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 la réalisation 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'analyse dynamique 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 premiére version du code (section 2)
2. Ajout de tests unitaires à la première version du code (section 3), et vérification de la couverture des tests
3. Refactoring du code pour en améliorer la qualité et la structure (section 4)
4. Analyse des améliorations de qualité et tests de régression (section 5)
5. Extension du logiciel et ajout des tests unitaires pour cette extension (section 6)
6. Analyse de la qualité de cette extension et tests de régression (section 7)
7. Etude de l'historique de la qualité logicielle entre toutes les versions du code (section 8)

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 [1] Eclipse [2] et IntelijIdea [3] et les programmes CodePro [4], InCode [5] et VisualVM [6].

2.1 Enoncé

Le but de la première étape clé est d'analyser la qualité du logiciel pour avoir une première idée de ce qu'il faudra corriger si l'on désire améliorer la qualité. Cette première analyse est également l'occasion de comprendre l'architecture et la dynamique du système. Pour cette étape clé il est demandé de :

1. Déterminer les classes et les méthodes qui sont couvertes par des tests unitaires, et mettre en évidence les méthodes pour lesquelles de nouveaux tests unitaires devraient être créés prioritairement.
2. Analyser les performances du système en terme d'utilisation CPU et de consommation de mémoire. Repérez les parties du code créant un goulot d'étranglement et précisez les modules qui devraient être retravaillés afin de procéder à un déoulottage du système.

Décrivez la qualité globale du système. Quel serait, selon vous, le coût nécessaire à sa maintenance et à son évolution ?

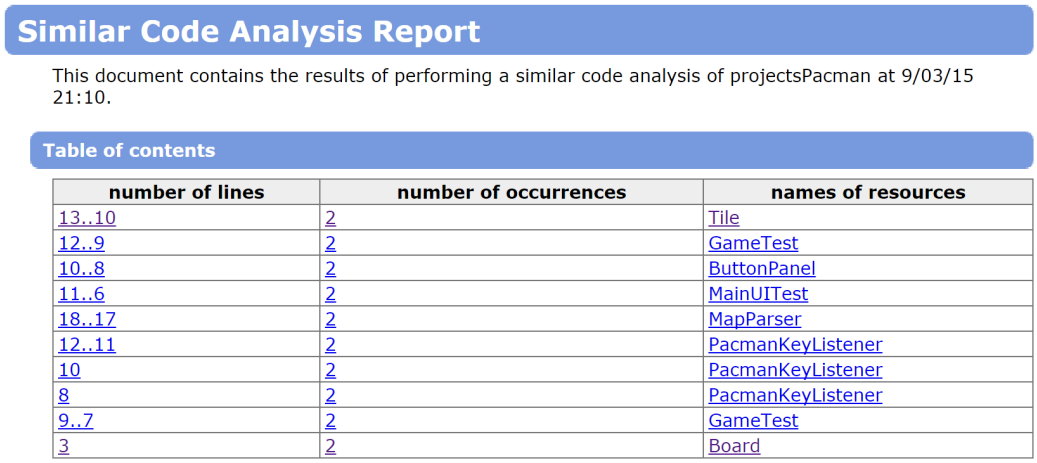
2.2 Résultat

2.2.1 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 mau-

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

vaise qualité parce que ça rend le code plus difficile à changer, à maintenir, à comprendre,...
Les solutiuons qui sont offertent par les languages 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 ana-



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

lyser cette métrique, nous avons utiliser le programme CodePro à partir de l'interface d'Eclipse (Eclipse -> CodePro Tools -> Find Similar Code).
Nous pouvons observer sur la figure 1 (page 6) que cet outils a détécté 10 blocs de code. Les figures de l'annexe A (page 34) 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.
Ce résultat n'est pas bon, mais on peut observer que les blocs se trouve chaque fois dans une même classe. Il sera donc probablement possible de créer des fonction pour chacun de ses cas.

