

Software evolution JPacman framework

Ducruet Corentin
Gallois Florent
Ledru Santorin

Année académique
2015 - 2016

Table des matières

1	Introduction	2
2	Supergomme - Ducruet Corentin	2
2.1	Programme initial et objectifs	2
2.2	Démarche suivie	2
3	Série de labyrinthe - Ledru Santorin	3
3.1	Programme initial et objectifs	3
3.2	Démarche suivie	3
3.2.1	Still alive !	3
3.2.2	Plusieurs niveaux	3
3.2.3	Sauvegarde de progression	4
4	IA pour fantômes - Gallois Florent	4
4.1	Programme initial et objectifs	4
4.2	Démarche suivie	4
5	Difficultés liées au merge	5
6	Analyse de la qualité du code	5
6.1	Outils utilisés	5
6.1.1	Google CodePro AnalitiX	5
6.1.2	EclEmma	6
6.1.3	CodeCity	6
6.1.4	PMD	6
6.2	Analyse des dépendances	6
6.2.1	Projet de base	6
6.2.2	Projet modifié	6
6.3	Code dupliqué	6
6.4	Test & Code Coverage	7
6.4.1	Projet de base	7
6.4.2	Projet modifié	7
6.5	Métriques	7
6.5.1	Tableaux de métriques	7
6.5.2	CodeCity : méthodes par classes	7
6.5.3	CodeCity : complexité cyclomatique	7
6.5.4	CodeCity : nombre de lignes de code	7
6.6	Bad Smells	8

1 Introduction

Dans le cadre du cours de Software Evolution, nous devons mettre en pratique les concepts d'évolution logicielle vus en cours. Le projet qui nous est confié consiste à récupérer un projet de pacman et d'y implémenter plusieurs fonctionnalités. Ces mêmes fonctionnalités doivent être faites tout en suivant un processus de développement dirigé par les tests. Après la réalisation de ces tâches, il nous est demandé de rassembler ces différents travaux dans le logiciel. Enfin, une analyse de la qualité ainsi qu'une amélioration du code doit nous permettre de terminer le logiciel. Dans un premier temps, nous étudierons chaque tâche individuelle : dans un premier temps nous parlerons de la réalisation de la supergomme, puis de la série de labyrinthe et enfin l'implémentation de l'IA des fantômes. Dans un deuxième temps, nous parlerons des difficultés rencontrées lors du merge des trois fonctionnalités et enfin nous parlerons de l'analyse du code et des améliorations.

2 Supergomme - Ducruet Corentin

2.1 Programme initial et objectifs

Dans la version de initial du framework Jpacman, il n'y avait qu'un seul type de gomme qui avait pour seul effet d'augmenter notre score de 10 points. Il nous a été demandé d'ajouter la fonctionnalité "Supergomme". Cette fonctionnalité ajoute un nouveau type de gomme (les "Supergomme"). celles-ci se distinguent des autres gommes par leur effet. Tout d'abord, elles sont rouges pour pouvoir les différencier visuellement. Ensuite, lorsqu'elles sont mangées par Pacman, le score est augmenté de 50 points et les rôles sont inversés. En effet, Pacman devient chasseur et les fantômes deviennent les proies (pendant 7 secondes lors des deux premières "Supergomme" mangées, 5 secondes pour les deux dernières). Lorsque les fantômes sont des proies, ils sont bleus et fuient Pacman. Si Pacman arrive à manger un fantôme, cela lui rapporte des points.(200 pour le premier, 400 pour le deuxième, 800 pour le troisième et 1600 pour le quatrième).

2.2 Démarche suivie

Pour réaliser cette fonctionnalité, nous avons tout d'abord créé une classe "SuperPellet" qui hérite de Pellet. Nous l'avons ajoutée à la MapParser pour que les "Supergomme" soient ajoutées à la carte.

Nous avons créé une classe "VulnerableGhost" qui représente les fantômes qui peuvent être chassés quand une "Supergomme" est mangée. Blinky, Clyde, Inky et Pinky héritent donc maintenant de "VulnerableGhost". Nous avons dans ce même temps, créé une classe "VulnerableGhostFactory" qui hérite de "GhostFactory" qui sert à instancier les fantômes et cela permet de respecter le design pattern factory.

