qualité globale du système. Une forte cohésion engendre un couplage faible, ce qui signifie que les deux métriques sont donc corrélées entre elles.

a) Couplage et cohésion

L'analyse de couplage peut se faire via la métrique CBO (*coupling between object classes*) qui permet de mesurer le nombre d'autres classes couplées à une classe donnée. En effet, CBO représente le couplage en comptant le nombre de classes différentes rattachées à une classe précise autant en entrée qu'en sortie.

En ce qui concerne l'analyse de la cohésion des classes, la métrique LCOM (*Lack of Cohesion in Methods*) peut être utilisée pour mesurer la cohésion entre les méthodes d'une classe. Cette métrique représente le manque de cohésion au sein d'une classe (une grande valeur pour cette métrique représente un manque de cohésion dans cette classe).

Logiciel initial

Par souci de lisibilité dans ce rapport, nous afficherons seulement une partie des classes du projet pour visualiser leurs couplages et leurs cohésions. La figure 18 affiche une moyenne de 8.78 pour la métrique CBO et 1.58 pour la métrique LCOM du projet.

On constate également une valeur moyenne de 2.02 pour la métrique DIT (*Depth Inheritence Tree*) qui sert de mesure structurelle sur la hiérarchie des classes du systèmes. Pour une classe donnée, DIT représente la longueur du chemin pour se rendre de la racine à cette classe en passant par les liens d'héritage.

Et pour finir, on observe une valeur moyenne de 7.73 pour la métrique WMC (*Weighted Methods per Class*) qui sert de mesure de taille et de complexité puisqu'elle additionne le nombre de méthodes d'une classe pondéré par un facteur de complexité quelconque.

class	CBO	DIT	LCOM	NOC	RFC	WMC
nl.tudelft.jpacman.npc.ghost.GhostFactory	10	1	1	0	10	5
nl.tudelft.jpacman.npc.ghost.Inky	9	4	2	0	14	9
nl.tudelft.jpacman.npc.ghost.Navigation	9	1	2	0	22	16
nl.tudelft.jpacman.npc.ghost.Navigation.N	3	1	2	0	7	6
nl.tudelft.jpacman.npc.ghost.Pinky	8	4	2	0	14	6
nl.tudelft.jpacman.npc.NPC	8	2	2	1	2	0
nl.tudelft.jpacman.PacmanConfigurationE>	3	4	0	0	4	2
nl.tudelft.jpacman.sprite.AnimatedSprite	6	1	1	0	16	15
nl.tudelft.jpacman.sprite.EmptySprite	4	1	4	0	4	4
nl.tudelft.jpacman.sprite.ImageSprite	3	1	1	0	15	8
nl.tudelft.jpacman.sprite.PacManSprites	13	2	1	0	17	9
nl.tudelft.jpacman.sprite.Sprite					4	
nl.tudelft.jpacman.sprite.SpriteStore	5	1	2	1	15	7
nl.tudelft.jpacman.ui.Action					1	
nl.tudelft.jpacman.ui.BoardPanel	7	5	1	0	20	7
nl.tudelft.jpacman.ui.ButtonPanel	3	5	0	0	6	3
nl.tudelft.jpacman.ui.PacKeyListener	2	1	3	0	7	5
nl.tudelft.jpacman.ui.PacManUI	10	6	1	0	22	5
nl.tudelft.jpacman.ui.PacManUiBuilder	7	1	1	0	13	11
nl.tudelft.jpacman.ui.ScorePanel	4	5	1	0	16	8
nl.tudelft.jpacman.ui.ScorePanel.ScoreFor					1	
Total						**348**
Average	8,78	2,02	1,58	0,35	10,88	7,73

FIGURE 18 – Métriques de cohesion et couplage

Logiciel final

Effectuons cette même analyse sur la version finale où les résultats sont reportés dans la figure 19. On observe que la moyenne de la métrique CBO du projet est désormais de 10.63 contre 8.78 dans la version initiale. La valeur moyenne de la métrique LCOM du projet est maintenant de 1.83 contre 1.58 précédemment. La métrique DIT atteint la valeur moyenne de 1.98 contre 2.02. Pour terminer, la moyenne de la métrique WMC du projet est désormais de 10.14 contre 7.73 dans la version initiale.

Mise à part la métrique DIT, on constate que les métriques de cohésion et de couplage ont légèrement augmenté par rapport à la version initiale. Ces résultats ne sont pas étonnant car avec l'ajout de trois extensions qui ont modifié considérablement le jeu initial, il est normal que les couplages et cohésions entre classes augmentent ne serait-ce que légèrement.

class	CBO	DIT	LCOM	NOC	RFC	WMC
nl.tudelft.jpacman.sprite.SpriteStore	5	1	2	1	15	7
nl.tudelft.jpacman.strategy.AIStrategy	11	2	4	2	10	5
nl.tudelft.jpacman.strategy.AStar	5	1	1	1	21	12
nl.tudelft.jpacman.strategy.AStar.Path	2	1	2	0	5	5
nl.tudelft.jpacman.strategy.AStarPath	12	2	3	0	27	23
nl.tudelft.jpacman.strategy.HumanControllerStrategy	2	3	0	12	5	
nl.tudelft.jpacman.strategy.PacManhattan.	14	3	2	0	50	40
nl.tudelft.jpacman.strategy.PacmanStrateg	10	1	4	2	6	4
nl.tudelft.jpacman.strategy.PacmanStrateg	7		0		0	0
nl.tudelft.jpacman.ui.Action					1	
nl.tudelft.jpacman.ui.BoardPanel	7	5	1	0	20	7
nl.tudelft.jpacman.ui.ButtonPanel	2	5	0	0	9	2
nl.tudelft.jpacman.ui.MyJDialogStrategy	8	6	2	0	23	4
nl.tudelft.jpacman.ui.MyJDialogStrategy.M	5	1	1	0	8	3
nl.tudelft.jpacman.ui.PacKeyListener	2	1	3	0	7	5
nl.tudelft.jpacman.ui.PacManUI	13	6	3	0	48	10
nl.tudelft.jpacman.ui.PacManUiBuilder	9	1	1	0	12	8
nl.tudelft.jpacman.ui.PlayerInfosPanel	5	5	1	0	17	8
nl.tudelft.jpacman.ui.PlayerInfosPanel.Scor					1	
Total						**639**
Average	10,63	1,98	1,83	0,38	15,49	10,14

Figure 19 – Métriques de cohesion et couplage

b) Dépendance

Cette métrique permet de connaître les différentes dépendances entre classes. Ces dépendances doivent toujours être plus ou moins faibles afin de garder une bonne modularité du logiciel.

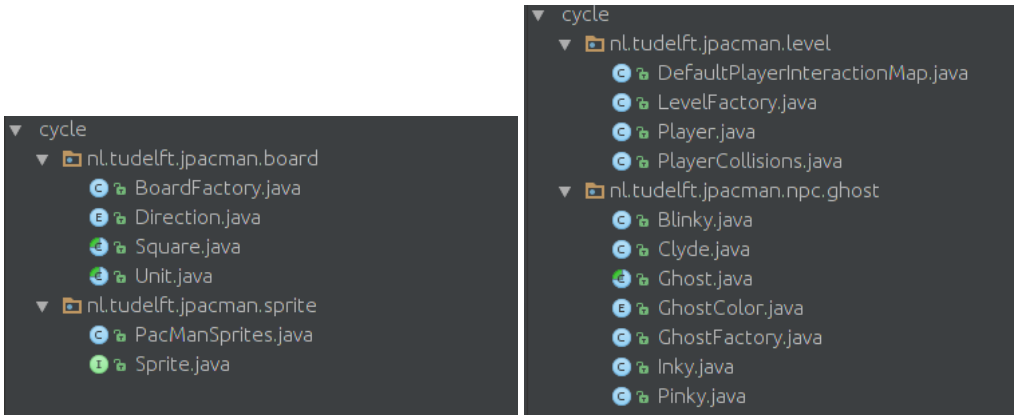
Logiciel initial

Par souci de lisibilité dans ce rapport, nous afficherons seulement une partie des classes du projet pour visualiser leurs dépendances. La figure 20 montre que les dépendances sont relativement correcte.

