UMONS
Université de Mons

Faculté des Sciences

Rapport du projet Pac-Man

Projet réalisé dans le cadre
de la 1ère Master en Sciences Informatiques
pour le cours de « Software Evolution »

Réalisé par

CAMBIER	Robin	ROBIN.CAMBIERR@student.umons.ac.be
OPSOMMER	Sophie	SOPHIE.OPSOMMER@student.umons.ac.be

Faculté
des Sciences

Sous la direction de: *Titulaire* : T. MENS
Assistant : M. CLAES

Année Académique 2014-2015

ACADÉMIE
UNIVERSITAIRE
WALLONIE-
BRUXELLES

Résumé

Ce *rapport* est rendu dans le cadre du cursus de première année de « Master en Sciences Informatiques »pour le cours de *Software Evolution* (dont le titulaire est Mr. *T. Mens* et l'assistant est Mr. *M. Claes* en année académique 2014-2015) . Le but de ce rapport est de présenter les résultats de l'amélioration du projet Pac-Man.

Table des matières

1 Introduction

1.1 Problème posé

A partir d'un code existant, ce projet consiste à :

- analyser la qualité du logiciel, en utilisant des techniques d'analyse statique du code (par exemple, la détection du code dupliqué et des bad smells, les diverses métriques de qualité) et les outils d'analyses dynamiques du code (par exemple, le profilage, la couverture du code et des tests) ;
- améliorer la qualité et la structure du code (en utilisant des refactorings, en introduisant des design patterns, et en modularisant le code) ;
- étendre le logiciel avec de nouvelles fonctionnalités (évolution), et étudier l'effet de cela sur la qualité du code ;
- tester le logiciel avant et après chaque modification. Ceci implique que vous devez ajouter des tests unitaires (unit tests) pour au moins les fragments du code modifiés ou ajoutés, et d'appliquer des tests de régression à chaque modification.

1.2 Etapes clés

Les étapes clés du projet sont les suivantes : (chronologiquement)

1. Analyse de la qualité de la première version du code (section 2 -Etape 1 : Première analyse de la qualité du logiciel- (page 6))
2. Ajout de tests unitaires à la première version du code (section 3 -Etape 2 : Ajout de tests unitaires- (page 30)), et vérification de la couverture des tests
3. Refactoring du code pour en améliorer la qualité et la structure (section 4 -Etape 3 : Refactoring en vue d'améliorer la qualité
   Etape 4 : Analyse de la qualité du logiciel- (page 31))
4. Analyse des améliorations de qualité et tests de régression (section 4 -Etape 3 : Refactoring en vue d'améliorer la qualité
   Etape 4 : Analyse de la qualité du logiciel- (page 31))
5. Extension du logiciel et ajout des tests unitaires pour cette extension (section 5 -Etape 5 : Extensions- (page 44))
6. Analyse de la qualité de cette extension et tests de régression (section 6 -Etape 6 : Analyse de la qualité du logiciel- (page 49))
7. Etude de l'historique de la qualité logicielle entre toutes les versions du code (section ?? -??- (page ??))

1.3 Le jeu Pacman

Pac-Man est un jeu vidéo créé en 1980 [1]. Le joueur dirige un personnage jaune en forme de camembert appelé Pac-Man. Ce jeu vidéo a connu un grand succès en salle d'arcade, et de nombreux clones et variantes du jeu ont été réalisés sur diverses plateformes, y compris sur PC. Le but du jeu est de terminer une série de niveaux. Chaque niveau est constitué d'un labyrinthe dans lequel Pac-Man se promène. Le labyrinthe est également peuplé de fantômes qui, la plupart du temps, tentent de toucher (et ainsi de tuer) Pac-Man. Celui-ci doit éviter les fantômes et manger toutes les gommes se trouvant sur son chemin. Un niveau est terminé dès que toutes les gommes du labyrinthe ont été mangées.
Dans chaque labyrinthe se trouvent quatre fantôomes, chacun ayant une couleur et un nom uniques. Il s'agit de :

- Blinky, le fantôme rouge ;
- Pinky, le fantôme rose ;
- Inky, le fantôme bleu ;
- Clyde, le fantôme orange.

1. `http://fr.wikipedia.org/wiki/Pac-Man`

2 Etape 1 : Première analyse de la qualité du logiciel

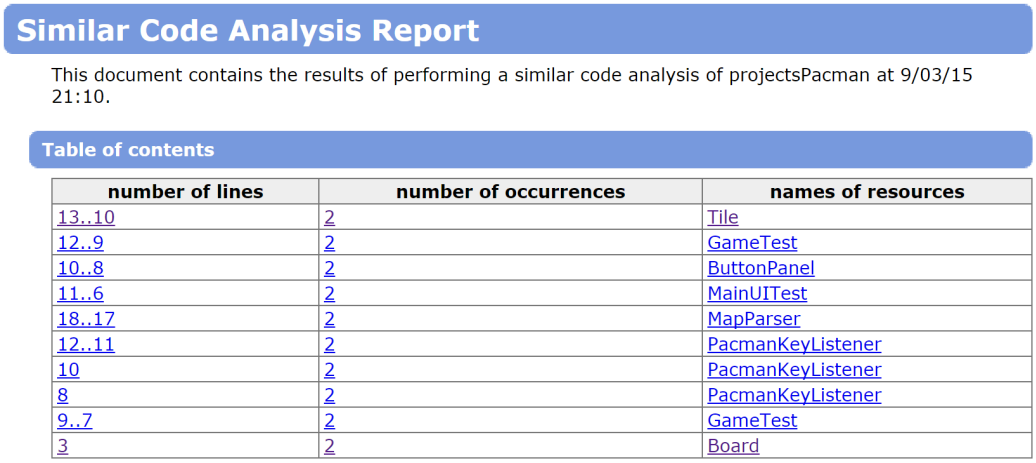
Avant toute modification, il convient d'analyser l'état de la qualité du logiciel afin de se rendre compte des améliorations à effectuer, des corrections à appliquer si des mauvaises pratiques sont observées. Cette analyse se fera à l'aide d'outils d'analyse de code tel que les IDE [2] Eclipse [3] et IntelijIdea [4] et les programmes CodePro [5], InCode [6] et VisualVM [7].

Cette section regroupe donc les unes après les autres toutes les analyses qui ont pu être réalisées à travers les différents outils sur le projet JPacman. Elles ont généralement été regroupées par type de remarques mais parfois aussi par outil.

2.1 Code dupliqué

Du code dupliqué consiste à trouver au sein d'un projet des blocs de lignes de code identique en plusieurs exemplaires. C'est un facteur de mauvaise qualité parce que ça rend le code plus difficile à changer, à maintenir, à comprendre,...
Les solutions qui sont offertes par les langages de programmation sont les méthodes, les fonctions, les librairies, l'encapsulation des objets. Eviter de dupliquer du code permet d'avoir un programme plus cohésif.
Pour analyser cette métrique, nous avons utilisé le programme CodePro à partir de l'interface d'Eclipse (Eclipse -> CodePro Tools -> Find Similar Code).
Nous pouvons observer sur la figure 1 (page 7) que cet outil a détécté 10 blocs de code. Les figures de l'annexe A (page 52) permettent de visualiser les différents blocs de code. Les classes concernées sont : Tile.java, GameTest.java, ButtonPanel.java, MainUITest.java, MapParser.java, PacmanKeyListener.java, PacmanKeyListener.java, PacmanKeyListener.java, GameTest.java, Board.java.

2. IDE : Integrated Development Environment (Environnement de développement).
3. Eclipse : `https://eclipse.org/` version : Eclipse Luna SR2 (4.4.2).
4. InteligIdea : `https://www.jetbrains.com/idea` version : Community Edition 14.0.3
5. CodePro : `https://marketplace.eclipse.org/content/codepro-analytix` version : CodePro Analytix 7.1.0.r37
6. InCode : `https://marketplace.eclipse.org/content/incode-helium` version 2.0.1
7. VisualVM : `http://visualvm.java.net/` version 1.3.8



Similar Code Analysis Report

This document contains the results of performing a similar code analysis of projectsPacman at 9/03/15 21:10.

Table of contents

number of lines	number of occurrences	names of resources
13..10	2	Tile
12..9	2	GameTest
10..8	2	ButtonPanel
11..6	2	MainUITest
18..17	2	MapParser
12..11	2	PacmanKeyListener
10	2	PacmanKeyListener
8	2	PacmanKeyListener
9..7	2	GameTest
3	2	Board

FIGURE 1 – Résultat de l'analyse de "code dupliqué" par CodePro

Ce résultat n'est pas bon, mais on peut observer que les blocs se trouvent chaque fois dans une même classe. Il sera donc probablement possible de créer des fonctions pour chacun de ces cas.

2.2 Dépendances cycliques

Une dépendance cyclique peut-être appelée dépendance cyclique directe ou indirecte et elle a la même définition qu'il s'agisse de dépendances entre des projets, entre des packages ou entre des classes. Quand on a une dépendance directe, on a un élément X qui dépend d'un élément Y qui dépend lui-même de X. Contrairement à la dépendance cyclique indirecte où la situation dans laquelle on se trouve est tel que X dépend de Y, Y dépend de Z et Z dépend de X.
Du point de vue de la compilation, plus une dépendance est à haut niveau, plus elle est à traiter en priorité. En effet entre deux classes, ce n'est pas très grave et ça ne pose généralement pas de problème. Entre deux package c'est fortement déconseillé même s'il est généralement possible de compiler le projet. Par contre entre deux projets, l'issue est fatale puisque chaque projet doit-être compilé avant de pouvoir compiler l'autre.
Du point de vue de la maintenance, une dépendance d'un élément A à un élément B et vice-versa impose que pour pouvoir modifier A, il faut commencer par modifier B et pour pouvoir modifier B, il faut commencer par retravailler A. L'évolution de ces éléments est donc compliquée.
Pour pallier à ce genre de problème, plusieurs pistes sont possible : déplacer les éléments (les classes si le problème concerne deux packages ou la(les)

méthode(s) si le soucis se situe entre deux classes), redécouper certains éléments (pour mieux associer les blocs de code aux éléments qui en ont besoin), regrouper les éléments (pour n'en former plus qu'un seul),... Cet métrique

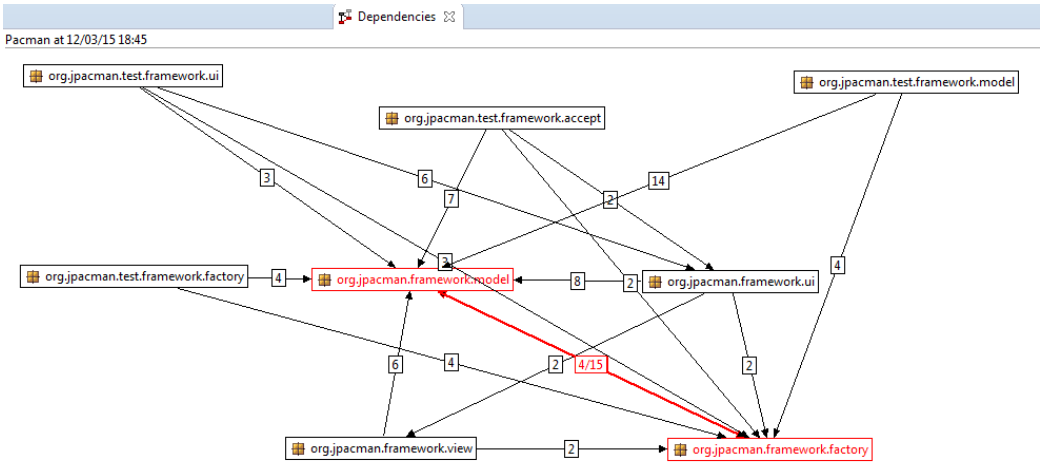


FIGURE 2 – Détail de l'analyse des dépendences cycliques des packages du projet.

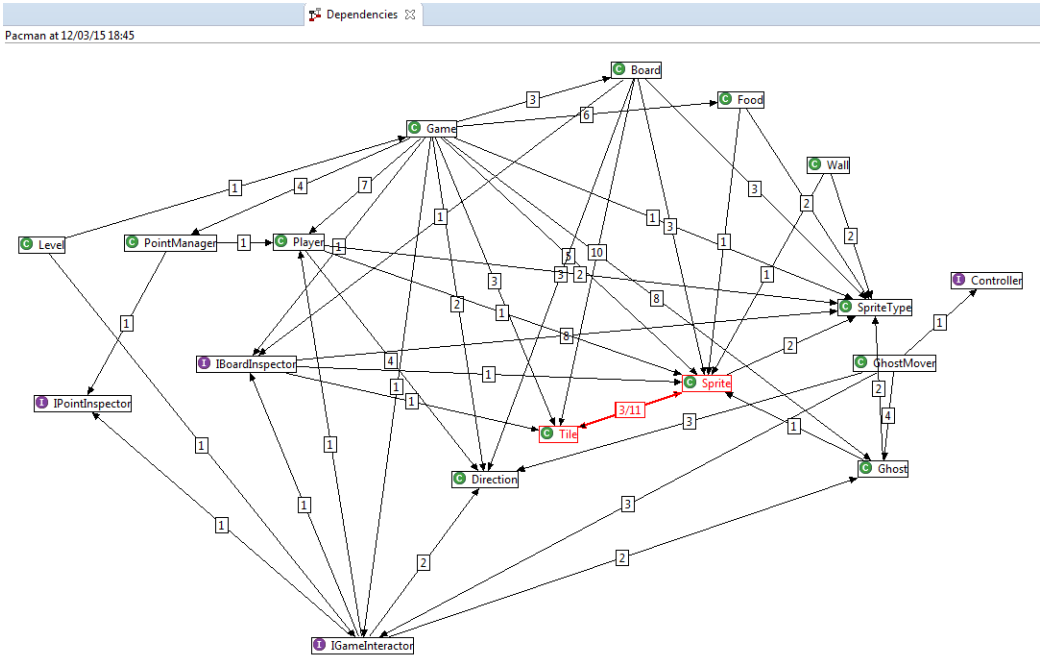


FIGURE 3 – Détail de l'analyse des dépendences cycliques du package Model.

a aussi été visualisée à partir de l'outil CodePro depuis Eclipse (Eclipse -> CodePro Tools -> Analyse Dependencies). On peut observer sur la figure 2 (page 8) et la figure 3 (page 8) qu'il existe des dépendances cycliques au sein du package "model" (entre les classes Sprite et Tile) et entre le package "model" et le package "factory".
L'annexe B (page 57) contient toutes les autres visualisations qui n'ont pas révélé de problème de dépendances.

2.3 Code inutile

Du code inutile, aussi appelé "Dead Code", correspond à des lignes de code qui sont compilées mais qui ne sont jamais utilisées.
C'est fréquemment du code qui a été utile à une fonctionnalité et lorsque cette fonctionnalité a été supprimée, réécrite, déplacée,... ce code est resté. Le problème dans ce cas est que ça ralentit la compréhension du développeur lors de la lecture, ça gaspille des ressources au compilateur et lors de l'exécution.
La solution est généralement de supprimer ces lignes de code. L'outil utilisé

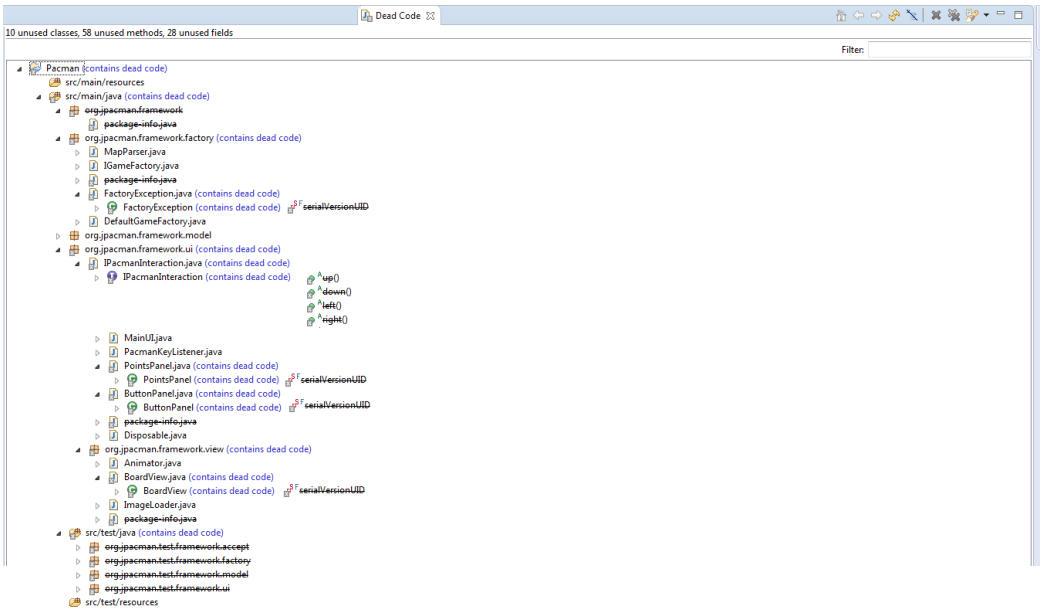


FIGURE 4 – Détail de l'analyse des parties de code non utilisé lors de l'éxécution du programme.

reste CodePro depuis Eclipse (Eclipse -> CodePro Tools -> Find dead code). Attention tout de même à ne pas tout supprimer sans réfléchir, en effet, on observe sur la figure 4 (page 9) que les packages contenant les tests sont considérés comme inutile. Ils le sont en effet lors de l'exécution du programme, mais ne le sont pas au bon développement du programme. Il en va de même pour les variables "serialVersionUID". Ces variables, bien que inutile lors de l'exécution, doivent être présentes dans les classes qui étendent (directement ou indirectement) la classe "Sérializable". Leur valeur est indispensable pour des applications qui transitent par le réseau mais leur existence ne peut causer aucun tort. Pour ce qui est des fichiers "package-info.java", ils sont à nouveau inutiles lors de l'exécution du code, mais permettent de contenir les commentaires contenant l'information relative au package en vue de la création de la javadoc. Ces fichiers sont donc à conserver (et à complêter dans certains cas).
L'analyse a aussi été effectuée par l'outil Code Inspector de IntelijIdea et donne d'autres résultats complémentaires. Ceux-ci font référence à des méthodes complètes qui ne sont jamais appelées.
La solution, la plus rapide, est simplement de supprimer ces méthodes. Seulement si lors d'une future amélioration, on se rend compte qu'elles auraient pu être nécessaire, on doit recommencer le travail. Une autre solution pourrait être de mettre ces fonctions en commentaire afin de ne pas devoir réécrire ces fonctions.
Les modifications à éffectuer, bien que nombreuses, sont donc mineures.

