



Faculté  
des Sciences

**UMONS**  
Université de Mons

# Software Evolution

## Rapport de projet

### Année académique 2015-2016

Timur YAROL : timur.yarol@student.umons.ac.be

Alexandre LEBRUN : alexandre.lebrun@student.umons.ac.be

Florent BARRACO : florent.barraco@student.umons.ac.be

09 mai 2016



# Table des matières

<b>Introduction</b>	<b>3</b>
<b>1 Outils utilisés</b>	<b>4</b>
<b>2 Compréhension du logiciel</b>	<b>5</b>
<b>3 Extensions individuelles</b>	<b>6</b>
3.1 Fonctionnalité des Super Gommes . . . . .	6
3.2 Fonctionnalité de la map infinie . . . . .	12
3.3 Cases et fruits spéciaux . . . . .	14
<b>4 Intégration et analyse de qualité des extensions</b>	<b>18</b>
4.1 Fusion des fonctionnalités supers gommes et map infinie . . . . .	18
4.1.1 Fusion des codes et gestion des conflits . . . . .	18
4.1.2 Amélioration des performances . . . . .	19
4.2 Fusion des 3 fonctionnalités . . . . .	19
4.2.1 Fusion des codes et gestion des conflits . . . . .	20
4.2.2 Amélioration des performances et ajout de mini-fonctionnalité .	20
4.2.3 Correction de bugs et refactoring de la structure du code . . .	21
4.2.4 Nettoyage du code fusionné . . . . .	22
4.3 Analyse de qualité du code . . . . .	22
4.3.1 Comparaison du code initial et pré-final . . . . .	23
4.3.2 Analyse du code final . . . . .	45
<b>5 Difficultés rencontrées et bugs connus</b>	<b>49</b>
5.1 Difficultés rencontrées . . . . .	49
5.2 Bugs connus . . . . .	49
5.3 Cheat Mode . . . . .	50
<b>Conclusion</b>	<b>51</b>

# Introduction

Le but de ce projet est de mettre en pratique les différents concepts vus au cours théorique de Software Evolution en étendant un logiciel existant en y ajoutant certaines fonctionnalités. Le logiciel fourni initialement est une version très simplifiée du jeu Pac-man.

Au cours de ce projet, plusieurs choses nous ont été demandées. En voici la liste :

- Comprendre et se familiariser avec la structure du code source d'un projet logiciel déjà existant.
- Étendre le logiciel en ajoutant individuellement une nouvelle fonctionnalité. Chacun d'entre nous a du implémenter l'une de ces fonctionnalités.
- Suivre un processus de développement dirigé par les tests tel que vu au cours.
- Intégrer nos modifications individuelles avec les fonctionnalités des autres membres du groupe.
- Utiliser un système de contrôle de versions distribué Git (hébergé sur GitHub).
- Analyser la qualité du code source final afin de le comparer avec celui que nous avions initialement.

Chacune de ces parties seront détaillées au cours de ce rapport.

# Chapitre 1

## Outils utilisés

Tout d'abord, nous allons brièvement décrire les outils que nous avons du utiliser au cours de ce projet.

L'outil de développement, dont une licence universitaire nous a été fournie, se nomme "IntelliJ". Cet IDE possède de base toute une série de plugins permettant de faciliter le développement de notre programme. L'un d'entre eux est Maven.

Nous avons également utilisé un système de contrôle de versions distribué imposé pour ce projet : Git. Ce logiciel nous a permis de faciliter le développement de notre logiciel en nous autorisant de sauver notre projet à chaque nouvel ajout de fonctionnalité. Ainsi, si un soucis survenait au cours de notre développement (ce qui a d'ailleurs été le cas), nous pouvions rebrousser chemin pour mieux repartir. De plus, Git nous a permis de fusionner nos codes de manière plus efficace que manuellement.

Les outils utilisés par la suite pour ce projet ont principalement été des outils d'analyses de codes proposés dans le PDF sur le projet tel que :

- PMD : Pour vérifier la qualité de la structure et du style du code.
- IntelliJ : Pour la détection des duplications de codes, pour le refactoring de codes comme vu lors des TPs, pour l'analyse de dépendance, pour les codes et tests coverages effectués tout au long de ce projet et finalement pour l'analyse des métriques.
- JProfiler : Pour l'analyse technique dynamique.

Ces outils nous auront été très utile tout au long de ce projet et spécialement lors de l'analyse de qualité du code final.

## Chapitre 2

# Compréhension du logiciel

L'un des premiers objectifs de ce projet a été de se familiariser avec un code source existant afin d'y intégrer individuellement de nouvelles fonctionnalités. Ce code source était de base bien structuré et optimisé. Sa compréhension complète et totale fut longue mais malgré tout facile, chaque classes et méthodes étant correctement commentées et structurées. Une fois que nous avons été familiarisé avec le code, nous sommes passé à l'ajout d'une fonctionnalité individuelle en essayant au mieux de ne pas dégrader les performances et la structure du code initial. Ce ne fut pas aussi simple que prévu pour certaines de ces fonctionnalités qui impliquaient de bien réfléchir afin de ne pas dégrader de trop la qualité globale du programme. De plus, nous devions également prévoir de modifier un minimum le code source existant afin de minimiser les conflits lors de la fusion de nos différentes fonctionnalités. Nous nous sommes donc imposé de base de ne modifier qu'en cas de stricte nécessité les signatures de méthode, et de tout faire pour ne pas toucher aux valeurs de retour des méthodes. Nous avons donc essayé au maximum d'ajouter nos fonctionnalités en utilisant uniquement de nouvelles méthodes et classes s'appuyant en grande partie sur ce qui existait déjà lorsque c'était utile.

Une fois d'accord sur ce principe, nous sommes passés à nos implémentations individuelles.

# Chapitre 3

## Extensions individuelles

Les fonctionnalités qui nous ont été imposées, ainsi que leur répartition au sein du groupe, sont les suivantes :

- L'ajout de Super Gommes (Yarol Timur).
- La possibilité de jouer sur une map infinie (Lebrun Alexandre).
- L'ajout de fruits dans le jeu (avec ponts et téléporteurs) (Florent Barraco).

Pour chacune de ces fonctionnalités, le rapport a été rédigé par la personne responsable de sa fonctionnalité.

Pour le reste de ce rapport, la majorité a été rédigée par Yarol Timur avec quelques interventions de Lebrun Alexandre et Barraco Florent lorsque c'était nécessaire.

### 3.1 Fonctionnalité des Super Gommes

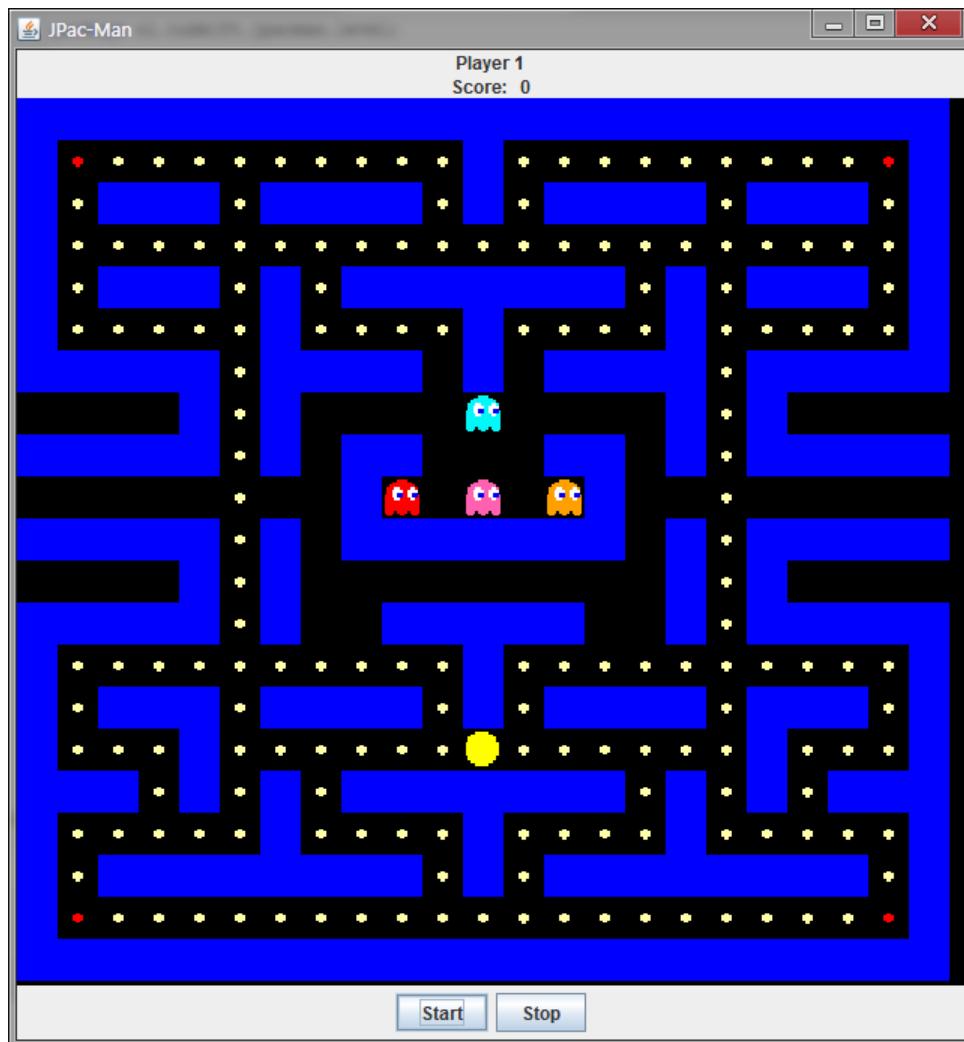
Le but de cette fonctionnalité est de permettre au jeu de changer de mode lorsque Pacman mange une super gomme. En effet, dans ce cas, le jeu passe en mode fuite et les fantômes sont effrayés. Pendant un cours instant, le joueur de Pacman peut manger aussi bien les gommes que les fantômes. Un fantôme mangé se doit de réapparaître après un certain temps dans son "antre" au centre de la map. Chaque fantôme mangé rapporte également des points au joueur.

Voici comment j'ai procédé pour l'implémentation de cette fonctionnalité :

Pour commencer, j'ai décidé de gérer la fonctionnalité des Supers Gommes dans le jeu en utilisant un maximum les méthodes existantes. Voici à présent, étape par étape, comment j'ai développé le logiciel. Tout d'abord, j'ai modifié la classe *MapParser* afin que la méthode *addSquare(...)* de cette classe gère le cas où une super gomme est lue dans le fichier *board.txt*. Ensuite, ce fichier a été modifié afin d'y placer des super

gommes (représentées par des "o") à chaque coin de la map. Un comportement initial a été attaché à ces supers gommes afin de les différencier d'une gomme normal en y attribuant un score plus grand lorsque un joueur en mange une (50 points pour une super gomme contre 10 pour une simple gomme). Cela m'a permis également de vérifier via les tests unitaires qu'une super gomme était bien différenciée d'une simple gomme grâce au score obtenu par un joueur. J'ai alors rapidement modifié les sprites de la map afin de pouvoir différencier une gomme d'une super gomme. Pour cela, j'ai changé la couleur des gommes de base jaune afin d'obtenir des supers gommes de couleur rouge. La méthode nouvellement créée *createSuperPellet(...)* permet à présent de créer une gomme de 50 points avec le sprite qui lui est correctement associé dans le fichier des ressources contenant les sprites. Chaque méthode créée à ce stade s'est appuyée sur ce qui existait déjà afin de rester en logique avec le code initial.

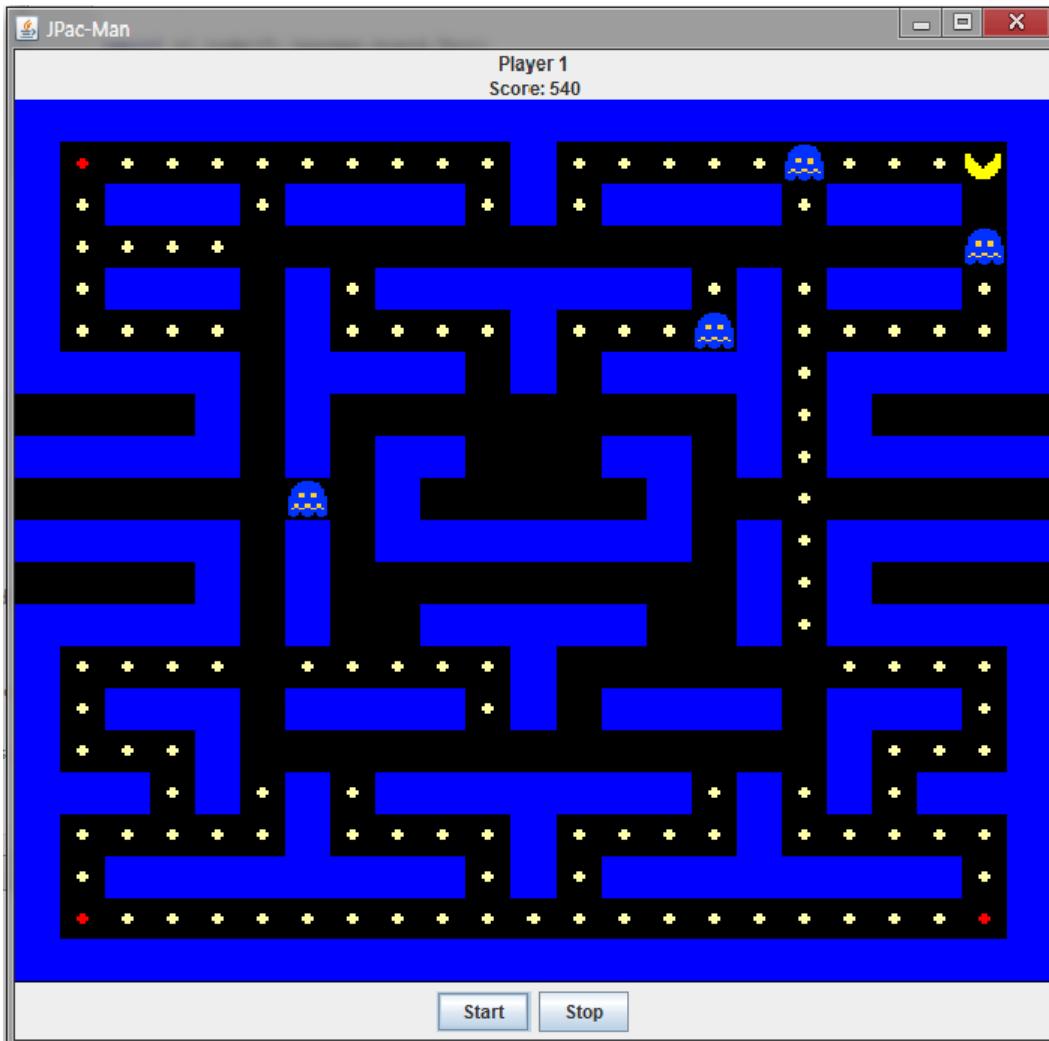
Les supers gommes rouges sont bien visibles aux coins de la map :



A ce stade, je me suis penché plus en profondeur sur le code afin de mieux réfléchir à comment implémenter le reste de ma fonctionnalité. Tout d'abord, j'ai effectué un code coverage initial afin de voir combien de % du code était utilisé pour le moment. J'ai tout d'abord été étonné de voir que seulement 84-85% du code était utilisé lors d'une partie de pacman se terminant par une défaite, et qu'également ce même pourcentage était utilisé lors d'une victoire. J'ai alors pu remarquer que certaines classes existantes n'étaient jamais utilisée (tel que *CollisionInteractionMap* ou encore *DefaultPlayerInteractionMap* qui sont des implémentations possibles et plus optimisées pour gérer la collision entre les objets que celle utilisée par le jeu dans la classe *PlayerCollisions*). J'ai également pu remarquer que certaines méthodes pouvaient ne pas être appelée en fonction de la partie jouée (si un joueur gagne, la méthode s'occupant de gérer la défaite d'un joueur n'est pas appelée, et inversement). Tout ceci mit ensemble fait que seul 85% du code est utilisé en moyenne par partie. Un test coverage a été ensuite effectué. J'ai pu remarquer que de la même manière, le *LauncherSmokeTest* utilisait à peu près 80% du code en moyenne. Après cette partie d'analyse du code, j'ai pu avoir les idées plus claires sur quelles classes modifier et sur ce que je devrais ajouter afin d'implémenter au mieux ma fonctionnalité.

Je me suis donc lancé sur la création d'un mode Hunter pour le joueur. Ce mode s'active automatiquement à chaque fois qu'un joueur mange une super gomme. Dans ce mode, les fantômes sont effrayés et le joueur peut alors les manger pendant un court laps de temps.