Class	Cyclic	Dcy	Dcy*	Dpt	Dpt*
nl.tudelft.jpacman.board.Board	0	1	4	8	17
nl.tudelft.jpacman.board.BoardFactory	0	6	14	4	5
nl.tudelft.jpacman.board.BoardFactory.Gr	0	3	4	1	6
nl.tudelft.jpacman.board.BoardFactory.Wa	0	3	4	1	6
nl.tudelft.jpacman.board.Direction	0	0	0	21	45
nl.tudelft.jpacman.board.Square	1	3	3	23	43
nl.tudelft.jpacman.board.Unit	1	3	3	24	43
nl.tudelft.jpacman.game.Game	0	4	16	7	7
nl.tudelft.jpacman.game.GameFactory	0	4	24	1	2
nl.tudelft.jpacman.game.SinglePlayerGame	0	3	17	1	3
nl.tudelft.jpacman.Launcher	0	18	57	1	1
nl.tudelft.jpacman.level.CollisionInteractio	0	10	19	1	1
nl.tudelft.jpacman.level.CollisionInteractio	0	3	5	1	2
nl.tudelft.jpacman.level.DefaultPlayerInter	0	8	28	0	0
nl.tudelft.jpacman.level.Level	1	10	15	10	13
nl.tudelft.jpacman.level.Level.NpcMoveTa	1	3	15	1	13
nl.tudelft.jpacman.level.LevelFactory	0	11	31	3	4
nl.tudelft.jpacman.level.LevelFactory.Rand	0	3	6	1	5
nl.tudelft.jpacman.level.MapParser	0	8	35	2	3
nl.tudelft.jpacman.level.Pellet	0	2	4	5	16
nl.tudelft.jpacman.level.Player	0	4	6	15	24
nl.tudelft.jpacman.level.PlayerCollisions	0	5	13	1	5
nl.tudelft.jpacman.level.PlayerFactory	0	2	12	2	3
Total					
Average	0,09	4,58	11,53	4,29	10,98

FIGURE 20 – Métrique de dépendance

On peut également s'intéresser à la dépendance cyclique, une métrique qui met en évidence les classe dépendantes l'une de l'autre. Les différentes classes possédant une dépendance cyclique sont représentées dans la figure 21. En analysant le fonctionnement de ces différentes classes, on remarque que leur interdépendance est plutôt logique. Par exemple, il est normale que les classes du package "ghost" soient liées aux classes du package "sprite" car les fantômes sont représentés visuellement dans le jeu.



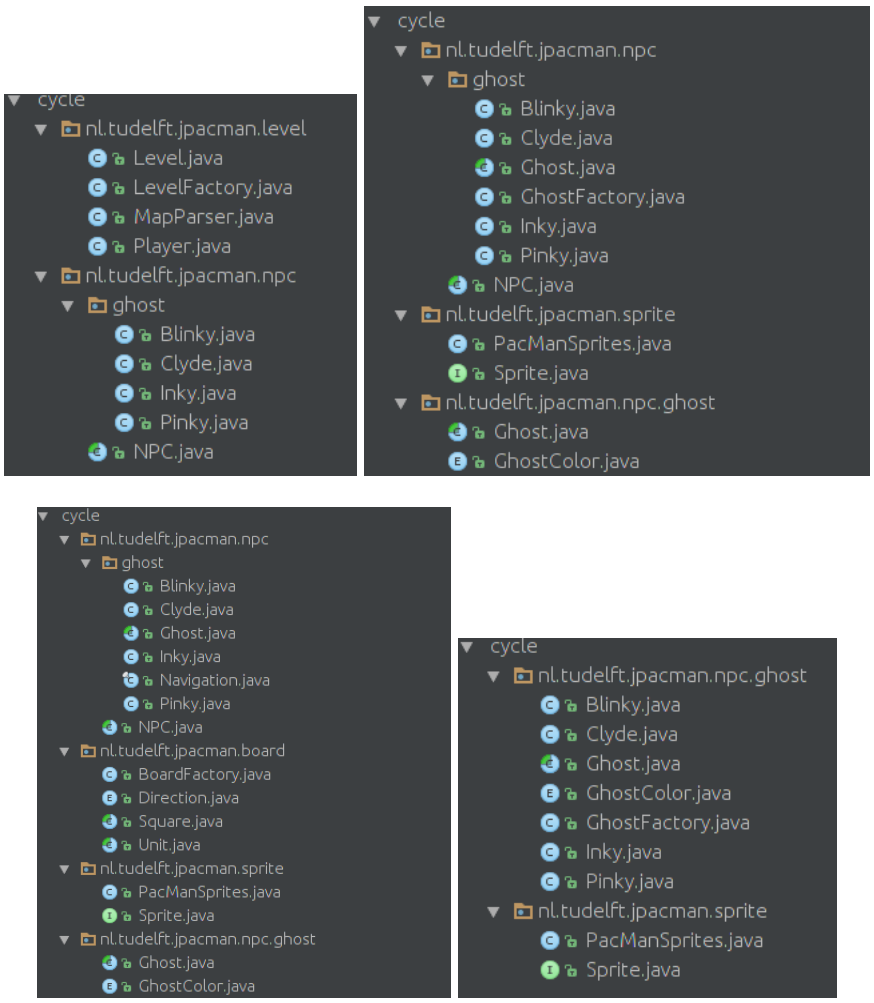


FIGURE 21 – Dépendances cycliques

La matrice de dépendance qui est représentée par la figure 22 permet de mieux visualiser ces dépendances entre classes. Dans cette matrice, les cases bleues représentent les dépendances unidirectionnelles tandis que les cases rouges représentent les dépendances bidirectionnelles (cycles). Il est donc conseillé d'éviter au maximum les cycles.

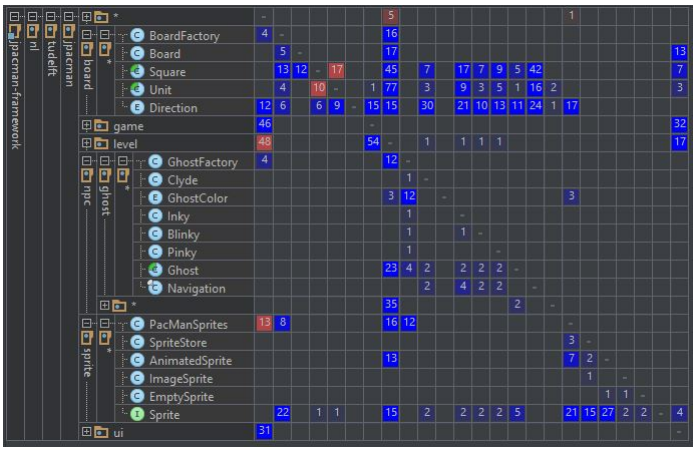


FIGURE 22 – Matrice de dépendances

Logiciel final

Encore une fois, seule une partie des classes seront affichées par souci de lisibilité dans le rap-

port. La figure 23 montre les dépendances entre les classes après intégration des trois fonctionnalités. On peut facilement observer que le nombre de dépendances a fortement augmenté.

class	Cyclic	Dcy	Dcy*	Dpt	Dpt*
nl.tudelft.jpacman.board.Board	40	4	70	12	57
nl.tudelft.jpacman.board.BoardFactory	40	8	70	4	57
nl.tudelft.jpacman.board.BoardFactory.Ground	0	3	4	1	58
nl.tudelft.jpacman.board.BoardFactory.Wall	0	3	4	1	58
nl.tudelft.jpacman.board.Direction	0	0	0	32	71
nl.tudelft.jpacman.board.ItemsColor	0	0	0	2	59
nl.tudelft.jpacman.board.Square	1	3	3	31	70
nl.tudelft.jpacman.board.Unit	1	3	3	33	70
nl.tudelft.jpacman.FileChecker	40	1	70	1	57
nl.tudelft.jpacman.game.Achievement	40	1	70	6	57
nl.tudelft.jpacman.game.Game	40	19	70	24	57
nl.tudelft.jpacman.game.Game.PlayerMoveTask	40	5	70	1	57
nl.tudelft.jpacman.game.GameFactory	40	3	70	1	57
nl.tudelft.jpacman.game.HallOfFame	0	0	0	3	59
nl.tudelft.jpacman.game.SinglePlayerGame	40	5	70	3	57
nl.tudelft.jpacman.Launcher	40	12	70	11	57
nl.tudelft.jpacman.level.AILevel	40	12	70	6	57
nl.tudelft.jpacman.level.AILevel.NpcMoveTask	40	3	70	1	57
nl.tudelft.jpacman.level.AILevel.PlayerMoveTask	40	8	70	1	57
nl.tudelft.jpacman.level.CollisionInteractionMap	0	10	19	1	1
Total					
Average	24,79	5,05	45,79	6,03	55,46

FIGURE 23 – Métrique de dépendance

Si on analyse les dépendances cycliques entre les packages, on constate qu'elles ont elles aussi augmenté considérablement. Les figures 24 à 31 représentent les dépendances cycliques entre les différentes classes des packages.