2.4 Javadoc

La javadoc est une documentation standard au format HTML pour les programmes développés en JAVA. Elle est créée de façon automatique par la plupart des outils de développement en se basant sur les tags placés dans le code au dessus de la déclaration de chaque classe et de chaque méthode (pour la documentation des packages, elle se trouve dans les fichiers "paquage-info.java"). L'utiliser est un plus mais ne consiste en rien en une obligation (mais alors il sera tout de même fortement conseillé d'utiliser les commentaires classiques pour constituer un code suffisamment documenté à la compréhension).
Grâce à l'outil CodePro d'Eclipse, il a été observé (non illustré parce que toutes les classes sont à revoir) que en règle générale, les classes sont documentées. L'outil détecte bien quelques manquement, mais ce sont généralement l'un ou l'autre tag qu'il détecte manquant mais qui ne perturbe pas la compréhension ainsi que les classes DefaultGameFactory, Game, IBoardInspector, IPointInspector, Player et PointManager dont les méthodes ne

sont pas commentées et les méthodes issues d'une interface (sous l'annotation "@Overhide").

2.5 Test unitaire

Les tests unitaires permettent de vérifier le bon fonctionnement du programme en testant les lignes de codes du logiciel. C'est pourquoi, que dans cette section,une analyse est réalisée sur les différents tests unitaires en fesant un test de couverture.
Tout d'abord, lorsque les tests sont lancés (avec Intellij IDEA), tous les tests sont réussis sauf trois qui sont ignorés.
Ensuite, grâce à l'outil de couverture de test dans Intellij IDEA, il est possible de voir les méthodes testées et celles qui ne le sont pas. Les tableaux qui suivent montrent le résultat de la couverture de test.
Comme le montre le tableau 1, l'analyse de couverture des tests montre que 96% du code est couvert. Ce qui correspond à 90% des méthodes et 88% de lignes.
Des tests devrons être ajoutés pour couvrir et tester les méthodes dans la classe "MapParser". Ceux-ci permettent de voir si les exceptions sont bien lancées lorsqu'une map est mal encodée.
Après, l'analyse montre que les méthodes withFactory et withButtonPanel dans la classe MainUI ne sont pas testées.

De plus, les évènements du clavier, se trouvant dans la classe "PacmanKeyListener", ne sont pas testés et devront l'être. Tout comme pour les boutons "play", "stop" et "exit" se trouvant dans la classe ButtonPanel.

Package	Classe,%	Methode,%	Ligne,%
Toutes les classes	96,6% (28/29)	90% (198/220)	88,3% (704/797)

TABLE 1 – Résumé de l'analyse de couverture

Puis, le tableau 2 montre plus précisément la répartition de couverture entre les différents packages. Le tableau révèle que toutes les classes sont testées sauf une qui est la classe FactoryException se trouvant dans le package factory. De plus, il souligne que la plus part des méthodes sont testées mais toutes ne le sont pas.

Package	Classe,%	Methode,%	Ligne,%
org.jpacman.framework.factory	66,7% (2/3)	84,2% (16/19)	75% (69/92)
org.jpacman.framework.model	100% (13/13)	93,5% (87/93)	96,1% (273/ 284))
org.jpacman.framework.ui	100% (8/8)	82,9% (63/76)	81,3% (226/278)
org.jpacman.framework.view	100% (5/5)	100% (32/32)	95,1% (136/143)

TABLE 2 – Répartition de couverture des tests

2.6 Flux de conception

A l'aide de l'outil InCode intégré à Eclipse, on peut entre-autre observer la conception du programme sur la figure 5 (page 13) dont la légende se trouve à l'annexe 48 (page 61) La figure 5 (page 13) illustre donc que il n'y a pas de gros problèmes dans l'application(point de vue structure) cependant 2 classes ont tout de même un comportement inadéquat : Game et PacmanKeyListener. Ils sont répertoriées comme des classes schizophréniques.
Ces classes ont la particularité d'être utilisées par des groupes disjoints de classes de clients utilisant des fragments disjoints de la classe.
Plusieurs solutions sont possible pour résoudre ce genre de problème :

- Regrouper les éléments qui sont utilisés par des groupes disjoints de clients et en faire 2 classes distinctes.
- Revoir l'accessibilité des éléments qui la contiennent.
- Regardez l'aperçu de la vue "couplage" pour identifier toutes les dépendances basées sur les appels entre la classe courante et les classes externes.

2.7 Complexité

A l'aide de l'outil InCode intégré à Eclipse, on peut entre-autre observer la complexité de l'application sur la figure 6 (page 14) dont le détail de la légende se trouve à l'annexe 49 (page 62) Cette figure met en avant la classe PacmanKeyListener qui encourt la plus forte complexité et met un attention sur les classes BoardView et MapParser (à cause de leur grand nombre d'attributs).

2.8 Encapsulation

A l'aide de l'outil InCode intégré à Eclipse, on peut entre-autre observer la complexité de l'application sur la figure 7 (page 15) dont le détail de la légende se trouve à l'annexe 50 (page 63) Cette figure contient beaucoup d'informations, nous en retiendrons quelques unes :

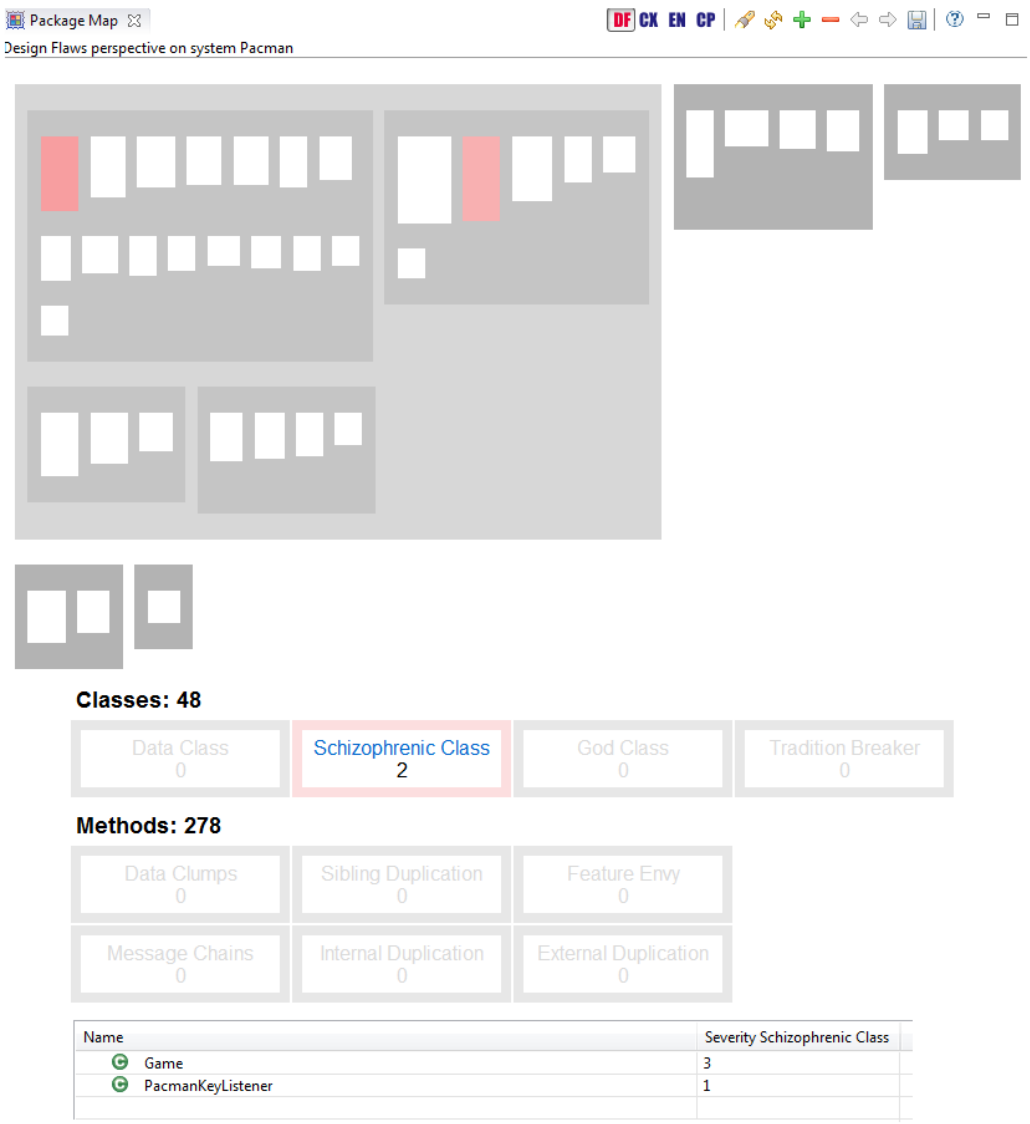


FIGURE 5 – Analyse de la conception du programme (sa structure) par In-Code.

- La classe "MainUI" a beaucoup de clients (c'est-à-dire beaucoup d'autres classes qui accèdent aux données publiques de cette classes) dont les principaux sont "PacmanKeyListener" et "IGameInteractor". On ne sait rien concernant le nombre de fournisseurs (c'est-à-dire beaucoup de données publiques d'autres classes auxquelles celle-ci accède) sauf qu'il est inférieur au nombre de clients.
- La classe "Player" n'a aucun fournisseur et a beaucoup de clients dont

FIGURE 6 – Analyse de la complexité par InCode.

les principaux sont "Game" et "BoardView".
- La classe "Sprite" n'a aucun fournisseur et a beaucoup de clients dont les principaux sont "Game", "Board" et "Tile".
- Les classes telles que "Ghost", "Controller", "Wall", "ImageLoader", "Animator", "MapParser", "IGameFactory", "FactoryException", "IPacmanInteraction" et "Disposable" n'ont ni clients ni fournisseurs.

2.9 Couplage

A l'aide de l'outil InCode intégré à Eclipse, on peut entre-autre observer la complexité de l'application sur la figure 8 (page 16) dont le détail de la légende se trouve à l'annexe 51 (page 64) Cette figure 8 (page 16) contient aussi beaucoup d'informations, nous en retiendrons quelques unes :
- La classe "MainUI" a beaucoup de fournisseurs[8] (c'est-à-dire beaucoup de méthodes publiques d'autres classes auxquelles celle-ci accède) dont les principaux sont "GhostMover", "IGameInteractor", "Level", "BoardView", "Animator", "PacmanKeyListener", "ButtonPanel" et "PointsPanel". On ne sait rien concernant le nombre de clients (c'est-

8. >provider

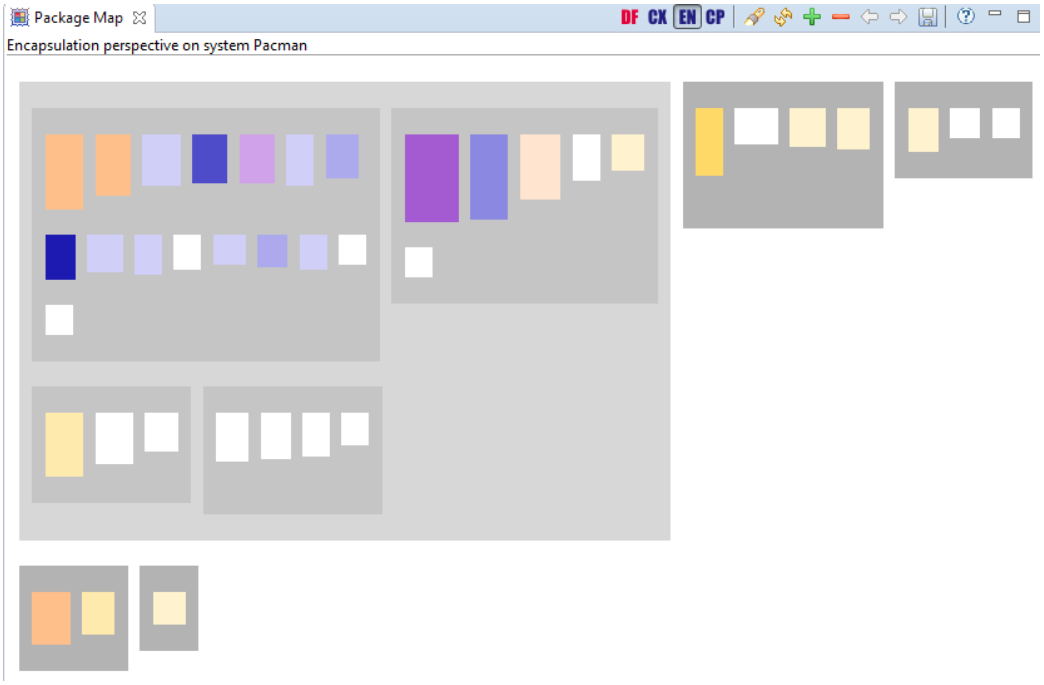


FIGURE 7 – Analyse de la complexité par InCode.

à-dire beaucoup d'autres classes qui accèdent aux méthodes publiques de cette classes) sauf qu'il est inférieur au nombre de clients.
- La classe "Tile" n'a aucun fournisseur et beaucoup de clients dont les principaux sont "Game", "Board" et "Sprite".
- La classe "Board" a beaucoup de clients dont les principaux sont "Game", "MapParser" et "DefaultGameFactory". On ne sait rien concernant le nombre de fournisseurs sauf qu'il est inférieur au nombre de clients et que les principaux sont "Sprite" et "Tile".

2.10 Pyramide des métriques

A l'aide de l'outil InCode intégré à Eclipse, on peut entre-autre observer les valeurs de l'application sur la figure 9 (page 16) dont le détail de la légende se trouve à l'annexe C (page 65). Comme le précise l'interprétation de la figure 9 (page 16), l'arbre que constitue les classes est grand et étroit. Les classes ont tendance à contenir un nombre moyen de méthodes et à être organisées avec quelques classes par paquet.
Les méthodes tendent à être longue et avec une logique assez simple et à appeler de nombreuses méthodes (à forte intensité de couplage) de quelques autres classes (à faible dispersion de couplage).

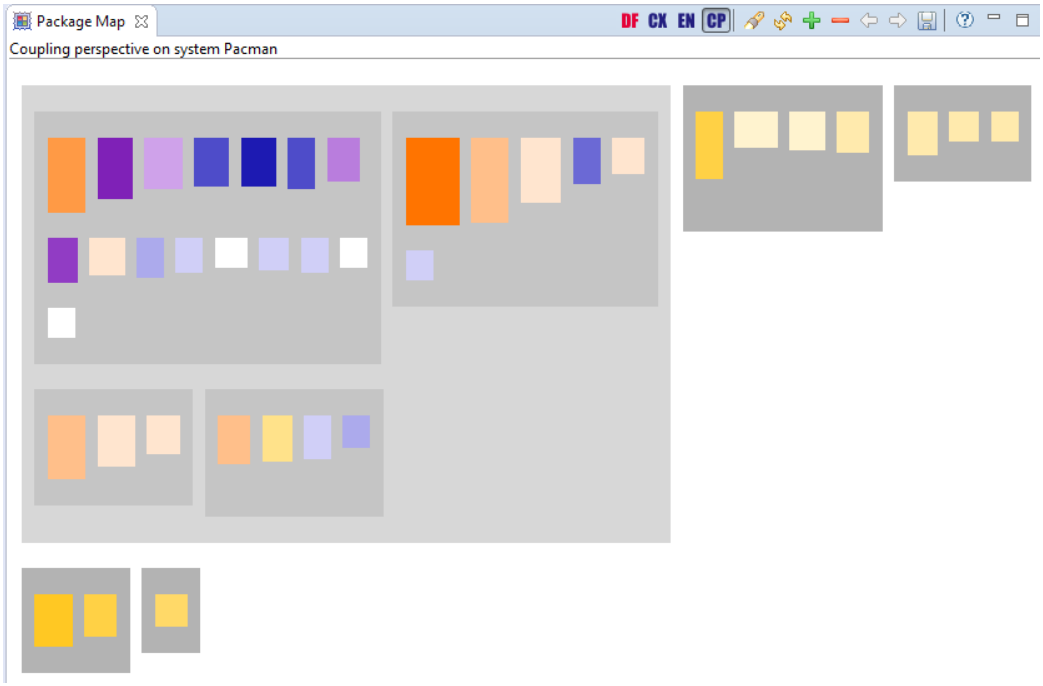


FIGURE 8 – Analyse de la complexité par InCode.

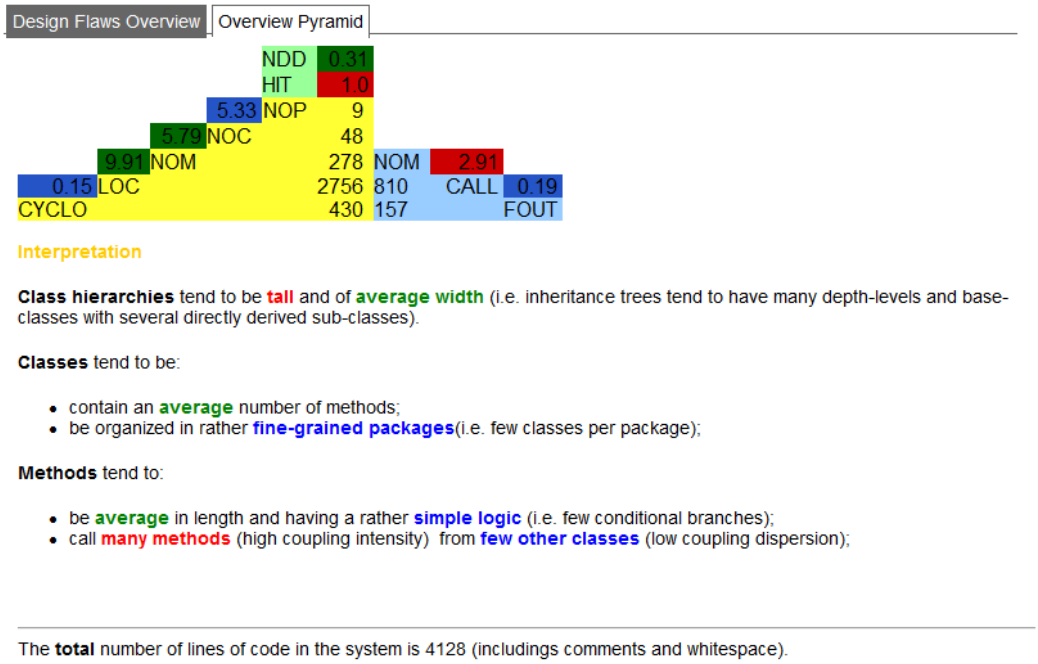


FIGURE 9 – Pyramide des valeurs calculées par InCode.