Initialement, j'ai voulu créer une classe *EatableGhost* afin de gérer le cas où un joueur mange un fantôme mangeable ou non. Cette classe héritait de la classe *Ghost* comme le font les classes des autres fantômes (Blinky, Clyde, ...). Cette classe *Eatable-Ghost* devait me permettre de différencier le comportement d'un fantôme normal par rapport à un fantôme effrayé. Une de ces différences rapidement implémentée a été le changement de sprite du fantôme. Un fantôme effrayé possède le sprite bleu ci-dessous :



De plus, la vitesse des fantômes effrayés devait être de 50% pour tous les fantômes. Ceci a facilement été possible grâce aux variables et méthodes existantes initialement dans la classe *Ghost* et celles héritant de *Ghost*. C'est la méthode *getInterval()* spécifique au type de fantôme qui permet d'obtenir sa vitesse, c'est donc cette méthode qui a été réutilisée dans le cas d'un fantôme effrayé en adaptant les variables *INTERVAL\_VARIATION* et *MOVE\_INTERVAL*. La *GhostFactory* a été également modifiée pour créer un *EatableGhost* dans le cas où le jeu change de mode. Comme demandé pour ce mode, les fantômes se mettent à bouger aléatoirement lorsqu'ils sont effrayés. Ceci a également été possible en utilisant une nouvelle méthode de déplacement spécifique à ce mode.

Ces différents changements apportés ont finalement été revus par la suite car ils ne m'ont pas permis de gérer correctement le jeu. J'expliquerai ceci plus en détail par la suite. Cette partie m'a néanmoins permis de déjà gérer un changement de mode correctement et d'y associer les sprites requis. Un code et test coverage ont à nouveau été

effectués à ce stade, car pas mal de changement ont déjà été apporté. Aucune différence par rapport à avant n'est à signaler.

Par la suite, je me suis attaché à gérer le mode Hunter en y attachant un Timer. Ce Timer se lance dès qu'une super gomme est mangée et permet d'appeler après 7 ou 5 secondes (en fonction du nombre de super gommes restantes) la méthode s'occupant d'arrêter ce mode Hunter. De plus, c'est à ce stade que je me suis rendu compte qu'il était difficile de gérer la fin du mode Hunter dans le sens où ma création d'*EatableGhost* semblait être une bonne idée mais qu'elle ne me permettait pas de retrouver quel fantôme devait reprendre la place de tel *EatableGhost*. J'ai donc remonté la plupart des méthodes de la classe *EatableGhost* dans la classe *Ghost* afin de pouvoir gérer l'instance des fantômes plus facilement (comme ces derniers héritent de *Ghost*).

Une première suppression de code inutile a été effectuée à ce stade grâce au code coverage avec les méthodes devenues inutiles que j'avais déjà pu créer ainsi qu'avec la suppression de la classe *EatableGhost* dont le comportement a été remonté dans la classe *Ghost*. Une autre manière de faire aurait été de faire hériter les classes de *Blinky*, *Pinky*, *Inky* et *Clyde* de *EatableGhost*, elle-même héritant de la classe *Ghost*.

Maintenant les idées bien fixées, la suite de l'implémentation de la fonctionnalité s'est faite plus aisément. Tout d'abord, un Timer a été ajouté de la même manière que celui qui gère le mode Hunter mais cette fois pour le respawn d'un fantôme. En effet, lorsqu'un fantôme est mangé, il doit réapparaître après 7 secondes au centre de la map dans son antre. Pour cela, une méthode a été créée afin de calculer le centre de la map dans la classe *Board*. Pour gérer le fait qu'un fantôme doit réapparaître, un compteur de fantôme en vie est mis en place dans le jeu. Si ce compteur tombe en dessous du nombre de fantômes initiaux créés lors du parage de la map au lancement du jeu, un observer est appelé afin de lancer le Timer pour recréer un fantôme. Pour cela, une instance du fantôme mangé par le joueur est donné en paramètre à ce Timer afin de savoir quel type de fantôme doit réapparaître. Ensuite, la classe *PlayerCollisions* s'occupant de gérer les collisions possibles a été modifiée afin de gérer les points gagnés par le joueur en fonction du nombre de fantôme mangé lors d'un mode Hunter. Un nouveau nettoyage du code a été effectué et un code coverage m'a permis de voir que le code était toujours utilisé en moyenne à 80%. Des premières optimisations de méthodes ont également été effectuées.

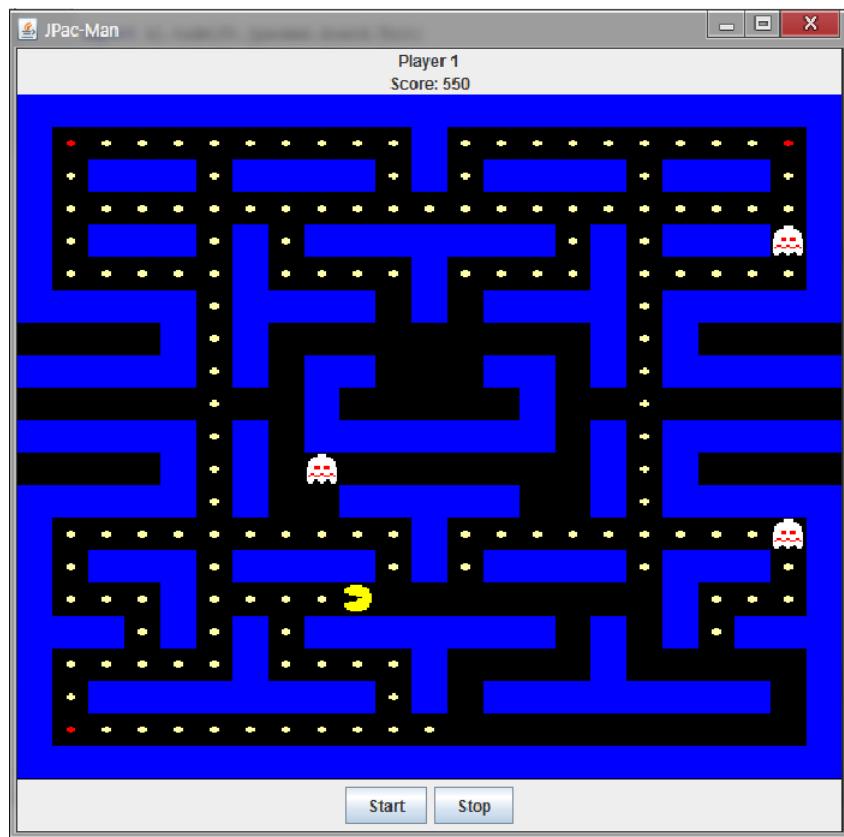
Le commit suivant s'est simplement occupé de gérer le déplacement des fantômes effrayés. À la base, les fantômes se mettaient à bouger aléatoirement dans les directions possibles, ce qui donnait lieu à des comportements étranges lorsque les fantômes se trouvaient dans un couloir et ne se déplaçaient qu'entre 2 positions pendant un moment. Pour cela, une nouvelle méthode a été mise en place afin d'obliger les fantômes à avancer tout droit dans un couloir et de choisir une nouvelle direction aléatoire lors d'un embranchement hormis celle d'où ils viennent. Ainsi, les fantômes ne peuvent plus

retourner sur leur pas, ce qui règle le soucis initial.

Ensuite, après avoir testé le jeu et vérifié son bon comportement, je me suis mis à commenter toutes mes méthodes et variables créées (ce que j'aurai du faire initialement) afin de permettre aux lecteurs du code de mieux comprendre à quoi servent mes méthodes et variables.

Tout au long du développement de cette fonctionnalité, j'ai cherché à gérer un effet qui n'était pas demandé, à savoir le fait qu'un fantôme puisse "clignoter" à la fin du mode Hunter, afin de prévenir le joueur que les fantômes vont retrouver leur état normaux très prochainement. Néanmoins, j'ai énormément eu de mal à gérer cela. Pour le commit final pour l'évaluation individuelle, cette fonctionnalité supplémentaire a été ajoutée correctement.

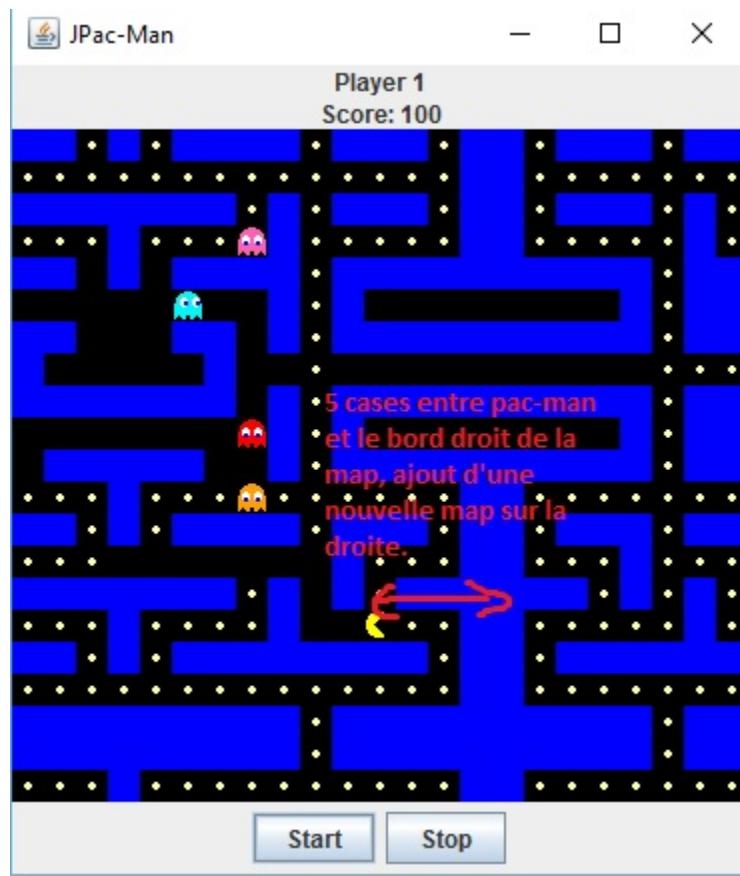
Voici ce que donne un fantôme qui va retrouver un comportement normal :



## 3.2 Fonctionnalité de la map infinie

1<sup>ière</sup> étape :

Étant donné que la map allait devenir plus grande que ce que le jeu affiche à l'écran, la première étape a été de centrer l'affichage du jeu sur la position courante de pac-man. Pour réaliser cela la méthode *render(...)* de la classe *BoardPanel* a été modifiée pour prendre en compte la position de pac-man lors du dessin du plateau de jeu. Ensuite lorsque pac-man se rapproche à moins de 5 cases d'un des bords du plateau, ce dernier s'agrandit de la taille d'une map dans la direction du bord vers lequel se dirige pac-man comme on peut le voir sur l'image ci-dessous.



2<sup>ième</sup> étape :

Cette partie concerne l'agrandissement du board. Tout d'abord j'ai ajouté dans la classe *Square* des variables correspondants aux coordonnées des squares sur le plateau de jeu, ainsi que des tests unitaires pour vérifier si ces coordonnées étaient bien gérées correctement. Ensuite dans la classe *Board* j'ai rajouté une méthode *boardCopy(Square[][] grid)* qui a pour but de recopier le plateau de jeu actuel dans un autre plateau plus

grand, ainsi qu'une méthode *setPositions(Square[][] grid)* qui met à jour les coordonnées *x* et *y* des squares du plateau.

3<sup>ième</sup> étape :

Cette étape consistait en la création du nouveau plateau à ajouter à l'actuel. Pour réaliser cette étape j'ai décidé de me servir de la méthode *makeLevel(...)* présente dans la classe *Launcher*. Pour ce faire j'ai ajouté une variable static me permettant de récupérer l'instance du Launcher ainsi qu'une seconde variable pour pouvoir changer le fichier texte qui servira de base pour la création du plateau. Une fois le nouveau level créé, j'ai ajouté des méthodes afin de recopier les squares de ce nouveau level aux bons endroits dans le plateau de jeu agrandi. En dernier lieu, afin que pac-man et les fantômes puissent se déplacer sur les nouveaux squares présent dans le plateau de jeu, j'ai repris comme base la méthode *setLink()* de la classe *LevelFactory* que j'ai modifié afin qu'elle s'occupe de faire son travail sur les nouveaux squares présent sur le plateau de jeu.

4<sup>ième</sup> étape :

Pour implémenter la fonctionnalité concernant l'augmentation de vitesse des fantômes j'ai ajouté une variable *speed* de type float et initialisée cette dernière à 1.0 dans la classe *Ghost*. Cette variable va servir à diviser l'intervalle de temps entre deux mouvements des fantômes. Cette variable s'incrémentera à chaque agrandissement du plateau de jeu. Le fait d'initialiser la variable à 1.0 permettait en vue de la future mise en commun du code de ne pas modifier le comportement des fantômes si on jouait sans map infinie car la méthode *getInterval()* des différents fantômes a été modifiée.

5<sup>ième</sup> étape :

Afin d'implémenter l'ajout de fantôme sur le plateau de jeu, j'ai mit un caractère 'G' dans les fichiers textes qui serve à construire le nouveau level, ce qui permet au fantôme d'être automatique créer par les mécanisme déjà présent dans l'application. Ensuite il a fallu récupérer la liste des NPCs du nouveau level afin d'ajouter le fantôme de ce level dans la liste des NPCs du level courant afin qu'il soit gérer avec et comme les 4 fantômes de bases. Ceci se fait dans la méthode *addGhost(Level l)* de la classe *Level*.

6<sup>ième</sup> étape :

Cette dernière étape a surtout consisté en l'ajout de tests unitaires afin de tester les fonctionnalités ajoutées au programme et à l'optimisation et la correction de celles-ci si nécessaire.

Au niveau des problèmes rencontrés, il y a le fait que l'affichage de plateau lorsque

celui-ci s'agrandit manque de fluidité (ce qui a été corrigé pour la partie commune du projet), ainsi que le fait les fantômes ajouté suite à l'agrandissement du plateau ne s'arrêtaien pas lorsqu'on clique sur le bouton Stop. Ceci a aussi été corrigé lors de la mise en commun. Et enfin il y a une chute des performances lorsque l'on commence à approcher des 10 fantômes présents sur le plateau.

La version finale de cette fonctionnalité sur trouve sur GitHub au commit du 17 avril 2016 nommé « Last commit : ok pour évaluation ».

### 3.3 Cases et fruits spéciaux

Pour rappel il fallait implémenter 3 type de cases spéciales : Un pont, un téléporteur et un piège ainsi que six types de fruits ayant des effets spéciaux quand ils sont mangés par le joueur.

Le piège à pour effet de paralyser un fantôme ou pacman si ceux-ci le touchent. Il est représenté sur la carte par un grand cercle au contour de couleur jaune. Pour gérer le fait qu'un personnage soit immobile ou non, les classes *Player* et *Ghost* contiennent une variable d'instance appelée *mobile* qui vaut vrai si un personnage peut bouger ou faux quand il est immobile. La méthode *move()* de la classe *Level* est la fonction du programme qui bouge un personnage sur l'écran, celle-ci a été modifiée pour ne pas bouger Pacman ou un Fantôme (qui implémentent l'interface *DirectionCharacter*, mettant en commun les comportements entre pacman et les fantômes) si celui-ci est immobile. Le piège est géré par la classe *Hole* qui hérite de *Unit*. Son effet s'active de la manière suivante : La variable d'instance *mobile* est fixée à faux et un timer est démarré pour activer une action dans une durée égale à l'effet du piège. Cette action va fixer de nouveau la variable d'instance *move* à vrai pour permettre de nouveau à l'unité de se déplacer. La gestion de cet effet est la même pour le fruit "poisson" vu que son effet est identique à celui du piège. Pour pouvoir ajouter des piège sur le plan un nouveau caractère a été introduit pour représenter le piège, la lettre H.

De nombreux effets temporaires de ces cases et fruits spéciaux sont gérés de la même manière : une variable d'instance pour indiquer si le personnage est dans cet état ou non, quand le joueur change d'état, une action sera lancé dans une durée égale à la durée de l'effet pour remettre la variable d'instance à son état initial et les différentes fonctions du jeu vérifient l'état des personnages pour voir s'il faut appliquer l'effet ou pas.

Pour implémenter la fonctionnalité du pont il faut gérer les règles suivantes : Deux unités ne rentrent pas en collisions si l'une est sur un pont et l'autre ne l'est pas, si un objet se trouve sous un pont il ne doit pas être dessiné vu que le jeux est affiché en

vue aérienne. un pont peut avoir une orientation verticale et horizontale et contenir un objet en dessous, il ne peut pas être possible de se jeter d'un pont et de monter sur un pont autrement qu'en passant par les cases reliés par ce pont.

les première et deuxièmes règles sont gérées par l'ajout d'une variable d'instance booléenne appelée *onBridge* qui indique si un objet est sur un pont ou non, Avant de traiter une collision entre deux objets sur une même case, la fonction *collide(...)* de la classe *PlayerCollision* vérifie que les deux objets ont des valeurs égales pour la variable d'instance *onBridge* si c'est le cas le, alors traitement de la collision se poursuivra.