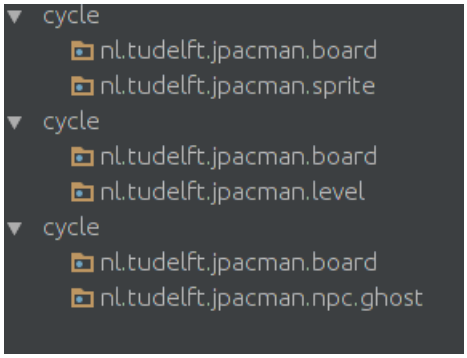


FIGURE 24 – Dépendances cycliques pour le package *board*

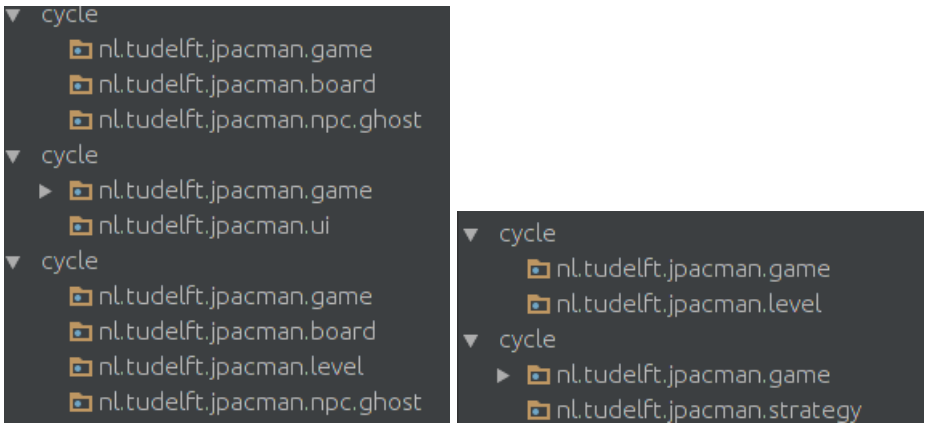


FIGURE 25 – Dépendances cycliques pour le package *game*

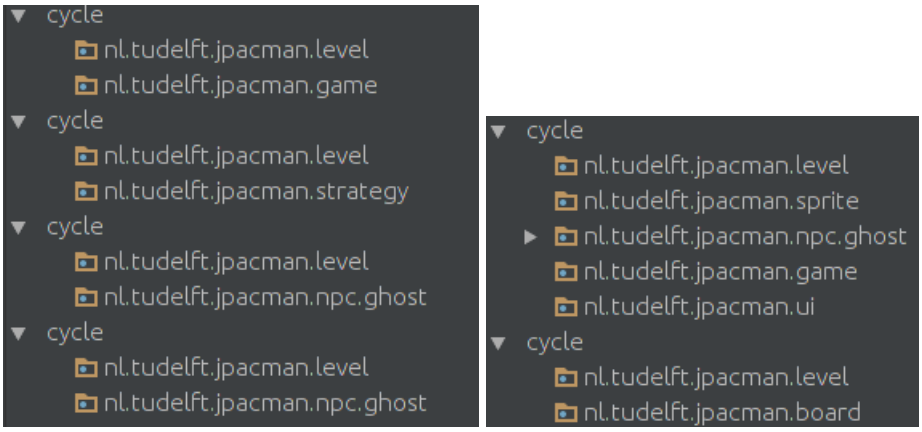


FIGURE 26 – Dépendances cycliques pour le package *level*

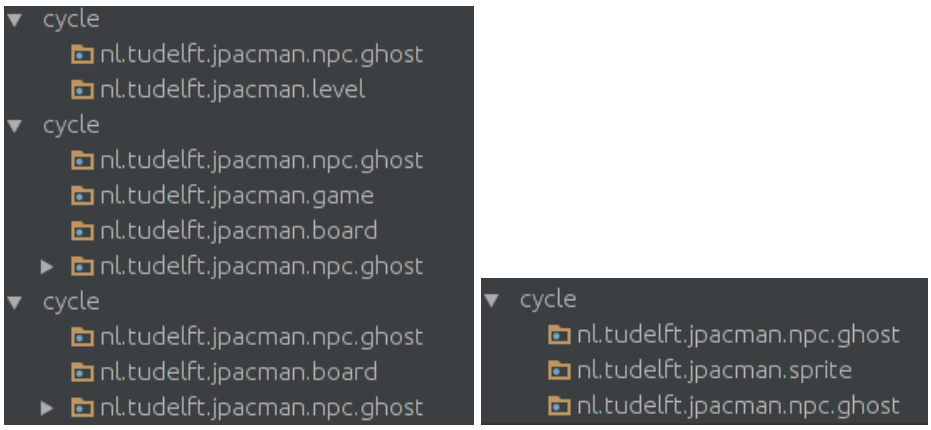


Figure 27 – Dépendances cycliques pour le package *npc*

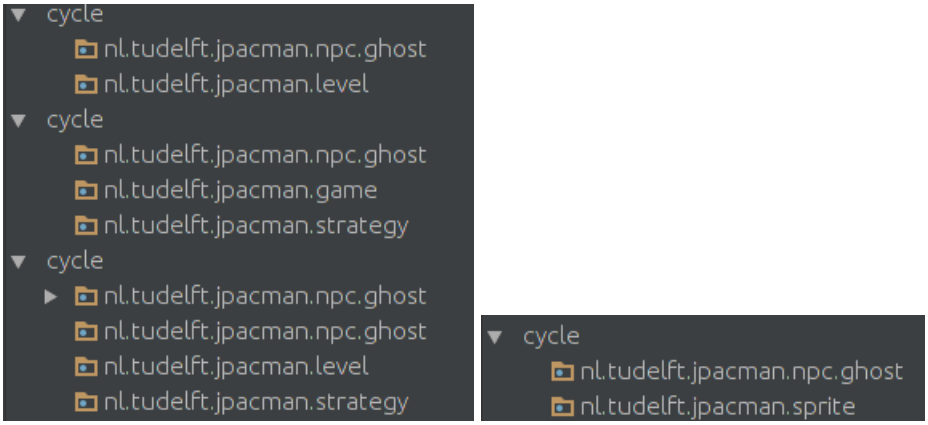


Figure 28 – Dépendances cycliques pour le package *npc.ghost*

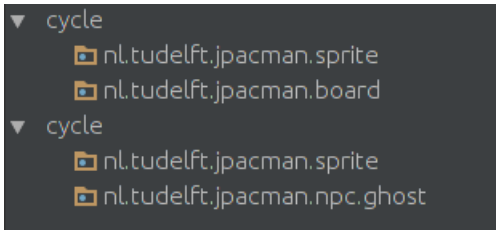


Figure 29 – Dépendances cycliques pour le package *sprite*

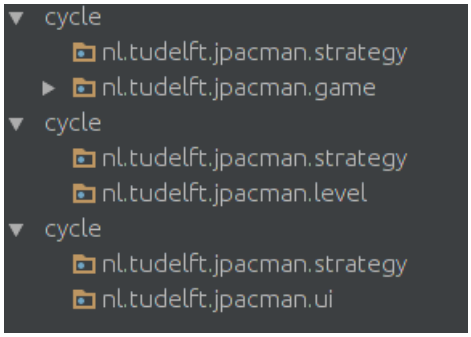


Figure 30 – Dépendances cycliques pour le package *strategy*

La matrice de dépendance qui est représentée par la figure 32 permet de mieux visualiser ces dépendances entre classes. On observe assez facilement que le nombre de dépendances cycliques a augmenté par rapport à la version initiale. Encore une fois, ce résultat peut être expliqué par l'ajout

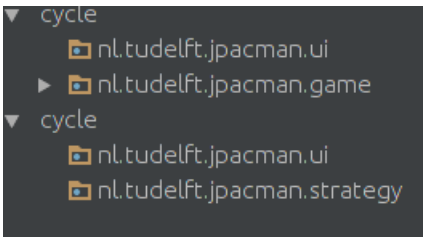


Figure 31 – Dépendances cycliques pour le package *ui*

de nombreuses modifications par rapport à cette version. En effet, la version initiale implémente un jeu très basique avec une seule fonctionnalité (un mode de jeu sur un niveau) tandis que la version finale implémente un jeu plus "élaboré" contenant plusieurs fonctionnalités (plusieurs modes de jeu et niveaux,...). Il est donc évident que cette amélioration a un impact direct sur les dépendances entre classes car les fonctionnalités ajoutées interagissent ensemble. On peut donc en conclure que ces résultats sont tout à fait convenables et cohérents si on prend en considération l'ampleur des modifications qui ont été effectuées.