2.11 Métriques

Grâce à l'outil de calcul des métriques d'Eclipse, on peut observer les résulats présent à la figure 10 (page 18), la figure 11 (page 19) et la figure 12 (page 20).

2.11.1 Complexité cyclique moyenne

Il s'agit de la moyenne de la complexité cyclomatique de chacune des méthodes. La complexité cyclomatique d'une méthode unique est une mesure du nombre de chemins distincts de l'exécution dans le procédé. Elle est mesurée par l'ajout d'une voie pour la méthode avec chacun des chemins créés par des instructions conditionnelles (telles que "if" et "for") et les opérateurs (tels que " ? :"). Pour chacun des cas illustrés dans la figure 13 (page 21) et la figure 14 (page 22), il sera nécessaire lors du refactoring (voir section 4 -Etape 3 : Refactoring en vue d'améliorer la qualité
Etape 4 : Analyse de la qualité du logiciel- (page 31)) d'analyser et de voir s'il est possible de la diminuer.

2.11.2 Nombre moyen de méthodes par type

C'est la moyenne du nombre de méthodes définies pour chaque classe. On remarque que certaines classes ont trop de méthodes, en particulier les classes "Game", "ButtonPanel", "MainUI", "PacmanKeyListener" et "BoardView". Lors d'une analyse précédente (voir section 5 il avait été observé que les classes "Game" et "PacmanKeyListener" était schizophréniques. Le problème se résolvera donc probablement naturellement lors du traitement de ce problème. Pour ce qui est des autres classes, il sera nécéssaire lors du refactoring d'étudier l'utilité de chacune des méthodes et d'aviser au cas par cas le travail à effectuer sur chacune.

Metrics

Pacman at 9/03/15 21:03

Metric	Value
⊞ Abstractness	14.5%
⊞ Average Block Depth	0.84
⊟ Average Cyclomatic Complexity	1.53
⊞ org.jpacman.framework.factory	1.69
⊞ org.jpacman.framework.model	1.26
⊟ org.jpacman.framework.ui	2.08
⊞ ButtonPanel.java	1.05
Disposable.java	1.00
IPacmanInteraction.java	1.00
MainUI.java	1.08
⊟ PacmanKeyListener.java	4.25
MatchState	1.00
PacmanKeyListener	4.54
PointsPanel.java	1.00
⊟ org.jpacman.framework.view	1.85
⊞ Animator.java	1.00
BoardView.java	2.21
ImageLoader.java	1.77
⊞ org.jpacman.test.framework.accept	1.00
⊞ org.jpacman.test.framework.factory	1.00
⊞ org.jpacman.test.framework.model	1.00
⊞ org.jpacman.test.framework.ui	1.25
⊞ Average Lines Of Code Per Method	6.14
⊞ Average Number of Constructors Per Type	0.35
⊞ Average Number of Fields Per Type	2.00

FIGURE 10 – Détails de l'analyse des métriques du projet (partie 1).

⊟ Average Number of Methods Per Type	5.43
⊞ org.jpacman.framework.factory	5.00
⊟ org.jpacman.framework.model	5.41
Board.java	12.00
Controller.java	3.00
Direction.java	0.00
Food.java	1.00
Game.java	18.00
Ghost.java	1.00
GhostMover.java	8.00
⊞ IBoardInspector.java	2.50
IGameInteractor.java	9.00
IPointInspector.java	3.00
Level.java	4.00
Player.java	8.00
PointManager.java	6.00
Sprite.java	6.00
Tile.java	7.00
Wall.java	1.00
⊟ org.jpacman.framework.ui	7.20
⊟ ButtonPanel.java	4.25
	1.00
	1.00
	1.00
ButtonPanel	14.00
Disposable.java	1.00
IPacmanInteraction.java	7.00
MainUI.java	22.00
⊟ PacmanKeyListener.java	11.00
MatchState	1.00
PacmanKeyListener	21.00
PointsPanel.java	3.00
⊟ org.jpacman.framework.view	6.00
⊞ Animator.java	2.00
BoardView.java	13.00
ImageLoader.java	7.00
⊞ org.jpacman.test.framework.accept	8.00
⊞ org.jpacman.test.framework.factory	2.00
⊟ org.jpacman.test.framework.model	3.85
BoardTileAtTest.java	2.00
GameTest.java	15.00
PointManagerTest.java	4.00
⊞ SpriteTest.java	1.50
⊞ org.jpacman.test.framework.ui	2.66

FIGURE 11 – Détails de l'analyse des métriques du projet (partie 2).

⊞ Average Number of Parameters	0.49
⊞ Comments Ratio	15.9%
⊞ Efferent Couplings	31
⊞ Lines of Code	2,188
⊞ Number of Characters	108,548
⊞ Number of Comments	349
⊞ Number of Constructors	17
⊞ Number of Fields	124
⊞ Number of Lines	4,128
⊞ Number of Methods	261
Number of Packages	19
⊞ Number of Semicolons	1,208
⊞ Number of Types	48
⊞ Weighted Methods	427

FIGURE 12 – Détails de l'analyse des métriques du projet (partie 3).

Minimum and Maximum | Method Complexities | Description

Name	Value
PacmanKeyListener()	1
keyTyped()	1
keyPressed()	12
keyReleased()	1
start()	16
stop()	16
exit()	8
up()	1
down()	1
left()	1
right()	1
movePlayer()	16
controlling()	1
getCurrentState()	1
withDisposable()	1
withGameInteractor()	1
stopControllers()	1
startControllers()	1
getGame()	1
update()	1
updateState()	16
updateState()	1

FIGURE 13 – Détails de l'analyse de la complexité cyclique de la classe "PacmanKeyListener".

Minimum and Maximum | Method Complexities | Description

Name	Value
worldWidth()	1
worldHeight()	1
BoardView()	1
windowWidth()	1
windowHeight()	1
createDrawArea()	2
paint()	1
drawCells()	3
drawCell()	3
fullArea()	1
centeredArea()	1
spriteColor()	8
spriteImage()	5
nextAnimation()	2

FIGURE 14 – Détails de l'analyse de la complexité cyclique de la classe "BoardView".

2.12 Audit

Cet intitulé reprend tous les problèmes et les erreurs de code tel que les erreurs non capturées, la sérialisation les imports inutiles, les droits d'accès,... Voici celles détectées par l'outil CodePro d'Eclipse :

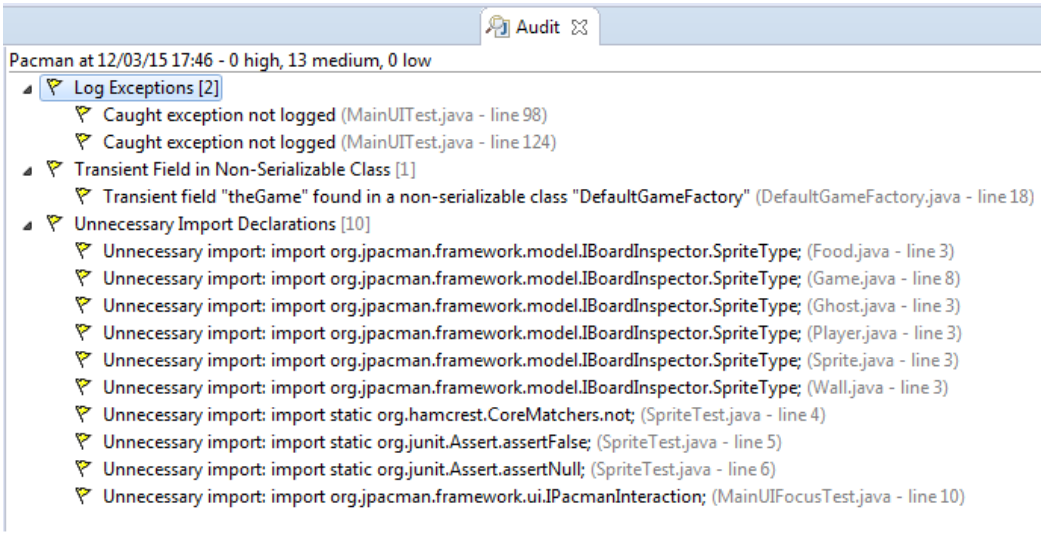


FIGURE 15 – Détail de l'analyse d'audit faite par Eclipse.

2.12.1 Erreur non capturée

Il s'agit de morceaux de code à risque (qui peuvent provoquer l'arrêt du programme avec une erreur fatale) qui n'est pas protégé. Ici, les deux cas repérés sont encadrés par un bloc "try...catch" mais ils ne capturent qu'un seul type d'exception. Pour résoudre ce problème il suffira d'ajouter un second bloc "catch" qui prend en charge l'ensemble des autres exceptions.

2.12.2 Sérialisation

Un objet est sérialisable quand il implémente la classe "Serializable". Au sein d'un tel objet, tous les éléments doivent, par défaut, pouvoir l'être aussi. Dans le cas contraire, une variable globale peut-être qualifée avec le mot clé "‘transient"‘ pour déclarer cette variable non sérialisable. Dans ce cas-ci, l'objet DefaultGameFactory n'implémente pas la classe "Serializable" donc il n'y a aucune raison de déclarer une variable "transient".

2.12.3 Import inutile

Les imports sont les premières lignes d'une classe. Ils permettent d'informer au compilateur d'où viennent les méthodes, les objets,... utilisés qui ne sont pas créés au sein de la classe courante. Egalement repris plus loin dans les warnings d'Eclipse, les classes comportent plusieurs imports qui ne sont jamais utilisés. La meilleur solution est simplement de les supprimer.
Ceux détectés par l'outil Code Inspector d'IntelijIdea se trouve à la figure 16 (page 24).

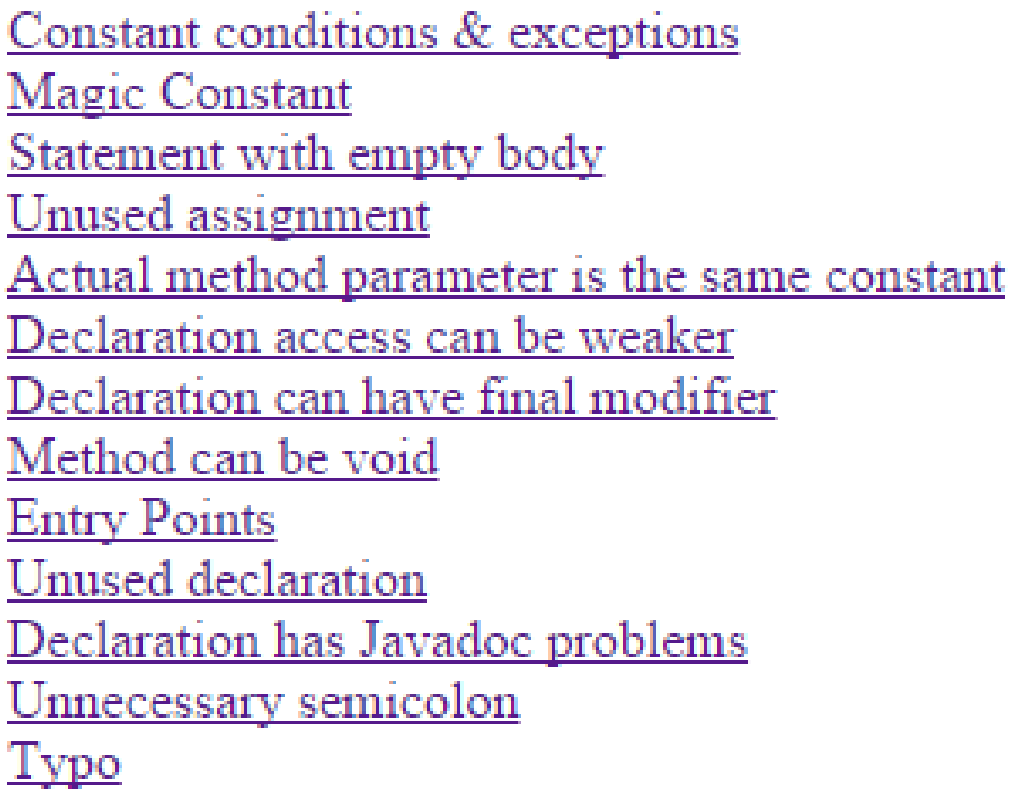


FIGURE 16 – Détail de l'analyse d'audit faite par IntelijIdea.

2.12.4 Droit d'accès

Chaque élément d'un code (classe, sous-classe, méthode, variable,...) est qualifié d'un mot-clé qui permet de définir l'accessibilité tel que (pour une variable globale ou une méthode [9]) :

9. un descriptif similaire peut-être fait pour une classe

- public :
  - Est accessible dans la classe
  - Peut être accessible depuis une autre classe du package où elle se trouve.
  - Peut être accessible depuis n'importe quelle classe extérieure au package.
- protected :
  - Est accessible dans la classe
  - Peut-être accessible depuis une autre classe du package où elle se trouve (et des classes enfents).
  - N'est pas accessible depuis n'importe quelle classe extérieure au package.
- private :
  - est accessible dans la classe
  - Ne peut pas être accessible depuis une autre classe du package où elle se trouve.
  - N'est pas accessible depuis n'importe quelle classe extérieur au package.

Une mauvaise pratique est de tout mettre en publique. Ca permet une vue d'ensemble sur son programme mais implique parfois certaines erreurs d'utilisation.
Ce n'est donc pas un problème grave à l'heure actuelle puisque le jeu fonctionne correctement mais lors de l'ajout de fonctionnalités cela peut le devenir.

2.12.5 Optimisation

En java, une variable qui ne doit pas être modifiée pendant le temps d'une exécution peut-être qualifiée avec le mot-clé "final". Ceci permet d'optimiser le code et de ne pas être confronté à une mauvaise utilisation de la variable par la suite.

2.12.6 Restructuration

Cet outil informe que la fonction "public int addPoints(int)" de la classe "org.jpacman.framework.model.Player" retourne une valeur qui n'est jamais utilisée et qui dont pourrait être transformée en "public void addPoints(int)". Cette modification est à étudier avant d'agir pour vérifier s'il n'est pas nécéssaire pour une utilisation postérieur de garder cette valeur de retour.
Il informe aussi que deux autres fonctions reçoivent toujours la même valeur dans leur paramètre et que donc cette valeur pourrait devenir une constante.

Chacun des cas sera à étudier pour savoir si cette information est disponible à la méthode et que celle-ci doit alors être refactorée ou si elle a été mise en paramètre en vue d'une amélioration future.

2.12.7 Code inutile

Ce sujet a déjà été traité plus haut (Voir 2.3).

2.12.8 Javadoc

Ce sujet a déjà été traité plus haut (Voir 2.4).

2.12.9 Caractère inutile

Ce sont des éléments du code qui sont sous-entendu par le reste de l'architucture du code.
Dans ce projet, un seul cas a été resencé, il s'agit d'un ";" dans la classe "IBoardInspector" du package "org.jpacman.framework.model". Lors du refactoring, il suffira de le retirer.

2.12.10 Typographie

Les problèmes de typographies reprennent les erreurs de formatage des différents éléments. Ce ne sont que des erreurs de conventions parce que elles ne changent en rien le comportement de l'application. Cependant, un bon respect des conventions permet une meilleur compréhension lors d'une relecture, d'une modification, de l'étude du code par un nouveau développeur sur le projet,...
L'outil Code Inspector intégré à IntelijIdea a permis d'en recenser plusieurs. A l'étude de celle-ci, il a été observé qu'elles sont souvent dans les commentaires. Exemple, dans le fichier "GhostMover.java" du package "org.jpacman.framework.model", entre la ligne 28 et la ligne 30, le mot "randomizer" a été identifié avec une majuscule dans le commentaire et sans majuscule dans les lignes de code.

Il est important de signaler aussi, que l'outil Eclipse soulève certaines attentions à l'aide de "warnings".
Les types d'erreurs sont :

- Empty block should be documented x2
- Javadoc : Missing comment for public declaration x 51
- Redundant specification of type arguments <...> x6
- The import ... is never used x4

- The method ... of type ... should be tagged with @Override since it actually overrides a superinterface method x13
- The parameter ... is hiding a field from type ... x 7

Leurs emplacements :

Package	# warning	Classe	# warnings
/main/java/.../model	60	Game.java	15
		IBoardInspector.java	13
		Direction.java	8
		Player.java	7
		Board.java	4
		IPointInspector.java	3
		PointManager.java	3
		Tile.java	3
		GhostMover.java	2
		Sprite.java	1
		Food.java	1
/main/java/.../ui	15	ButtonPanel.java	8
		PacmanKeyListener.java	5
		MainUI.java	2
/test/java/.../model	5	SpriteTest.java	5
main/java/.../factory	2	MapParser.java	2
/test/java/.../ui	1	MainUIFocusTest.java	1

TABLE 3 – Emplacement des "warnings" signalé par Eclipse

2.13 Profilage

2.13.1 Ressources CPU

Comme visible sur la figure 17 (page 28), c'est la chargement des images qui prend le plus de temps. Ceci étant une fonction externe au projet, il n'est pas possible d'y effectuer une modification. La méthode, sur laquelle il serait peut-être possible d'effectuer des changements, s'appelle "displayPoint" de la classe "org.jpacman.framework.ui.PointsPanel". Celle-ci calcule à chaque mise-à-jour de l'affichage le nombre de points déjà récoltés sur le nombre de points totaux. Cependant, la fonction est simple et ne prend en elle-même que très peu de temps. Ce qui pénalise est le nombre d'appel, qui lui ne peut pas être modifié puisqu'il est est necéssaire que ce score soit toujours à jour. Il n'est donc probablement pas possible d'accélérer l'application.