2.2.2 Dépendences cycliques

Une dépendance cyclique peut-être appelée dépendance cyclique directe ou indirecte et elle a la même difinition 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 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 B. L'évolution de ses 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) méthodes si le soucis se situe entre deux classes), redécouper certaines 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 a

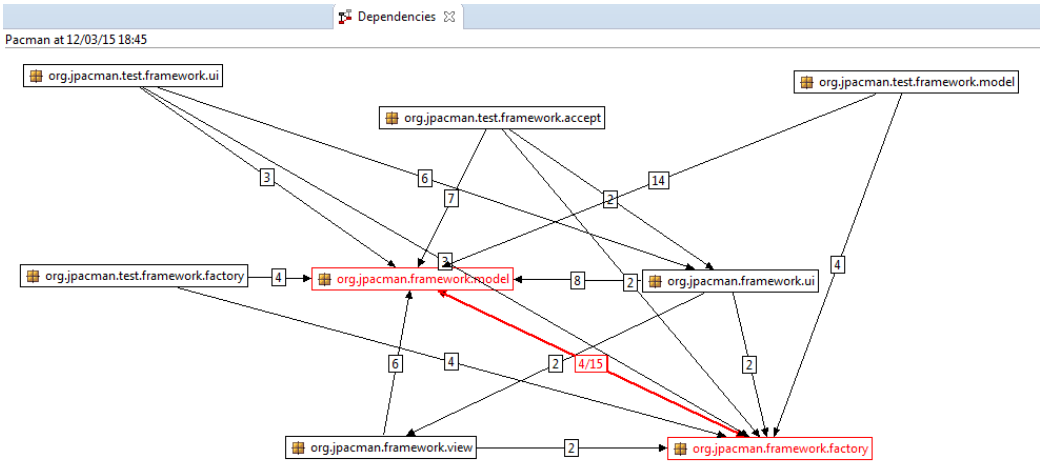


FIGURE 2 – Détail de l'analyse des dépendences cycliques des packages du projet.

aussi été visualisée à partir de l'outil CodePro depuis Eclipse (Eclipse -> CodePro Tools -> Analyse Dependencies). On peut observer sur la figure 2 (page 7) et la figure 3 (page 8) qu'il existe des dépendences 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 38) contient toutes les autres visualisations qui n'ont pass révélé de problème de dépendances.

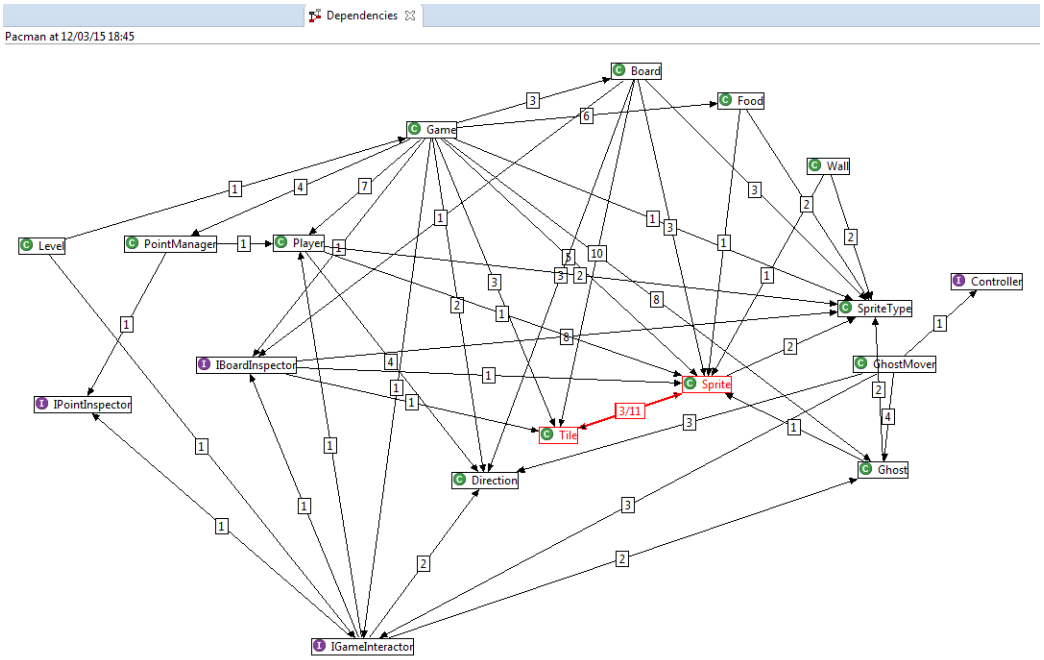


FIGURE 3 – Détail de l'analyse des dépendences cycliques du package Model.