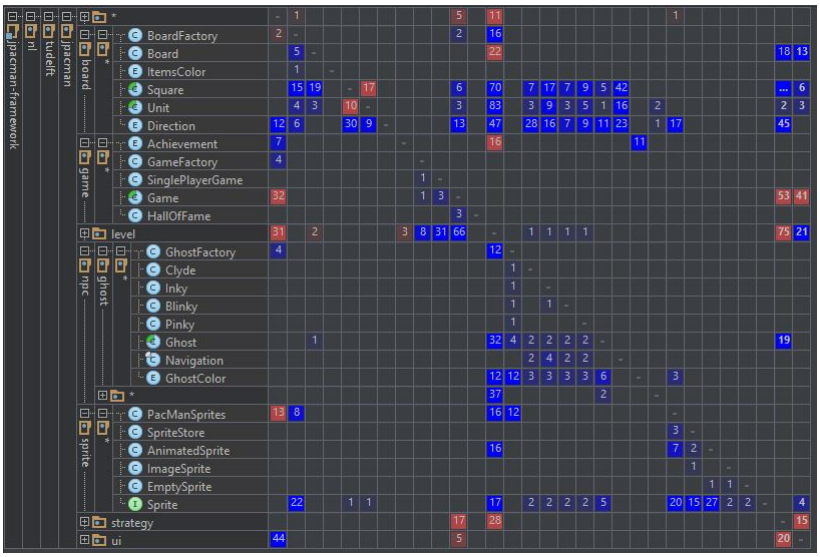


Figure 32 – Matrice de dépendances

Analyse de SonarQube

Avant de terminer cette analyse au niveau des métriques, le logiciel *SonarQube* a été utilisé pour avoir les résultats d'un autre logiciel d'analyse de qualité du code. *SonarQube* est un outil tout-en-un, indépendant des IDEs, qui permet de calculer et monitorer la qualité logicielle, de voir l'évolution entre les versions du logiciel et aussi de comparer ces versions entre elles très facilement. C'est un outil open source qui a pour objectif de fournir une analyse complète de la qualité d'une application en fournissant de nombreuses statistiques (ou métriques) sur un projet. Ces données permettent d'évaluer la qualité du code, et d'en connaître l'évolution au cours du développement.

Nous pouvons ainsi encore une fois comparer les résultats obtenus (pour certaines métriques) pour la version finale avec le version initiale. Il est important de spécifier que les métriques utilisés par ce logiciel ne sont pas nécessairement les mêmes que celles utilisées précédemment, ce qui implique que les résultats ne sont pas exactement les mêmes que lors des analyses précédentes.

Logiciel initial

Effectuons une analyse de certaines métriques sur la version initiale dont on a reporté les résultats dans les figures 33 à 36.

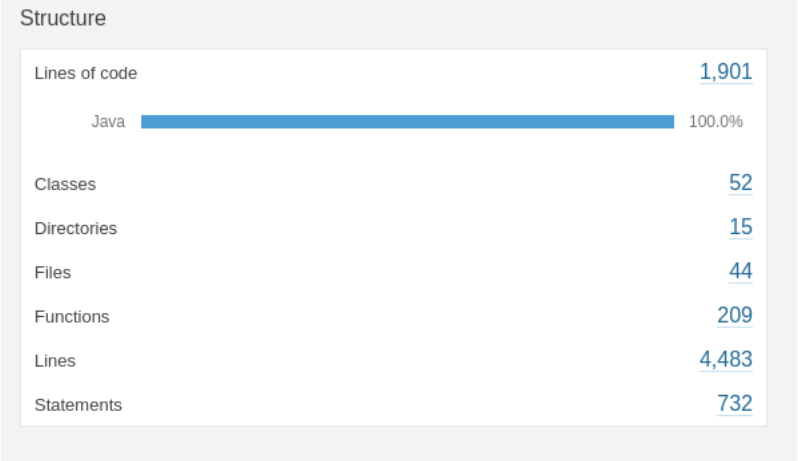


FIGURE 33 – Caractéristique de la structure

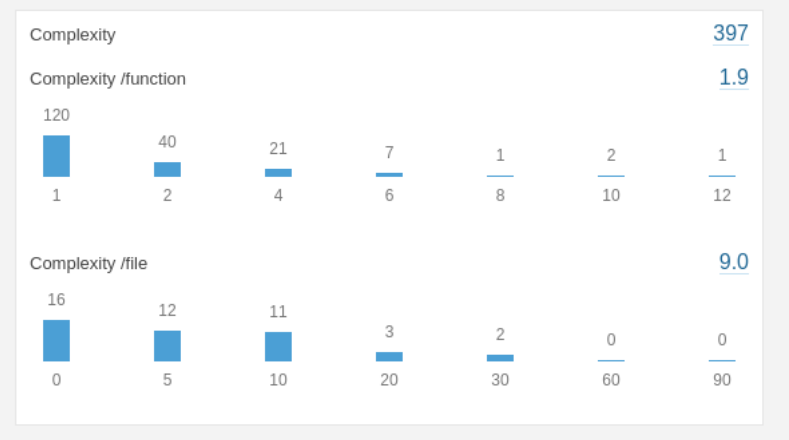


FIGURE 34 – Caractéristique de la complexité

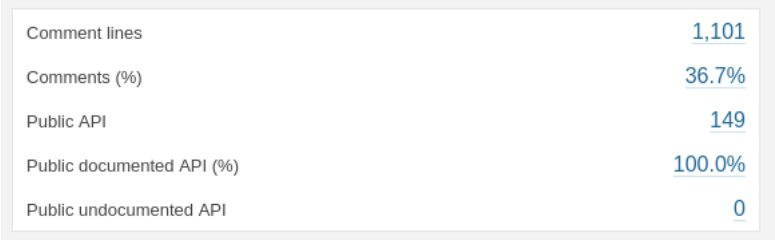


FIGURE 35 – Pourcentages des commentaires et de la documentation

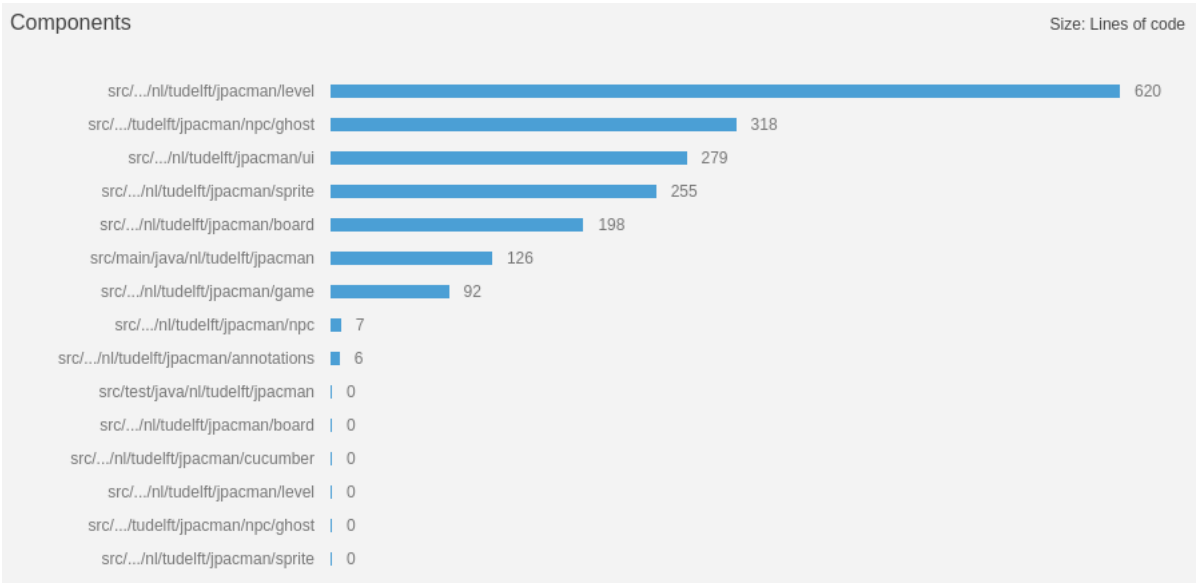


FIGURE 36 – Répartition des lignes de codes

Logiciel final

Effectuons, à présent, la même analyse avec la version finale dont on a reporté les résultats dans les figures 37 à 40.