Ressources CPU

Hot Spots - Method	Self Time [%]	Self Time	Total Time	Invocations
sun.awt.image.ImageFetcher.**run** ()		15.431 ms (50 %)	15.431 ms	1
javax.swing.RepaintManager$ProcessingRunnable.**run** ()		14.480 ms (46,9 %)	14.844 ms	2.131
org.jpacman.framework.ui.PointsPanel.**displayPoints** ()		302 ms (1 %)	302 ms	941
org.jpacman.framework.model.Tile.**topSprite** ()		219 ms (0,7 %)	219 ms	1.686.080
org.jpacman.framework.ui.MainUI.**update** (java.util.Observable, Object)		145 ms (0,5 %)	145 ms	943

Ressources CPU avec filtre "org.jpacman"

Hot Spots - Method	Self Time [%]	Self Time	Total Time	Invocations
org.jpacman.framework.ui.PointsPanel.**displayPoints** ()		302 ms (1 %)	302 ms	941
org.jpacman.framework.model.Tile.**topSprite** ()		219 ms (0,7 %)	219 ms	1.686.080
org.jpacman.framework.ui.MainUI.**update** (java.util.Observable, Object)		145 ms (0,5 %)	145 ms	943
org.jpacman.framework.model.Board.**tileAt** (int, int)		96,2 ms (0,3 %)	96,2 ms	1.685.153
org.jpacman.framework.model.Wall.**getSpriteType** ()		37,5 ms (0,1 %)	37,5 ms	830.156

Ressources mémoire

Class Name - Live Allocated Objects	Live Bytes [%]	Live Bytes	Live Objects	Generations
java.awt.**Rectangle**		258.688 B (30,8 %)	8.084 (29,5 %)	2
java.awt.**Point**		160.584 B (19,1 %)	6.691 (24,4 %)	1
java.awt.**Dimension**		159.336 B (19 %)	6.639 (24,2 %)	1
sun.java2d.**SunGraphics2D**		28.080 B (3,3 %)	130 (0,5 %)	1
char[]		19.120 B (2,3 %)	306 (1,1 %)	3
java.lang.**Object[]**		19.056 B (2,3 %)	583 (2,1 %)	8
java.awt.**Insets**		18.560 B (2,2 %)	580 (2,1 %)	1
java.security.**AccessControlContext**		18.200 B (2,2 %)	455 (1,7 %)	1
sun.java2d.pipe.**Region**		16.240 B (1,9 %)	406 (1,5 %)	1
java.awt.geom.**AffineTransform**		15.912 B (1,9 %)	221 (0,8 %)	2
byte[]		10.752 B (1,3 %)	56 (0,2 %)	3
java.util.**TreeMap$Entry**		8.960 B (1,1 %)	224 (0,8 %)	1
java.io.**ObjectStreamClass$WeakClassKey**		8.608 B (1 %)	269 (1 %)	1
int[]		8.208 B (1 %)	36 (0,1 %)	3
java.awt.event.**InvocationEvent**		5.568 B (0,7 %)	87 (0,3 %)	2
java.util.**HashMap**		3.984 B (0,5 %)	83 (0,3 %)	8
java.util.**Vector**		3.648 B (0,4 %)	114 (0,4 %)	8
java.lang.**String**		3.504 B (0,4 %)	146 (0,5 %)	3

FIGURE 17 – Détail de l'analyse des ressources CPU et mémoire par VisualVM.

2.13.2 Consommation mémoire

Les ressources mémoires utilisées par l'application sont visible dans la seconde moitié de la figure 17 (page 28). Les observations ne sont pas nombreuses si ce n'est que l'utilisation de la mémoire concerne majoritairement l'affichage et les éléments graphiques de l'application. A moins donc de diminuer la qualité ou la vitesse de rafraichissement (ce qui est fortement déconseillé), il n'y a pas grand chose à faire.

2.14 Conclusion

Suite à cette analyse, il est evident de constater que ce code n'est pas parfait, cependant, dans l'ensemble il est correctement écrit, bien structuré, généralement commenté et facilement compréhensible. De ce point de vue il est donc normal d'estimer que certe certaines améliorations sont conseillées tel que les dépendances cycliques entre les packages et l'ajout de test unitaire mais dans l'ensemble, c'est un bon projet qui devrait être facile à maintenir et à poursuivre.

3 Etape 2 : Ajout de tests unitaires

Maintenant que l'analyse est terminée et avant de commencer à corriger les problèmes soulevé, il convient d'en vérifier le comportement à l'aide de test unitaire. Ces tests vont permettre de créer des scénario avec tous les cas de figure pour tester lors exécution. Les test unitaire permettent ainsi de s'assurer que les modifications apporté au code source ne modifieront pas le comportement du logiciel. Tout d'abord, les tests qui étaient ignorés ont été rajoutés. Et les deux méthodes, se trouvant dans la classe MainUI, qui n'etait pas couverte, le sont maintenant.
Ensuite, des tests ont été ajoutés dans la nouvelle classe MapParserTest[10]. Ils permettent de vérifier que lorsqu'une mauvaise map est mal encodé, les exceptions se lancent bien. Par exemple, lorsq'une map est vide, contient de mauvais caractères, n'a pas une taille "rectiligne",...
Après, des tests ont été rajoutés pour voir si les boutons "play", "stop" et "exit" fonctionnent correctement. Pour cela, un premier test regarde que le joueur ne peut pas bouger lorsque que le bouton pause est appelé et un autre test vérifie que la fenêtre se ferme bien lorsque le bouton "exit" est utilisé.
Enfin, des tests ont été rajoutés pour vérifier le bon fonctionnement des raccourcis clavier. C'est-à-dire que les tests regardent que l'action à effectuer est bien réalisée. Donc les touches haut, bas, gauche, droite, x,s,q sont testées.
Pour conclure, le tableau 4 montre bien que toutes les classes et presque toutes les méthodes sont couvertes. Les méthodes qui ne sont pas couvertes sont des accesseurs (getters et setters) et des méthodes toString().
Grâce aux tests ajoutés, la couverture du code est nettement améliorée.

Package	Classe,%	Methode,%	Ligne,%
Toutes les classes	100% (29/29)	96,4% (212/220)	96,5% (769/797)

TABLE 4 – Résumé de l'analyse de couverture après l'ajout de tests

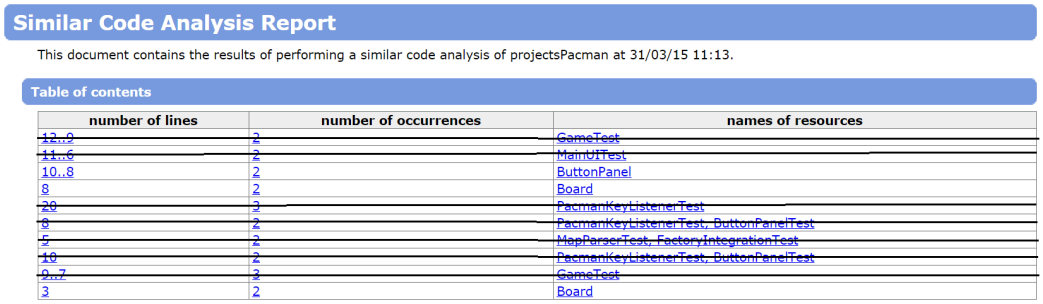
10. Les test sont trié. Chaque classe du package "src.main.java.org.jpacman.framework.*" est reliée à la classe de même nom suivit du mot-clé "Test" dans le package "src.test.java.org.jpacman.test.framework.*".

4 Etape 3 : Refactoring en vue d'améliorer la qualité
Etape 4 : Analyse de la qualité du logiciel

Maintenant que l'ensemble du code est couvert par les test, il est réaliste de vouloir modifier le code pour l'améliorer. Grâce aux différents tests, on peut garantir une certaine stabilité du logiciel et que celui-ci continuera à fonctionner. Après chaque modification, un lancement de l'ensemble des tests sera réalisé pour pouvoir éventuellement faire marche arrière si la modification rétrograde le logiciel.
Cette section va reparcourir chacune des sous-sections de la section section 2 -Etape 1 : Première analyse de la qualité du logiciel- (page 6) et détaillera les chagements réalisés et les changements non réalisés. Pour une meilleur clarté, l'analyse de la qualité du logiciel sera détaillée simultanément [11].
N.B. : Les analyses et le refactoring ne concernent pas les tests.

4.1 Code dupliqué

Il est important de traiter cette section parce que pour la maintenance d'un projet, il est impératif de ne pas devoir effectuer un chagement plusieurs fois et risquer de ne pas effectuer le changement quelque part. Nous pouvons


Similar Code Analysis Report

This document contains the results of performing a similar code analysis of projectsPacman at 31/03/15 11:13.

Table of contents

number of lines	number of occurrences	names of resources
12..9	2	GameTest
11..6	2	MainUITest
10..8	2	ButtonPanel
8	2	Board
20	3	PacmanKeyListenerTest
6	2	PacmanKeyListenerTest, ButtonPanelTest
5	2	MapParserTest, FactoryIntegrationTest
10	2	PacmanKeyListenerTest, ButtonPanelTest
9..7	3	GameTest
3	2	Board

FIGURE 18 – Résultat de l'analyse de "code dupliqué" par CodePro après refactoring

observer sur la figure 18 (page 31) que cet outils ne détécte plus que 3 blocs de code (en excluant les blocs provenant des test). Les blocs qui ne parraissent plus dans le raport de résultat ont été résolu en créant une nouvelle méthode au sein de la classe courante. Ces méthodes sont :

– "checkSprite() ;" pour la classe "...model.Tile" ;

11. L'analyse a été réalisée une fois tout le refactoring réalisé

- "getMap(String filename);" pour la classe "...factory.MapParser" ;
- "invariant();" pour la classe "...ui.PacmanInteraction" (Cette duplication était dans la classe "PacmanKeyListener" voir explicationdans la sous-section 20).

Les blocs encore visible sont détaillé à l'annexe 38 (page 56). Ils n'ont pas été modifié parce que :

- ce sont des test ;
- bien que la syntaxe soit semblable, ils ont un rôle bien différent ;
- les éléments différents ne savent pas (à moins de complexifier le code) être passer en paramètre.

4.2 Dépendances cycliques

Ceci est probablement la modification la plus importante à faire. En effet des dépendances cyclique entre les packages peuvent être fatales et il vaut toujours mieux les résoudre dès que l'on s'en rend compte.
Pour résoudre le problème de dépendance entre les packages "model" et "factory", la classe "level" qui se trouvait dans le package "model" a été transféré dans l'autre. La figure 2 (page 8) permet d'observer que la dépendance cyclique a maintenant disparu.
Rien n'a été fait pour la dépendence entre les classes "Tile" et "Sprite" parce que ce problème a été jugé mineur.

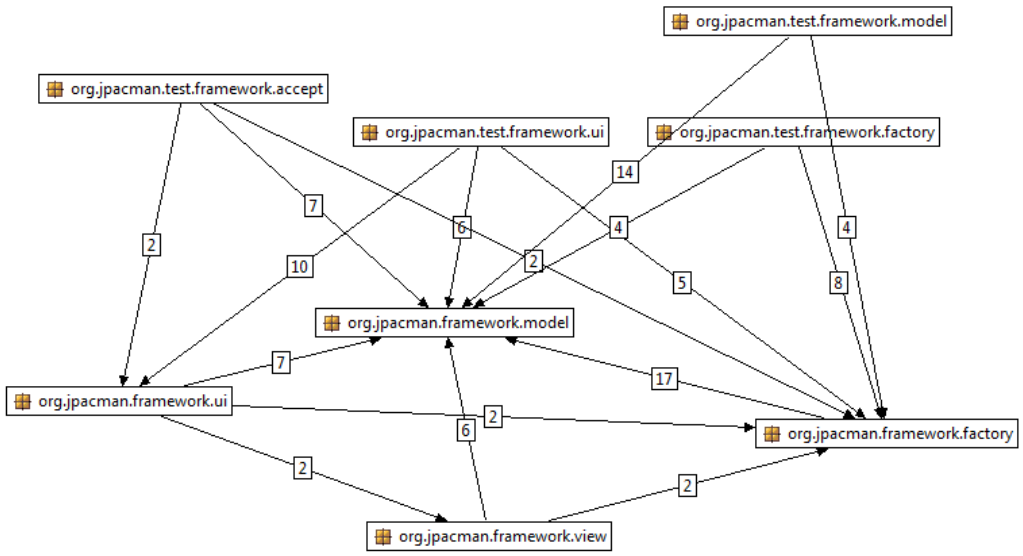


FIGURE 19 – Détail de l'analyse des dépendences cycliques des packages du projet.

4.3 Code inutile

Ces modifications sont subtile et n'ont pas grand intérêt ni valeur ajoutée, et tel que déjà sugéré lors de l'analyse (voir sous-section 2.3), il n'y a finalement que peux de travail à effectuer ici. Les packages de test, les variables "serialVersionUID", les fichiers "package-info.java", sont important donc ne peuvent pas être modifiés.
Concrétement, ont été modifié :

- un " ;" a été supprimé dans le classe "...model.IBoardInspector" ;
- une déclaration redondante dans la classe "...view.ImageLoader" et dans la classe "...factory.MapParser".

4.4 Javadoc

Comme précisé dans l'énoncé, le temps disponible est assez court donc ce critère a été évalué comme mineur. Certains ajout se feront peut-être mais au fur et à mesure du travail sur les autres missions mais pas façon systématique. En voici la liste :

4.5 Test unitaire

Tout d'abord, les méthodes se trouvant dans la classe GameTest ont été renommées pour plus de clareté .En effet, les méthodes étaient simplement numérotées. De plus, la javadoc a été mis à jour dans le but de mieux correspondre à ce qui est testé dans les méthodes. Enfin, suite à d'autres modification, deux tests ont été commentés suite à la suppression de deux méthodes (withFactory() et withButtonPanel()) dans la classe "MainUI".

4.6 Flux de conception

En regardant la sévérité des classes schizophréniques, il est évident que la classe "Game" doit-être travaillée en priorité. Ce fut chose faite en déplaçant les méthodes "movePlyer()" et "moveGhost()" respectivement dans les classes "Player" et "Ghost". La figure 20 (page 34) montre le résultat de se changement.
Pour l'autre classe, une première séparation a été faite entre la partie gestion des événements clavier et la partie interaction avec pacman en créant la classe PacmanInteractor. C'est donc maintenant cette dernière qui est répertoriée comme schizophréniques. Une seconde séparation est donc surement possible, mais le choix est de la garder tel qu'elle est.

FIGURE 20 – Analyse de la structure du code par InCode.

4.7 Complexité

La figure 21 (page 35) permet de voir que la complexité est restée sur la classe "PacmanKeyListener". C'est dù au fait que dans l'une des méthodes se trouve un switch qui est composée de beaucoup de cas.

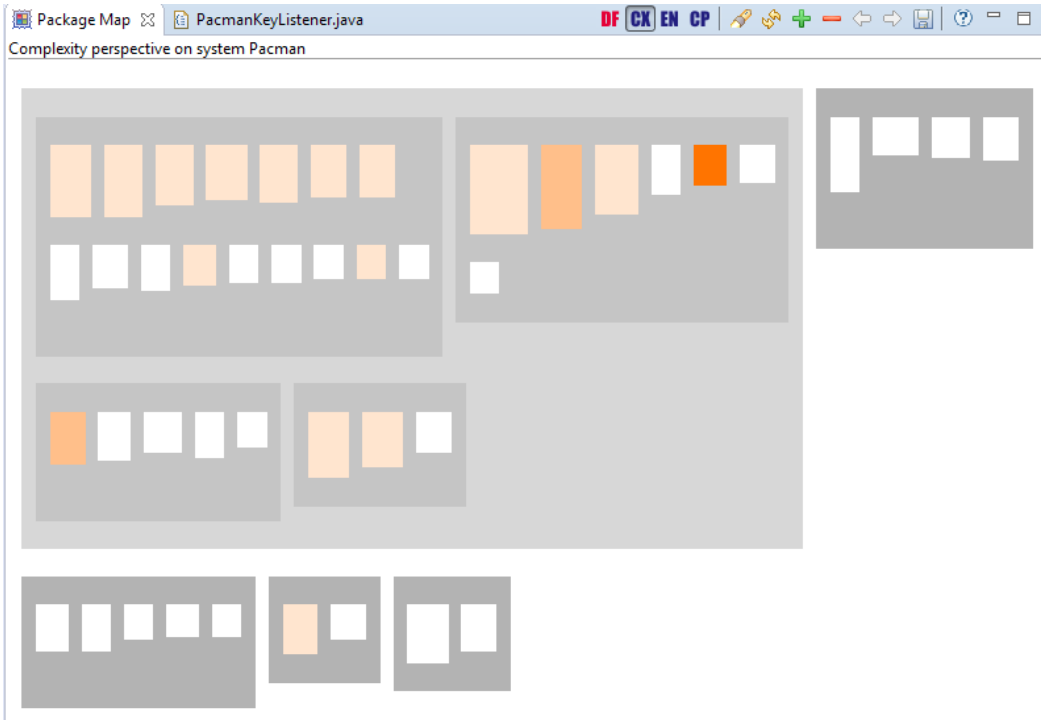


FIGURE 21 – Analyse de la complexité par InCode.

4.8 Encapsulation

Aucun travail n'a réellement été réalisé sur ce critère, on remarque tout de même certaines modifications dont :

- La classe "MainUI" s'est affirmée avec encore plus de clients.
- La classe "Game" avait tendance à avoir plus de client et maintenant elle a plus de fournisseur.

4.9 Couplage

La figure 23 (page 37) nous montre, comparé à la figure 8 (page 16), que il n'y a pas eu de gros changement de ce côté-là.

4.10 Pyramide des métriques

La figure 24 (page 38) montre que la seule différence significative est que l'arbre que constitue les classe s'est élargit.

FIGURE 22 – Analyse de la complexité par InCode.