Pour gérer l'alternance des modes, nous avons créé un timer que l'on peut mettre en pause. Ainsi, lorsque le jeu est en mode pause, le timer ne continue pas de s'exécuter.

Les collisions sont gérées dans "PlayerCollisionSuperPellet". Si Pacman mange une "Supergomme" le mode "chasseur" est lancé, les fantômes deviennent bleus et fuient Pacman. Pendant ce temps, un timer est lancé pour qu'après 7 secondes (pour les deux premières

"Supergommes", 5 secondes pour les dernières) le jeu revienne à la normal. Si Pacman mange un fantôme pendant le mode "chasseur", il gagne le score voulu(expliqué plus haut).

Enfin, nous avons créer un launcher("LauncherSuperPellet") pour lancer le jeu avec les paramètres pour que les "Supergommes" soient activés.

3 Série de labyrinthe - Ledru Santorin

3.1 Programme initial et objectifs

Initialement, Pac-Man mourrait et la partie s'achevait dès qu'un fantôme rentrait en contact avec Pac-Man. La partie s'achevait également lorsque le joueur avait ramassé toutes les gommes du labyrinthe de base proposé. Le premier objectif visait à donner à Pac-Man trois vies, et lorsqu'il mourrait de le téléporter à une case aléatoire à plus de quatre cases du fantôme le plus proche. Pac-Man doit également gagner une vie tous les dix milles points accumulés.

Ensuite, il fallait mettre en place un système de niveaux et, lorsqu'un niveau est terminé, la partie s'enchaîne sur le niveau suivant. Pac-Man conserve son nombre de vies et ses points lors du changement de niveau.

Enfin, la progression du joueur devait pouvoir être sauvegardée pour qu'il puisse reprendre sa partie au dernier niveau atteint.

3.2 Démarche suivie

3.2.1 Still alive !

Tout d'abord, une simple variable est ajoutée dans la classe Player pour tenir le compte des vies restantes de Pac-Man. Le décompte des vies se fait grâce à la méthode `setAlive(boolean)` de la classe Player. Le respawn de Pac-Man est effectué dans la méthode `levelLost()` de la classe `SinglePlayerGame`, un Observer est déjà en place, autant s'en servir. Dans cette méthode, si le joueur possède encore une vie, une case aléatoire du Board est choisie et grâce à la méthode `nearestValidRespawnPoint()` de `Square`, on déplace Pac-Man sur la position de respawn valide la plus proche.

3.2.2 Plusieurs niveaux

On gère d'abord le chargement de plusieurs niveaux. Les niveaux chargés seront tous les fichiers nommés `board*.txt` présent dans le dossier ressources des fichiers du jeu. ils seront toujours présentés au joueur dans l'ordre alphabétique des noms de fichiers. On les charge dans un tableau de Levels et on modifie `SinglePlayerGame` pour qu'il puisse gérer plusieurs niveaux. Il existe deux moyen de changer de niveau, des boutons ont été implémentés pour permettre au début de la partie de choisir son niveau. Une fois en jeu, il ne reste qu'un seul moyen d'accéder au niveau suivant, gagné le niveau en cours. A ce moment, le changement de niveau se fait toujours grâce à l'Observer présent dans la classe `SinglePlayerGame`, mais cette fois dans la méthode `levelWon()`. On peut voir à la [Figure 1](#) les différents niveaux ajoutés au jeu.

3.2.3 Sauvegarde de progression

Au début de la partie, le programme demande le nom de l'utilisateur. On vérifie ensuite s'il existe déjà un fichier de sauvegarde associé à l'utilisateur, sinon on le crée. A chaque niveau est associé un indice. On va stocker le plus grand indice des niveaux terminé par le joueur dans le fichier de sauvegarde. A sa prochaine partie, si l'utilisateur entre le même nom, et grâce aux boutons de changement de niveau de l'interface, il pourra choisir le niveau qu'il désire s'il l'a déjà débloqué.

Un design pattern Observer a été implémenté, par le biais de l'interface GameObserver, afin de permettre de notifier le joueur par des pop-up lorsque certains événements surviennent, tels que le fait de ne pas pouvoir démarrer un niveau (car le joueur ne l'a pas débloqué), de notifier le joueur de la fin de la partie et de sauvegarder sa progression.