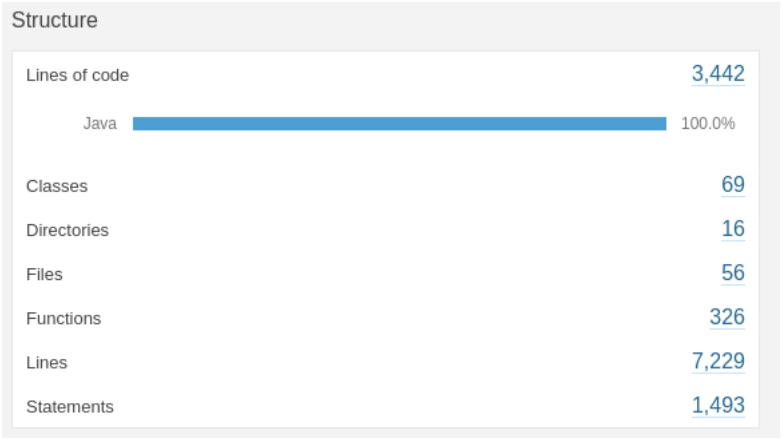


FIGURE 37 – Caractéristique de la structure

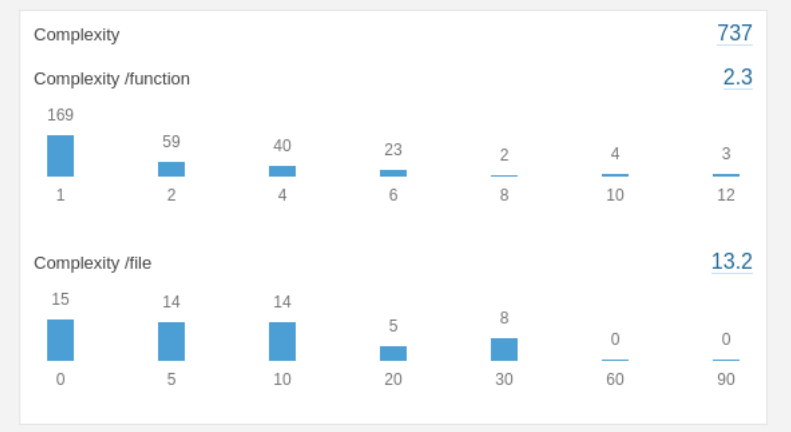


FIGURE 38 – Caractéristique de la complexité

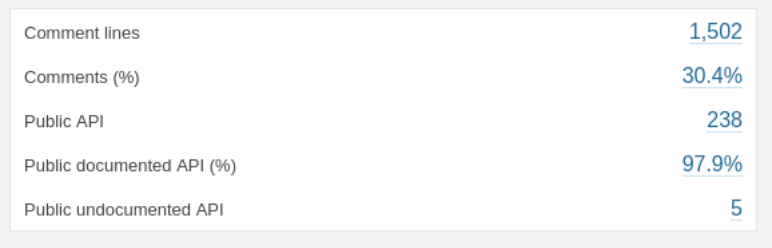


FIGURE 39 – Pourcentages des commentaires et de la documentation

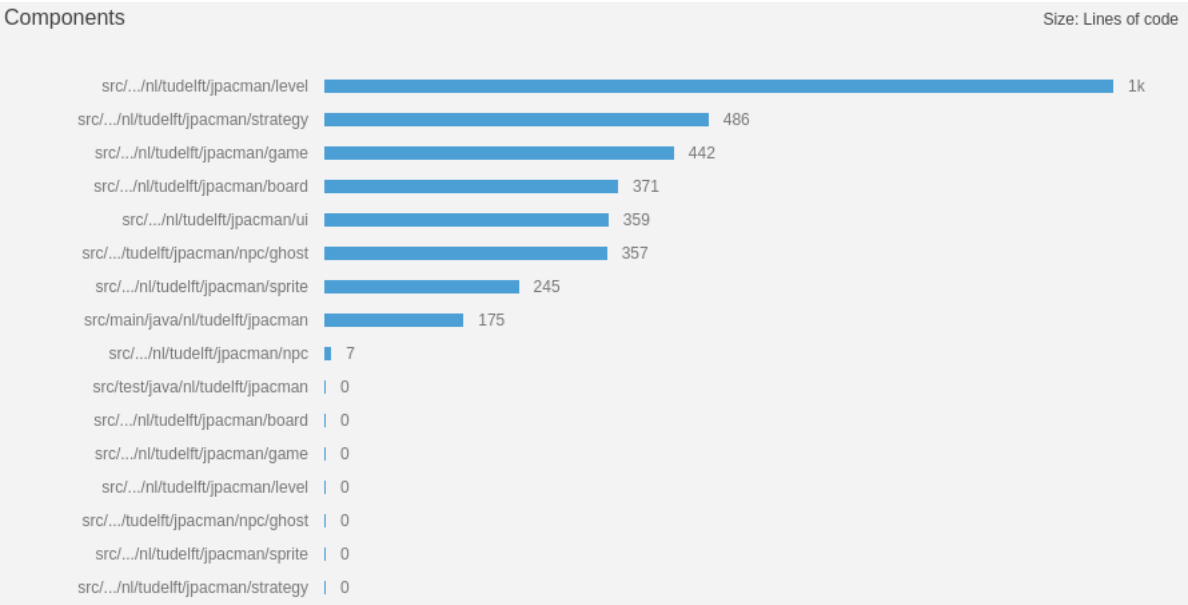


FIGURE 40 – Répartition des lignes de codes

Le logiciel *SonarQube* peut également effectuer une comparaison globale des deux versions dont on a reporté les résultats dans la figure 41. Cette comparaison est très utile car elle permet de mettre en évidence la relation directe des résultats obtenus pour les métriques des deux versions.

On constate, assez facilement, que les résultats obtenus lors de cette analyse sont similaires aux résultats précédents.

	INITIALVERSION 6.4.0 15:04	FINALVERSION 7.3.0 15:05
Lines of code	1 901	3 442
Complexity	397	737
Comments (%)	36,7%	30,4%
Duplicated lines (%)	0,0%	1,1%
Duplicated lines	0	83
Classes	52	69
Technical Debt	1d	3d 5h
Files	44	56
Unit tests		87
Complexity /function	1,9	2,3
Complexity /file	9,0	13,2
Comment lines	1 101	1 502
Functions	209	326
Lines	4 483	7 229
Directories	15	16