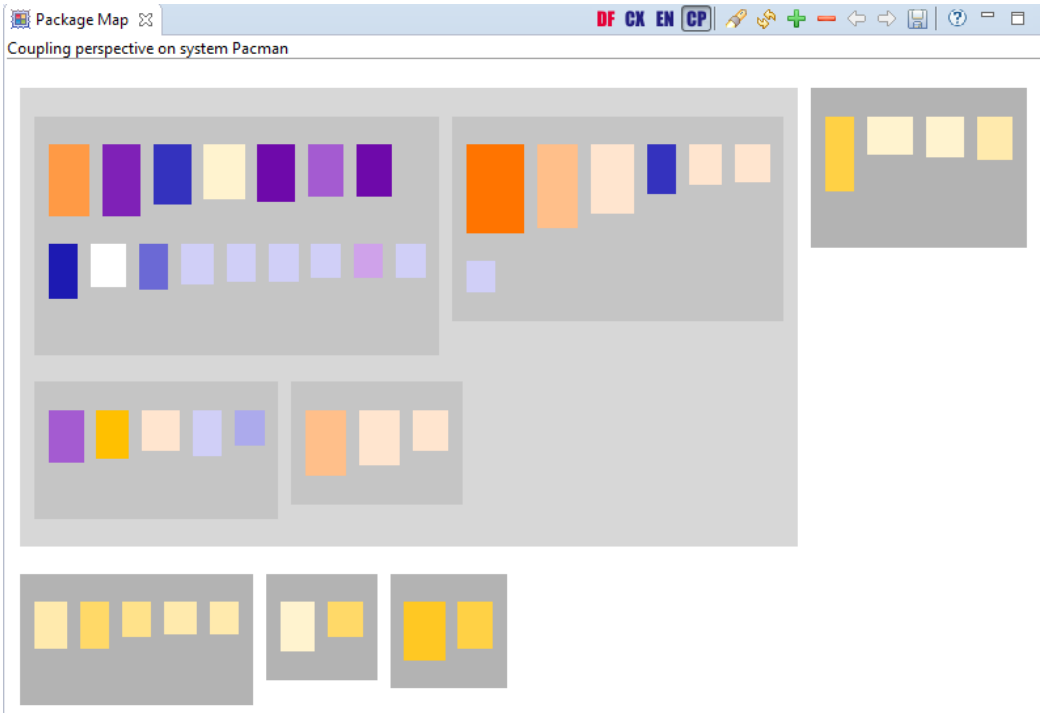


FIGURE 23 – Analyse de la complexité par InCode.

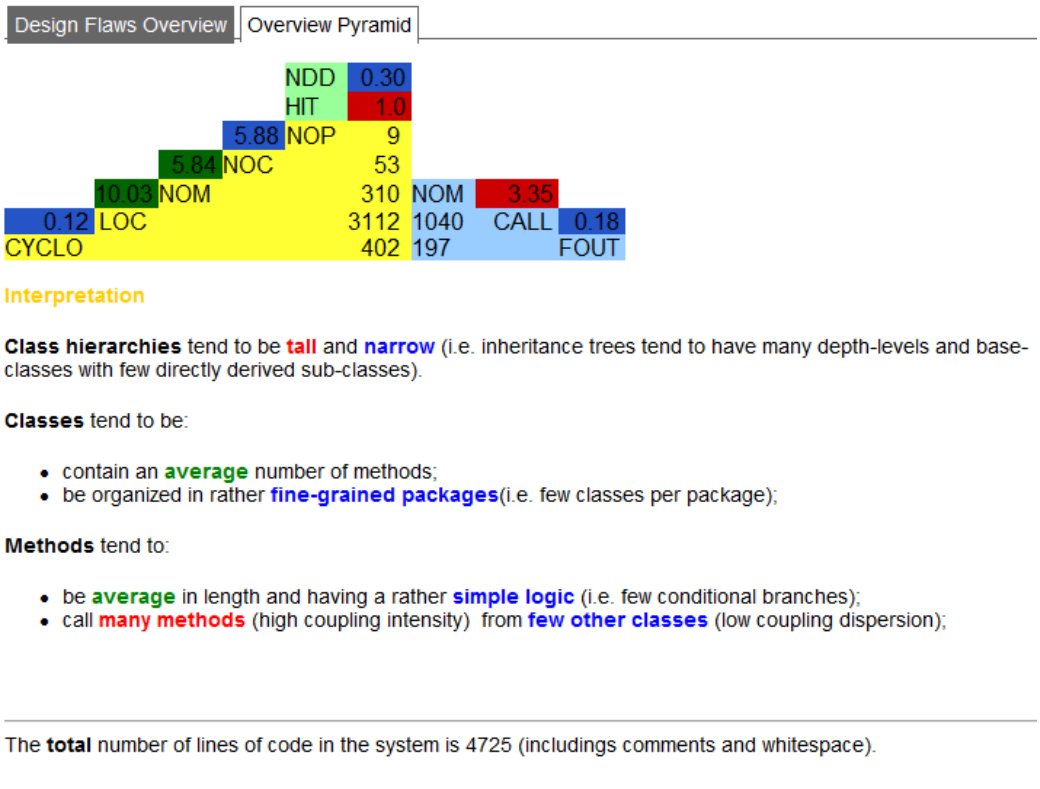


FIGURE 24 – Pyramide des valeurs calculées par InCode.

4.11 Métriques

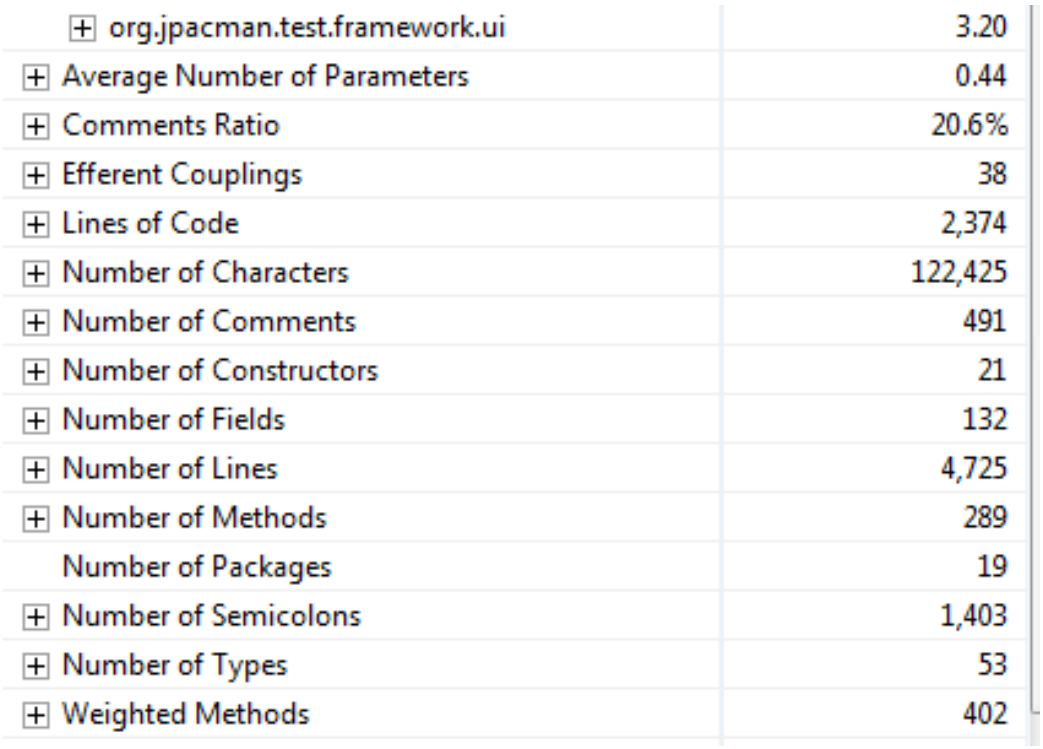
Suite au refactoring du code dupliqué, on observe que le nombre de problème n'a pas vraiment diminué. Certains cas ont été déplacé, mais c'est tout.

Metric	Value
⊞ Abstractness	15%
⊞ Average Block Depth	0.81
⊟ Average Cyclomatic Complexity	1.29
⊟ org.jpacman.framework.factory	1.55
DefaultGameFactory.java	1.00
FactoryException.java	1.00
IGameFactory.java	1.00
Level.java	1.00
MapParser.java	2.87
⊞ org.jpacman.framework.model	1.21
⊟ org.jpacman.framework.ui	1.44
⊞ ButtonPanel.java	1.05
IDisposable.java	1.00
IPacmanInteraction.java	1.00
MainUI.java	1.09
⊟ PacmanInteraction.java	1.85
MatchState	1.00
PacmanInteraction	1.94
PacmanKeyListener.java	4.00
PointsPanel.java	1.00
⊞ org.jpacman.framework.view	1.62
⊞ org.jpacman.test.framework.accept	1.00
⊞ org.jpacman.test.framework.factory	1.11
⊞ org.jpacman.test.framework.model	1.00
⊞ org.jpacman.test.framework.ui	1.12
⊞ Average Lines Of Code Per Method	5.91
⊞ Average Number of Constructors Per Type	0.39
⊞ Average Number of Fields Per Type	1.88

FIGURE 25 – Détails de l'analyse des métriques du projet (partie 1).

Metric	Value
⊟ Average Number of Methods Per Type	5.45
⊞ org.jpacman.framework.factory	4.79
⊟ org.jpacman.framework.model	6.00
Board.java	14.00
Direction.java	4.00
Food.java	3.00
Game.java	15.00
Ghost.java	1.00
GhostMover.java	8.00
⊞ IBoardInspector.java	3.00
IController.java	3.00
IGameInteractor.java	9.00
IPointInspector.java	3.00
Player.java	10.00
PointManager.java	8.00
RandomGhostMover.java	1.00
Sprite.java	7.00
Tile.java	9.00
Wall.java	1.00
⊟ org.jpacman.framework.ui	6.45
⊞ ButtonPanel.java	4.25
IDisposable.java	1.00
IPacmanInteraction.java	7.00
MainUI.java	20.00
⊞ PacmanInteraction.java	10.00
PacmanKeyListener.java	3.00
PointsPanel.java	3.00
⊟ org.jpacman.framework.view	5.75
⊞ Animator.java	2.00
BoardView.java	12.00
ImageLoader.java	7.00
⊞ org.jpacman.test.framework.accept	8.00
⊞ org.jpacman.test.framework.factory	4.50
⊞ org.jpacman.test.framework.model	4.00

FIGURE 26 – Détails de l'analyse des métriques du projet (partie 2).



⊞ org.jpacman.test.framework.ui	3.20
⊞ Average Number of Parameters	0.44
⊞ Comments Ratio	20.6%
⊞ Efferent Couplings	38
⊞ Lines of Code	2,374
⊞ Number of Characters	122,425
⊞ Number of Comments	491
⊞ Number of Constructors	21
⊞ Number of Fields	132
⊞ Number of Lines	4,725
⊞ Number of Methods	289
Number of Packages	19
⊞ Number of Semicolons	1,403
⊞ Number of Types	53
⊞ Weighted Methods	402

FIGURE 27 – Détails de l'analyse des métriques du projet (partie 3).

4.12 Audit

Cet intitulé reprend tous les problèmes et les erreurs de code tel que les erreurs non capturées, la sérialisation les imports inutiles, les droit d'accès,... Voici celles détectées par l'outil CodePro d'Eclipse :

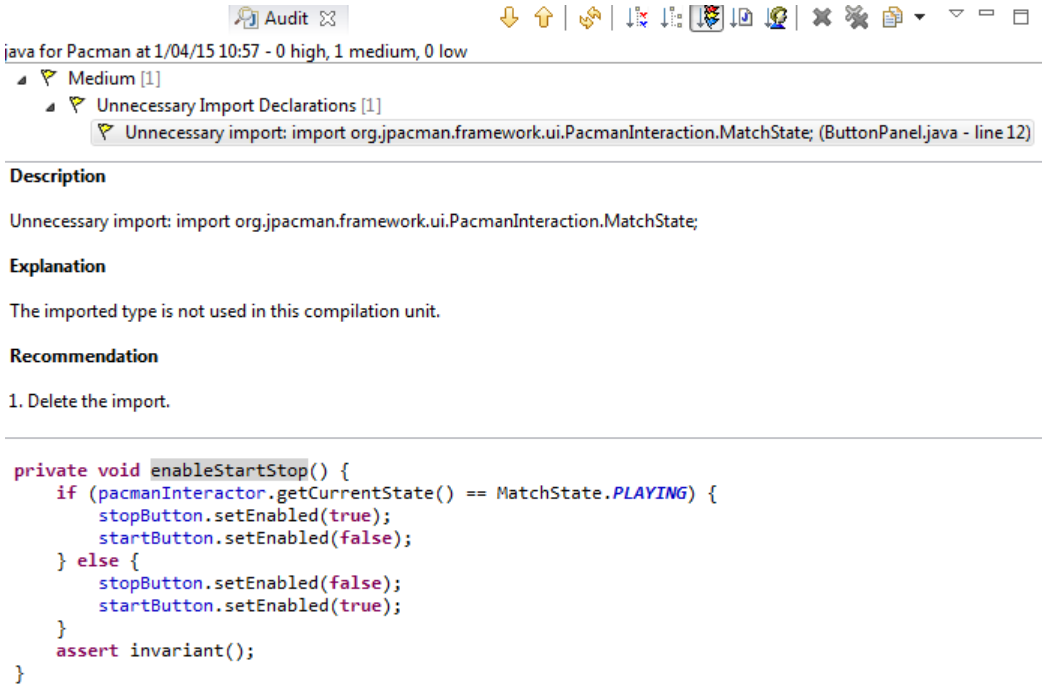


FIGURE 28 – Détail de l'analyse d'audit faite par Eclipse.

4.12.1 Import inutile

Un seul import reste dans l'analyse. La justification est que lors de sa suppression, une erreur est signalée au niveau de la fonction illustrée dans le bas de l'image.

4.13 Conclusion

Après ces quelques heures de travail, il est évident que il y a un mieux, cependant, il n'est pas significatif.

5 Etape 5 : Extensions

Maintenant que le logiciel a été étudié, réparer et réétudié, on peut commencer à l'améliorer et à ajouter de nouvelles fonctionnalités.

5.1 IA fantômes

5.1.1 Règle

Actuellement, le comportement des fantômes est assez erratique, car la direction empruntée par chacun d'eux est déterminée aléatoirement. Commencez par attribuer une couleur à chacun des quatre fantômes du jeu, afin que le joueur puisse les différencier. Créez des IA pour les fantômes afin qu'ils prennent des décisions plus intéressantes pour le joueur. Une décision de direction n'est prise que lorsqu'un fantôme arrive à un embranchement. La décision est basée sur le calcul de la plus courte distance entre la position de case de l'embranchement et la position que le fantôme souhaite atteindre. Si plusieurs directions sont pareillement préférables, un fantôme préférera toujours aller en haut, puis à gauche, puis en bas. Un fantôme ne peut donc aller à droite que si cette direction est la seule représentant la plus petite distance jusqu'à la destination.
Le comportement des fantômes alterne entre un mode de **poursuite** et un mode de **dispersion**.
En mode poursuite, chaque fantôme a un comportement et une vitesse spécifiques et déeterministes [12] :

- **Blinky** a une philosophie très simple : "Droit au but ! ". En toute circonstance, il tente de se rendre sur la case où se trouve Pac-Man par le chemin le plus court. Il a une vitesse de déplacement de 100%.
- **Pinky** aime tendre des embuscades. Il devine où sera Pac-Man 4 mouvements plus tard et se rend à cette position. Il a une vitesse de déplacement de 80%.
- **Inky** est dificilement prévisible pour un humain, car son comportement dépend des positions et des directions de Pac-Man et de Blinky. Imaginez un vecteur joignant la position de Blinky à celle où se trouvera Pac-Man dans 2 mouvements. Doublez la longueur de ce vecteur. La position que Inky cherche à atteindre est à l'extrémité de ce vecteur. Il a une vitesse de déplacement de 100%.
- **Clyde** ne semble pas se préoccuper des autres. Il a en fait deux modes de fonctionnement, et il passe de l'un à l'autre en fonction de la distance

12. http ://gameinternals.com/post/2072558330/understanding-pac-man-ghost-behavior

existant entre Pac-Man et lui. S'il se trouve à plus de 8 cases de Pac-Man, Clyde a le même comportement que Blinky. Dans le cas contraire, il se rend sur la case située dans le coin inférieur gauche du labyrinthe. Il a une vitesse de déplacement de 100%.

En mode dispersion, chaque fantôome se dirige vers sa position "maison". Ces positions sont situées aux quatre coins du labyrinthe :

- Blinky se rend dans le coin supérieur droit.
- Pinky se rend dans le coin supérieur gauche.
- Inky se rend dans le coin inférieur droit.
- Clyde se rend dans le coin inférieur gauche.

Une fois sa position "maison" atteinte, le fantôme va se déplacer dans un circuit qui consiste à toujours emprunter le chemin de gauche pour Pinky et Clyde, et celui de droite pour Blinky et Inky (cf. figure 29 (page 46)) L'alternance entre ces modes est définie comme suit :

- Dispersion pendant 7 secondes, puis poursuite pendant 20 secondes.
- Dispersion pendant 7 secondes, puis poursuite pendant 20 secondes.
- Dispersion pendant 5 secondes, puis poursuite pendant 20 secondes.
- Dispersion pendant 5 secondes, puis poursuite indéfiniment.

5.1.2 Implémentation

Les premiers changements appliqués pour cette extension sont la création de quatre classes dans le package Model. Ces classes sont respectivement GhostBlinky, GhostClyde, GhostInky, et GhostPinky. Toutes les quatres hérite de la classe Ghost.

De plus, des nouveaux SpriteType sont ajoutés intitulé GHOSTBLINKY, GHOSTCLYDE, GHOSTINKY et GHOSTPINKY. Chacun est représenté par une image correspondant à la couleur qui lui est propre pour pouvoir les distinguer. C'est pourquoi les classes BoardView et ImageLoader ont été modifié. Si les images ne sont pas disponible lors de l'exécution, chacun utilise une case de sa couleur défine (respectivement rouge, orange, cyan et rose). Sur une map (dans le fichier ".txt" qui est lu par la classe "factory.MapParser", Blinky est représenté par le symbole "R", Clyde par le symbole "O", Inky par le symbole "C" et Pinky par le symoble "M".