2.2.3 Code inutile

Du code inutile, aussi appelé "Dead Code", correspond à des lignes de code qui sont compilée mais qui ne sont jamais utilisée. C'est fréquement du code qui a été utile à une fonctionnalité et lorsque cette fonctionnalité à 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'éxécution.
La solution est généralement de supprimer ses lignes de code. L'outil utilisé reste CodePro depuis Eclipse (Eclipse -> CodePro Tools -> Find dead code).
Attention tout de même à ne pas tout supprimer sans réfléchir, en effet, on observer sur la figure 4 (page 9) que les packages contenant les test sont considéré comme inutile. Ils le sont en effet lors de l'éxécution 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 présente dans les classes qui étendent (directement ou indirectement) la classe "Sérializable". Leur valeur est indispensable pour des applications qui transittent par le réseau mais leur existence ne peut causer aucun tort. Pour ce qui est des fichiers "package-info.java", il sont a nouveau inutule lors de l'exécution du code, mais permettent de contenir les

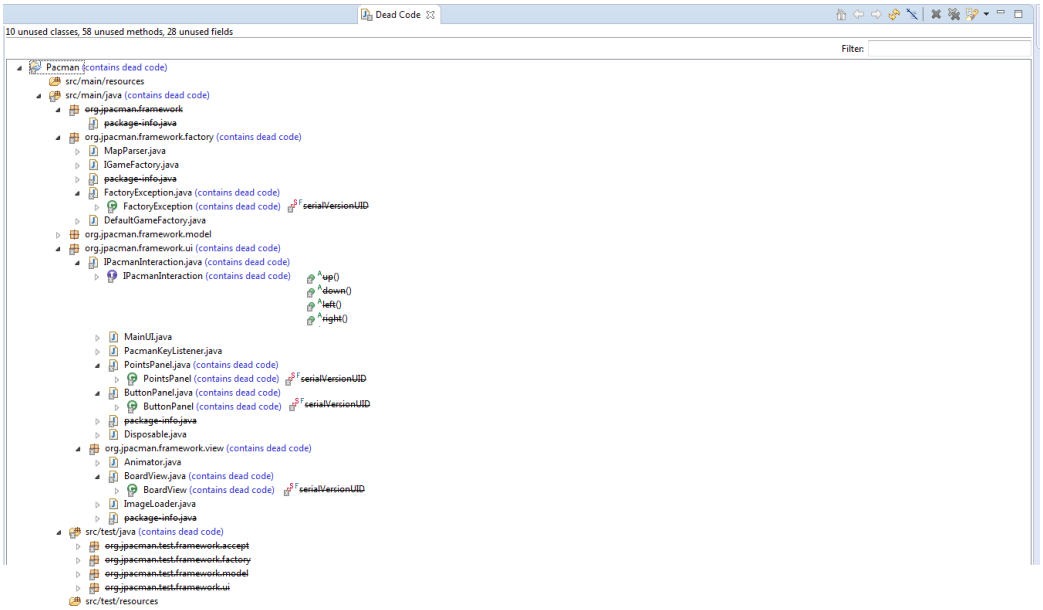


FIGURE 4 – Détail de l'analyse des parties de code non utilisé lors de l'éxécution du programme.

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 font référence à des méthodes complètes qui ne sont jamais appelées.
La solution la plus rapide, est simplement de supprimer ses 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 mettres ces fonctions en commentaire afin de ne pas devoir réécrire ces fonctions.
Les modifications à éffectuées, bien que nombreuses, sont donc mineures.

2.2.4 Javadoc

La javadoc est une documentation standard au format HTML pour les programmes developpé 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 classique pour constituer un code suffisemment documenter à la compréhension).
Grace à l'outil CodePro d'Eclipse, il a été observé (non illustré parce que toutes les classes sont a revoir) que en règle générale, les classes sont documentée. 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 commentée et les méthodes issues d'une interface (sous l'annotation "@Overhide").

2.2.5 Test unitaire

Cet analyse sera détaillée dans la section suivante 3.