FIGURE 41 – Comparaison globale entre la version initiale et la version finale

5.1.2 Code dupliqué

Puisque nous utilisons tous les trois IntelliJ IDEA, l'outil intégré a été utilisé dans un premier temps. La figure 42 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 :

```
1 assertEquals(square.getSquareAt(Direction.WEST).getY(), 15);
2 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. Les parties de code identifiées comme « duplicats » et qui nous ont semblé nécessiter un refactoring en ont fait l'objet.

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 concernait la classe *AStarPathTest* pour laquelle les méthodes *hTest* et *gTest* contenaient, 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).
    getSquareAt(Direction.EAST);

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

Ce problème de duplication de code a été résolu en regroupant le contenu des méthodes *hTest* et *gTest* au sein d'une unique méthode de test. En effet, ces deux méthodes ayant un comportement très proche l'un de l'autre, il est évident qu'on puisse les regrouper ensemble.

5.1.3 Audit de code

1) CheckStyle

CheckStyle est un outil de contrôle de code permettant de vérifier le style d'un code source écrit en langage Java.

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).

La liste suivante présente ces warnings par ordre de nombre de « violations » :

- 148 violations de type *JavadocStyle* : en réalité toutes des « First sentence should end with a period. »→ la totalité de ces warnings ont été réglés.
- 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 → Ces éléments ont été ajoutés.
- Des violations relatives à des tableaux déclarés à la « mode C » plutôt qu'à la « mode Java »→ Des modifications ont été effectuées.
- D'autres violations plus « isolées » non reprises ici.

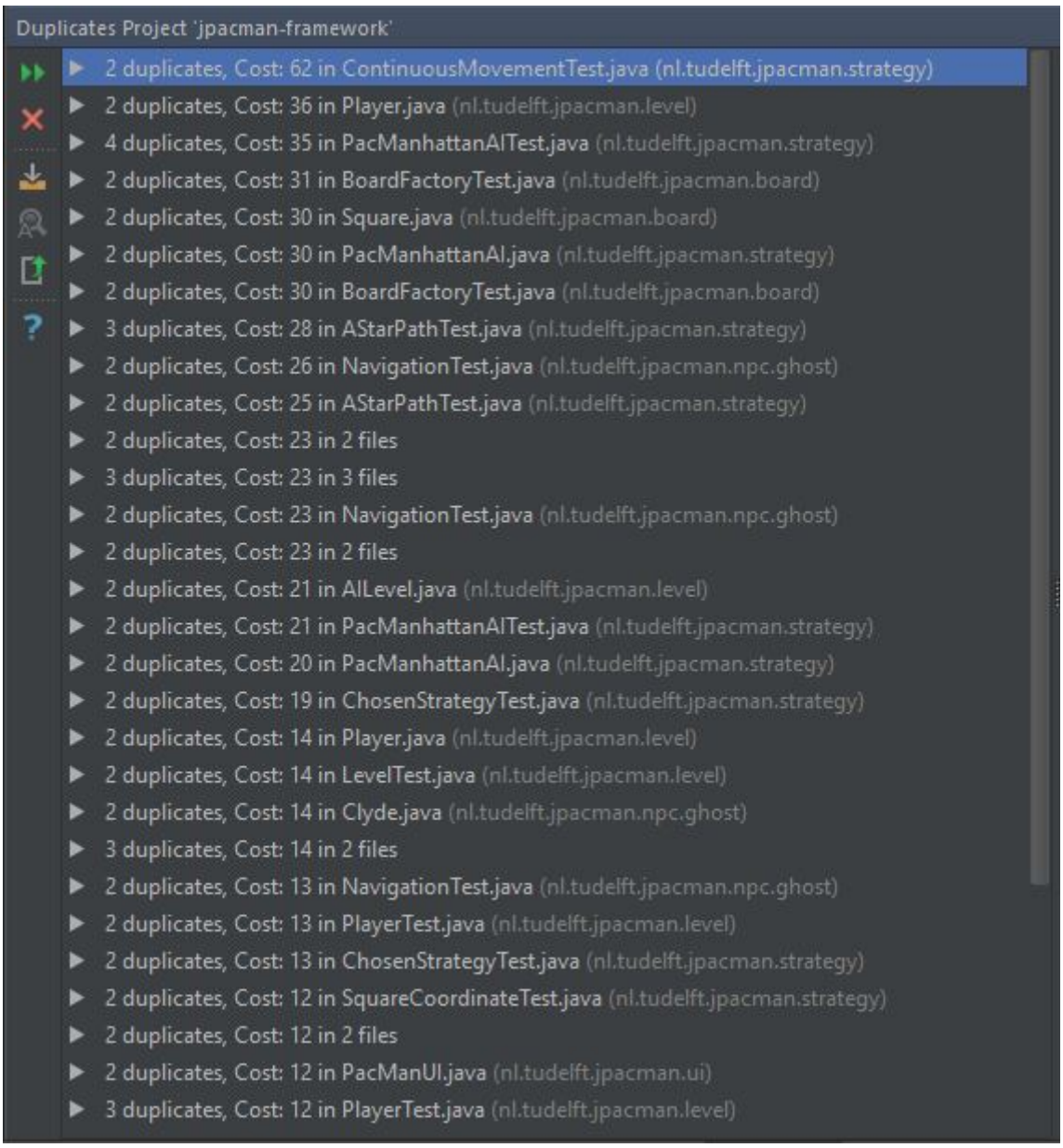


FIGURE 42 – Recherche de code dupliqué via l'outil intégré à *IntelliJ IDEA*

Comme mentionné ci-dessus, les problèmes de style ont été réglés et on a retrouve donc bien la situation du code d'origine : aucun avertissement pour de tels problèmes.

2) IntelliJ Code inspection

C'est un outil d'analyse de code qui applique des inspections sur celui-ci. Il permet de détecter les erreurs de compilations mais aussi les différentes inefficacité présents dans le code.

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 pouvant 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 »).

Altérations mineures 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,
- 2 paramètres n'étaient jamais utilisés et furent ôtés,
- 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. De même dans *AStarPathTest*,
- 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 testent maintenant correctement les fonctionnalités qui leur sont assignées.

« 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*).

Classe obsolète : La classe *DefaultPlayerInteraction* était non utilisée et, comme nous avons modifié les interactions lors de collisions, elle n'avait pas été mise à jour. Elle a donc été supprimée.

3) PMD

PMD est un framework qui permet d'analyser le code source Java. Il contient un certain nombre de règles qui assurent la qualité du code : le code inutile, les imbrications trop complexes,... Il permet d'obtenir les résultats par le biais d'un rapport.

« 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 [5], 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.

La classe *Player* regroupait l'implémentation de deux fonctionnalités séparées (score et labyrinthe) en plus des services de base, dès lors, il a été préféré de la diviser en plusieurs : le comportement lié à la fonctionnalité « score » a été déplaçé dans la classe descendante *IdentifiedPlayer* et les méthodes de vérification de contenu de fichiers ont été déplaçées vers *FileChecker*. A ce jour, il ne reste plus de « god class » dans l'implémentation.

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

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

Loi de Demeter : PMD détecte de nombreuses violations par rapport à cette 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.

4) FindBugs

FindBugs est un logiciel libre d'analyse statique de bytecode Java. Il a pour objectif de trouver des bugs dans les programmes Java en identifiant des patterns reconnus comme étant des bugs.

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

Cet outil a détecté que plusieurs opérations d' écriture et lecture de fichiers ne spécifiaient pas le « charset »utilisé et que certaines opérations de concaténation de chaînes de caractères étaient effectuées dans des boucles. les opérations de lecture et d' écriture de fichier spécifient maintenant le « charset » et les concaténations de chaînes de caractères ont été remplacées par des « StringBuilder.append() ».

5.2 Analyse dynamique

Nous allons maintenant exécuter le code et observer le comportement du logiciel dans ce cas. Nous allons également évaluer la portée des tests unitaires afin de vérifier que tout le code (ou une bonne partie) est couverte et donc testées.

5.2.1 Couverture du code

La couverture du code est importante pour connaître les portions de code inutiles, non testées, ou jamais appelées à l'exécution du logiciel.

Couverture globale Avant l'ajout de nos fonctionnalités, le pourcentage de code couvert par des tests était comme suit :

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	88.1% (59/ 67)	86.5% (346/ 400)	80.4% (1438/ 1788)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
nl.tudelft.jpacman.npc	100% (1/ 1)	100% (1/ 1)	100% (1/ 1)
nl.tudelft.jpacman.strategy	100% (8/ 8)	95.9% (47/ 49)	87.8% (223/ 254)
nl.tudelft.jpacman.game	100% (6/ 6)	95.7% (44/ 46)	85.7% (197/ 230)
nl.tudelft.jpacman.board	100% (8/ 8)	94.6% (53/ 56)	81.8% (171/ 209)
nl.tudelft.jpacman.npc.ghost	100% (9/ 9)	94.6% (35/ 37)	92.5% (148/ 160)
nl.tudelft.jpacman.sprite	100% (5/ 5)	86.8% (33/ 38)	88.9% (96/ 108)
nl.tudelft.jpacman.ui	100% (10/ 10)	85% (34/ 40)	85.1% (177/ 208)
nl.tudelft.jpacman.level	58.8% (10/ 17)	74.5% (82/ 110)	70.9% (385/ 543)
nl.tudelft.jpacman	66.7% (2/ 3)	73.9% (17/ 23)	53.3% (40/ 75)

FIGURE 43 – Rapport de couverture de code généré par *IntelliJ IDEA* sur le code initial

Actuellement, ces pourcentages ont évolué de la manière suivante :

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	92.2% (59/ 64)	88.7% (347/ 391)	82.4% (1469/ 1782)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
nl.tudelft.jpacman	66.7% (2/ 3)	73.9% (17/ 23)	57.3% (43/ 75)
nl.tudelft.jpacman.board	100% (8/ 8)	98.2% (54/ 55)	88.7% (180/ 203)