Ensuite, étant donnée que les fantômes ont des comportements différents, le design pattern Strategy a été implementé. Pour cela un package, intitulé strategy, a été crée. Celui-ci contient l'interface IStrategy. Celle-ci contient quatre méthodes qui sont les quatre comportements des fantomes qui sont implémentés par les classes suivantes :

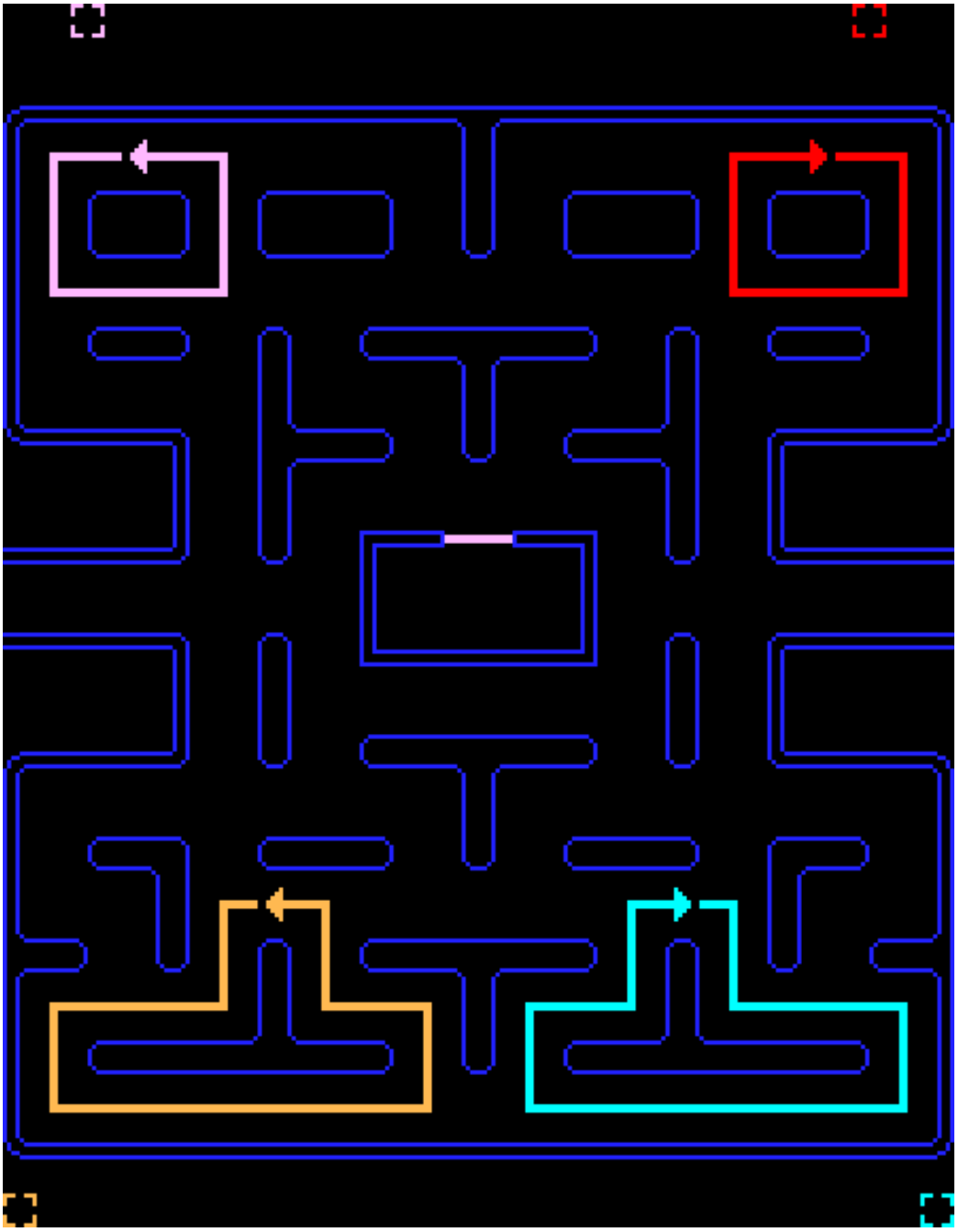


FIGURE 29 – Circuits empruntés par les fantômes en mode dispersion.

1. Escape :

Cette classe, qui implemente IStrategy, redéfinit les méthodes et permet aux fantômes d'avoir le comportement adéquat. C'est-à-dire, qu'ils choisent leur chemin aléatoirement à chaque embranchement.

2. Dispersion :
   Cette classe, qui implemente IStrategy, redéfinit les méthodes et permet aux fantômes d'aller dans leur "maison" respective. C'est-à-dire,que Blinky va dans le coin supérieur droit, si c'est déjà le cas, il se déplace en tournant toujours à droite. Pour Pinky, il se rend dans le coin supérieur gauche et emprunte toujours le chemin de gauche. Pour Pour Inky, il se rend dans le coin inférieur droit et se déplace dans un circuit quiconsite à toujours emprunter le chemin de droite. Et pour il se rend dans le coin inférieur gauche et tourne toujours vers la gauche.

3. Tracking :
   Cette classe, qui implemente IStrategy, redéfinit les méthodes et permet aux fantômes de choisir leur direction pour attraper Pacman.

La stratégie a utilisé est appelé depuis la classe "NormalGhostMover", qui se trouve dans le package "controller". Cette classe hérite de la classe abstraite "AbstractGhostMover". La classe "NormalGhostMover" redéfinit la méthode doTick() qui permet aux fantômes d'appeler la stratégie adéquate par rapport aux évènements comme les timers ou pacman qui mange une super gomme.

Pour finir, la classe MyTimer a été crée pour gérer les différents timers des fantômes. Cette classe se trouve dans le package controller.

5.2 Super gommes

5.2.1 Règle

Modifiez le jeu afin qu'il prenne en compte la présence de supergommes. Lorsque Pac-Man mange une supergomme, elle lui rapporte 50 points.
Pendant un court moment, les règles du jeu changent (le jeu se met en mode fuite) et la proie devient chasseur. Lorsque Pac-Man mange une supergomme, les fantômes sont effrayés. Le chronomètre du mode dans lequel ils se trouvent s'arrête et les fantômes entrent en mode fuite : ils deviennent bleus et leur vitesse de déplacement est réduite à 50% (y compris pour Pinky). En mode

fuite, chaque fantôme décide aléatoirement de la direction à prendre à chaque embranchement. Si Pac-Man touche un fantôme en mode fuite, celui-ci disparaît du jeu et réapparaît au milieu du labyrinthe après 5 secondes.
Si Pac-Man touche un fantôme en mode fuite, il le mange, ce qui lui rapporte :

- 200 points pour le premier fantôme mangé ;
- 400 points pour le second fantôme mangé ;
- 800 points pour le troisième fantôme mangé ;
- 1600 points pour le quatrième fantôme mangé.

Il y a quatre supergommes par niveau. Les deux premiéres à être avalées effraient les fantômes pendant 7 secondes, tandis que les deux dernières les effraient pendant 5 secondes. Après ce délai, les fantômes retournent dans le mode dans lequel ils étaient avant de fuir. Le chronomètre associè à ce mode est repris là où il s'était arrêté.

5.2.2 Implémentation

Pour l'ajout de cette fonctionnalité, une classe SuperGum, qui hérite de la classe Sprite, a été ajouté. Une SpriteType est également ajouté sous le nom de SUPERGUM. Une image est aussi associé a la supergum. De plus, les fantômes deviennent bleu ce qui a demandé un ajustement des classes BoardView et ImageLoader. La couleur défini pour la supergum est magenta.

Etant donnée, que la supergum peut être mangée et rapporte des points, la classe PointManager a été modifié.

Lorsque pacman mange une super gomme, les fantômes changent d'état et passent en mode fuite. A ce moment précis, un timer est lancé dans la classe "NormalGhostMover".

Lorsque pacman mange un fantome, le jeux s'arrête parce que nous avons rencontré une erreur que nous n'avons pas su et pas eu le temps de débugger.

6 Etape 6 : Analyse de la qualité du logiciel

Voici à nouveau l'analyse du code maintenant les améliorations réalisées.

6.0.3 Dépendance

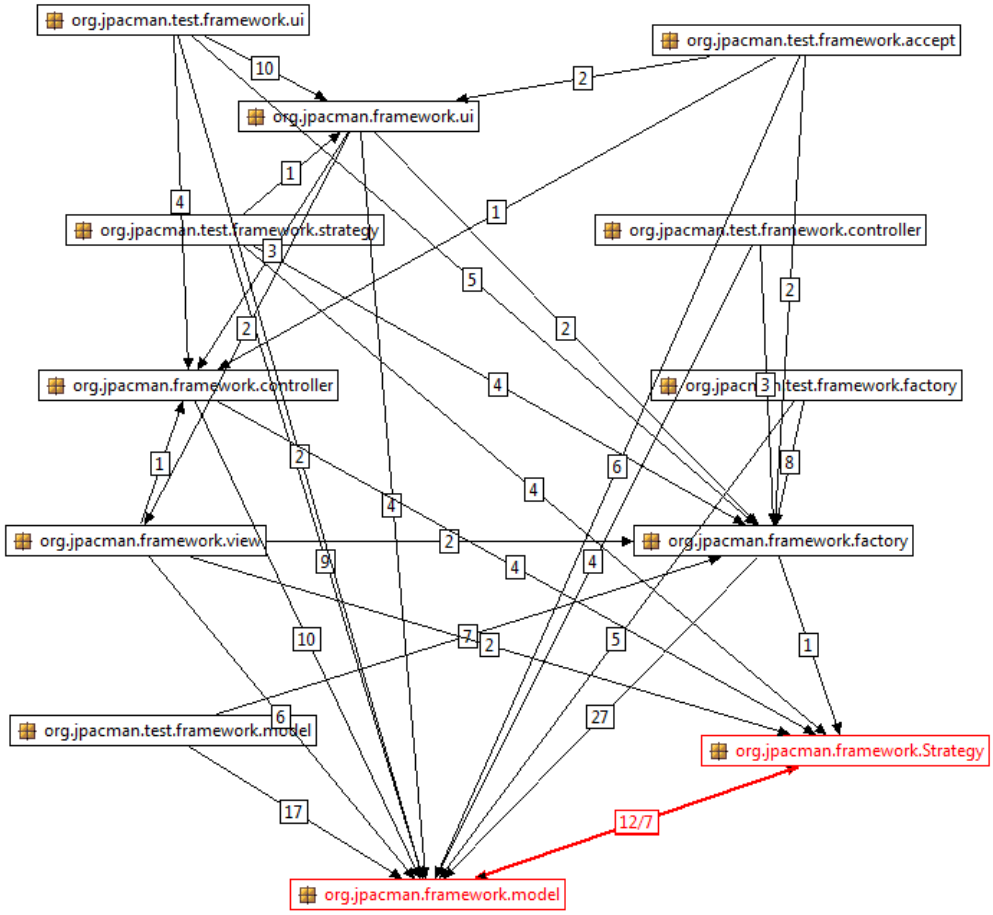


FIGURE 30 – Dépendances entre les packages

6.0.4 Test unitaire

Tout d'abord les anciens tests unitaires ont été modifié pour qu'ils réussisent toujours. Car comme le code évolue, les tests le doive aussi. Par exemple, pour la super gomme, tous les tests ou une map était nécessaire devait être modifé car s'il n'y a pas exactement quatre supergums sur la map

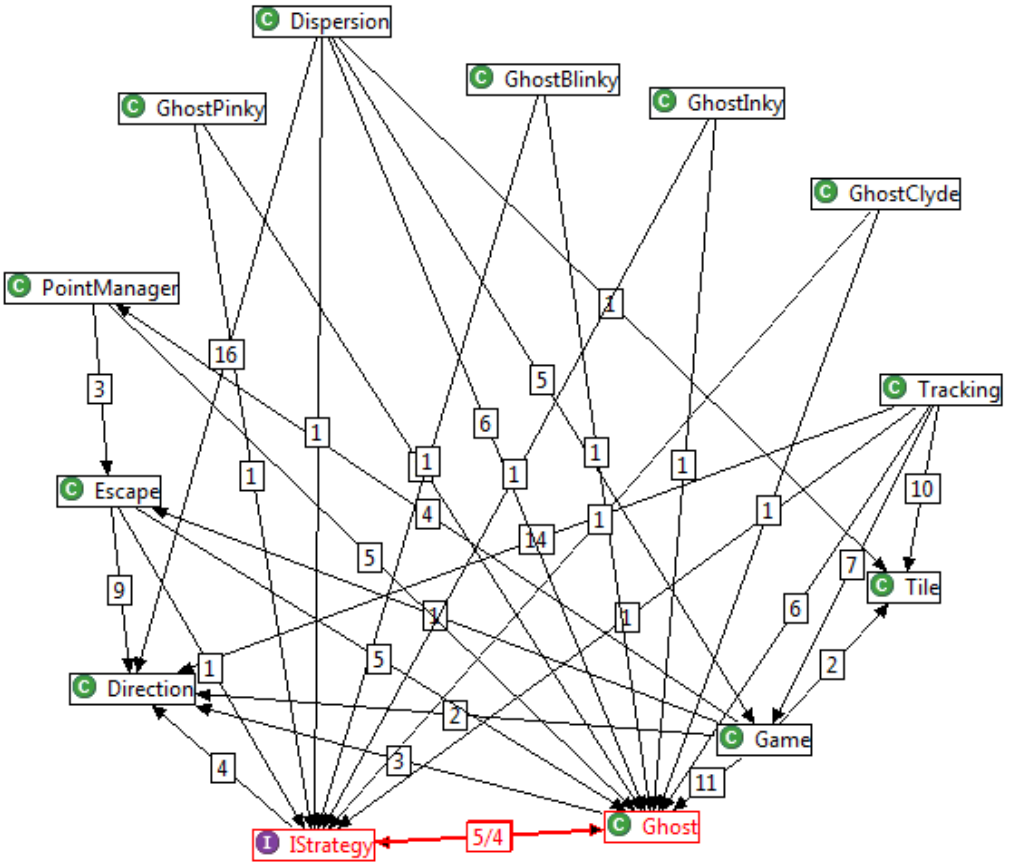


FIGURE 31 – Dépendances entre les packages Model et Strategy

alors les tests ne passent pas.

Ensuite des tests ont été ajoutés au fur et à mesure pour tester le logiciel et essayer de couvrir un maximum le code. L'image reftestCouverture montre la couverture du code actuellement.

L'anlyse montre que toutes les classes sont toujours couvertes mais toutes les méthodes n'ont pas su être testées.

Remarque : Lorsque les tests sont lancés, sur Intellij IDEA, il y a 15 tests failed sur un total de 68 tests. Mais lorsque que les tests sont lancés un par un, tous les tests fonctionnent. Ce problème n'a pas su résolu et a pris énormément de temps pour ensuite être abondonné.

Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	100% (61/ 61)	88,6% (364/ 411)	78,5% (1526/ 1944)

Coverage Breakdown

Package	Class, %	Method, %	Line, %
org.jpacman.framework.Strategy	100% (3/ 3)	28,6% (6/ 21)	13,6% (16/ 118)
org.jpacman.framework.controller	100% (5/ 5)	65,2% (15/ 23)	33,2% (94/ 283)
org.jpacman.framework.factory	100% (4/ 4)	93,1% (27/ 29)	94,7% (125/ 132)
org.jpacman.framework.model	100% (16/ 16)	85,2% (104/ 122)	89,7% (305/ 340)
org.jpacman.framework.ui	100% (9/ 9)	97,3% (73/ 75)	96% (263/ 274)
org.jpacman.framework.view	100% (5/ 5)	97,2% (35/ 36)	96,1% (146/ 152)
org.jpacman.test.framework.accept	100% (2/ 2)	95% (19/ 20)	60% (39/ 65)
org.jpacman.test.framework.controller	100% (1/ 1)	100% (6/ 6)	93% (53/ 57)
org.jpacman.test.framework.factory	100% (2/ 2)	100% (12/ 12)	79,6% (39/ 49)
org.jpacman.test.framework.model	100% (7/ 7)	100% (36/ 36)	89,9% (142/ 158)
org.jpacman.test.framework.strategy	100% (2/ 2)	100% (10/ 10)	92,1% (116/ 126)
org.jpacman.test.framework.ui	100% (5/ 5)	100% (21/ 21)	98,9% (188/ 190)

FIGURE 32 – Résumé de couverture

7 Annexes

A Annexe : Code Dupliqué

Dans cette section se trouve les différentes annexes qui permettent d'identifier les blocs de code dupliqués détectés par CodePro. Chaque image illustre un bloc de code mis à part la dernière qui illustre les 6 derniers blocs de code et est issue du rapport généré par CodePro (parce que Eclipse les masque). N.B. : La figure reprenant tous les blocs identifiés se trouve à la sous-section 2.1