La méthode *render(...)* de la classe *Boardpanel* s'occupe de dessiner les objets présents sur chaque case, si l'objet est détecté comme étant sous un pont cette méthode ne dessinera pas l'objet.

Pour pouvoir ajouter des ponts sur plan en respectant la troisième règle, la spécification du fichier texte décrivant le plateau a été étendue, un pont est représenté par le caractère B et le fichier de description du plan est désormais séparé en trois parties : la première qui décrit le plan, la seconde décrit les endroits pointés par les téléporteurs et la dernière spécifie l'orientation et les objets sous les ponts.

Chaque entrée de cette section se compose d'une ligne avec pour premier caractère l'orientation du pont : H si le pont est horizontal et V si le pont est vertical, suivi d'un espace, puis un caractère indiquant ce qui se trouve en dessous du pont : N pour rien du tout, P pour une gomme (ou pellet) et F si un fruit peut apparaître sous ce pont.

La gestion de la position d'un joueur en fonction d'un pont dépend de la direction du joueur et celle du pont, Dans certaines circonstances un pont peut empêcher le joueur de bouger par exemple si le joueur est sous le pont et qu'il souhaite aller dans la même direction que le pont ou s'il est sur le pont et qu'il souhaite aller dans la direction opposée à celle du pont ces règles sont gérées par une méthode de la classe *Bridge* appelée *blockedByBridge*, cette méthode est appelée par la méthode *move(...)* de la classe *Level* pour déterminer si le joueur est bloqué par le pont, si c'est le cas la méthode *move(...)* ne déplacera pas le joueur.

Pour monter sur un pont, un joueur ou fantôme doit entrer en collision avec le pont tout en étant dans la même direction que celui-ci, par exemple si Pac-Man descend vers le bas et qu'un pont vertical se trouve sur la prochaine case il sera considéré comme étant sur un pont, à chaque fois qu'un personnage bouge sa variable d'instance *onBridge* est automatiquement fixé à fausse, pour gérer la possibilité de quitter un pont, mais comme un pont est toujours en premier dans la liste des occupants, la collision se fera toujours avec le pont en premier ce qui évite à un joueur de toucher des ennemis sous ce pont alors qu'il est censé être sur celui-ci.

La case du téléporteur permet à un joueur de se téléporter sur une case prédéfinie dans le fichier txt, le téléporteur est représenté par le caractère T et la section suivante indique les cases où conduisent chaque téléporteur dans l'ordre de haut en bas et de gauche à droite, chaque entrée de cette section consiste en deux nombres entiers séparés par un seul espace indiquant les coordonnées de la case où conduit le téléporteur. Le téléporteur est représenté par la classe *Teleport* et contient en variable d'instance un objet de type *Square* appelée référence, le carré où conduit ce téléporteur, quand Pac-Man entre en collision avec un téléporteur, il est déplacé sur la case dans la variable d'instance référence étant donné que d'autres objets pourraient se trouver sur cette case. Une collision est effectuée avec les objets de la case de destination sauf les téléporteurs afin d'éviter une boucle infinie dans le cas où un téléporteur amène à une case où se trouve un autre téléporteur qui amène à la case de départ.

Toutes les variétés de fruits spéciaux héritent d'une classe appelée *Fruit*. À part le sprite de ce fruit, la classe contient aussi comme variable d'instance, la durée de vie du fruit, c'est à dire le temps que ce fruit reste sur la carte avant d'être enlevé ainsi que la durée de l'effet de ce fruit, la durée des effets est gérée de la même manière qu'avec le trou, avec un changement de variable d'instance suivi d'un timer planifié qui remet la variable d'instance à son état initial. La seule chose qui change entre chaque fruit est le contenu de la méthode *fruitEffect()* et parfois l'ajout de la liste des NPC du niveau pour pouvoir appliquer des opérations sur les fantômes.

Pour permettre l'apparition des fruits sur le plateau, le caractère F a été introduit pour indiquer qu'un fruit peut apparaître à cet endroit. Si il n'y a aucune case marquée par un F aucun fruit n'apparaîtra dans le jeu.

La classe *FruitFactory* récupère les cases où les fruits peuvent apparaître et à pour but de fournir de manière contrôlée des références de fruits. Un fruit apparaît quand un joueur totalise un score de 50 et 1500 points. Cette condition est vérifiée dans la fonction *updateObservers* de la classe *Level* et la classe *LevelObserver* a été étendue pour inclure une fonction *fruitEvent* pour déclencher l'apparition d'un fruit aléatoirement choisis sur une case aléatoirement choisie.

Le poivron et la patate sont respectivement modélisés par la classe *BellPepper* et *Potato* et ont pour effets respectifs de l'accélérer temporairement Pac-Man et les fantômes une variable d'instance booléenne appelée *acceleration* indique si un personnage doit se déplacer de manière accélérée ou non. Ces fruits auront pour effet de fixer temporairement la valeur de cette variable d'instance à vrai.

Les classes *Player* et celles des différents fantômes ont deux variables d'instances toutes les deux des variables constantes indiquant les intervalles de temps entre chaque mouvement pour une vitesse normale ou accélérée (*MOVE\_INTERVAL* et *ACCELERATED\_MOVE\_INTERVAL* où cette dernière est censée être de valeur plus petite)

en fonction de la valeur de la variable d’instance *acceleration* l’une des deux valeur sera utilisée pour calculer l’intervalle de temps entre chaque mouvement de pacman et les fantômes.

Dans le jeu d’origine, Pac-Man ne se déplace pas de manière automatique et avance d’une seule case à chaque pression d’une touche directionnelle. Vu que le code contient tout pour faire déplacer un objet de manière automatique grâce à la classe NPC, la classe *Player* a été modifiée pour hériter de cette classe afin de pouvoir bénéficier de ces fonctionnalités. La fonction faisant avancer Pac-Man a été modifiée juste pour ne plus que changer la direction de celui-ci. Cette modification était nécessaire pour donner un sens au poivron.

Le haricot rouge est modélisé par la classe *KidneyBean* a pour effet de fixer temporairement à vrai la variable d’instance *shooting* de l’objet *Player*. la fonction *updateObservers()* de la classe *Level* vérifie avec la fonction *isAnyPlayerShooting()* si Pac-Man peut tirer des balles. si oui la fonction *FruitEvent()* du *LevelObserver* va ajouter des objet de types *Bullet* à l’emplacement du joueur toutes les secondes, les objets de types *Bullet* sont des *NPC* se déplacent tout le temps dans la même direction, si l’un d’entre eux entre en collision avec un fantôme celui-ci explose et est retiré du jeux, si une balle se cogne contre un mur, l’objet disparaît également. La fonction *updateObservers()* vérifie également avec la *deadNPCs()* si des fantômes ont explosés où des balles doivent disparaître et appelle la méthode *NPCCleanEvent()* du *LevelObserver* pour les faire disparaître du plateau de jeu.

La grenade est modélisée par a classe *Pomgranate*, cette classe contient une liste des *NPC* actifs au moment de la création du niveau c’est à dire les fantômes. Quand le joueur mange ce fruit, tout les fantôme se trouvant dans un rayons de quatre cases explosent. Pour déterminer si un fantôme doit exploser, le plus court chemin en autorisant la capacité à passer entre les murs est calculé pour chaque fantôme entre chaque fantôme si la longueur du chemin est inférieur ou égale à quatre, le fantôme explose.

La tomate est modélisée par la classe *Tomato* et a pour effet de fixer temporairement à vrai la variable d’instance invincible de l’objet *Player* à vrai. Quand une collision entre un objet *Ghost* et un objet *Player* se produit, cette collision est ignorée si l’objet *Player* est invincible.

# Chapitre 4

## Intégration et analyse de qualité des extensions

Dans cette partie nous expliquerons en détail comment nous avons procédé afin de fusionner les différentes fonctionnalités grâce à Github mais également les différentes améliorations et corrections apportées au code. Ensuite, nous effectuerons une analyse de la qualité de notre code final par rapport à celui initial en expliquant en détail les résultats obtenus.

Malgré le fait que cette partie est une partie commune aux 3 membres du groupe, nous préciserons bien qui s'est occupé de quelle partie à chaque fois.

### 4.1 Fusion des fonctionnalités supers gommes et map infinie

#### 4.1.1 Fusion des codes et gestion des conflits

Cette fusion des codes et gestion des conflits a été effectuée par Lebrun Alexandre avec la participation de Yarol Timur.

Tout d'abord, nous nous sommes occupés de la fusion des codes de map infinie et des supers gommes. Nous avons résolu les différents conflits signalés par Github. Cette fusion ne nous a posé aucun problème majeur bien que le code ne semblait plus fonctionner correctement. Pour cela, nous avons analysé plus en détail les erreurs obtenues et corrigés ces dernières assez facilement. De plus, une amélioration du Launcher permet au joueur de choisir si il désire jouer au mode normal ou bien à celui de la map infinie. En effet, étant donné que les règles du jeu sont différentes dans un mode ou dans l'autre, séparer ces deux comportement de jeu nous a semblé logique à ce stade.

Néanmoins, la fusion des codes a mis en évidence une perte énorme des perfor-

mances lorsque nous jouions sur la map infinie. En effet, énormément de lecture de la map étaient effectuée par le code des supers gommes, ce qui est gérable dans le cas d'une map basique, mais beaucoup plus difficile à maintenir lorsque la map s'agrandit continuellement. De plus, énormément de Timers semblaient être créé (plus que prévu).

#### 4.1.2 Amélioration des performances

Ces améliorations ont été effectuées par Yarol Timur.

Une lecture du code fusionné a été effectuée afin de voir à quel endroit le code pourrait être optimisé. Pour cela, chaque endroit du code ou une lecture de la map complète était effectuée et analysé plus en profondeur afin de voir s'il était possible de diminuer cette lecture au maximum. Pour cela, une réduction de la lecture complète a été faite en faisant en sorte que seul ce qui apparaissait à l'écran ne soit analysé. Cela a permis d'améliorer grandement les performances, bien qu'une latence se faisait toujours ressentir. Une nouvelle analyse a donc été effectuée et a mis en évidence l'utilisation de méthodes obsolètes pour le cas de la map infinie tel que la lecture du nombre de gommes restantes. En effet, dans le mode infini, le joueur ne peut plus gagner en mangeant toutes les gommes, il faut donc ne pas utiliser ce comportement dans le cas du mode infini. Une variable a donc été mise en place afin d'empêcher l'utilisation de ces méthodes inutiles en fonction de si l'on se trouve dans un mode ou dans l'autre. Au terme de cette analyse, un maximum de lectures inutiles ont donc été enlevées. Les latences restantes sur le jeu semblent provenir du nombre de fantômes à gérer qui croît continuellement.

Une autre amélioration apportée a été celui des fichiers utilisés pour l'extension de map qui ont été modifiés afin d'augmenter les possibilités de fuite du joueur.

Finalement, initialement une lecture complète de la map était effectuée pour le comptage du nombre de gommes restantes, et une autre lecture complète pour le nombre de supers gommes restantes. Cette seconde lecture est au final redondante, et une solution plus simple et efficace est de faire comme pour les fantômes et d'utiliser une variable maintenant à chaque instant le nombre de supers gommes présente sur la map.

Après toutes ces améliorations, le code semble être plus fonctionnel, bien que certains bugs restants aient été finalement corrigés par Lebrun Alexandre.

### 4.2 Fusion des 3 fonctionnalités

Une fois la fusion de la map infinie et des supers gommes effectuées, nous nous sommes penchés sur la fusion avec le mode de jeu incluant les fruits et les téléporteurs

ainsi que les ponts.

#### 4.2.1 Fusion des codes et gestion des conflits

Cette fusion a été entièrement effectuée par Lebrun Alexandre. Il s'est également occupé de gérer les conflits entre les différents codes.

La fusion des codes s'est ici passée plus difficilement. En effet, le nombre de conflit rencontré entre le mode avec les supers gommes et celui avec les fruits a été très important. Ces deux modes se reposent sur la plupart des mêmes méthodes, ce qui a impliqué un nombre de conflit assez important. Par exemple la méthode s'occupant de gérer la création d'une super gomme a été utilisée pour la création d'un fruit, ou encore d'un téléporteur ou d'un pont, il a donc fallu gérer ce conflit en acceptant les deux parties du code dans la version fusionnée. De plus, quelques signatures de méthodes ont été modifiées, ce qui a eu un impact sur le projet en entier, il a donc fallu également gérer ce conflit en divisant ces méthodes aux signatures modifiées en 2 méthodes, une avec la signature initiale, utilisée pour le jeu de base, et une seconde avec la signature modifiée qui s'occupera de gérer le mode de jeu avec les fruits. Gérer tout les conflits a donc été très long, mais assez facile.

#### 4.2.2 Amélioration des performances et ajout de mini-fonctionnalité

Au niveau de l'ajout de fonctionnalité, il y a le fait qu'initialement, lors de l'agrandissement du board, nous choisissons aléatoirement entre 4 extensions de board possible, donc l'une contenant un 'G' signalant qu'il fallait ajouter un Ghost à la partie. Cependant, on s'est vite rendu compte que cette façon de faire menait à des soucis. En effet, lorsque nous avons voulu limiter le nombre de fantômes par partie à 10 maximum, nous avons remarqué que dès que le fichier contenant un fantôme était choisi, il y avait une incrémentation étrange du nombre de fantômes sur le jeu du au nombre de passage dans la méthode s'occupant de cela. Nous avons essayé à la base de régler ce soucis en faisant en sorte qu'un seul ajout soit effectué mais sans succès. De plus, un autre problème qu'apportait cette façon de faire était que le fantôme choisi était toujours *Blinky*. Ceci est du au fait que la réutilisation du comportement de *LevelFactory* passait toujours par le premier case du switch qui crée un *Blinky*. Nous avons voulu corriger cela en permettant l'apparition d'un fantôme aléatoire parmi les 4 existants.

L'ajout des fonctionnalités ci-dessous ont été effectuées par Yarol Timur.

J'ai décidé, afin de pallier aux problèmes cités ci-dessus, de gérer l'apparition des fantômes en mode infini de manière totalement aléatoire. Pour cela, j'ai modifié la méthode *addGhost()* initial en une méthode *addGhostTask()* qui s'occupera de faire apparaître toutes les X secondes (X étant une variable dépendant du nombre de fantômes déjà présent sur la map) un nouveau fantôme au sommet de la map comme demandé pour

ce mode de jeu. De plus, le fantôme sera choisi de manière aléatoire parmi les 4 possibles.

Ce comportement d'apparition de fantômes aléatoire a été utilisé pour faire apparaître de la même manière les fruits aléatoirement sur le board via une nouvelle méthode `addFruitTask()`.

Bien que ces méthodes utilisent des constantes difficile à comprendre (calculées grâce à la position du joueur à chaque déplacement), elles ne sont pas adaptées en cas de modification de la taille du board. Cette façon de faire n'est donc pas la meilleure et ne respecte pas les principes vus au cours, mais un refactoring a été effectué plus tard par mes soins afin de corriger ce problème.

Une troisième fonctionnalité ajoutée a été le fait que les fantômes gagnent en vitesse toutes les X secondes (X dépendant du nombres de fantômes déjà présent). Cette fonctionnalité est effectuée grâce à la méthode `speedUpTask()`.

#### 4.2.3 Correction de bugs et refactoring de la structure du code

Tout d'abord, il est utile de préciser qu'après la fusion de nos codes individuelles, quelques bugs et illogismes au niveau de la structure du code on été découverte. Par exemple, dans la version avec les fruits, Pacman est censé se déplacer automatiquement avec une vitesse de 100%, or, dans notre version fusionnée, ce déplacement automatique faisait planter le jeu, on a donc du le supprimer temporairement. Un exemple d'illogisme dans la structure du code est le fait que dans la version avec les fruits, la classe *Player* étendait une nouvelle classe qui n'était plus *Unit* mais *NPC*. La logique derrière était que certains comportements comme le déplacement automatique des *NPCs* était utile pour le *Player*. Nous avons donc décidé de nous concerter tout les 3 afin de discuter du refactoring de la structure du code et de comment gérer les bugs et conflits restant.

Les refactorings suivants ont été effectués par Florent Barraco :

Le refactoring de la structure n'a pas été très difficile. Nous avons décidé qu'au lieu que *Player* extend de *NPC*, ces deux classes étendraient une nouvelle classe nommée *MovableCharacter*. Ainsi le comportement commun a ces classes se trouveront dans une nouvelle classe et leur comportement individuels restera unique à ces classes.

Les fantômes et pacman partagent les caractéristiques de se déplacer à intervalle régulier et de pouvoir se déplacer à une vitesse accélérée, ces fonctionnalités ont été déplacées de *NPC* à *MovableCharacter*, les objets s'occupant du déplaçant les *NPC* ont été changées pour supporter les objets *MovableCharacter* à la place.

Pour optimiser les fonctions qui testent différents comportements sur les *NPC* comme le nettoyage de balles, une liste ne contenant que les balles, une ne contenant que les fantômes et une ne contenant que les joueurs ont été ajoutées.

Après avoir fusionné le code nous avons également remarqués que presque tout

les tests échouaient, par conséquent il fallait corriger les différents comportements du programmes pour pouvoir les rendre de nouveau conformes aux tests.