4 IA pour fantômes - Gallois Florent

4.1 Programme initial et objectifs

Actuellement, les comportements des 4 fantômes du jeu Pac-man sont erratiques : leur trajectoire est déterminée aléatoirement.

Nous allons créer des IA pour les fantômes afin de leur donner des trajets plus intéressants pour le jeu. Pour cela, l'objectif est d'affecter à chaque fantôme 2 modes : un mode de poursuite pendant lequel il suivra une stratégie de poursuite et un mode de dispersion pendant lequel il suivra une trajectoire bien définie. Le mode poursuite ayant déjà été réalisé dans le logiciel, il ne nous reste à créer plus que le mode dispersion : Chaque fantôme doit régulièrement se rendre dans un coin : Pinky en haut à gauche, Blinky en haut à droite, Clyde en bas à gauche et enfin Inky en bas à droite. Cependant, nous ne devons pas l'implémenter n'importe comment, mais nous devons réaliser un strategy design pattern afin d'implémenter le comportement des fantômes. Un changement de stratégie permettra d'alterner entre la poursuite et la dispersion. Ces alternements devront se faire suivant les périodes de temps indiqués dans le polycopié : 7 sec de dispersion puis 20 sec de poursuite, 7 sec de dispersion ...

4.2 Démarche suivie

Le problème principal de cette réalisation est d'arriver à construire élégamment les IA des fantômes de tel sorte qu'un ajout de stratégie reste facile. Comme il nous l'est suggéré, nous nous sommes intéressées au strategy design pattern.

Ce pattern est très utile lorsqu'il faut permuter dynamiquement des algorithmes. Ou par exemple lorsque des classes ne diffèrent que par leur comportement. Ainsi, pour éviter de dupliquer le code, on crée une seule classe comportant les éléments de base ainsi qu'une interface implémentées par diverses classes décrivant les différents comportements de la classe centrale.

De prime abord, il a été pensé de créer pour chacun des fantômes, une interface qui était implémentée par 2 classes : une classe de poursuite et une classe de dispersion. Or cela engendrait toujours du code dupliqué : les algorithmes de dispersion sont tous les mêmes.

Pour la première version du logiciel, un pattern comme celui qui suit a été créé :

4 classes associées au mode poursuite pour chaque fantôme ont été créées dans lesquelles la méthode de poursuite a été copiée collée. Ces 4 classes héritaient de la classe abstraite `PoursuiteMode`. 4 autres classes ont vu le jour représentant le mode dispersion de chaque fantôme. Ces 4 classes héritaient de la classe abstraite `DispersionMode`. Et enfin, `PoursuiteMode` et `DispersionMode` implémentaient l'interface `Strategy`.

Pour la 2ème version du logiciel, l'algorithme de dispersion a été créé. Afin de pouvoir l'exécuter, des variables d'instance ont été aménagées dans la classe `Ghost`. Un `String` `atteintHome` indique si le fantôme a atteint sa case maison (la case du coin), un tableau de directions mémorise le chemin qu'il devra emprunter après avoir visité sa case maison. Un autre tableau de directions représente le chemin qu'il doit prendre à la prochaine occurrence et enfin un `square` représentant sa case "maison". Cet algorithme est placé dans la classe `DispersionMode`. Les classes filles font appel à cette même méthode.

La 3ème version, nous avons mis en place la variable `strategy` nous permettant de jongler entre la dispersion et la poursuite. Ainsi, un `String` est placé dans la classe `NPC` représentant la stratégie en cours : `ModeDispersion` ou `ModePoursuite`. Lorsque la stratégie change, au cours du temps, cette variable est modifiée grâce à `setStrategy`. Ainsi la méthode `nextMove()` de `blinky` par exemple appelle directement la méthode de sa classe de dispersion ou sa classe de poursuite associée en fonction de la valeur de cette variable. Par conséquent, si l'on souhaite modifier la stratégie d'un fantôme, nous n'avons plus à changer le code dans toutes les classes fantômes mais à rectifier les appels aux méthodes.