A.0.5 Avant refactoring

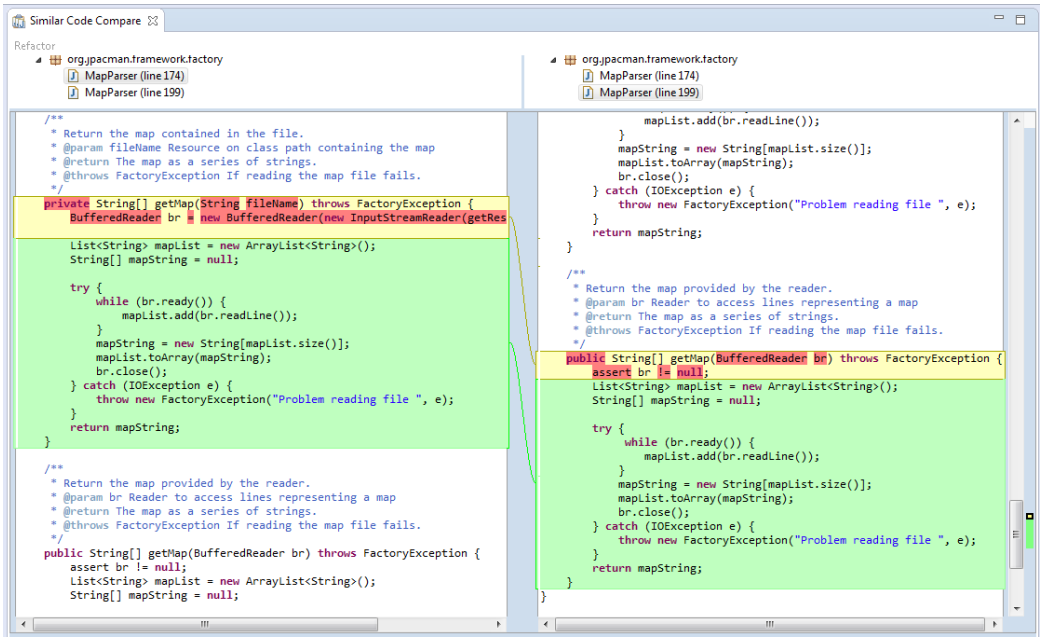


FIGURE 33 – Détail de l'analyse de code redondant par CodePro

A.0.6 Après refactoring

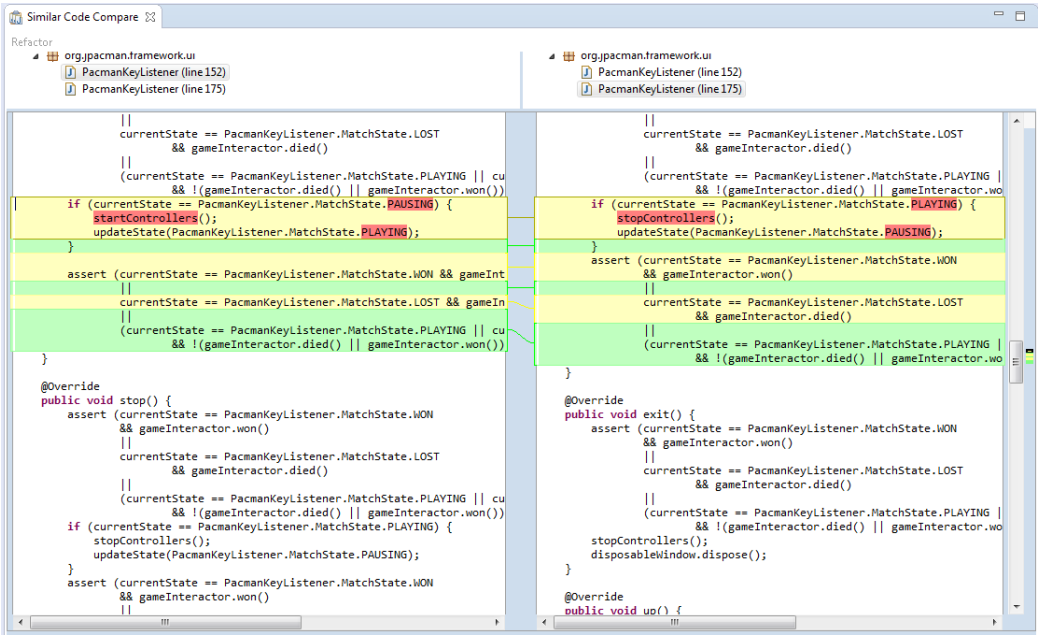


FIGURE 34 – Détail de l'analyse de code redondant par CodePro

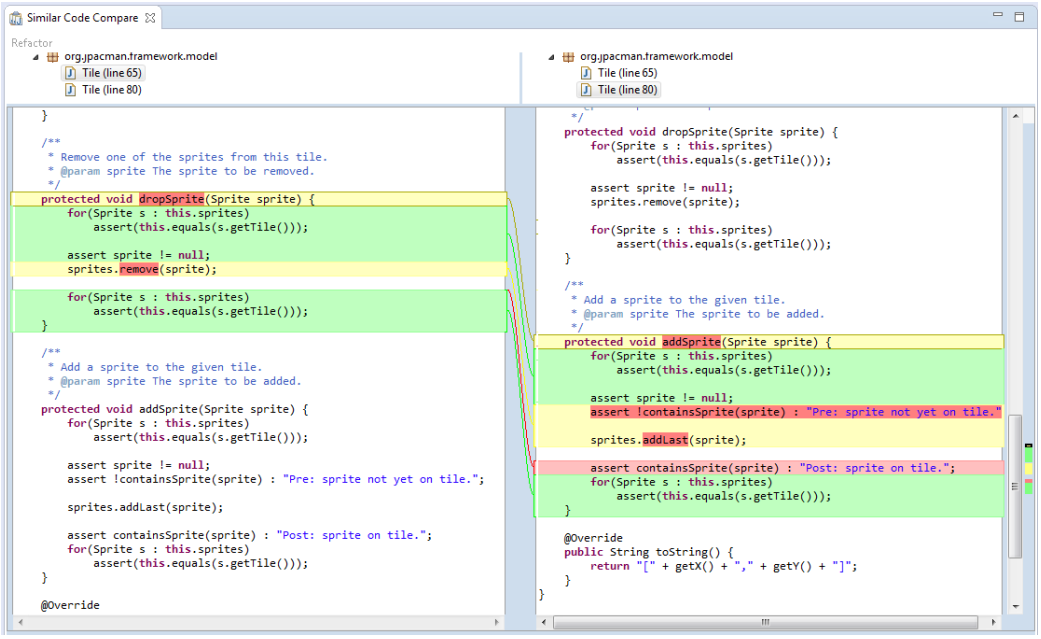


FIGURE 35 – Détail de l'analyse de code redondant par CodePro

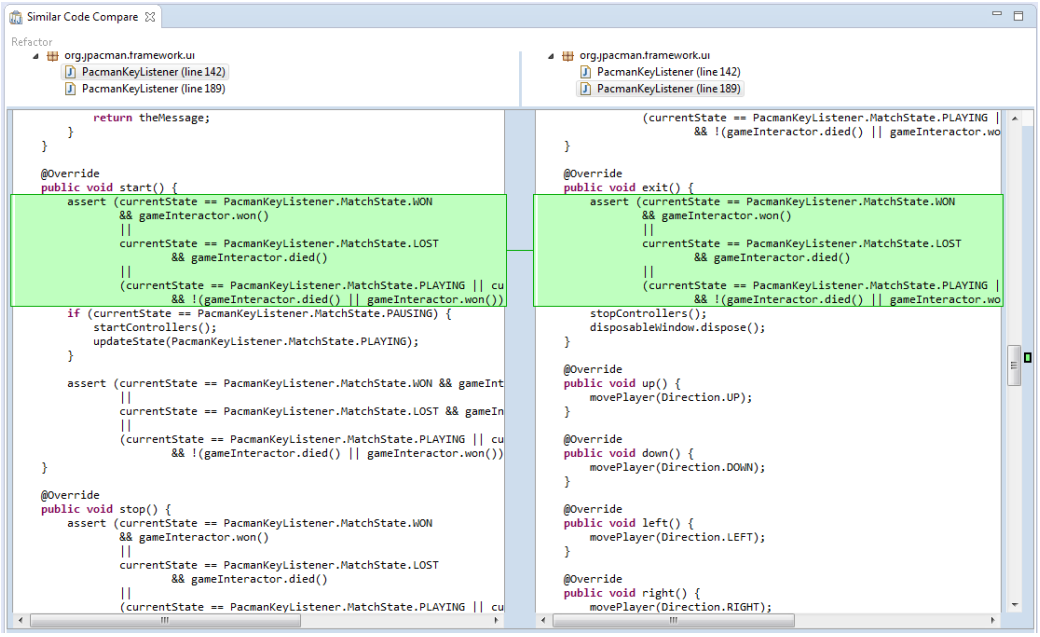


FIGURE 36 – Détail de l'analyse de code redondant par CodePro

FIGURE 37 – Détail de l'analyse de code redondant par CodePro (élément dont la visualisation n'est pas possible dans Eclipse)

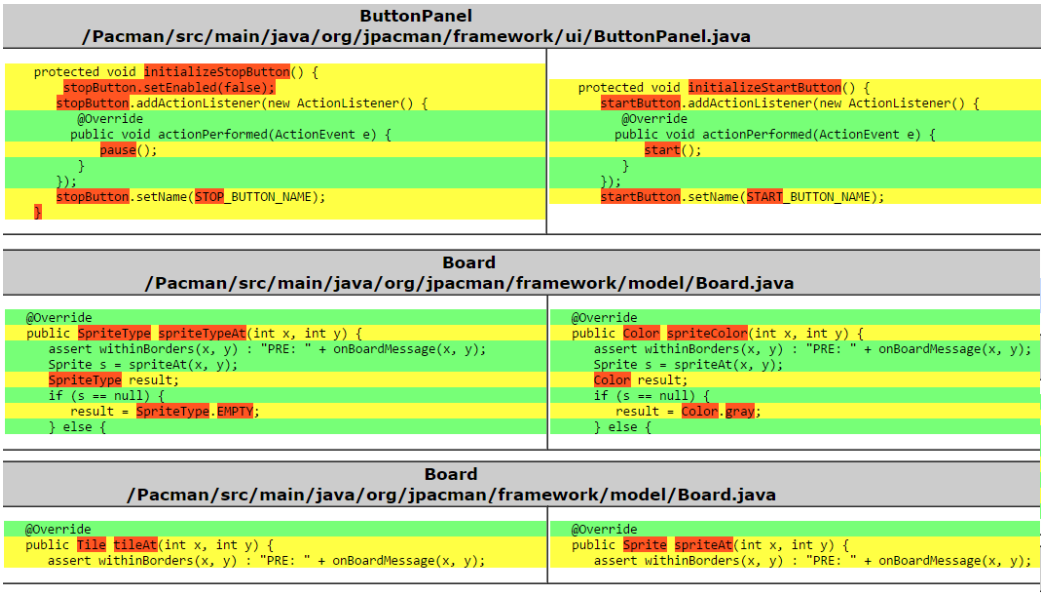


FIGURE 38 – Détail de l'analyse de code redondant par CodePro

B Annexe : Dépendances

Ces figures permettent de visualiser les dépendances entre les différents éléments du projet. N.B. : Les figures des dépendances entre les packages et des dépendances au sein du package Model se trouvent à la sous-section 2.2.

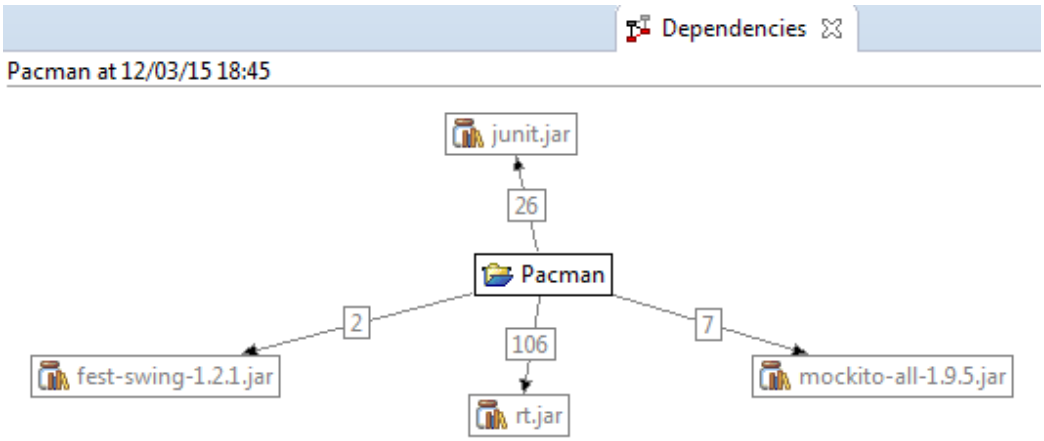


FIGURE 39 – Détail de l'analyse des dépendences cycliques du projet.

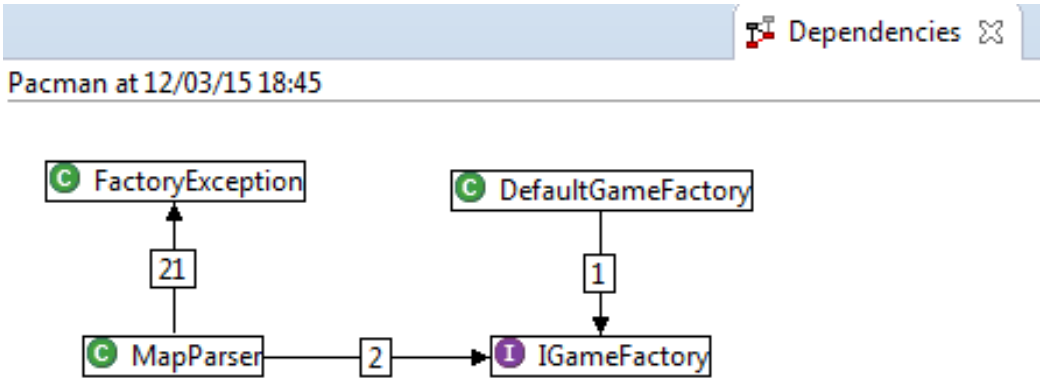


FIGURE 40 – Détail de l'analyse des dépendences cycliques du package Factory.

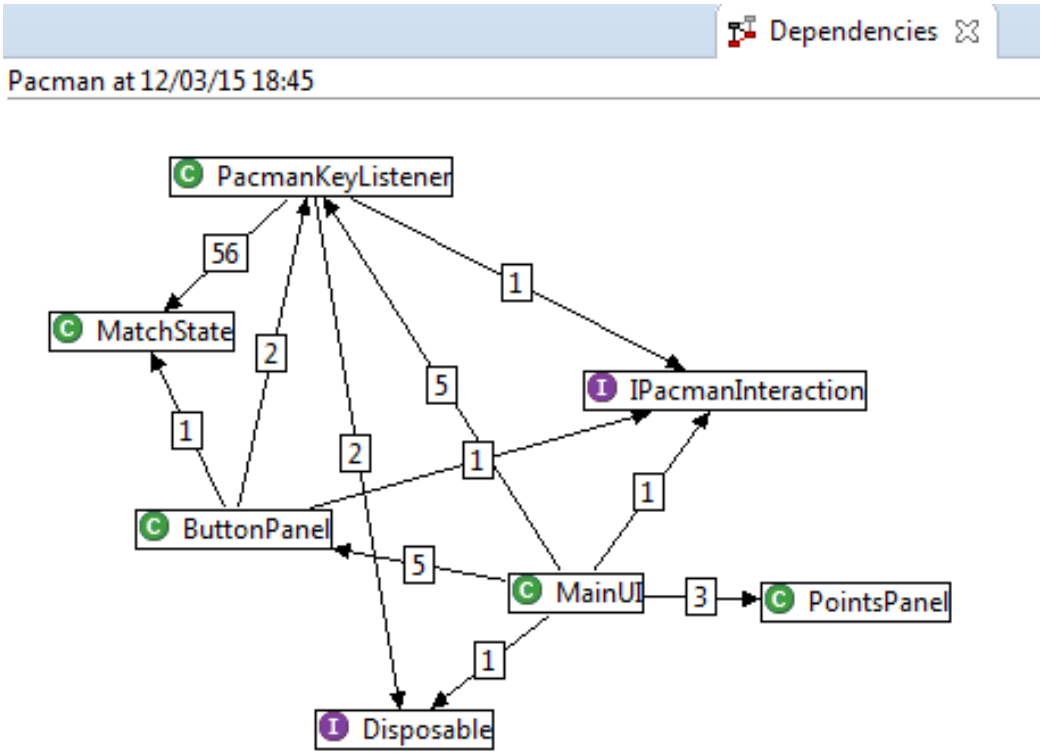


FIGURE 41 – Détail de l'analyse des dépendences cycliques du package UI.

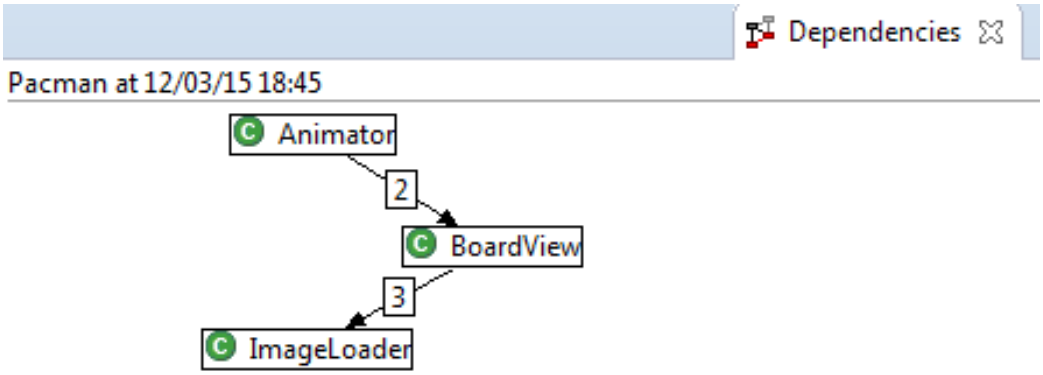


FIGURE 42 – Détail de l'analyse des dépendences cycliques du package Vieuw.

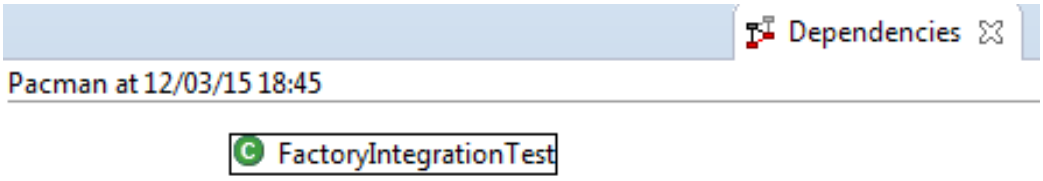


FIGURE 43 – Détail de l'analyse des dépendences cycliques du package Factory (Test).

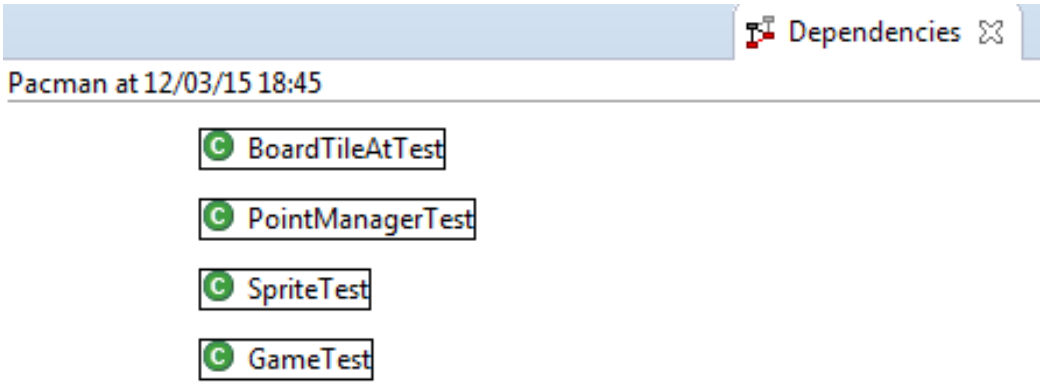


FIGURE 44 – Détail de l'analyse des dépendences cycliques du package Model (Test).

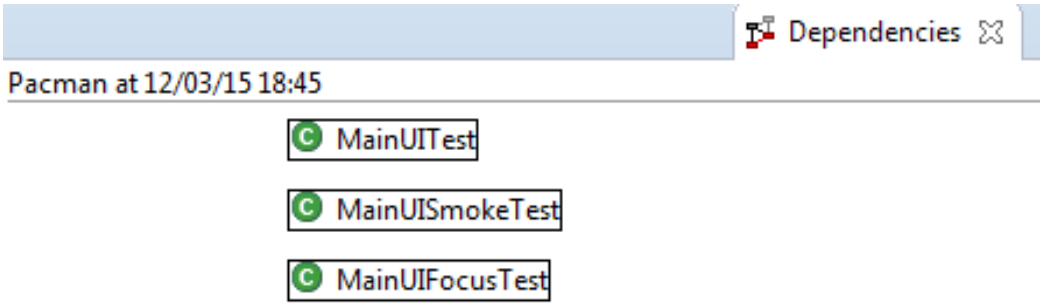


FIGURE 45 – Détail de l'analyse des dépendences cycliques du package UI (Test).