2.2.6 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 11) dont la légende se trouve à l'annexe 31 (page 42) La figure 5 (page 11) illistre 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 utilisée par des groupes disjoints de classes de clients utilisant des fragments disjoints de la classe.
Plusieurs solution sont possible pour résoudre ce genre de problème :

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

2.2.7 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 12) dont le détail de la légende se trouve à l'annexe 32 (page 43) Cette figure met en avant la classe PacmanKeyListener qui encourt la plus forte complexité et met un

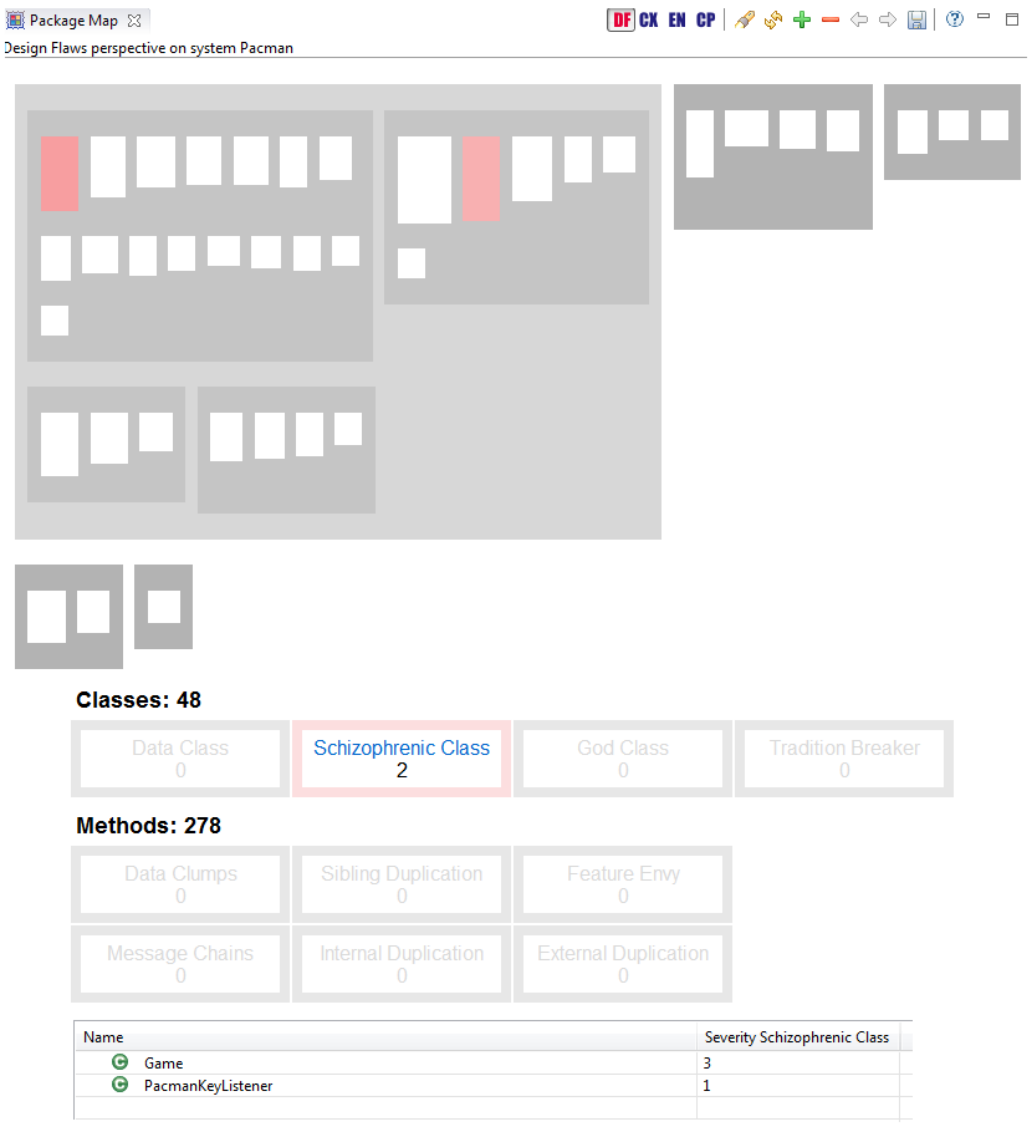


FIGURE 5 – Analyse de la conception du programme (sa structure) par InCode.

attention sur les classes BoardView et MapParser (à cause de leur grand nombre d'attributs).

2.2.8 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 13) dont le détail de la légende se trouve à l'annexe 33 (page 44) Cette figuer contient beaucoup

FIGURE 6 – Analyse de la complexité par InCode.

d'information, nous en retiendrons quelques unes :

- 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[7] (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 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.