#### 4.2.4 Nettoyage du code fusionné

Cette section a été effectuée par Yarol Timur.

Un refactoring du code a été entièrement effectué à ce stade afin de rendre plus propre le code. Par exemple, les variables créées depuis le début de ce projet ont souvent été liée à la classe *Level*, ce qui a rendu cette classe particulièrement complexe. Un premier refactoring a donc été de déplacer les variables de cette classe dans les classes correspondantes. Par exemple, la variable contenant le nombre de fantômes en vie peut être stockée dans la classe *Ghost* à la place. De la même manière, la variable contenant le nombre de supers gommes sur la map peut également être stockée ailleurs. Bien que ces valeurs soient liées au level en cours, il est plus facile de les garder dans la classe qui leur correspond afin d'alléger le code de la classe *Level*.

Un second refactoring a été d'optimiser le code en supprimant certains bad smell. Un exemple déjà cité a été celui de l'ajout aléatoire de fantômes et de fruits se basant sur la taille de la map fixée. En utilisant des variables existantes dans d'autres classes, j'ai pu corriger ce bad smell.

### 4.3 Analyse de qualité du code

Cette section se concentrera sur une analyse complète de la qualité du code, divisée en 2 parties. Une première partie s'occupera d'analyser le code initial et de le comparer à notre analyse du code pré-final. Ensuite, après avoir fait cette analyse au complet, nous examinerons en détail chaque point noir détecté dans notre projet et feront notre possible pour les corriger. Après ce grand refactoring du projet, nous effectuerons une nouvelle analyse de la qualité du code final, que nous détaillerons dans la seconde partie de cette section.

Une grande partie de l'analyse de qualité du code a été effectuée par Lebrun Alexandre. La comparaison entre le code initial et le code pré-final a été effectuée par Lebrun Alexandre et Yarol Timur. Le refactoring du code pour l'amélioration de la qualité a été en grande partie effectuée par Yarol Timur avec l'aide de Lebrun Alexandre. Finalement, l'analyse du code final a été effectuée par le groupe en entier.

### 4.3.1 Comparaison du code initial et pré-final

#### Code coverage

Tout d'abord, nous avons effectué un code coverage du code initial du projet.

Voici les résultats :

Coverage Launcher				
	Element	Class, %	Method, %	Line, %
nl.tudelft.jpacman	81% (48/59)	82% (213/258)	81% (785/965)	

Comme nous pouvons le constater, initialement, seul 81-82% du code était exécuté dans le cas d'une partie perdue. Nous avons vérifié (même si non présenté ici) et nous avons les mêmes résultats pour une partie gagnée. Analysons plus en détail ce code coverage initial :

Coverage Launcher				
	Element	Class, %	Method, %	Line, %
board	87% (7/8)	90% (37/41)	85% (90/105)	
game	100% (3/3)	66% (12/18)	75% (39/52)	
level	50% (8/16)	71% (45/63)	70% (228/322)	
npc	90% (9/10)	97% (33/34)	87% (143/164)	
sprite	100% (5/5)	83% (30/36)	90% (99/110)	
ui	100% (11/11)	87% (34/39)	90% (140/154)	
Launcher	100% (5/5)	87% (21/24)	86% (45/52)	
PacmanConfigurationException	0% (0/1)	0% (0/3)	0% (0/6)	

On voit très clairement que le package *Level* est celui qui nous fait perdre en coverage de code. Nous avons donc décidé d'analyser ce package plus en détail.

Coverage Launcher				
	Element	Class, %	Method, %	Line, %
CollisionInteractionMap	0% (0/2)	0% (0/9)	0% (0/56)	
DefaultPlayerInteractionMap	0% (0/3)	0% (0/4)	0% (0/15)	
Level	66% (2/3)	94% (17/18)	91% (104/114)	
LevelFactory	33% (1/3)	50% (4/8)	71% (15/21)	
MapParser	100% (1/1)	100% (8/8)	92% (61/66)	
Pellet	100% (1/1)	100% (3/3)	100% (6/6)	
Player	100% (1/1)	100% (6/6)	95% (19/20)	
PlayerCollisions	100% (1/1)	100% (5/5)	94% (18/19)	
PlayerFactory	100% (1/1)	100% (2/2)	100% (5/5)	

Deux classes sont inutilisées ici. Ceci a déjà été expliqué lors de la partie individuelle sur les supers gommes. En effet, les classes *CollisionInteractionMap* et *DefaultPlayerInteractionMap* sont des implémentations de différentes manières de gérer les collisions

mais qui ne sont pas utilisées en pratique par le jeu. Elles ont été créés en vue d'une amélioration de la gestion des collisions. Nous avons gardé ces classes dans le cadre de notre projet dans le cas où nous finirions pas les utiliser. Ca n'a finalement pas été le cas.

Un autre cas de code inutilisé est celui de *LevelFactory* dans lequel nous avons le code censé gérer un fantôme aléatoire sans comportement particulier. Cependant, ce type de fantôme n'apparaît pas dans le jeu de base. Ce code n'est donc jamais utilisé.

Au final, après une analyse complète du code de base, on se rend rapidement compte que jamais plus de 82-83% du code ne sera utilisé.

Passons à présent à l'analyse du code coverage de notre code pré-final :

Coverage Launcher				
	Element	Class, %	Method, %	Line, %
70% classes, 69% lines covered in package 'nl.tudelft.pacman'				
board	70% (7/10)	73% (47/64)		58% (121/206)
fruit	46% (7/15)	41% (15/36)		34% (37/107)
game	60% (3/5)	51% (14/27)		52% (47/89)
level	64% (18/28)	73% (106/145)		70% (592/837)
npc	75% (9/12)	75% (44/58)		77% (215/276)
sprite	100% (5/5)	71% (38/53)		84% (111/131)
ui	100% (11/11)	85% (34/40)		76% (145/189)
Launcher	100% (5/5)	89% (25/28)		85% (59/69)
PacmanConfigurationException	0% (0/1)	0% (0/3)		0% (0/6)

Comme nous pouvons le constater ici, seul 70-71% du code a été couvert lors d'une partie normale avec les fruits et les supers gommes. Analysons plus en détail le package *Level* :

Coverage Launcher				
	Element	Class, %	Method, %	Line, %
64% classes, 70% lines covered in package 'level'				
Bridge	100% (1/1)	100% (5/5)		100% (21/21)
CollisionInteractionMap	0% (0/2)	0% (0/9)		0% (0/56)
DefaultPlayerInteractionMap	0% (0/3)	0% (0/4)		0% (0/18)
Hole	100% (2/2)	83% (5/6)		85% (17/20)
Level	77% (7/9)	75% (37/49)		73% (265/361)
LevelFactory	33% (1/3)	66% (8/12)		76% (19/25)
MapParser	100% (1/1)	92% (13/14)		81% (141/172)
MovableCharacter	100% (1/1)	83% (5/6)		80% (8/10)
Pellet	100% (1/1)	100% (3/3)		100% (6/6)
Player	100% (1/1)	86% (13/15)		85% (34/40)
PlayerCollisions	50% (1/2)	73% (11/15)		70% (61/86)
PlayerFactory	100% (1/1)	100% (2/2)		100% (5/5)
Teleport	100% (1/1)	80% (4/5)		88% (15/17)

On voit que l'utilisation des méthodes de la classe est un peu moins importante. Cela s'explique par le fait que la classe *Level* gère le mode normal mais également le mode avec la map infinie. C'est à cause de cela que certaines méthodes ne sont pas appelées en fonction de si l'on joue à un mode ou bien à l'autre (par exemple, il est inutile de compter le nombre de gommes restantes en mode map infinie, cette méthode s'occupant de les compter ne sera donc jamais appelée).

Pour les autres packages, les raisons pour lesquelles ils sont moins utilisés sont les suivantes :

- Package Board : Etant donné que nous avons lancé une partie normale, sans le principe de map infinie, certaines méthodes spécifiques à ce mode n'ont jamais été appelée, il semble donc logique que l'on perde en code coverage dans ce cas-ci.
- Package Fruit : Le package fruit s'occupe de gérer l'apparition des fruits sur le board. Le nombre de fruits possible est assez important (8), il est difficile de jouer une partie faisant appel à l'apparition de tout ces fruits. Cela explique pourquoi ce package est utilisé aussi peu.

Voici le code coverage pour le mode de la map infinie :

Coverage Launcher				
	Element	Class, %	Method, %	Line, %
nl.tudelft.jpacman	65% (60/91)	61% (280/454)	59% (1133/1910)	

Comme expliqué précédemment, dans le cas du mode map infinie, pas mal de comportement du jeu de base sont perdus (compter les gommes restantes, faire respawn les fantômes, les supers gommes absentes (pour le code pré-final)). C'est pour cette raison que le code est moins utilisé encore qu'en mode normal ou en mode avec les fruits.

## Test Coverage

Voici le test coverage appliquée au code initial :

Coverage All in jpacman-framework (1)				
	Element	Class, %	Method, %	Line, %
board	87% (7/8)	97% (40/41)	93% (98/105)	
game	100% (3/3)	66% (12/18)	76% (40/52)	
level	50% (8/16)	71% (45/63)	71% (230/322)	
npc	81% (9/11)	97% (33/34)	85% (141/164)	
sprite	80% (4/5)	86% (31/36)	89% (98/110)	
ui	100% (11/11)	76% (30/39)	83% (129/154)	
Launcher	100% (5/5)	79% (19/24)	76% (40/52)	
PacmanConfigurationException	0% (0/1)	0% (0/3)	0% (0/6)	

Et ici le code coverage appliquée à notre code pré-final :

Coverage All in jpacman-framework [2]					
	Element	Class, %	Method, %	Line, %	
+	board	70% (7/10)	79% (51/64)	64% (133/206)	
+	fruit	86% (13/15)	86% (31/36)	85% (92/107)	
+	game	60% (3/5)	44% (12/27)	48% (43/89)	
+	level	64% (18/28)	77% (113/145)	70% (593/837)	
+	npc	91% (11/12)	93% (54/58)	80% (223/276)	
+	sprite	80% (4/5)	90% (48/53)	90% (119/131)	
?	ui	100% (11/11)	75% (30/40)	70% (133/189)	
?	Launcher	100% (5/5)	82% (23/28)	78% (54/69)	
?	PacmanConfigurationException	100% (1/1)	33% (1/3)	33% (2/6)	

On constate que la couverture du code a été légèrement améliorée dans notre code pré-final.

## Duplication de code

Nous examinerons ici la duplication de code entre la version initiale et pré-finale de notre projet.

Voici l'analyse de duplication de code pour le code initial :

The screenshot shows the SonarQube Duplication tool interface. The left pane displays a list of file-level duplicates with their cost and count (e.g., 2 duplicates, Cost: 41 in 2 files). The right pane shows a detailed comparison between two code snippets, likely from different versions of the project. The snippets are labeled '#1 lines 109 to 119 in Inky' and '#2 lines 96 to 106 in Pinky'. The code is annotated with line numbers and color-coded highlights to indicate matching lines between the two versions.

On voit que la plupart des duplications de code se trouve dans les classes de test du projet. Une solution déjà à ce stade pour éviter cela aurait été d'appeler une méthode `@setUp()` qui s'occupe de gérer les parties communes dans une classe de test.

Voici à présent la duplication de code de notre projet pré-final :

Duplicates Project 'jpacman-framework'

```

2 duplicates, Cost: 60 in BridgeTest.java (nl.tudelft.jpacman.level)
2 duplicates, Cost: 52 in TeleportTest.java (nl.tudelft.jpacman.level)
2 duplicates, Cost: 46 in BoardTest.java (nl.tudelft.jpacman.board)
2 duplicates, Cost: 43 in BulletTest.java (nl.tudelft.jpacman.level)
2 duplicates, Cost: 41 in 2 files
2 duplicates, Cost: 39 in Ghost.java (nl.tudelft.jpacman.npc.ghost)
3 duplicates, Cost: 34 in FruitTest.java (nl.tudelft.jpacman.level)
2 duplicates, Cost: 34 in TeleportTest.java (nl.tudelft.jpacman.level)
2 duplicates, Cost: 32 in BridgeTest.java (nl.tudelft.jpacman.level)
2 duplicates, Cost: 32 in TeleportTest.java (nl.tudelft.jpacman.level)
2 duplicates, Cost: 31 in BoardFactoryTest.java (nl.tudelft.jpacman.bc)
2 duplicates, Cost: 30 in BoardFactoryTest.java (nl.tudelft.jpacman.bc)
3 duplicates, Cost: 27 in 3 files
2 duplicates, Cost: 25 in FruitTest.java (nl.tudelft.jpacman.level)
2 duplicates, Cost: 25 in BulletTest.java (nl.tudelft.jpacman.level)
2 duplicates, Cost: 24 in NavigationTest.java (nl.tudelft.jpacman.npc)
2 duplicates, Cost: 22 in Level.java (nl.tudelft.jpacman.level)
2 duplicates, Cost: 22 in 2 files
2 duplicates, Cost: 21 in NavigationTest.java (nl.tudelft.jpacman.npc)
2 duplicates, Cost: 20 in Ghost.java (nl.tudelft.jpacman.npc.ghost)
2 duplicates, Cost: 20 in BridgeTest.java (nl.tudelft.jpacman.level)
2 duplicates, Cost: 19 in 2 files
2 duplicates, Cost: 17 in NavigationTest.java (nl.tudelft.jpacman.npc)
3 duplicates, Cost: 17 in FruitTest.java (nl.tudelft.jpacman.level)
2 duplicates, Cost: 16 in LevelTest.java (nl.tudelft.jpacman.level)
2 duplicates, Cost: 16 in NavigationTest.java (nl.tudelft.jpacman.npc)
2 duplicates, Cost: 16 in PacManSprites.java (nl.tudelft.jpacman.sprite)
2 duplicates, Cost: 16 in MapParser.java (nl.tudelft.jpacman.level)

```

#1 lines 225 to 235 in BridgeTest (nl.tudelft.jpacman.level) (Read-only)

```

PacManSprites pms = new PacManSprites();
MapParser parser = new MapParser(new LevelFacto
    new GhostFactory(pms), new BoardFacto
Board b = parser.parseMap(Lists.newArrayList("#
    "#, "#####", "-----", "-----",
Square bridgeSquare = b.squareAt(2, 2);
Player p = new Player(pms.getPacManSprites(), pm
CollisionMap cm = new PlayerCollisions();
Unit bridge = bridgeSquare.getOccupants().get(
Direction[] dirs = {Direction.WEST, Direction.E
Direction[] dirs = {Direction.WEST, Direction.E
    Direction.SOUTH};

```

#2 lines 258 to 268 in BridgeTest (nl.tudelft.jpacman.level) (Read-only)

```

PacManSprites pms = new PacManSprites();
MapParser parser = new MapParser(new LevelFacto
    new GhostFactory(pms), new BoardFacto
Board b = parser.parseMap(Lists.newArrayList("#
    "#, "#####", "-----", "-----",
Square bridgeSquare = b.squareAt(2, 2);
Player p = new Player(pms.getPacManSprites(), pm
CollisionMap cm = new PlayerCollisions();
Unit bridge = bridgeSquare.getOccupants().get(
    Direction[] dirs = {Direction.WEST, Direction.E
    Direction.SOUTH};

```

Duplicates: Project 'jpacman-framework' Project 'jpacman-framework'

```

2 duplicates, Cost: 14 in Bridge.java (nl.tudelft.jpacman.level)
2 duplicates, Cost: 14 in 2 files
#1 lines 158 to 159 in BulletTest (nl.tudelft.jpacman.level)
#2 lines 315 to 316 in TeleportTest (nl.tudelft.jpacman.level)
16 duplicates, Cost: 14 in 2 files
#1 lines 99 to 102 in BridgeTest (nl.tudelft.jpacman.level)
#2 lines 116 to 119 in BridgeTest (nl.tudelft.jpacman.level)
#3 lines 132 to 135 in BridgeTest (nl.tudelft.jpacman.level)
#4 lines 148 to 151 in BridgeTest (nl.tudelft.jpacman.level)
#5 lines 164 to 167 in BridgeTest (nl.tudelft.jpacman.level)
#6 lines 179 to 182 in BridgeTest (nl.tudelft.jpacman.level)
#7 lines 194 to 197 in BridgeTest (nl.tudelft.jpacman.level)
#8 lines 209 to 212 in BridgeTest (nl.tudelft.jpacman.level)
#9 lines 94 to 97 in TeleportTest (nl.tudelft.jpacman.level)
#10 lines 110 to 113 in TeleportTest (nl.tudelft.jpacman.level)
#11 lines 126 to 129 in TeleportTest (nl.tudelft.jpacman.level)
#12 lines 141 to 144 in TeleportTest (nl.tudelft.jpacman.level)
#13 lines 157 to 160 in TeleportTest (nl.tudelft.jpacman.level)
#14 lines 173 to 176 in TeleportTest (nl.tudelft.jpacman.level)
#15 lines 188 to 191 in TeleportTest (nl.tudelft.jpacman.level)
#16 lines 203 to 206 in TeleportTest (nl.tudelft.jpacman.level)
2 duplicates, Cost: 13 in BoardPanel.java (nl.tudelft.jpacman.ui)
2 duplicates, Cost: 13 in MapParser.java (nl.tudelft.jpacman.level)
2 duplicates, Cost: 13 in NavigationTest.java (nl.tudelft.jpacman.npc)
5 duplicates, Cost: 12 in 2 files
3 duplicates, Cost: 12 in 2 files

```

Contents are identical

#1 lines 99 to 102 in BridgeTest (nl.tudelft.jpacman.level) (Read-only)

```

PacManSprites sprites = new PacManSprites();
MapParser parser = new MapParser(new LevelFacto
    new GhostFactory(sprites), new BoardFacto
thrown.expect(PacmanConfigurationException.class);

```

#2 lines 116 to 119 in BridgeTest (nl.tudelft.jpacman.level) (Read-only)

```

PacManSprites sprites = new PacManSprites();
MapParser parser = new MapParser(new LevelFacto
    new GhostFactory(sprites), new BoardFacto
thrown.expect(PacmanConfigurationException.class);

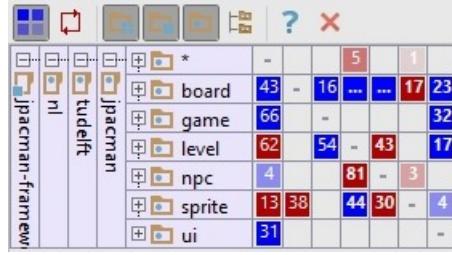
```

On a le sentiment que le notre de duplication a fortement augmenté. Cependant, comme pour le projet initial, nombre de duplication proviennent des classes de test ajoutée tout au long de ce projet. Dans la version finale rendue sur github, la plupart des duplications de code hors test ont été corrigée.