nl.tudelft.jpacman.game	100% (6/ 6)	95.7% (45/ 47)	83.8% (207/ 247)
nl.tudelft.jpacman.level	71.4% (10/ 14)	80.4% (82/ 102)	73% (386/ 529)
nl.tudelft.jpacman.npc	100% (1/ 1)	100% (1/ 1)	100% (1/ 1)
nl.tudelft.jpacman.npc.ghost	100% (9/ 9)	94.6% (35/ 37)	92.5% (149/ 161)
nl.tudelft.jpacman.sprite	100% (5/ 5)	86.8% (33/ 38)	89.8% (97/ 108)
nl.tudelft.jpacman.strategy	100% (8/ 8)	95.9% (47/ 49)	88.8% (223/ 251)
nl.tudelft.jpacman.ui	100% (10/ 10)	84.6% (33/ 39)	88.4% (183/ 207)

FIGURE 44 – Rapport de couverture de code généré par *IntelliJ IDEA* sur notre code

Comme on peut le voir, la couverture globale a non seulement été maintenue dans l'ensemble, mais a augmenté légèrement à tous les niveaux.

Classes où la couverture de code a diminué significativement :

- La classe *BoardFactory* a vu sa couverture diminuer de 6,7% au niveau des méthodes et 27,2% au niveau des lignes étant donné que certaines méthodes liées à la conversion d'images en fichiers de niveaux ne sont pas testées (la conversion en elle-même est testée mais les méthodes l'appelant non, car trop dépendantes des fichiers présents, qui peuvent varier).
- La classe *PlayerCollisions* a subi une légère baisse de sa couverture de lignes de 6,7% du fait qu'une clause « catch » ne soit évidemment pas testée.

Dans le cas des classes préexistantes non mentionnées ci-haut, la couverture s'est soit maintenue, soit a été accrue, parfois fortement.

Classes et packages ajoutés :

- La classe *MyJDialogStrategy* a une couverture de 100% au niveau des classes, 71,4% au niveau des méthodes et 69,8% au niveau des lignes. Les méthodes et lignes non testées dépendent d'une action du joueur.
- La classe *IdentifiedPlayer* est couverte à hauteur de 100% au niveau des classes, 85,7% au niveau des méthodes et 65,6% au niveau des lignes. Les méthodes et lignes non testées sont liées à des affichages graphiques.
- La classe *AILevel* jouit d'une couverture de 66,7% au niveau des classes, 61,5% au niveau des méthodes et 64,4% au niveau des lignes. Les méthodes et lignes non testées dépendent d'un jeu en exécution.
- La classe *HallOfFame* bénéficie d'une couverture de 100% au niveau des classes, 91,7% au niveau des méthodes et 72,7% au niveau des lignes. Les méthodes et lignes non testées sont liées à des affichages graphiques et gestions d' exceptions.
- L' énumération *Achievement* se voit testée pour 100% au niveau des classes, 100% au niveau des méthodes et 92,9% au niveau des lignes. Les lignes non testées correspondent à une gestion d' exceptions.
- L' énumération *ItemsColor* est entièrement couverte.
- La classe utilitaire *FileChecker* dispose d'une couverture de code de 100% au niveau des classes, 66,7% au niveau des méthodes et 51,2% au niveau des lignes. Le contenu non couvert correspond essentiellement à une méthode de lecture de fichier, en corrélation avec la nature de la classe.

— Le package *Strategy* est couvert pour 100% au niveau des classes, 95,9% au niveau des méthodes et 88,8% au niveau des lignes.

  Les classes ci-dessous appartiennent à ce package :

— Les classes *AIStrategy* et *HumanControllerStrategy* sont entièrement couvertes.
— La classe *AStar* a une couverture de 100% au niveau des classes, 100% au niveau des méthodes et 96,4% au niveau des lignes.
— La classe *AStarPath* a une couverture de 100% au niveau des classes, 100% au niveau des méthodes et 90,6% au niveau des lignes.
— La classe *PacmanStrategy* a une couverture de 100% au niveau des classes, 100% au niveau des méthodes et 90,9% au niveau des lignes.
— La classe *PacManhattanAI* a une couverture de 100% au niveau des classes, 92,9% au niveau des méthodes et 81,7% au niveau des lignes. Les méthodes et lignes non testées dépendent de la classe *AStar* qui n'a pas semblé utile de tester car c'est une classe template (voir section 2.3).

Mesures prises pour améliorer la couverture de code : L'analyse de couverture du code a révélé trois problèmes.

1. En premier lieu, nous avions « court-circuité » les tests de l'interface graphique du jeu en lui-même, ceux-ci ont donc été réinstaurés.
2. En deuxième lieu, une méthode (*Achievement#offerAchievements()*) ajoutée tardivement n'était pas testée, le test a donc été rajouté.
3. Enfin, des tests furent rajoutés pour la classe *PacManhattanAI*.

5.2.2 Profilage

Le profilage permet de connaître les parties de codes qui peuvent être problématiques en terme de performances. Les méthodes mises en évidence peuvent être mal optimisée et donc ralentir le logiciel de manière importante.

Pour cette partie de l'analyse dynamique, il sera fait usage de *Yourkit* (version d'évaluation complète). *Yourkit* est un profiler pour les applications écrites en Java créé par la société Yourkit LLC en 2003. On tente ici de déterminer si des endroits du code monopolisent les ressources CPU/-RAM et, le cas échéant, lesquels.

On réalisera 2 scénarios :
— Jeu en mode "IA" qu'on laisse tourner jusqu'à la mort définitive de Pac-man
— Jeu en mode "Joueur" pour lequel on crée un compte, auquel on est connecté automatiquement, lorsque Pac-man perd sa dernière vie, on recommence une fois la partie en chargeant le niveau 1

Processeur et threads : Pour les deux scénarios susmentionnés, l'analyse montre que la méthode *Navigation#shortestPath(Square, Square, Unit)* et en particulier l'appel à *HashSet.add(Object)* prend l'écrasante majorité du temps CPU. La figure 45 montre le résultat de l'analyse « Hot spots » de *Yourkit* sur le scénario "Joueur" (les résultats sur l'autre scénario étant similaires).

On constate donc que l'intelligence artificielle des fantômes (qui était présente dans la version originale du jeu) est gourmande en ressources par rapport au reste du programme, il serait donc

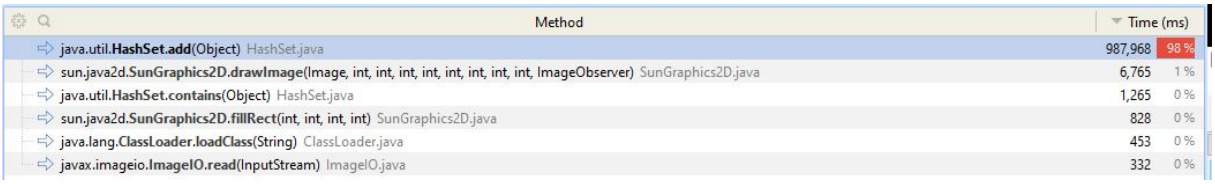


FIGURE 45 – Recherche d'« Hot spots » CPU avec *Yourkit*

intéressant de s'assurer que son comportement est optimisé. Cependant, il nous semble l'être et il n'est pas étonnant que le comportement d'intelligences artificielles prenne du temps CPU. On note également que des méthodes liées à l'affichage et à la gestion des entrées/sorties sont détectées comme « Hot spots » bien que leur impact soit négligeable en comparaison.

De manière générale, on peut voir l'évolution au cours du temps de l'utilisation du processeur et du nombre de threads sur les graphes de la figure 46. Toujours rien d'anormal à ce stade.

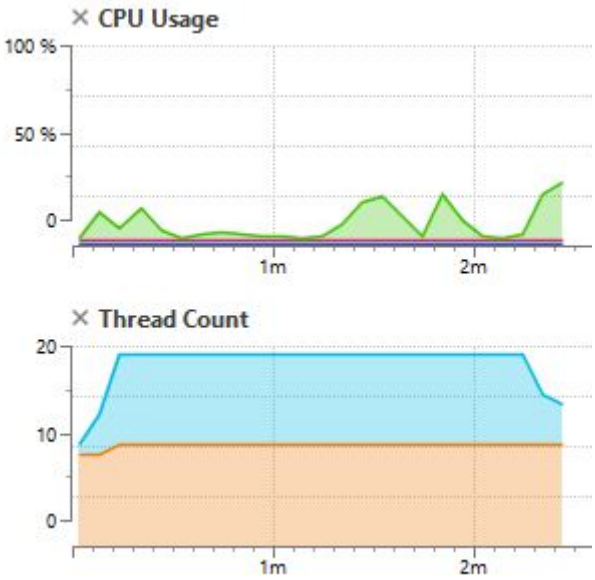


FIGURE 46 – Evolution de l'utilisation CPU et du nombre de threads avec *Yourkit*