Enfin, pour la 4ème et dernière version du logiciel, nous avons indiqué au programme de permuter entre les 2 stratégies en fonction du temps. Dans la classe `NpcMoveTask`, une variable `temps` est créée pour suivre l'écoulement du temps. Celle-ci est incrémentée à chaque occurrence de la fonction `run` de la moyenne d'intervalle entre 2 coups d'un fantôme. Dans la fonction `run`, en fonction de la valeur de cette variable, nous appelons la méthode `setStrategy` de la classe `NPC` afin de switcher la stratégie des fantômes.

5 Difficultés liées au merge

Pour la fusion entre la fonctionnalité "Supergomme" et la fonctionnalité "série de labyrinthe", nous n'avons eu aucun conflit et le jeu était en parfait état de marche avec les deux fonctionnalités en même temps.

Pour la fusion avec la troisième fonctionnalité, nous avons eu quelques conflits notamment dans les classes "MapParser" ainsi que dans les classes des fantômes (`Blinky`, `Clyde`, `Inky` et `Pinky`) mais cela s'est assez vite réglé car nos codes étaient tout de même compatibles.

6 Analyse de la qualité du code

6.1 Outils utilisés

6.1.1 Google CodePro AnaliTiX

Le premier outil utilisé est Google CodePro AnaliTiX, sous sa forme plugin Eclipse.

Il s'agit d'un outil très complet qui permet le calcul des métriques, la détection de code dupliqué, l'analyse des dépendances, la couverture du code et des tests. Ici, nous l'utiliserons pour le calcul des métriques, la détection de code dupliqué et l'analyse des dépendances.

6.1.2 EclEmma

Le second outil est Emma, sous sa forme plugin Eclipse (EclEmma).

Emma est un outils utilisé pour vérifier la couverture du code ainsi que des tests lors de l'exécution de ces derniers.

6.1.3 CodeCity

Le troisième outil est CodeCity.

CodeCity est un outil qui permet de visualiser les métriques sous forme graphique. Nous l'utiliserons pour constater les différences entre le projet de base et le projet modifié.

6.1.4 PMD

Le dernier outils utilisé est PMD, sous sa forme plugin Eclipse.

PMD est un logiciel permettant de détecter les mauvaises pratiques de programmation ("Bad Smells"). Nous regarderons la variation de l'apparition de ces "bad smells" lors de la modification du projet.

6.2 Analyse des dépendances

6.2.1 Projet de base

On peut voir sur la [Figure 4](#) un graphe des dépendances entre packages du projet de base. On constate sur celui-ci qu'il y a beaucoup de dépendances cycliques et que certaines classes ont une interdépendance forte (par exemple ghost et level).

6.2.2 Projet modifié

Un graphe des dépendances est représenté à la [Figure 5](#), on peut constater qu'en général les dépendances ont augmenté et que nous avons aussi ajoutés des interdépendance entre deux packages qui n'en possédaient pas.

6.3 Code dupliqué

Selon Google CodePro AnalitiX, il y a 54 lignes de code dupliquées dans le projet de base. Cette valeur est assez basse et montre la volonté des développeurs du projet d'éviter la programmation "copier-coller"

Google CodePro AnalitiX, pointe toujours 54 lignes de code dupliquées, nous n'avons donc pas altéré ce point précis du code lors de nos modifications.

6.4 Test & Code Coverage

6.4.1 Projet de base

EclEmma nous donne comme valeurs :

- test coverage : 80.9%
- code coverage (exécution sans jouer) : 41%
- code coverage (exécution partie type) : ~60%

6.4.2 Projet modifié

EclEmma nous donne comme valeurs :

- test coverage : 71%
- code coverage (exécution sans jouer) : 35.9%
- code coverage (exécution partie type) : ~60%

On peut constater que de manière générale le coverage a diminué, surtout au niveau du test coverage. Bien qu'un paradigme de programmation défensive à été utilisé, nous n'avons pas réussi à trouver de tests qui couvrent l'entièreté du code ajouté.

6.5 Métriques

6.5.1 Tableaux de métriques

La [Figure 6](#) montre les deux tableaux de métriques côte à côte, ce qui permet de comparer facilement les résultats.

On peut noter, entre autres, un légère augmentation de la complexité cyclomatique, un accroissement général de la taille des différentes classes (nombre de méthodes et nombre de lignes) et une légère diminution du ratio de commentaires.

6.5.2 CodeCity : méthodes par classes

Codecity nous permet de représenter graphiquement certaines métriques par classes sur l'ensemble du projet.