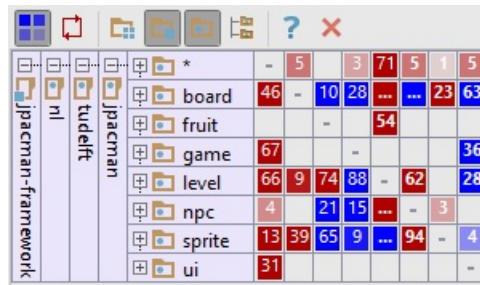
## Analyse des dépendances

Nous allons analyser ici plus en détail les dépendances entre nos classes.

Voici les dépendances entre classes du programme initial :



Et voici à présent celles de notre code pré-final :

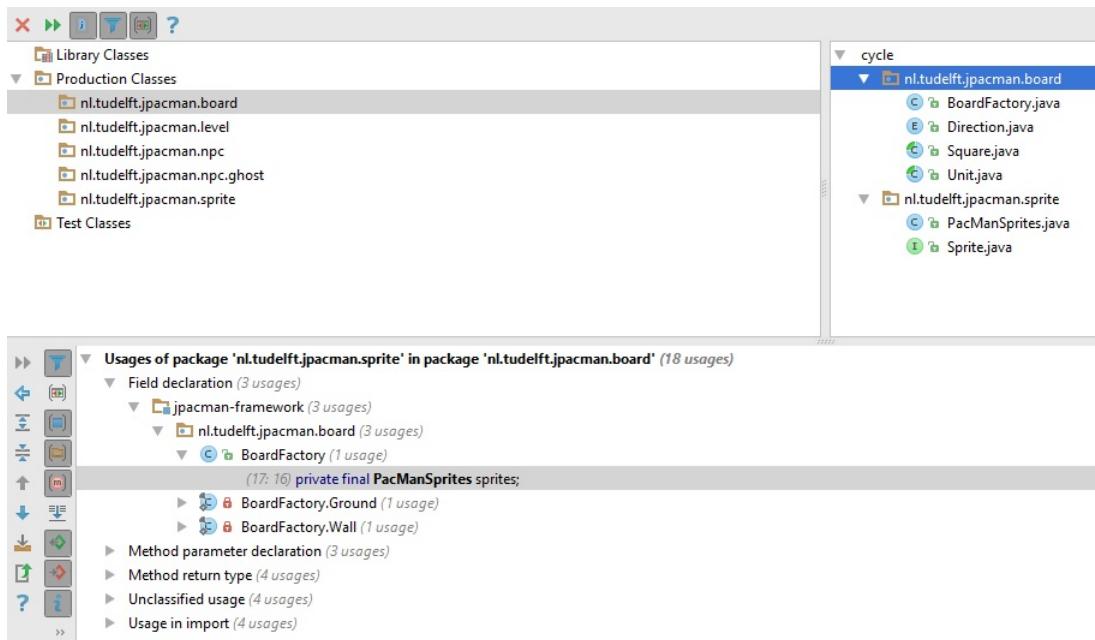


On voit en rouge les dépendances cycliques et en bleu les autres types de dépendances.

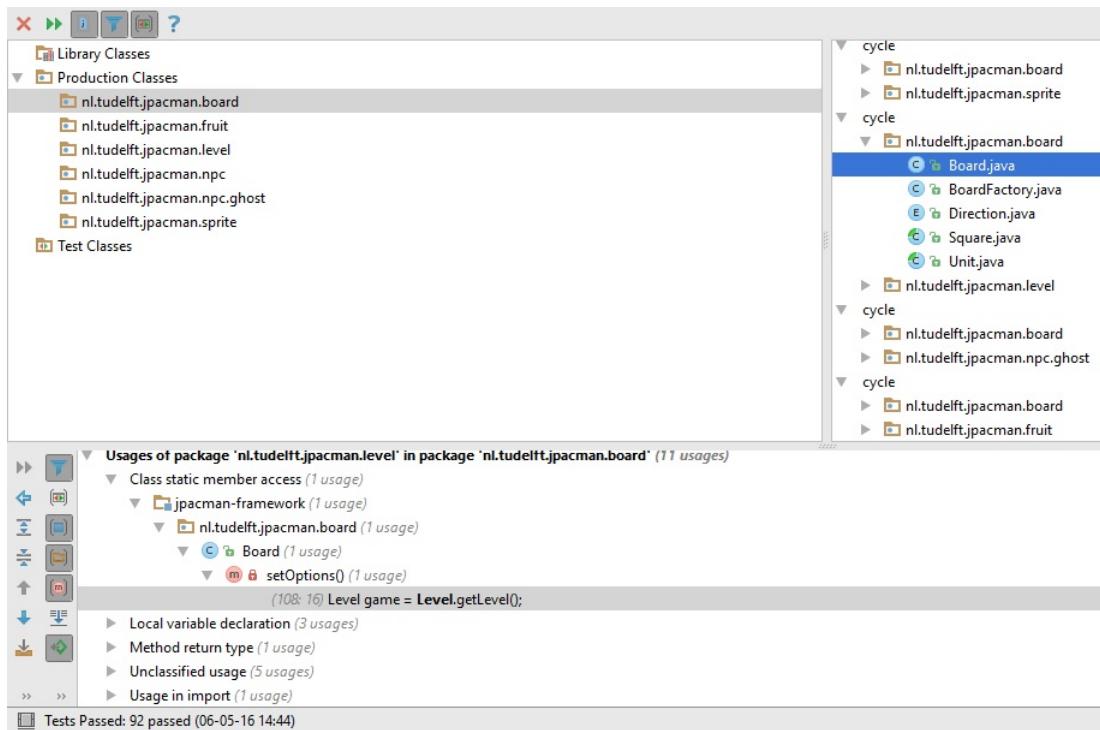
Comme nous pouvons le voir sur les matrice de dépendance du code initial et pré-final, d'une manière générale notre code a augmenté considérablement le nombre total de dépendance et en particulier entre certaines classes (*Level* et *Game* par exemple). De plus, il semblerait que certaines dépendances entre les packages soient devenue cyclique dans notre code, et enfin le nombre de dépendances cyclique sur l'ensemble du projet a également considérablement augmenté dans notre version.

### Dépendance cyclique :

Pour illustrer cela, regardons plus en détails les dépendances du package *board*. Comme nous pouvons le voir ci-dessous dans le code de base, il n'y a de dépendance cyclique qu'entre *board* et *sprite*. Par exemple, nous voyons bien que la classe *BoardFactory* a une dépendance avec la classe *PacManSprites* qui elle même en a avec la classe *Direction*.



Tandis que pour les mêmes packages, à savoir *board*, dans notre version du code, il a des dépendances cycliques avec 3 autres packages en plus de *sprite* comme par exemple la classe *Board* qui a une dépendance avec la classe *Level* du packages *level* et cette classe *Level* qui a elle même une dépendance avec la classe *Board*. Nous voyons donc bien là l'apparition d'un cycle.



## Analyse de métrique

Ligne de code (LOC) :

Comme nous pouvons le voir ci-dessous, pour le code de base il y a un total de 4666 lignes de code avec une moyenne de 583 lignes par packages dont 3890 (486 de moyenne) de ligne pour l'application et 776 (97 de moyenne) de code de test.

Metrics Lines of code metrics for Project 'jpacman-framework' from dim., 8 mai 2016 16:52:06 CEST						
package	LOC	LOC(rec)	LOCp	LOCp(rec)	LOCt	LOCt(rec)
nl		4.666		3.890		776
nl.tudelft		4.666		3.890		776
nl.tudelft.jpacman	309	4.666	209	3.890	100	776
nl.tudelft.jpacman.board	772	772	475	475	297	297
nl.tudelft.jpacman.game	199	199	199	199	0	0
nl.tudelft.jpacman.level	1.304	1.304	1.167	1.167	137	137
nl.tudelft.jpacman.npc	24	954	24	808	0	146
nl.tudelft.jpacman.npc.ghost	930	930	784	784	146	146
nl.tudelft.jpacman.sprite	594	594	498	498	96	96
nl.tudelft.jpacman.ui	534	534	534	534	0	0
<b>Total</b>	<b>4.666</b>		<b>3.890</b>		<b>776</b>	
Average	583,25		486,25		97,00	

Tandis que pour notre code, nous avons remarqué que le nombres de lignes a augmenté dans tous les packages pour atteindre un total de 8604 ligne (956 de moyenne) dont 6525 (725 de moyenne) de code d'application et 2079 (231 de moyenne) de code de tests.

Metrics Lines of code metrics for Project 'jpacman-framework' from dim., 8 mai 2016 16:50:46 CEST						
package	LOC	LOC(rec)	LOCp	LOCp(rec)	LOCt	LOCt(rec)
nl		8.604		6.525		2.079
nl.tudelft		8.604		6.525		2.079
nl.tudelft.jpacman	379	8.604	267	6.525	112	2.079
nl.tudelft.jpacman.board	1.082	1.082	763	763	319	319
nl.tudelft.jpacman.fruit	391	391	391	391	0	0
nl.tudelft.jpacman.game	334	334	269	269	65	65
nl.tudelft.jpacman.level	3.749	3.749	2.442	2.442	1.307	1.307
nl.tudelft.jpacman.npc	120	1.344	120	1.164	0	180
nl.tudelft.jpacman.npc.ghost	1.224	1.224	1.044	1.044	180	180
nl.tudelft.jpacman.sprite	709	709	613	613	96	96
nl.tudelft.jpacman.ui	616	616	616	616	0	0
<b>Total</b>	<b>8.604</b>		<b>6.525</b>		<b>2.079</b>	
Average	956,00		725,00		231,00	

Number of classes :

Comme nous pouvons le remarquer ci-dessous, le code de base possède un total de 72 classes repartie dans 8 packages avec 10 classes dédié au tests.

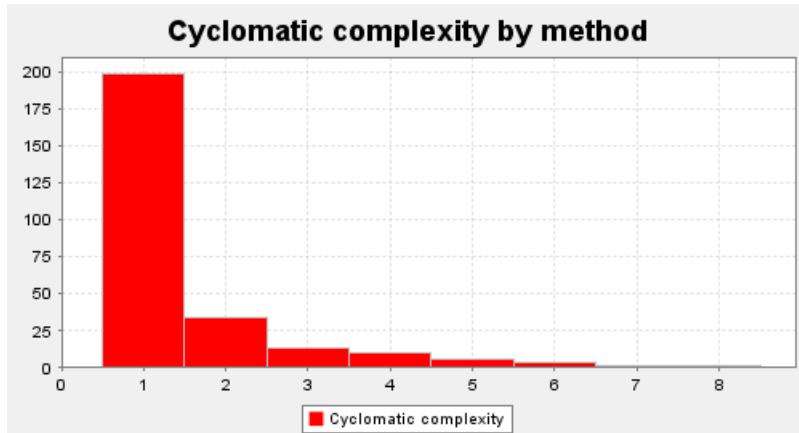
Metrics Class count metrics for Project 'jpacman-framework' from dim., 8 mai 2016 17:06:20 CEST						
package	C	C(rec)	Cp	Cp(rec)	Ct	TC(rec)
nl		72		62		10
nl.tudelft		72		62		10
nl.tudelft.jpacman	7	72	6	62	1	10
nl.tudelft.jpacman.board	13	13	7	7	6	6
nl.tudelft.jpacman.game	3	3	3	3	0	0
nl.tudelft.jpacman.level	18	18	17	17	1	1
nl.tudelft.jpacman.npc	1	11	1	10	0	1
nl.tudelft.jpacman.npc.ghost	10	10	9	9	1	1
nl.tudelft.jpacman.sprite	7	7	6	6	1	1
nl.tudelft.jpacman.ui	13	13	13	13	0	0
<b>Total</b>	<b>72</b>		<b>62</b>		<b>10</b>	
Average	9,00		7,75		1,25	

Tandis que pour notre code, nous avons 111 classes répartie sur 9 packages avec 16 classes dédiées aux tests. Nous pouvons dire que le nombre moyen de classes par packages a augmenté, ce qui semble logique lors de l'ajout de fonctionnalités. Le package qui a connu la plus grosse augmentation du nombre de classes lui appartenant est le package *level* avec une augmentation de 18 classes pour arriver à un total de 36 classes. Parmi les 18 nouvelles classes, il y en a 5 pour les tests.

#### Cyclomatic complexity :

Il s'agit du nombre de chemin différent dans le graphe de contrôle de flux du programme.

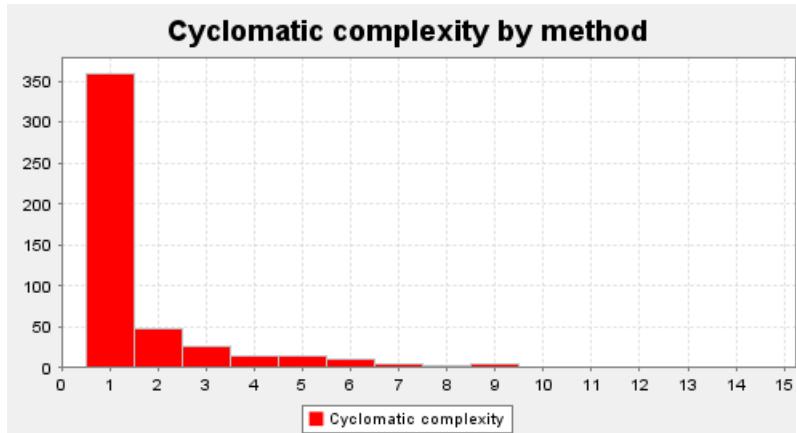
Comme nous pouvons le voir ci-dessous, la complexité cyclomatique de la plupart des méthodes du programme est de 1. Ensuite, il y a beaucoup moins de méthode avec 2 de complexité et ça continue de diminuer avec une complexité maximal de 8 (dernière colonne). C'est la méthode *nextMove()* de *Inky* qui possède la complexité maximal pour une complexité total de 415 et une moyenne de 1,55 par méthode.



nl.tudelft.jpacman.level.MapParser.checkMapFormat(List<Square>)	6	2	6
nl.tudelft.jpacman.npc.ghost.Navigation.findNearest(Cluster)	3	5	6
nl.tudelft.jpacman.level.CollisionInteractionMap.getInherentComplexity()	1	6	6
nl.tudelft.jpacman.level.LevelFactory.createGhost()	6	6	6
nl.tudelft.jpacman.level.MapParser.addSquare(Square[][], int)	2	2	7
nl.tudelft.jpacman.npc.ghost.Inky.nextMove()	5	8	8
<b>Total</b>	<b>325</b>	<b>373</b>	<b>415</b>
Average	1,21	1,39	1,55

Pour ce qui est de notre code, nous remarquons que le nombre de méthodes pour chaque complexité a augmenté et que la complexité maximal d'une méthode aussi est plus élevée. Nous pouvons le voir plus en détail ci-dessous (dernière colonne).