2.2.9 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 14) dont le détail de la

7. >provider

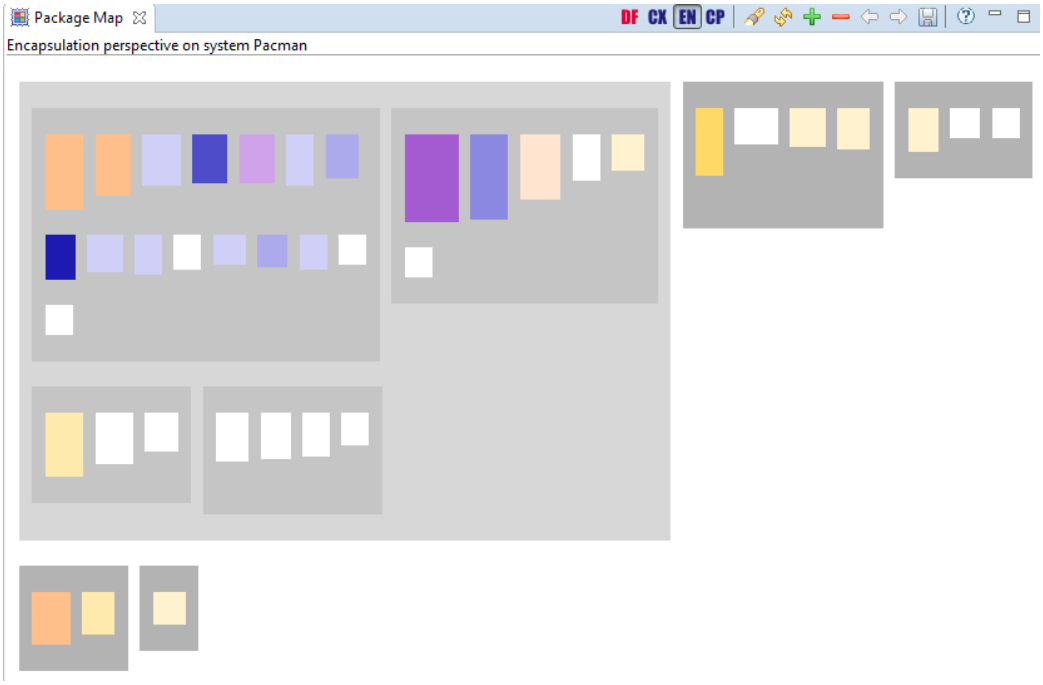


FIGURE 7 – Analyse de la complexité par InCode.

légende se trouve à l'annexe 34 (page 45) Cette figuer contient aussi beaucoup d'information, 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-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" a aucun fournisseur et beaucoup de client 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".