Ici([Figure 7](#)) nous nous intéressons au nombre de méthodes par classe, on voit que, de manière générale, celui augmente pour l'ensemble du projet.

6.5.3 CodeCity : complexité cyclomatique

On voit sur la [Figure 8](#) que la complexité cyclomatique reste sensiblement la même.

6.5.4 CodeCity : nombre de lignes de code

Sur la [Figure 9](#) on peut voir que le nombre de lignes de codes par classe est resté sensiblement identique, sauf pour deux d'entres elles (Launcher et Level), qui mériteraient d'être séparée en plus petites classes pour éviter le phénomène de "God Class".

6.6 Bad Smells

PMD a mis en évidence 827 violations des bonnes pratiques de programmation dans le projet initial. En ce qui concerne le projet final, on monte à 1442 violations. Les erreurs les plus fréquentes sont “variable or argument could be final” (39.6% des violations pour le projet initial contre 33% pour le projet modifié) et “law of demeter” (10% pour le projet initial contre 12% pour le projet modifié) et “short variable” (10% pour le projet initial contre 11% pour le projet modifié).

On a donc un accroissement de 74% de violations selon PMD, à savoir que le code est lui passé 2415 à 3825 lignes de code, c’est à dire un accroissement de 58% en ce qui concerne la taille du projet niveau lignes de codes (lignes de code effectives, selon Google CodePro AnalitiX). A noter aussi que PMD nous donne deux avertissements pour “God Class” pour les classes Level et Launcher, qui ont beaucoup crû pendant nos modifications respectives. Il semble qu’un refactoring soit nécessaire à ce niveau là pour pouvoir poursuivre la maintenance logicielle.