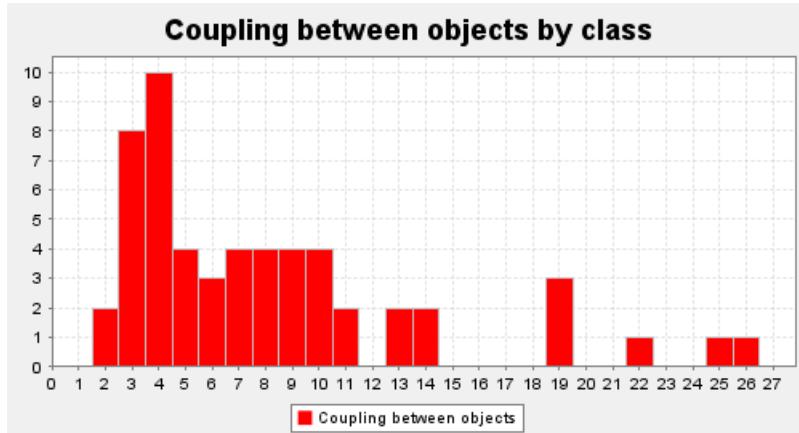
Dans notre cas, c'est la méthode *updateObservers()* qui a la complexité maximal avec 14, pour un total de 861 qui est logiquement lui aussi en augmentation et une moyenne de 1,76 qui a aussi légèrement augmenter dans notre cas. Vu que plus la complexité cyclomatique est faible plus le programme est facile à lire, tester et entretenir, dans notre cas, nous possédons un programme difficile à maintenir lorsque les classes possèdent une grosse complexité cyclomatique. C'est en effet ce que nous avons ressenti lors du refactoring final du code.



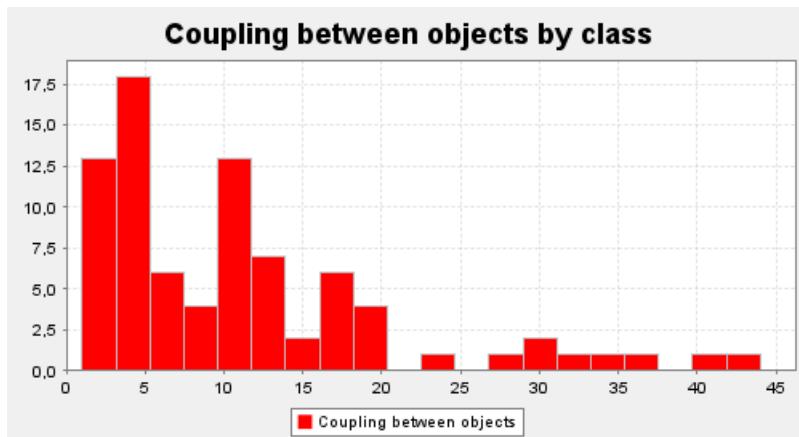
nl.tudelft.jpacman.level.PlayerCollisions.ghostColliding(	1	8	8
nl.tudelft.jpacman.fruit.FruitFactory.getRandomFruit()	8	8	8
nl.tudelft.jpacman.level.PlayerCollisions.playerColliding(	1	9	9
nl.tudelft.jpacman.npc.ghost.Inky.nextMove()	6	9	9
nl.tudelft.jpacman.ui.BoardPanel.renderInfinite(Board, Graphics)	1	9	9
nl.tudelft.jpacman.board.Board.getMiddleOfTheMap()	1	9	9
nl.tudelft.jpacman.level.MapParser.parseBridge(List<String>, MapParser)	4	6	10
nl.tudelft.jpacman.level.Level.addGhostTask()	2	7	11
nl.tudelft.jpacman.level.MapParser.addSquare(Square[][], int)	2	2	12
nl.tudelft.jpacman.level.Level.updateObservers()	1	14	14
<b>Total</b>	<b>586</b>	<b>785</b>	<b>861</b>
Average	1,20	1,61	1,76

Coupling between object (Mesure le nombre de classes auxquelles sont liées une classe données) :

Comme nous pouvons le voir sur le diagramme ci-dessous, le couplage minimum est de 2 c'est à dire qu'une classe donnée est liée à deux autres classes, le nombre maximum de lien au départ d'une classe est de 26 et la plupart des classes ont entre 3 et 10 liens avec d'autres classes, avec une moyenne à 8,15.

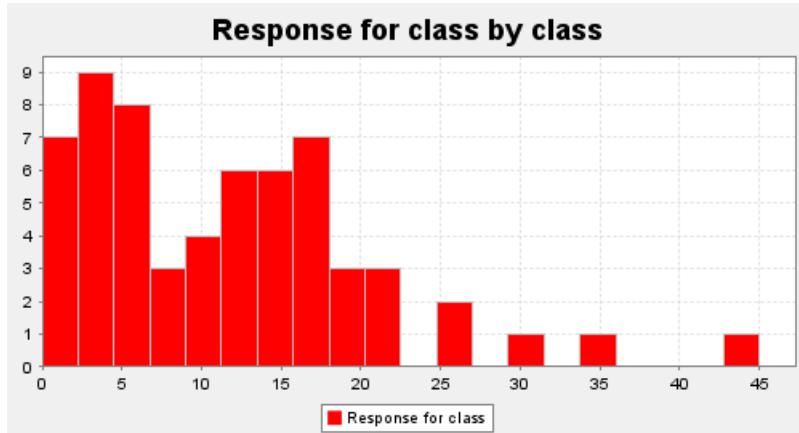


Pour notre code, nous voyons que le couplage maximal est de 44 et que la plupart des classes ont entre 2 et 21 autres classes auxquelles elles sont liées, moyenne à 11,16. Ici, le minimum est à 1 (Ce sont essentiellement les timers qui présentent ce couplage). On voit donc que certaines classes souffrent de couplage intensif, ce qui rend également le code difficile à lire et maintenir si cela est trop présent. Dans notre cas, c'est la classe Level qui souffre le plus de ce problème.

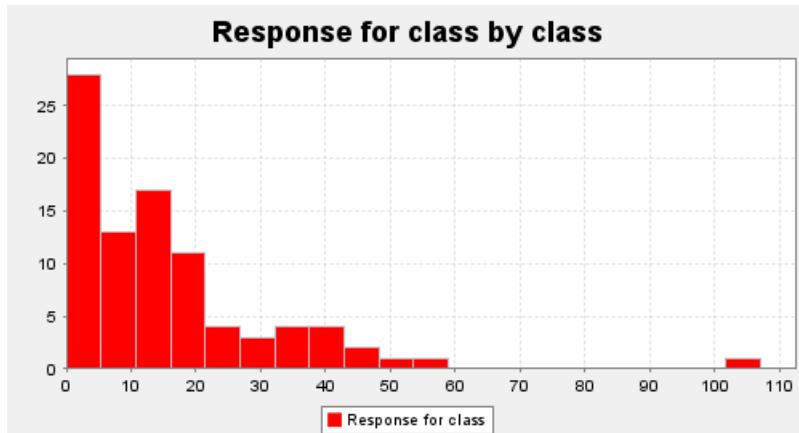


RFC (response for a class) :

Nous pouvons voir sur le diagramme ci-dessous que le code de base a un RFC maximal de 45 pour la classe *Level* et un RFC minimal de 1 pour 7 classes, à signaler que la classe *GhostColor* a un RFC de 0. De plus, la plupart des RFC se situent entre 1 et 22, avec une moyenne de 11,59.

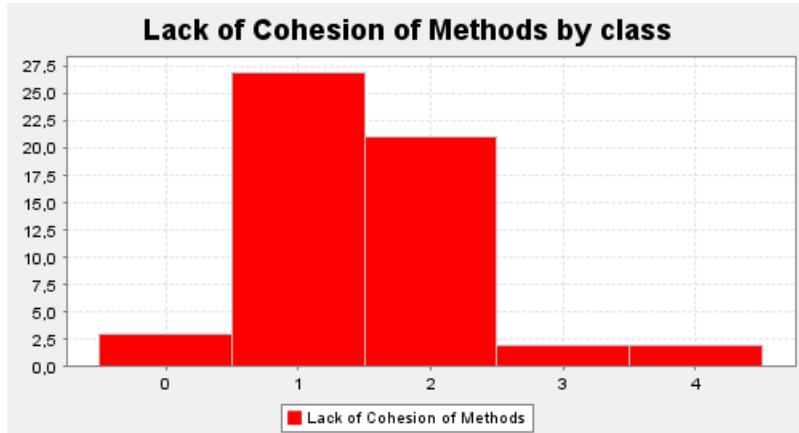


Pour notre code, nous remarquons un RFC maximal à 107 pour la classe *Level*. La plupart des autres RFC se situe entre 1 et 20, moyenne à 15,89. Les changements par rapport au code de base ne sont donc pas très importants en dehors de *Level* qui semble souffrir de gros problèmes à plusieurs niveaux à ce stade.

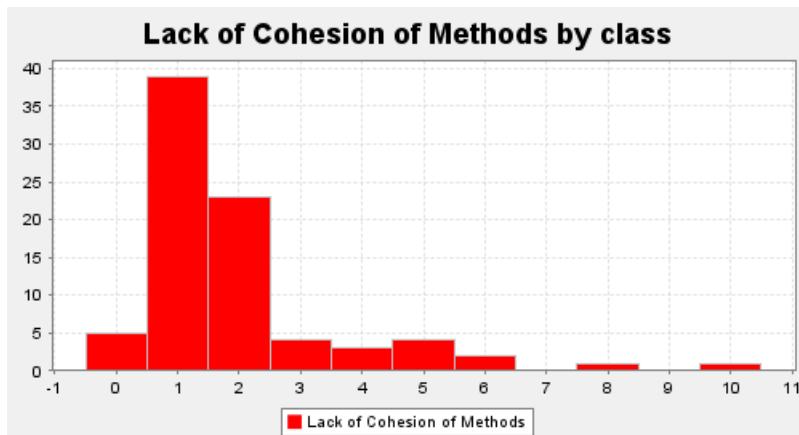


LCOM (lack of cohesion methods) :

Nous pouvons voir ici que pour le code de base possède un LCOM minimal de 0 pour 3 classes et un maximal de 4 pour 2 classes, la plupart des classes possèdent 1 ou 2 de LCOM, le tout pour une moyenne de 1,51.



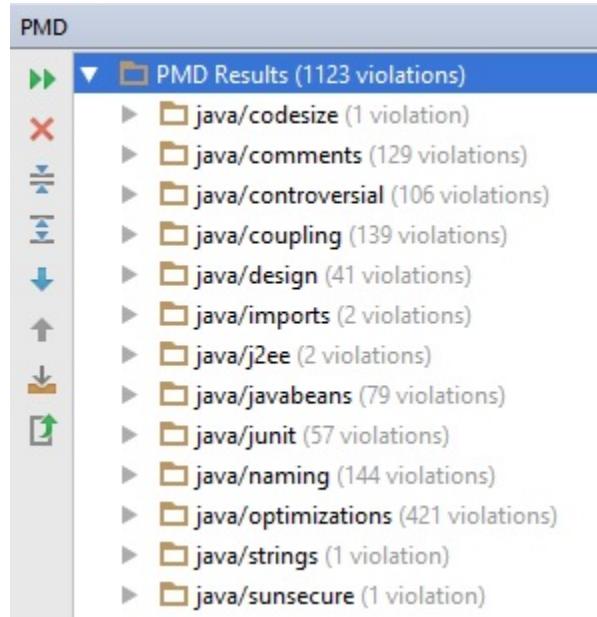
Nous voyons de suite que le LCOM maximal a augmenté et est maintenant de 10 pour la classe TeleportTest de notre code. Ensuite, la plupart des classes ont toujours entre 1 et 2 de LCOM et le minimum est toujours de 0 mais pour 5 classes cette fois-ci. Le tout pour une moyenne de 1,94.



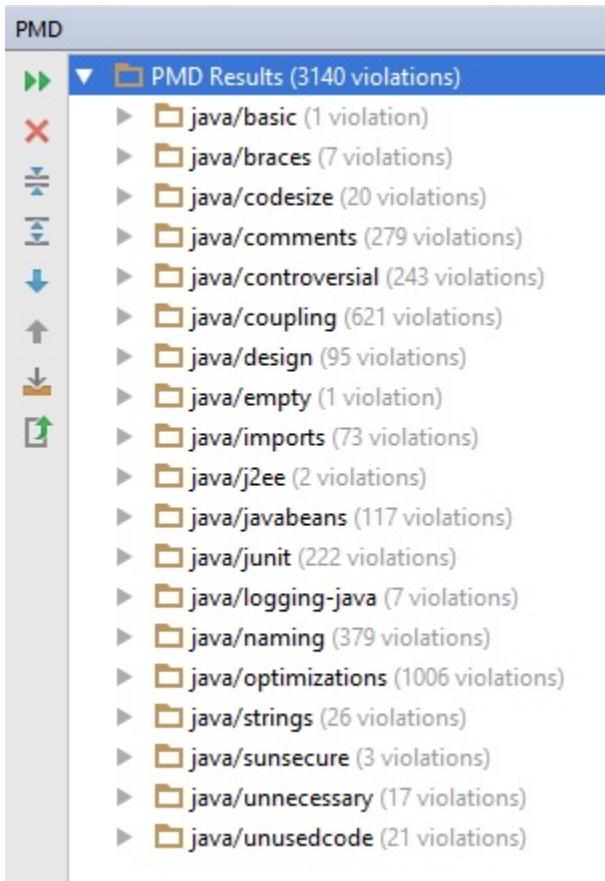
### Vérification de la qualité de la structure

Une grande partie du refactoring effectué avec l'aide du plugin PMD a été effectuée par Yarol Timur.

Voici quelques Bad Smell et violations rencontrés et signalés par PMD dans le code de base :



Et voici ce que nous avons analysé dans notre code pré-final :



On peut clairement constater que beaucoup de nouvelles catégories sont présentes pour notre code et que les catégories communes possèdent souvent beaucoup de violations en plus.

Détaillons rapidement chacune de ces catégories et comment nous les avons réglées.

- basic : Apparition dans le code de conditions "if" successives pouvant être fusionné en une seule condition "if". Ceci a facilement été corrigé grâce à l'utilisation d'une combinaison des deux conditions à l'aide du comparateur &&.
- braces : Apparition dans le code de conditions "if" ("else if", "else") sans {} (possible dans le cas où ces conditions ne contiennent qu'une seule ligne). Cette façon de faire est à proscrire car elle peut mener à des incompréhensions dans le code. Ces erreurs ont été rapidement corrigées en ajoutant simplement les accolades aux bons endroits.
- codesize : Certaines classes sont devenues trop grandes en terme de méthodes (Level et MapParser par exemple), ou de variable (Level par exemple). La complexité cyclique de certaines classes a également explosé. Ici il est beaucoup plus difficile de gérer le problème mais nous avons néanmoins fait notre possible pour réduire un maximum ces "bad smell" même si ce n'est pas toujours possible.
- comments : Taille de commentaires trop grand, trop de lignes de commentaires, commentaire mal formaté. Le soucis des commentaires mal formaté a été facilement résolu. Pour le reste, nous ne pouvions pas diminuer le nombre de ligne étant donné que nous respections les conventions de commentaire. Nous sommes donc retombé à un nombre de violation très proche de celui du code de base. Pour cette partie, nous pensons que ce n'est pas grave si le nombre de violation reste importante tant que les commentaires sont formatés correctement et compréhensibles.
- controversial : Plusieurs types de mauvaises façons de faire sont signalées ici. Par exemple le fait de coder en dur un test dans un "if" en utilisant un String (board == "monFichier") ou encore un int (ghostAlive < 10). En effet, si le nom du fichier change, le test ne fonctionnera plus, ou si on veut augmenter le nombre de fantome qui peuvent être en vie en même temps, il faut trouver dans le code ou changer la condition, ce qui peut être difficile. Ce genre de test n'est malheureusement pas toujours possible à éviter (par exemple dans la classe MapParser où l'on lit le fichier et où l'on fait quelque chose en fonction du symbole lu, car créer une variable static pour chaque symbole peut être au final plus couteux qu'avantageux). Une autre "mauvaise" façon de faire signalée par PMD est le fait qu'une méthode devant retourner quelque chose ne doit contenir qu'un seul return. Cela est facilement gérable bien que pas toujours utile. Par exemple dans le code ci-dessous :

```


    /**
     * Whatever happens, the squares on the board can't be null.
     *
     * @return false if any square on the board is null.
     */
    public boolean invariant() {
        for (int x = 0; x < board.length; x++) {
            for (int y = 0; y < board[x].length; y++) {
                if (board[x][y] == null) {
                    return false;
                }
            }
        }
        return true;
    }


```

On peut créer une variable booléenne au début de la méthode et lui donner une valeur false dans la boucle et true sinon et la retourner à la fin comme demandé. Cela corrigera cette mauvaise façon de faire selon PMD. Cela ne sera pas fait en pratique. Beaucoup d'autres façons de faire à proscrire sont encore signalées, comme le fait de ne faire qu'une déclaration de variable par ligne au lieu de int x = 0, y = 0, ou encore le fait d'appeler super() dans un constructeur d'une classe qui hérite d'une autre, etc. Tout ces points ont été au maximum amélioré lorsqu'il y avait une raison de le faire.