8. >provider

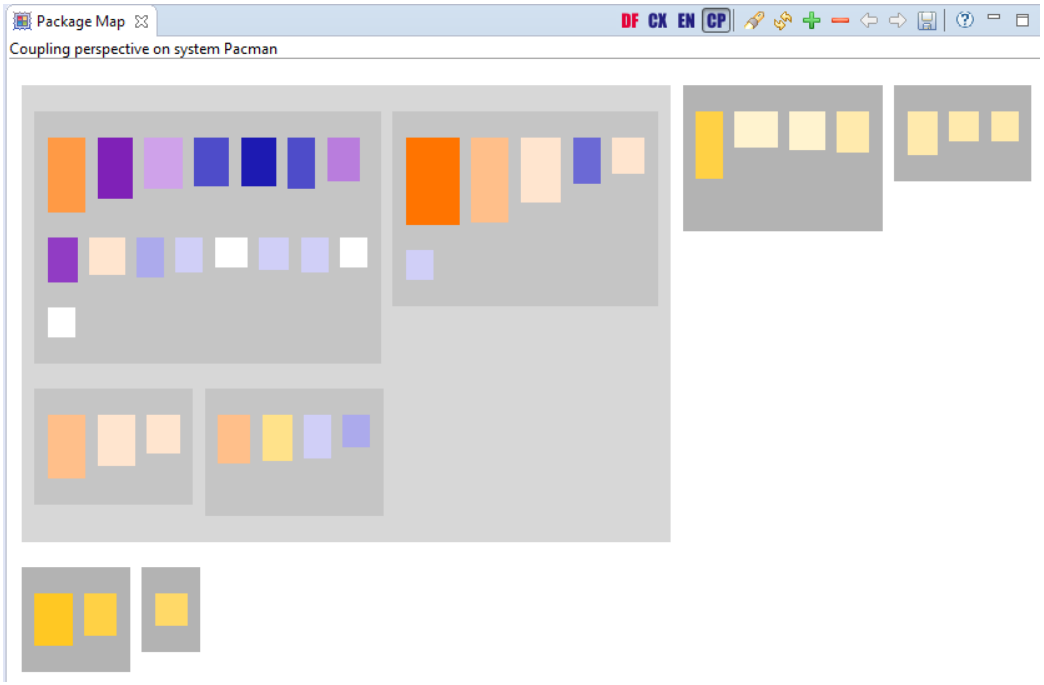


FIGURE 8 – Analyse de la complexité par InCode.

2.2.10 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 15) dont le détail de la légende se trouve à l'annexe C (page 46). Comme le précise l'interprétation de la figure 9 (page 15), L'arbre que constitue les classe est grand et large.
Les classes ont tendance à contenir un nombre moyen de méthodes et à être organisés 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).

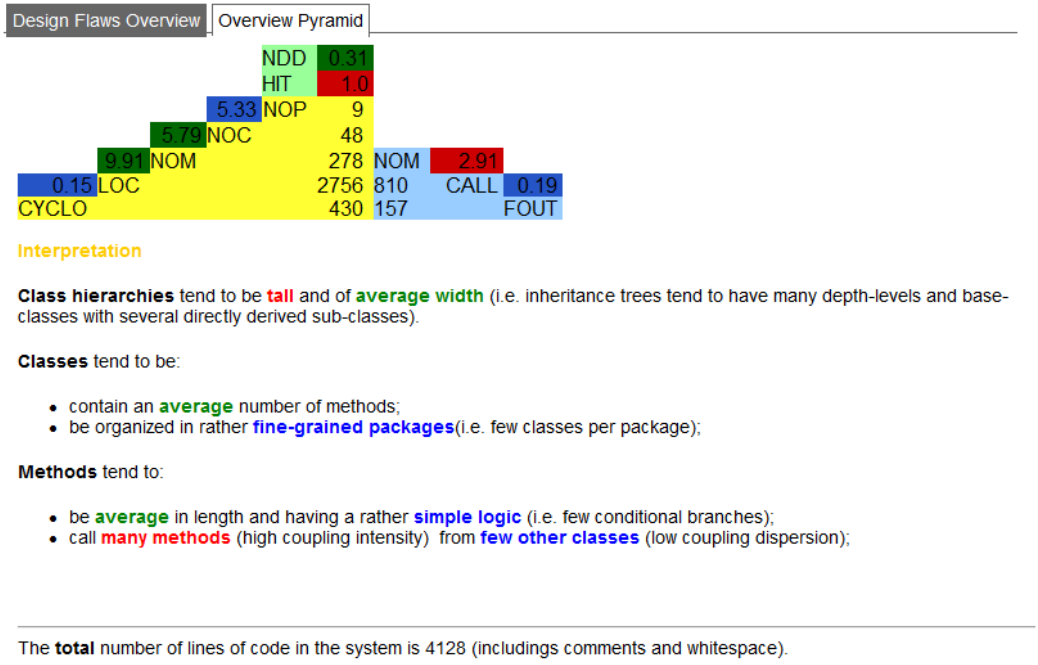


FIGURE 9 – Pyramide des valeurs calculée par InCode.

2.2.11 Métriques

Grace à l'outil de calcul des métriques d'Eclipse, on peut observer les résulats présent à la figure 10 (page 17), la figure 13 (page 21) et la figure 14 (page 22).

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 11 (page 18) et la figure 12 (page 19), il sera nécéssaire lors du refactoring (voir 4) d'analyser et de voir s'il sera possible de la diminuer.

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).

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 11 – 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 12 – Détails de l'analyse de la complexité cyclique de la classe "BoardView".

Nombre moyen de méthodes par type

C'est la moyenne du nombre de méthodes définies pour chaque type défini dans les éléments cibles. On remarque que certainnes classes ont trop de méthodes. !!!!!!!!!!!!!!!!!!!!!!!!!!!!

⊟ 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 13 – Détails de l'analyse des métriques du projet (partie2).