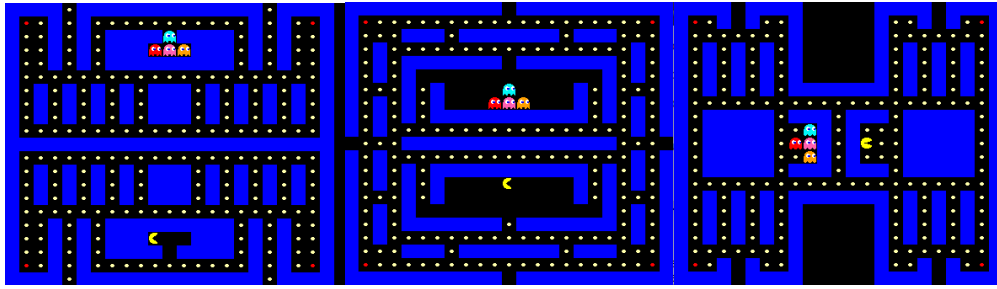


FIGURE 1 – Différents labyrinthes disponibles

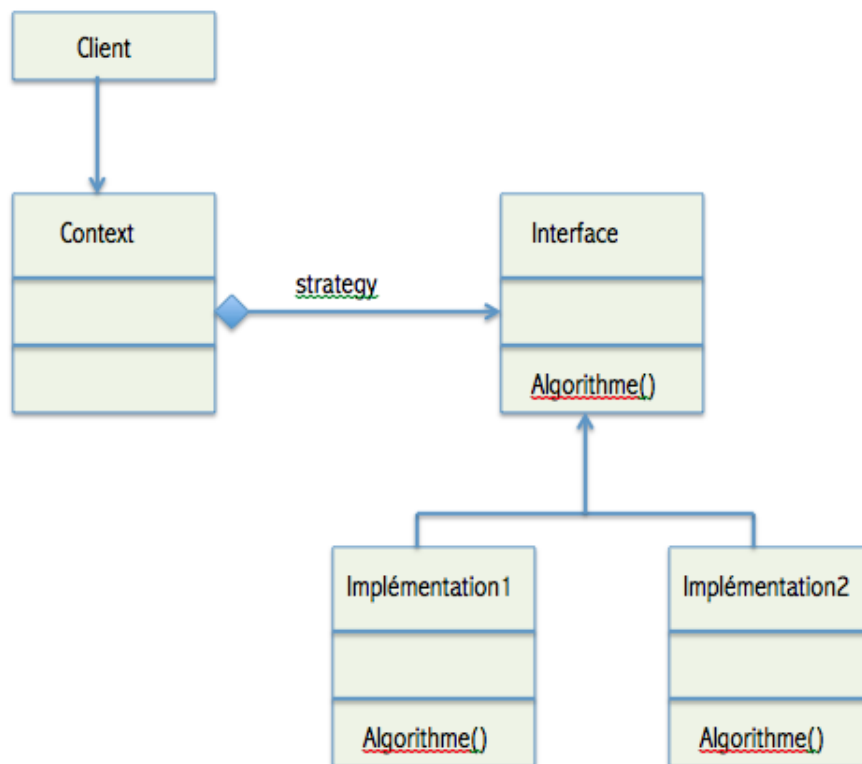


FIGURE 2 – Design pattern strategy

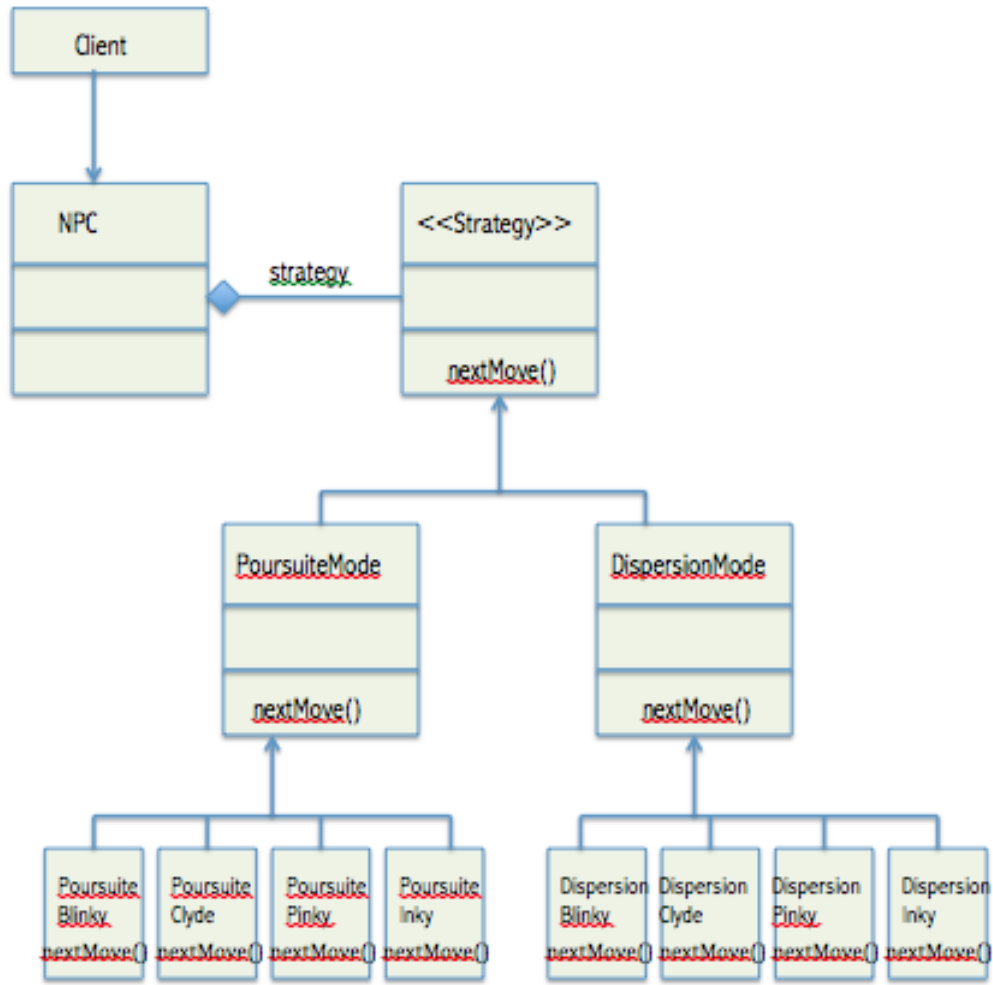


FIGURE 3 – Design pattern strategy appliqué

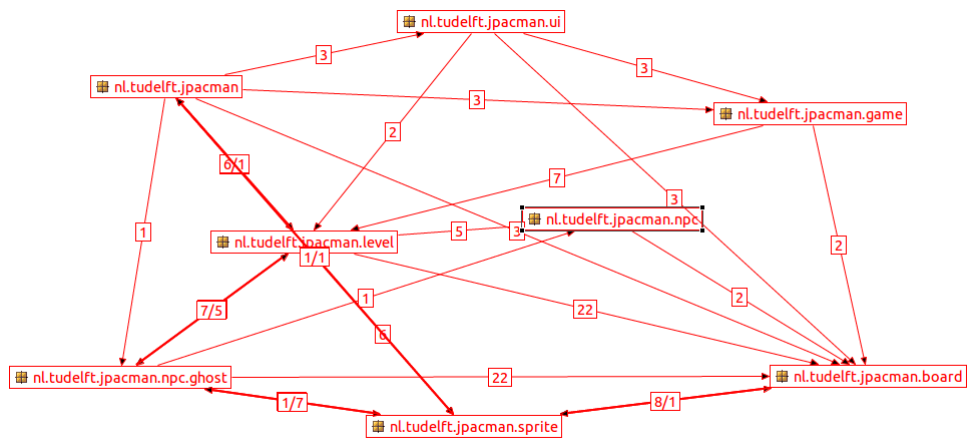


FIGURE 4 – Dépendances du projet de base

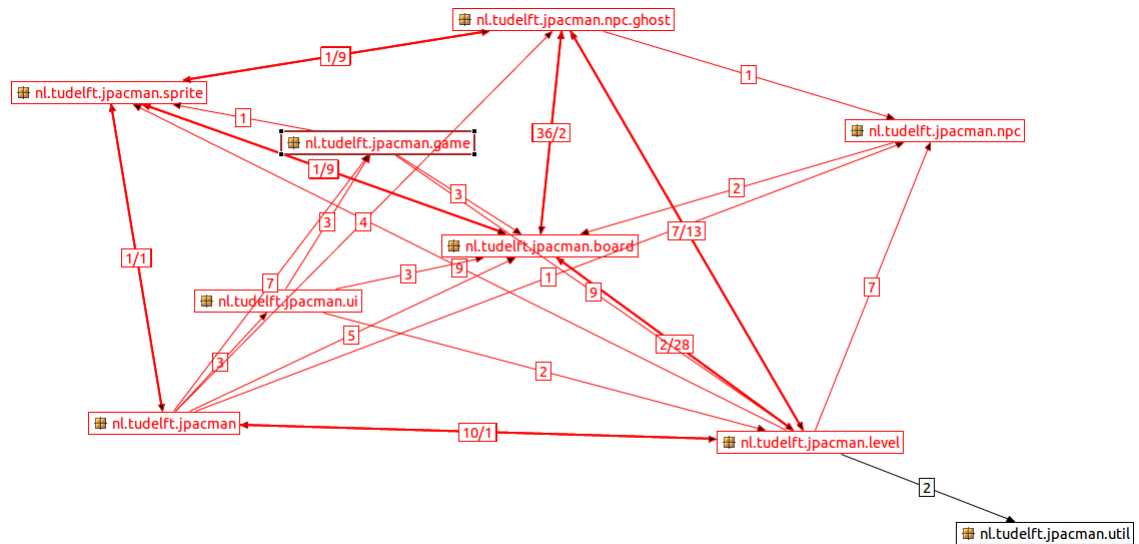


FIGURE 5 – Dépendances du projet modifié

Metric	Value	Metric	Value
⊕ Abstractness	15.2%	⊕ Abstractness	15.6%
⊕ Average Block Depth	0.98	⊕ Average Block Depth	1.01
⊕ Average Cyclomatic Complexity	1.42	⊕ Average Cyclomatic Complexity	1.52
⊕ Average Lines Of Code Per Method	6.30	⊕ Average Lines Of Code Per Method	6.44