Mémoire : Pour les deux scénarios considérés, on constate que la mémoire utilisée reste assez constante, tout comme le nombre de passages du « Garbage Collector ». La figure 47 rend compte des principaux résultats obtenus pour le scénario "IA".

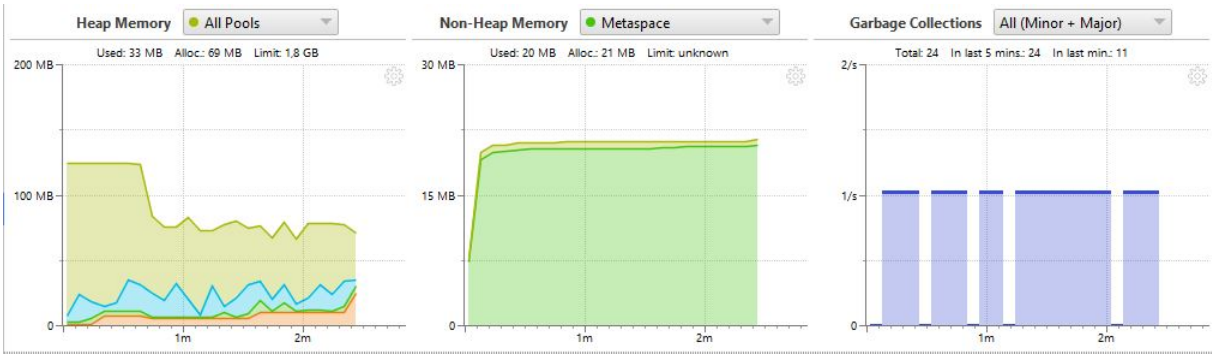


FIGURE 47 – Utilisation mémoire avec *Yourkit*

On ne constate donc toujours pas de problème pour cette partie ci de l'analyse. On notera, comme le montre l'image, qu'il a été choisi de ne montrer que la mémoire utilisée en terme de « metaspace » pour la mémoire « non-heap ». Cela vient du fait que *Yourkit* analyse également le « Code Cache », qui augmente continuellement (comportement de Java qui garde en cache du code) et donne donc une fausse impression d'augmentation.

Il ne nous a pas semblé intéressant de « profiler » la version de JPacman originelle puisque nous n'avons pas repéré de problèmes à ce stade ci alors que la version finale comprend une IA pour Pac-man, nécessitant donc naturellement plus de ressources (pour le moins en termes de CPU).

5.3 Interprétation de la qualité

Après avoir effectué cette analyse de qualité et la comparaison de celle-ci entre la version initiale et la version finale de JPacman, on peut en conclure que la qualité du code source après intégration (version finale) est tout à fait convenable par rapport aux modifications qui ont été effectuées sur le version initiale. En effet, que ce soit au niveau des métriques, du code dupliqué, de la couverture de code,... ou encore du profilage, on peut constater que les résultats obtenus lors des analyses sont tout à fait acceptables et sont parfois très semblables aux résultats de la version initiale grâce à d'importantes améliorations de la qualité qui ont été effectuées pour obtenir la version finale.

6 Analyse de l'évolution du logiciel

Dans cette section, une analyse de l'évolution du logiciel en terme de nombre de lignes de code est effectuée. Pour cela, l'outil *SLOCCount*[6] a été utilisé. Cet outil permet une première analyse très sommaire du code source d'un projet. En effet, il permet de compter les lignes de code que contiennent les fichiers de code source. Il permet donc d'analyser l'évolution du nombre de lignes de code en fonction du temps pour vérifier que celle-ci ne prenne pas de l'ampleur trop rapidement.

Nous allons appliquer cet outil sur trois versions et discuter des résultats qu'il fournit. Les trois versions qui seront analysées sont les suivantes :

1. La version initiale
2. la version juste après intégration des trois fonctionnalités
3. la version finale (après refactorings)

Les résultats retournés sont reportés en graphique dans la figure 48.

Au vu de ces résultats, nous pouvons constater qu'au fil des versions, le projet prend de l'ampleur. Ce qui est évidemment logique vu qu'au fil du temps, des fonctionnalités ont été ajoutées au logiciel initial. Ensuite, nous constatons qu'il y a une diminution du nombre de lignes de codes. Ce comportement de la courbe, peut être facilement expliqué par la phase de *refactoring* qui a succédé à l'intégration des fonctionnalités individuelles. On constate que cette phase a eut un impact positif sur le nombre de lignes de code car elle a permis de diminuer ce chiffre.

6. http ://www.dwheeler.com/sloccount/

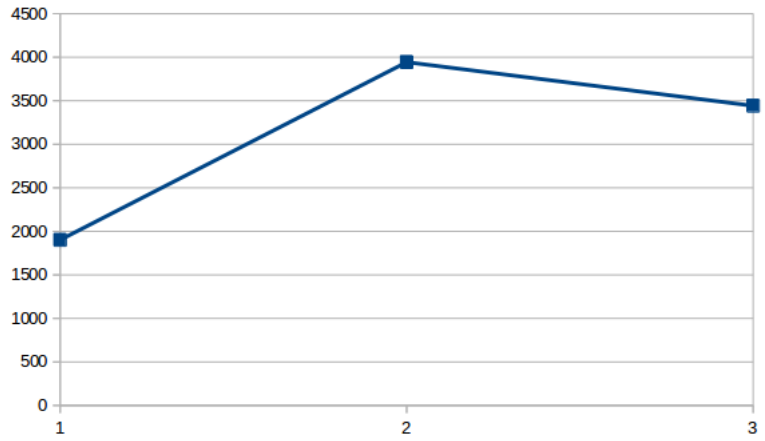


FIGURE 48 – Nombre de lignes de code calculés pour chaque version avec *SLOCCount*

7 Notes diverses et guide d'utilisation

7.1 Maven

Le projet utilisant Maven [7], 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).

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.

7. https://maven.apache.org/

7.2 Travis

Puisque le projet JPacman dans son état « initial » (avant modifications de notre part) intégrait déjà le fichier de configuration Travis [8], 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.

L' historique des « builds » (avec leurs logs) de la version individuelle de Nicolas Leemans est disponible à l'adresse : https://travis-ci.org/NicolasLeemans7/jpacman-framework/builds.

7.3 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 : Final_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`)

Le dépot commun pour la phase d'intégration est celui d'Adrien Coppens, la release finale étant taggée `merged-final-v7.1.0`.

7.4 Spécificités des ajouts individuels

7.4.1 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 [9] 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.

8. https://travis-ci.org/
9. https://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html

— Au lancement du jeu, si le fichier "board$n$.txt" est manquant mais qu'il existe un fichier "n.png", le « convertisseur de carte » mentionné au paragraphe ad hoc de la section 2.2 en page 6 crée le fichier "board$n$.txt" correspondant.

8 Conclusion

Ce projet nous a permis de mettre en pratique les différents concepts vu au cours de Software Evolution, notamment en matière de qualité d'un logiciel, maintenabilité du code, mise en place d'évolution du logiciel ainsi que la surveillance de l'évolution du code. Le projet nous a également permis de nous familiariser avec différents outils de gestion d'un code source, d'analyse et de refactoring. Ces outils nous ont permis d'identifier des problèmes fréquents et potentiels dans le code source, qui peuvent altérer la qualité et rendre l'évolution d'un système plus complexe.

JPacman était un projet relativement bien conçu à la base mais il y avait notamment des petits problèmes de qualité. Nous nous sommes rendus compte également qu'il fallait rester vigilant concernant les résultats fournis par des outils logiciels car ceux-ci peuvent parfois provoquer des problèmes, par exemple du code dupliqué qui ne l'est pas vraiment ou des avertissements que l'on peut considérer comme sans gravité. De plus, les tests unitaires ont permis de gérer les évolutions et les erreurs commises en gardant un contrôle tout au long du processus de développement en effectuant des tests de régression et de validation.

En ce qui concerne la version finale du logiciel, celle-ci a une qualité globalement satisfaisante selon les critères analysées. En effet, par rapport aux modifications qui ont été effectuées lors des extensions, nous sommes parvenus à conserver une complexité plutôt correcte par rapport à la difficulté des fonctions ajoutées telle que l'IA. Ce projet nous a montré que l'ajout de fonctionnalité supplémentaires rend particulièrement difficile la conservation d'une complexité raisonnable si l'on ne tient pas compte des critères d'évaluations de qualité.