- coupling : Lorsqu'il y a trop de communication entre les classes. Ceci a été le sujet de la présentation de Yarol Timur lors du cours avec l'Intensive Coupling et le Dispersed Coupling. Quelques violations ont été réduites ici en suivant les conseils donnés lors de la présentation.
- design : Quelques designs non respectés. Rien de dangereux ou vraiment mauvais.
- empty : Variable ou référence jamais utilisée. Il suffit simplement de la supprimer ou de l'utiliser quelque part si besoin.
- import : Certains imports ne sont pas utilisés. Il suffit de les supprimer.
- javabeans :
- logging-java : Les prints doivent être utilisés pour le débogage et supprimés ensuite. C'est ce que nous avons fait.
- naming : Convention de nommage. Ceci n'a pas été géré.
- optimizations : Beaucoup de manière d'optimiser le code en rendant certaines variables finales par exemple (dans le cas où elles sont utilisées une seule fois localement à une méthode), ou encore en évitant de déclarer une variable dans

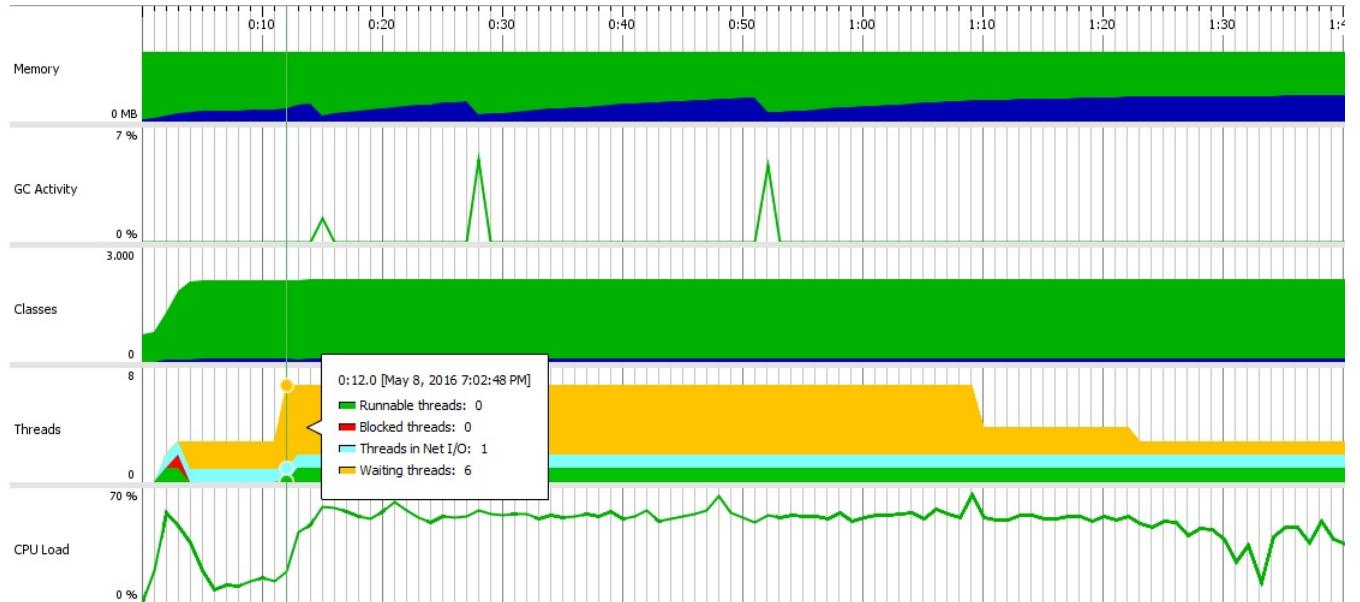
une boucle (il vaut mieux la déclarer à l'extérieur de la boucle et l'instancier à l'intérieur si besoin), etc. Nous avons compris chacune des manières d'optimiser le code selon cet analyse de PMD.

- string : La méthode equals() pour comparer deux objets est à préférer par rapport à `==` ou `!=`. C'est ce que nous avons fait.
- unusedcode : Partie du code non utilisée. Il suffit de la supprimer.

Beaucoup d'autres refactorings sont encore possible mais n'ont pas été effectués par manque de temps. Néanmoins, la plupart des violations signalées par PDM après ce refactoring sont pour une très grande partie des violations minimes (nom des variables, tailles des commentaires, ...) et n'ont donc pas lieu d'être résolues. Par contre, suite à ce refactoring complet du code, nous avons remarqué que certains problèmes présents depuis le début n'ont pas été réglé. Par exemple, lorsque l'on met le jeu en pause, les Timer ne s'arrêtent pas (pas de méthodes `pause()` ou `stop()` pour la classe Timer). C'est un soucis que nous aurions pu éviter si nous avions mieux pensé à la manière de gérer les modes de jeu à la base.

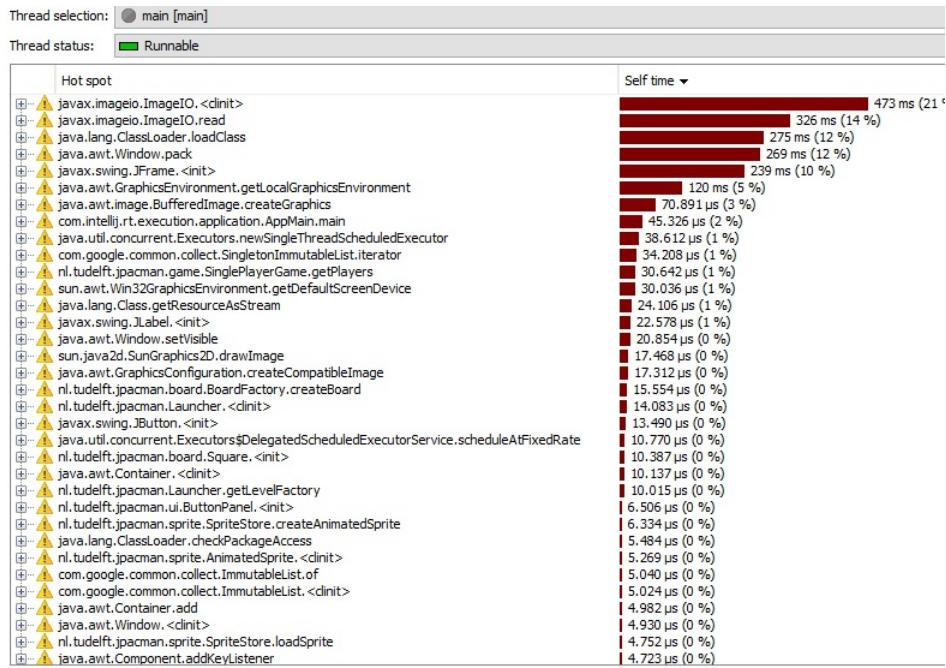
L'utilisation de cet outil nous a permis de mieux voir les mauvaises habitudes que nous avons et nous allons essayer dans le futur de gérer ces violations dès l'implémentation et non plus lors d'un refactoring qui peut prendre du temps et être difficile.

## Analyse des performances du programme de base

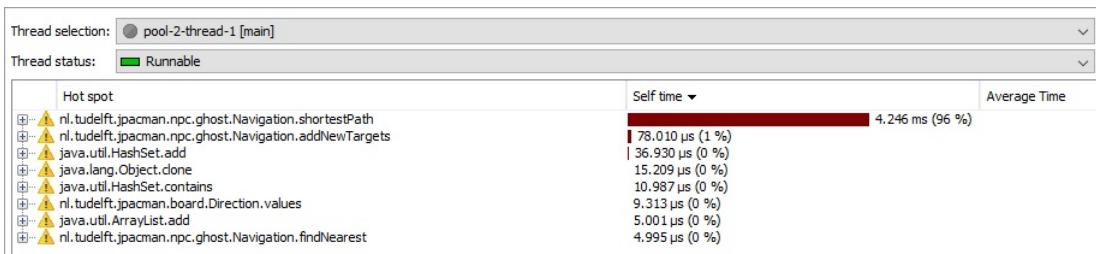


Au niveau de la mémoire, nous remarquons que la partie bleue du graphique qui correspond à la partie utilisée ne dépasse jamais 45Mo et qu'il en reste presque 100

de libre. Les activités au niveau de carte graphique correspondent à des « Garbage collector activity ». Au niveau des thread, nous remarquons que l'augmentation du nombre de thread en attente est due au lancement du jeu, tandis que la première baisse à 1min10 correspond au moment où on gagne ou perd la partie. Enfin, la charge CPU est à 70%. Ensuite, vers 1m23, le jeu a été coupé, impliquant une diminution de thread. Cependant, Jprofiler gardant la machine virtuelle java en vie pour ses enregistrement, nous ne pouvons pas voir la libération de la mémoire.



Ci-dessus, nous pouvons voir les activités du thread principal.



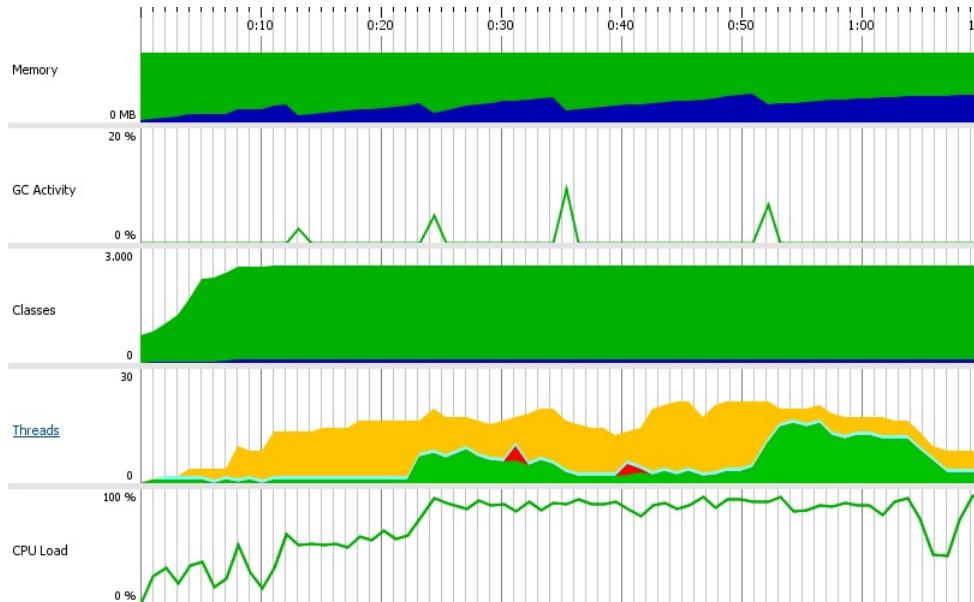
Ci-dessus, celle d'un thread de fantômes.

Name	Instance count ▾	Size
nl.tudelft.pacman.npc.ghost.Navigation\$Node	815.674	19.576 kB
java.lang.Object[]	98.419	4.390 kB
nl.tudelft.pacman.board.Direction[]	89.525	2.864 kB
java.util.HashMap\$Node	17.046	545 kB
com.google.common.collect.Iterators\$12	15.766	378 kB
com.google.common.collect.SingletonImmutableList	15.766	378 kB
char[]	13.039	786 kB
java.lang.String	12.172	292 kB
int[]	4.114	2.037 kB
java.lang.Class	2.473	284 kB
java.util.Hashtable\$Entry	1.677	53.664 bytes
java.security.AccessControlContext	1.499	59.960 bytes
java.util.ArrayList	1.273	30.552 bytes
java.lang.Integer	990	15.840 bytes
java.awt.Rectangle	934	29.888 bytes
java.util.concurrent.ConcurrentHashMap\$Node	885	28.320 bytes
java.lang.Object	812	12.992 bytes
java.lang.reflect.Field	785	56.520 bytes
java.lang.Class[]	690	18.016 bytes
java.lang.StringBuilder	655	15.720 bytes
java.util.HashMap\$Node[]	628	239 kB
sun.java2d.pipe.Region	586	23.440 bytes
java.lang.reflect.Method	486	42.768 bytes
java.util.EnumMap	484	19.360 bytes
java.util.HashMap	456	21.888 bytes
java.lang.String[]	433	17.608 bytes
java.lang.Short	405	6.480 bytes
java.lang.ref.SoftReference	379	15.160 bytes
byte[]	351	231 kB
sun.misc.FDBigInteger	341	10.912 bytes
<b>Total:</b>	<b>1.116.555</b>	<b>33.251 kB</b>

Et finalement, un récapitulatif détaillé des objets en mémoire en cours de partie.

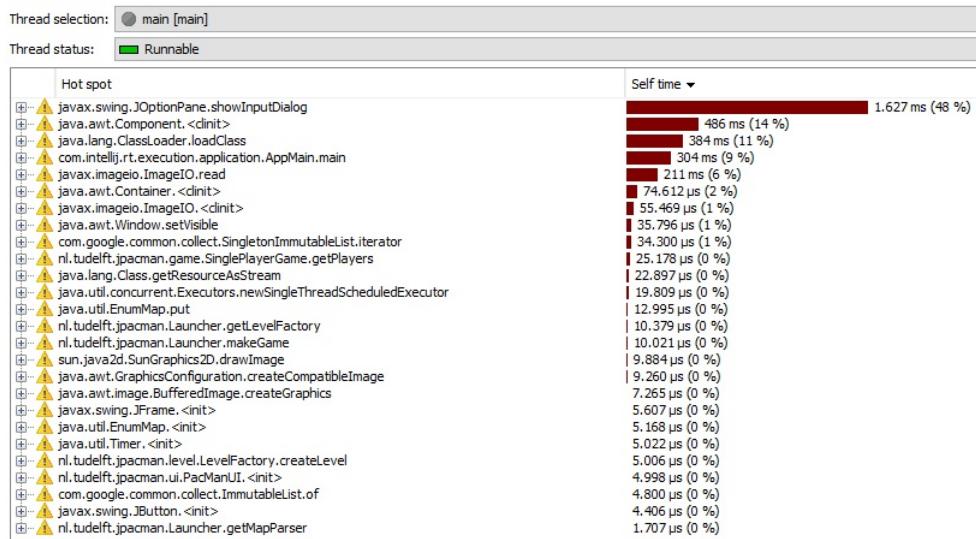
## Analyse des performances du programme pré-final en mode normal

Tout d'abord, l'analyse en mode normal avec la possibilité d'apparition de fruits :

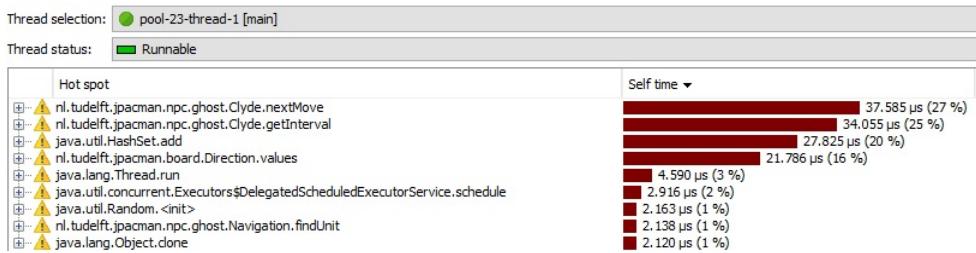


En moyenne, la quantité de mémoire utilisé est sensiblement la même. L'activité de la CG également. Le nombre de threads est plus élevé et fluctuant car les timers sont

considérés comme des thread. La charge CPU est elle aussi plus élevée à cause de ces timers.



Ci-dessus, les activités du thread principal qui semblent moins nombreuses que pour le code de base.



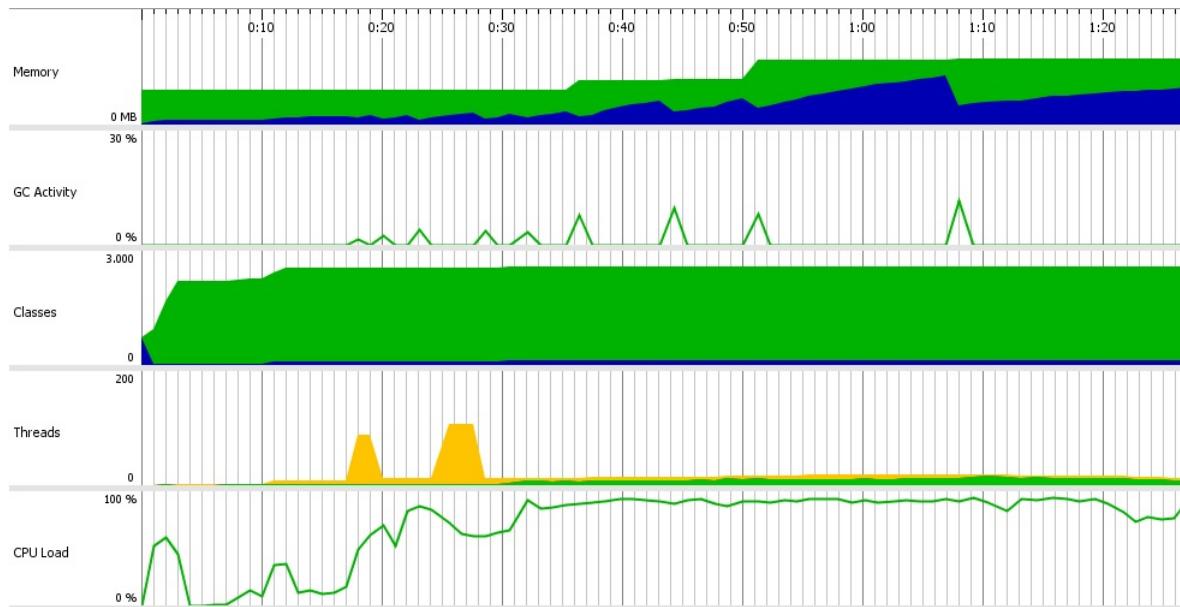
Les activités d'un fantôme sont similaires à celles du code de base.