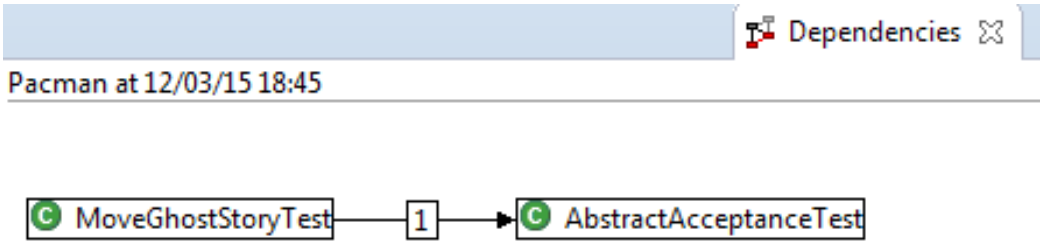


FIGURE 46 – Détail de l'analyse des dépendences cycliques du package Accept (Test).

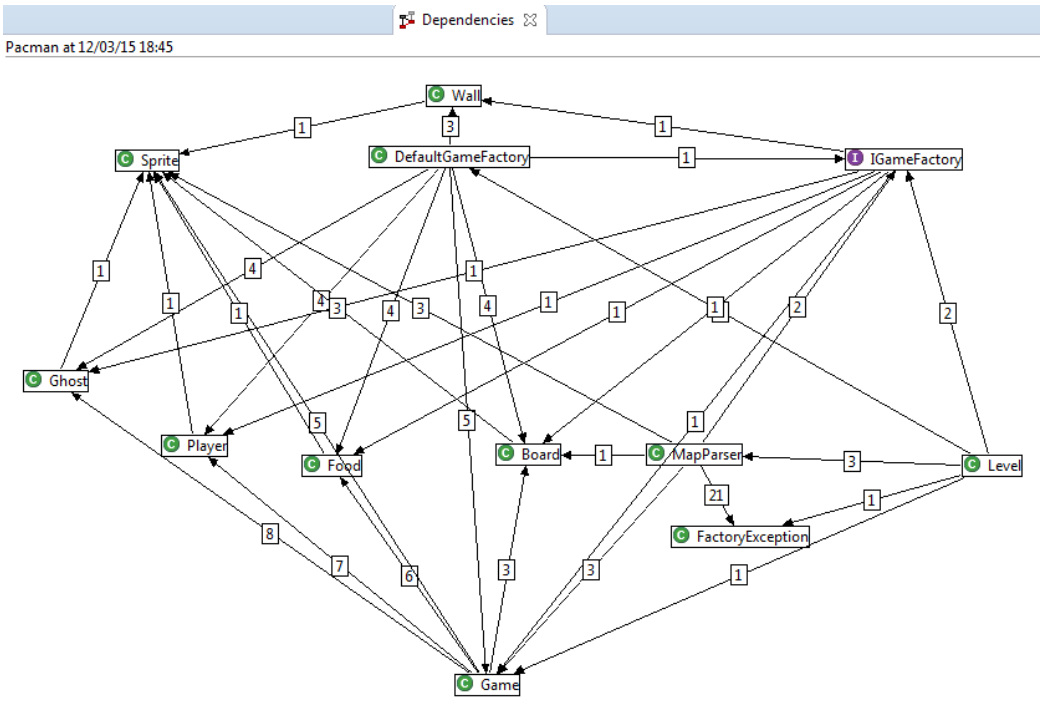
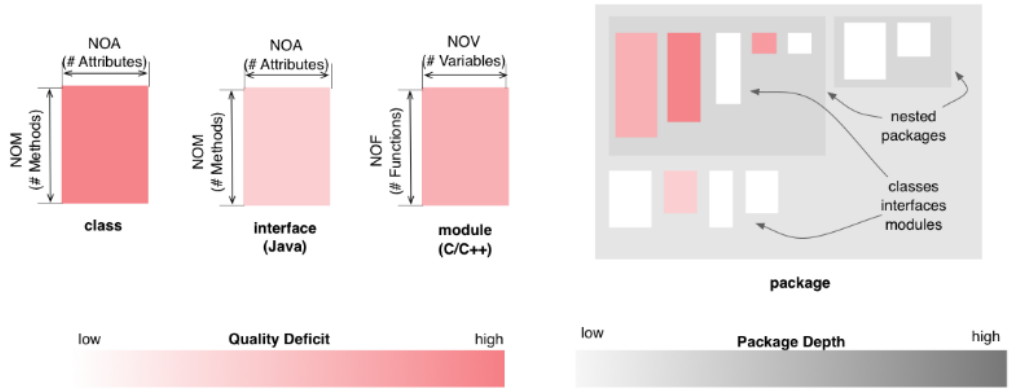


FIGURE 47 – Détail de l'analyse des dépendences cycliques entre le package Model et la package Factory.

C Annexe : Incode

Package Map - Design Flaws Perspective

The Design Flaws Perspective of the Package Map colors the classes, interfaces (Java) and modules (C and C++) based on the aggregated severity of all the design flaws affecting them. This coloring uses a white to red gradient, with darker shades of red for higher aggregated severity.



Entity selection

The user may select a class, an interface or a module in the map, in which case the selected entity is colored in green (with no borders). Everything else remains the same.

FIGURE 48 – Légende de l'outil InCode d'analyse de conception

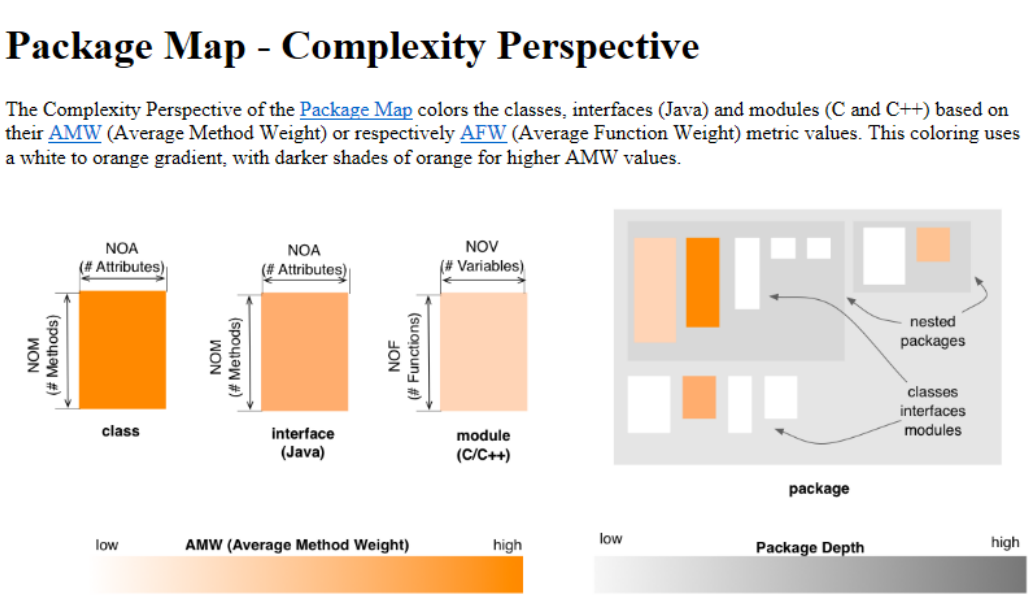


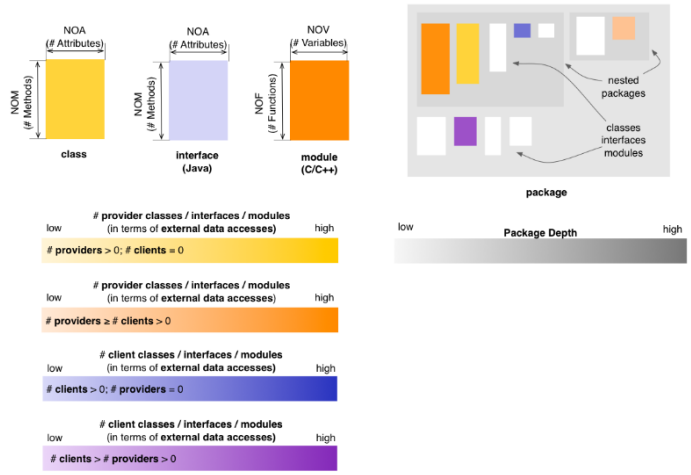
FIGURE 49 – Légende de l'outil InCode d'analyse de complexité

Package Map - Encapsulation Perspective

The Encapsulation Perspective of the Package Map provides insight into the way classes, interfaces (Java), or modules (C and C++) expose their data to external clients. In the default state, the Encapsulation Perspective will render classes, interfaces, and modules based on their predominant nature from the viewpoint of encapsulation, using four color gradients:

- if a class, interface, or module only accesses but does not itself expose data (i.e. it is a pure client), it is rendered in a shade of yellow
- if a class, interface, or module both exposes and itself accesses data from other classes, interfaces, or modules, it will be rendered in a color that depends on which aspect is predominant (i.e. mostly client shown in a shade of orange, or mostly provider shown in a shade of magenta)
- if a class, interface, or module only exposes but does not itself access data from other classes, interfaces, or modules (i.e. it is a pure provider) it is shown in a shade of blue

In this context the term "exposes data" means that the class, interface, or module has data that is either declared public, or accessible through a public accessor, and that there is at least one other class or module that accesses this data, either directly or through the provided accessor method. In other words, merely defining data as public is not considered as "exposing" that data, unless there is at least one client that actually accesses it.



Entity selection

The user may select a class, an interface or a module in the map, in which case the coloring of the map changes to reflect the encapsulation from the point of view of the selcted entity. The selected entity is colored in green (with no borders). Its collaborator classes, interfaces, and modules are colored using the four colors described below, based on their relation to the selected class, interface or module. In case of the Encpasulation Perspective, this relation is defined in terms of external data accesses. If a class, an interface, or a module has no relation to the selected entity, its coloring will be disabled.

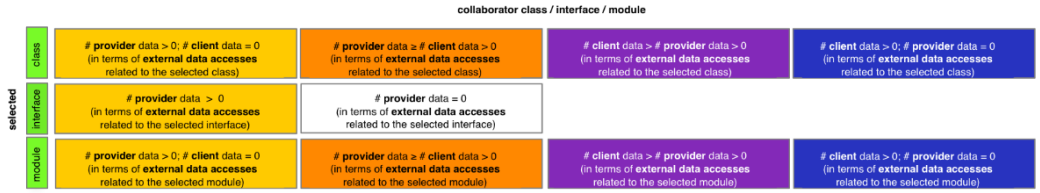
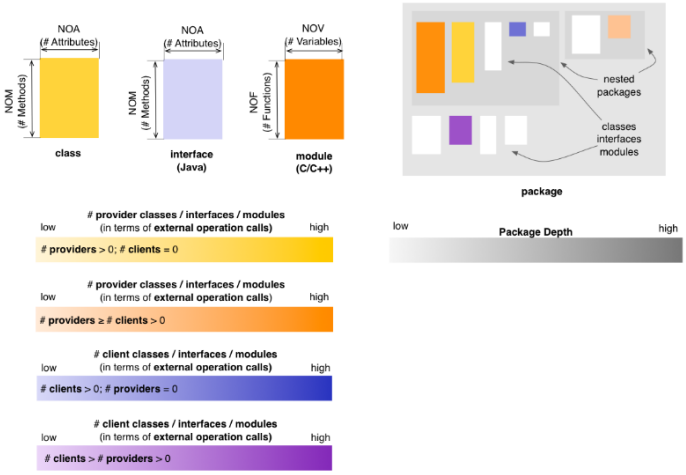


FIGURE 50 – Légende de l'outil InCode d'analyse d'encapsulation

Package Map - Coupling Perspective

The Coupling Perspective of the Package Map provides insight into the coupling that exists between classes, interfaces (Java), and modules (C and C++). In the default state, the Coupling Perspective will render classes, interfaces, and modules based on their predominant nature from the viewpoint of operation calls, using four color gradients:

- if a class or module only calls other operations but none of its operations are called (i.e. it is a pure client), it is rendered in a shade of yellow
- if a class or module both calls and its operations are called by other operations it will be rendered in a color that depends on which aspect is predominant (i.e. mostly client shown in a shade of orange, or mostly provider shown in a shade of magenta)
- if a class, interface, or module has its operations called by other operations and does not call other operations (i.e. it is a pure provider), it is shown in a shade of blue



Entity selection

The user may select a class, an interface or a module in the map, in which case the coloring of the map changes to reflect the coupling from the point of view of the selcted entity. The selected entity is colored in green (with no borders). Its collaborator classes or modules are colored using the four colors described below, based on their relation to the selected class, interface or module. In case of the Coupling Perspective, this relation is defined in terms of external operation calls. If a class, interface, or module has no relation to the selected entity, its coloring will be disabled.

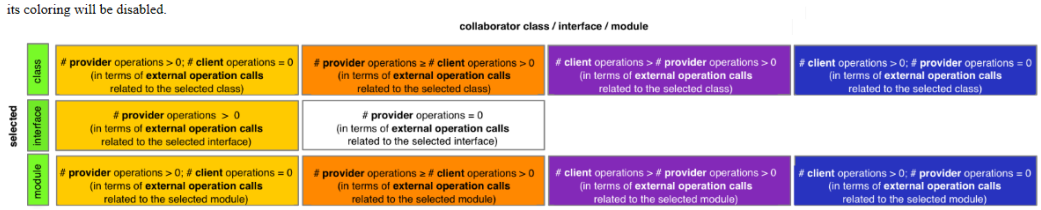


FIGURE 51 – Légende de l'outil InCode d'analyse de couplage

L'Aperçu Pyramide rassemble en un seul endroit les mesures les plus importantes sur un système orienté objet. Il se compose de trois parties, chacune quantifie un aspect important de la conception de systèmes orientés objet : la taille et la complexité, l'utilisation de l'héritage, et le couplage.
Le côté gauche de la pyramide aperçu (zone jaune) fournit des informations caractérisant la taille et la complexité du système. Ils comptent les unités les plus importantes de la modularité d'un système orienté objet, du plus haut niveau, jusqu'aux plus basses unités de mesures : • NOP (nombre total de packages définis dans le système) ; • CNP ou NOC (nombre total de classes définies dans le système, sans compter les classes de la bibliothèque) ; • NOM (nombre total de méthodes définies dans le système, y compris les méthodes et fonctions globales) ; • LOC (nombre total de lignes de code appartenant à l'exploitation) ; • CYCLO (somme des nombres cyclomatiques de toutes les opérations définies dans le système). Les chiffres indiqués à la gauche de ces paramètres sont calculés par un rapport entre les mesures directement placés en dessous et à droite. où par exemple le rapport NOM / CNP représente le nombre moyen de méthodes dans une classe.
La partie supérieure de la pyramide (la zone verte) est dédié à l'utilisation de l'héritage : • NDD (nombre moyen de descendants directs d'une classe, à l'exclusion des classes de la bibliothèque. Si une classe n'a pas de classes dérivées, alors la classe participe avec une valeur de 0) ; • HIT (moyenne de la métrique de HIT(= la longueur de trajet maximal d'une classe à sa plus profonde sous-classe) sur toutes les classes définies dans le système. Les classes autonomes sont considérés classes racines avec HIT = 0).
Le côté droit de la pyramide (la zone bleue) est dédié à l'aspect de couplage : • CALL (nombre total d'opérations distinctes d'appelle dans le système) ; • FOUT (somme de la métrique de FANOUT pour toutes les opérations définies dans le système)
Pour chaque rapport calculé, trois seuils sont calculés : • faible - bleu • Moyenne - vert • haute - rouge

D Annexe : Couverture par les test

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
Pacman	68,3 %	3.911	1.812	5.723
src/main/java	63,5 %	2.765	1.589	4.354
org.jpacman.framework.factory	59,3 %	254	174	428
DefaultGameFactory.java	68,1 %	77	36	113
FactoryException.java	0,0 %	0	9	9
MapParser.java	57,8 %	177	129	306
org.jpacman.framework.model	56,6 %	1.014	776	1.790
Board.java	34,7 %	166	313	479
Direction.java	94,0 %	79	5	84
Food.java	60,7 %	17	11	28
Game.java	86,8 %	197	30	227
Ghost.java	100,0 %	5	0	5
GhostMover.java	61,3 %	125	79	204
IBoardInspector.java	94,4 %	85	5	90
Level.java	59,8 %	49	33	82
Player.java	58,1 %	50	36	86
PointManager.java	53,4 %	63	55	118
Sprite.java	32,3 %	53	111	164
Tile.java	55,0 %	120	98	218
Wall.java	100,0 %	5	0	5
org.jpacman.framework.ui	61,7 %	884	549	1.433
ButtonPanel.java	69,4 %	202	89	291
MainUI.java	78,5 %	317	87	404
PacmanKeyListener.java	43,8 %	280	359	639
PointsPanel.java	85,9 %	85	14	99
org.jpacman.framework.view	87,2 %	613	90	703
Animator.java	100,0 %	38	0	38
BoardView.java	93,5 %	344	24	368
ImageLoader.java	77,8 %	231	66	297
src/test/java	83,7 %	1.146	223	1.369
org.jpacman.test.framework.accept	89,9 %	179	20	199
AbstractAcceptanceTest.java	83,1 %	98	20	118
MoveGhostStoryTest.java	100,0 %	81	0	81
org.jpacman.test.framework.factory	0,0 %	0	99	99
FactoryIntegrationTest.java	0,0 %	0	99	99
org.jpacman.test.framework.model	94,7 %	780	44	824
BoardTileAtTest.java	100,0 %	290	0	290
GameTest.java	88,5 %	339	44	383
PointManagerTest.java	100,0 %	51	0	51
SpriteTest.java	100,0 %	100	0	100
org.jpacman.test.framework.ui	75,7 %	187	60	247
MainUIFocusTest.java	100,0 %	45	0	45
MainUISmokeTest.java	100,0 %	36	0	36
MainUITest.java	63,9 %	106	60	166

FIGURE 52 – Détail de l'analyse de couverture du code par les tests unitaires.

Références