⊕ Average Number of Constructors Per Type	0.59	⊕ Average Number of Constructors Per Type	0.68
⊕ Average Number of Fields Per Type	1.36	⊕ Average Number of Fields Per Type	1.43
⊕ Average Number of Methods Per Type	3.50	⊕ Average Number of Methods Per Type	3.86
⊕ Average Number of Parameters	0.70	⊕ Average Number of Parameters	0.58
⊕ Comments Ratio	17.8%	⊕ Comments Ratio	15%
⊕ Efferent Couplings	45	⊕ Efferent Couplings	57
⊕ Lines of Code	2,415	⊕ Lines of Code	3,825
⊕ Number of Characters	129,605	⊕ Number of Characters	185,502
⊕ Number of Comments	431	⊕ Number of Comments	577
⊕ Number of Constructors	43	⊕ Number of Constructors	70
⊕ Number of Fields	135	⊕ Number of Fields	191
⊕ Number of Lines	5,337	⊕ Number of Lines	7,644
⊕ Number of Methods	252	⊕ Number of Methods	394
⊕ Number of Packages	20	⊕ Number of Packages	22
⊕ Number of Semicolons	1,278	⊕ Number of Semicolons	2,025
⊕ Number of Types	72	⊕ Number of Types	102
⊕ Weighted Methods	420	⊕ Weighted Methods	706

FIGURE 6 – Comparaison des tableaux de métriques (initial à gauche)

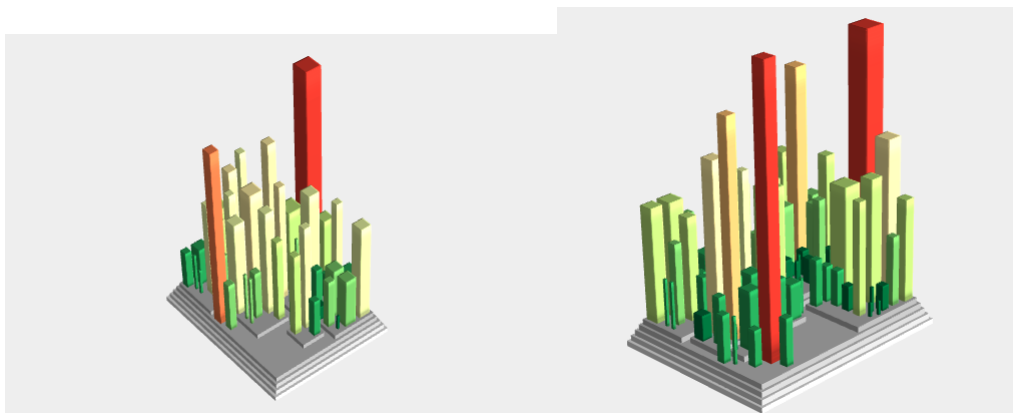


FIGURE 7 – Comparaison du nombre de méthodes par classe (initial à gauche)

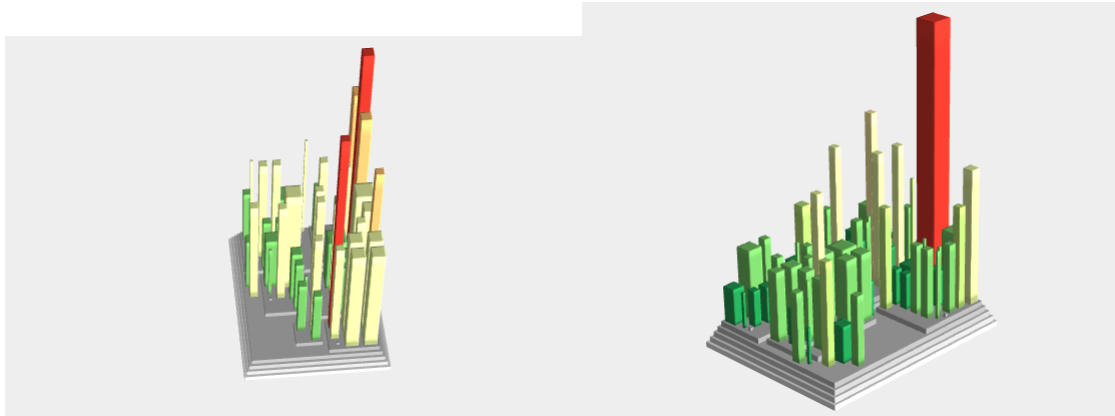


FIGURE 8 – Comparaison de la complexité cyclomatique par classe (initial à gauche)



FIGURE 9 – Comparaison du nombre de lignes de code par classe (initial à gauche)