Name	Instance count ▾	Size
nl.tudelft.jpacman.npc.ghost.Navigation\$Node	889.434	21.346 kB
nl.tudelft.jpacman.board.Direction[]	262.454	8.398 kB
java.lang.Object[]	165.064	7.323 kB
com.google.common.collect.SingletonImmutableList	32.679	784 kB
com.google.common.collect.Iterators\$12	32.578	784 kB
char[]	17.139	784 kB
java.lang.String	14.918	784 kB
java.util.HashMap\$Node	13.270	358 kB
int[]	10.573	424 kB
java.lang.Class	2.902	331 kB
java.security.AccessControlContext	2.738	109 kB
java.util.Hashtable\$Entry	2.562	81.984 bytes
java.awt.Rectangle	2.502	80.064 bytes
java.util.ArrayList	1.832	43.968 bytes
java.lang.StringBuilder	1.632	39.168 bytes
java.lang.Object	1.552	24.832 bytes
sun.java2d.pipe.Region	1.438	57.520 bytes
java.lang.Integer	1.240	19.840 bytes
java.util.concurrent.ConcurrentHashMap\$Node	992	31.744 bytes
java.awt.geom.AffineTransform	948	68.256 bytes
sun.java2d.SunGraphics2D	931	201 kB
java.lang.reflect.Field	913	65.736 bytes
java.lang.Class[]	812	20.696 bytes
java.util.HashMap\$KeyIterator	795	31.800 bytes
java.lang.ref.PhantomReference	674	21.568 bytes
sun.java2d.DefaultDisposerRecord	662	21.184 bytes
java.awt.Point	643	15.432 bytes
sun.awt.EventQueueItem	588	14.112 bytes
java.lang.reflect.Method	584	51.392 bytes
java.util.ArrayList\$1tr	570	18.240 bytes
<b>Total:</b>	1.497.957	47.624 kB

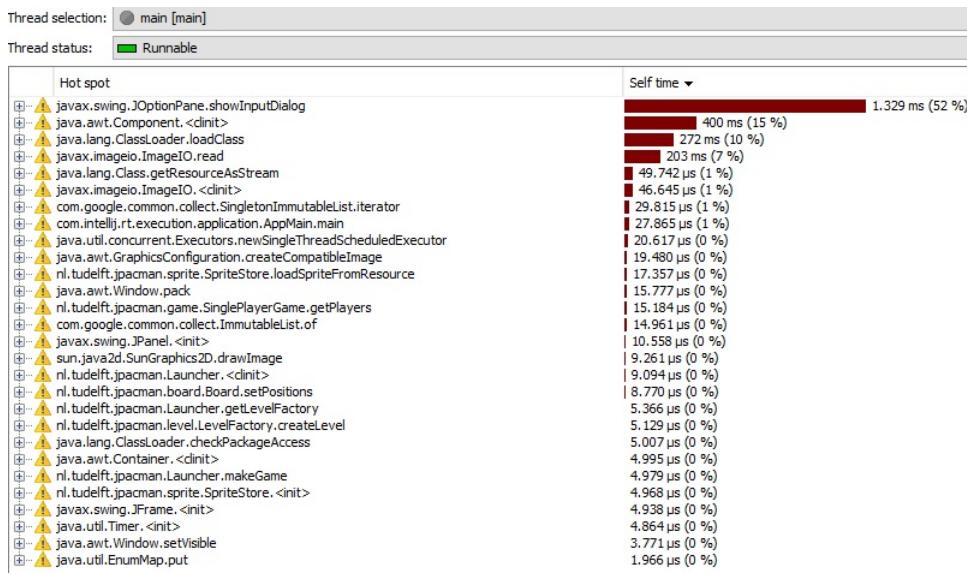
Un récapitulatif des objets en mémoire similaire au code de base sauf pour direction mais ceci est du au fait que le joueur peut se déplacer tout seul.

## Analyse des performances du programme pré-final en mode infini

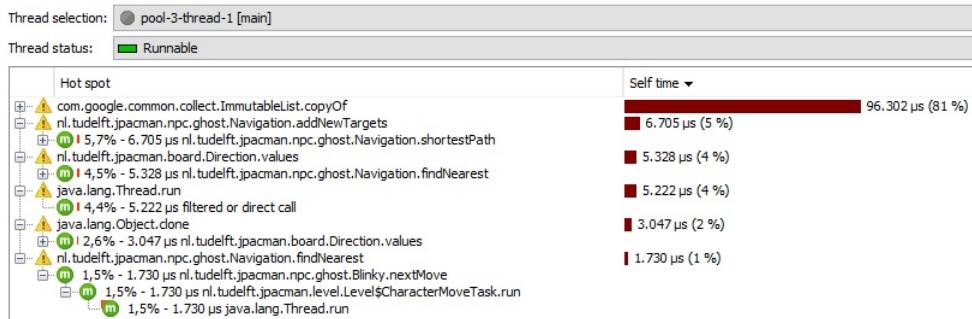
A présent, l'analyse en mode infini avec également la possibilité d'apparition de fruits :



Il y a beaucoup plus de mémoire utilisées dans ce cas. En effet, parfois cela dépasse les 100Mo. Il y a plus d'activités de garbage collector sur la GC. Le processeur est sollicité de la même façon que pour notre mode normal avec fruit, du au timer ici aussi. On peut également constater une explosion des threads au début du programme. Cela reste une énigme pour nous.



Voici un récapitulatif des activités du thread principal ci-dessus similaire au mode normal de notre code.



Les activités d'un fantôme sont similaires à celles de notre code en mode normal également.

Name	Instance count ▾	Size
nl.tudelft.jpacman.npc.ghost.Navigation\$Node	3.900.100	93.602 kB
nl.tudelft.jpacman.board.Direction[]	1.758.107	56.259 kB
java.util.HashMap\$Node	167.462	5.358 kB
java.lang.Object[]	123.159	9.367 kB
com.google.common.collect.Iterators\$12	35.048	841 kB
com.google.common.collect.SingletonImmutableList	35.048	841 kB
java.util.ArrayList	32.172	772 kB
java.util.EnumMap	30.913	1.236 kB
char[]	22.043	1.107 kB
java.lang.String	17.939	430 kB
nl.tudelft.jpacman.board.BoardFactory\$Ground	16.634	532 kB
nl.tudelft.jpacman.level.Pellet	16.419	525 kB
nl.tudelft.jpacman.board.BoardFactory\$Wall	14.278	456 kB
int[]	10.172	5.080 kB
java.lang.StringBuilder	2.933	70.392 bytes
java.lang.Class	2.929	335 kB
java.util.Hashtable\$Entry	2.013	64.416 bytes
java.lang.Integer	1.548	24.768 bytes
java.lang.Object	1.528	24.448 bytes
java.util.HashMap\$Node[]	1.031	2.212 kB
java.util.concurrent.ConcurrentHashMap\$Node	1.000	32.000 bytes
java.lang.reflect.Field	913	65.736 bytes
java.lang.Class[]	811	20.672 bytes
java.util.Formatter\$Flags	738	11.808 bytes
java.util.regex.Matcher	728	46.592 bytes
java.util.Formatter\$FormatString[]	728	17.472 bytes
java.util.Formatter\$Flags[]	728	17.472 bytes
java.util.LinkedHashMap\$LinkedKeyIterator	728	23.296 bytes
java.util.Formatter	728	23.296 bytes
java.text.DecimalFormatSymbols	728	46.592 bytes
<b>Total:</b>	<b>6.219.878</b>	<b>180 MB</b>

Ici, les objets en mémoire sont beaucoup plus nombreux, mais cela est du à la map infinie avec un board plus grand et au fait qu'il y a plus de fantomes qui spawn sur la map.

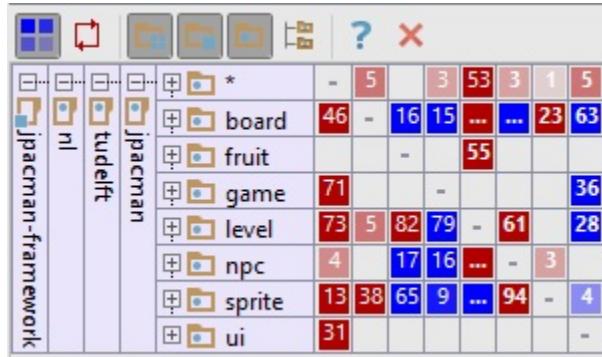
### 4.3.2 Analyse du code final

#### Refactoring

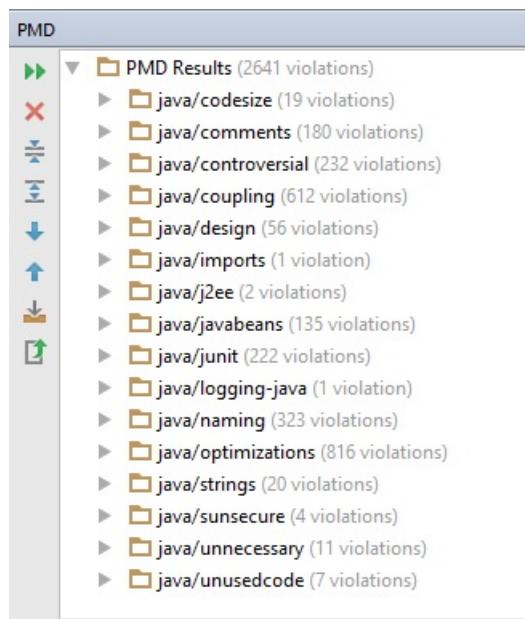
Voici certains gros refactoring effectué finalement sur notre code :

- Transfert de tous les Timers définis dans la classe Level dans une nouvelle classe TimerTasks. Cela a permis de réduire drastiquement le nombre de lignes et la lisibilité de la classe Level mais aussi de faciliter la compréhension des Timers qui sont mieux définis et plus propres.
- Gestion des Timers afin de les créer uniquement là où cela est nécessaire. Cela a permis de réduire drastiquement les consommations CPU et le nombre de Thread actif dans les différents modes de jeu comme nous le présentons ci-dessous dans l'analyse du code final.
- Correction des bugs majeurs survenant souvent lors des parties.
- Amélioration de la performance de nombreuses méthodes en diminuant leur complexité.

Après ce gros refactoring final afin d'améliorer les performances de notre code pré-final, voici rapidement quelques statistiques que nous avons recueillies :



Globalement, après le refactoring complet, les dépendances entre classes n'ont fondamentalement pas vraiment changées. Certaines sont passées d'une classe à l'autre, mais en moyenne, cela reste la même chose. Nous n'avons donc pas réussi à diminuer ces dépendances.



Comme expliqué dans la section concernant les violations et bad smell, nous avons réussi à corriger certaines manières de faire qui ne sont pas propre. Malgré tout, ce nombre impressionnant de violations comporte lui aussi un grand nombre de violations non importantes. De plus, le fait de gérer une violation comme proposée par PDM nous en créé une nouvelle qui nous renvoyait au cas initial. Cet outil nous a simplement permis de mieux analyser notre code.



A présent, nous constatons un retour d'utilisation du CPU identique à celui du programme de base dans le cas de notre code en mode normal (avec fruit), ce qui est une bonne chose après l'ajout de nos fonctionnalités. De plus, le nombre de Thread est ici beaucoup moins important, ce qui est aussi une bonne chose. Ces Threads sont également plus réguliers.



Pour le mode infini, une diminution drastique du nombre de Thread est visible. Ce nombre de Thread reste également plus ou moins constant tout le long de l'exécution de notre programme. Néanmoins, une utilisation du CPU à 100% se fait ressentir. Nous ne sommes pas parvenu à régler ce problème.

# Chapitre 5

## Difficultés rencontrées et bugs connus

### 5.1 Difficultés rencontrées

L'une des principales difficultés de ce projet a été tout d'abord la bonne compréhension du code initial fourni. Ensuite, pour chacune de nos parties individuelles, nous avons chacun rencontré la même difficulté sur comment implémenter notre fonctionnalité en impactant le moins négativement le projet. Au final, chacun d'entre nous a terminé sa partie individuelle avec de gros soucis de performances. C'est pour cette raison qu'un gros soucis a été la mise en commun de nos codes et la gestion de tous les bugs créés par cette fusion. Ils nous a fallu repenser plus proprement la plupart de nos méthodes et classes importantes pour le projet afin d'obtenir au final un résultat satisfaisant. Ce temps perdu aurait pu être évité si nous avions mieux pensé dès le départ à comment faire les choses proprement. C'est une leçon que nous retiendrons dans le cas ou de futurs projets dans le genre nous serons demandés.

### 5.2 Bugs connus

La plupart des bugs connus ont été résolus par Yarol Timur lors du refactoring final. Néanmoins, certains bugs restent présent et n'ont pas pu être résolu par manque de temps.

En voici la liste :

- Mettre en pause le jeu alors qu'un mode est en cours (l'effet d'un fruit, le mode Hunter) ne met pas en pause le Timer associé. Le jeu va donc tourner derrière bien que les unités seront figées par le bouton pause. Une solution aurait été d'utiliser une autre classe que celle de Timer permettant la mise en pause d'un Thread ou d'un Timer. Un manque de temps ne nous a pas permit de corriger ce détail.

- Les fantômes ne redémarrent pas correctement après leur respawn dans certaines situations. Ces situations sont visibles en particulier dans le mode Cheat et n'ont donc pas été corrigée par manque de temps également, les modes "normaux" fonctionnant correctement.
- L'apparition des fantômes en mode map infinie n'est pas bien gérée. Ils spawn à des endroits non visible de la map alors qu'ils devraient apparaître sur le sommet de la map. Il s'agit là d'un soucis de calcul sur l'apparition des fantômes.
- En mode map infinie, passé un certain nombre de fantôme, le jeu se met à freeze assez violemment, empêchant le joueur de continuer à jouer correctement. Nous supposons que le soucis provient des locks sur les instances de fantômes et n'avons pas réussi à régler le problème.
- Une difficulté de faire tourner pacman lorsqu'il se déplace dans un couloir afin de changer de direction. Ceci n'est pas vraiment un bug mais plutôt un manque de maniabilité. Nous n'avons pas su le résoudre.

Finalement, les tests fonctionnent tous sans problème en dehors du LauncherSmokeTest se basant sur un déplacement de pacman que nous avons supprimé (mit en commentaire) dans le code. Pour que ce test réussisse, il suffit de décommenter la ligne de code de la méthode *move(...)* de la classe *Game*.

Nous pensons avoir ici donné une liste des bugs connus les plus fréquent du jeu. Si d'autres bugs existent, nous n'en n'avons pas connaissance.

### 5.3 Cheat Mode

Ce mode est un petit bonus que nous avons intégré à notre projet afin de tester à la base certain comportement difficile à voir dans les autres modes. Nous l'avons intégré finalement aux autres modes sans raison particulière étant donné qu'ils ne nous a pris aucun temps à implémenter (une dizaine de lignes de code).

# Conclusion

Au terme de ce projet, nous pouvons clairement affirmer que ce ne fut pas une si-nécuré. Nous pensions à la base que ce projet nous prendrait moins de temps et serait plus facile à gérer. Nous sommes responsables de ces difficultés rencontrées tout au long du projet et citées dans la partie "Difficultés rencontrées" mais cela nous a permis également de nous rendre compte de mauvaises habitudes que nous avions.

De plus, au cours de ce projet, nous avons été amenés à utiliser des outils et logiciels nouveau qui nous seront utiles dans le cadre de futurs gros projets dans le genre.

Ce projet nous a également permis d'utiliser les principes vus au cours de Software Evolution et de mieux comprendre certains patterns tel que le design pattern Factory souvent utilisé dans le code.

Nous avons pu également mieux nous adapter à certains bad smell et à comment les résoudre mais aussi comment les éviter dans le futur.

Au terme de ce projet, nous pensons avoir acquis certaines connaissances et appris de nos erreurs. Si un tel projet était à refaire dans le futur, nous nous y prendrions probablement mieux dès le début et ne tomberions pas dans certains pièges.

Malgré tout cela, nous pouvons conclure en disant que nous avons respecté au mieux les consignes qui nous étaient données et que nous avons rendu un projet loin d'être parfait mais fonctionnel et nettoyé de la plupart des bads smells.