⊞ 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 14 – Détails de l'analyse des métriques du projet (partie3).

2.2.12 Audit

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

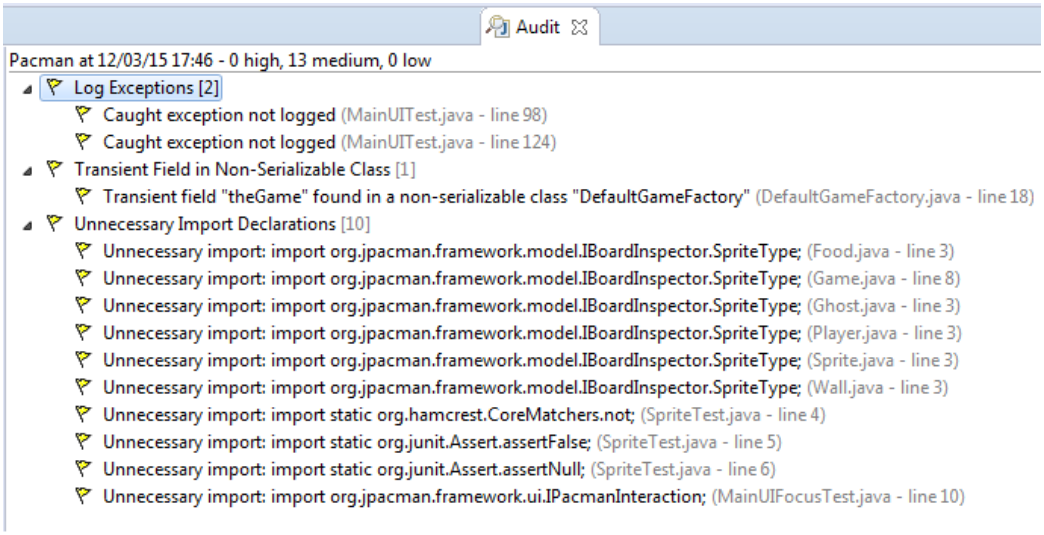


FIGURE 15 – Détail de l'analyse d'audit faite par Eclipse.

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 repéré sont encédré par un bloc "try...catch" mais ils ne capturent qu'un type d'exception. Pour résoudre ce problème il suffira d'ajouter un second bloc "catch" qui prenne en charge l'ensemble des autres exceptions.

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 contraite, 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".

Import inutile

Les imports sont les premières lignes d'une classe. Ils permettent d'informer au compilateur d'où viennent les méthodes, les objets,... utilisées qui ne sont pas créé au sein de la classe courante. Egalement repris plus loin dans les warnings d'Eclipse, les classes comportent plusieurs imports qui ne sont jamais utilisé. La meilleur solution est simplement de les supprimer.
Voici celles détectées par l'outil Code Inspector d'IntelijIdea :

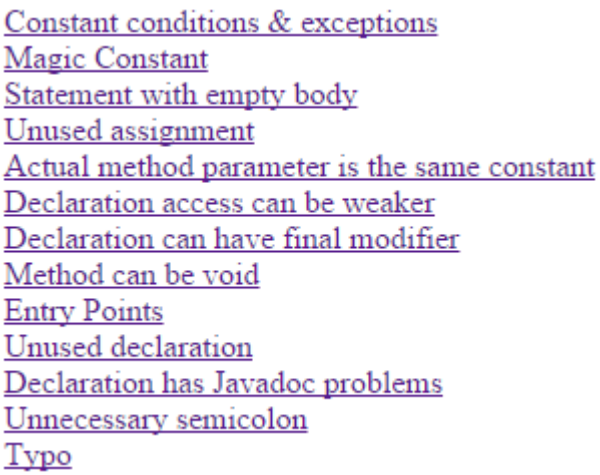


FIGURE 16 – Détail de l'analyse d'audit faite par IntelijIdea.

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]) :

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

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

se trouve.

- N'est pas accessible depuis n'importe quelle class 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 grave problème à l'heure actuelle puisque le jeux fonctionne correctement mais lors de l'ajout de fonctionnalité cela peut le devenir.

Optimisation

En java, une variable qui ne doit pas être modifié 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.

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 transformé 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 miseen paramètre en vue d'une amélioration future.

Code inutile

Ce sujet a déjà été traité plus haut (Voir 2.2.3).

Javadoc

Ce sujet a déjà été traité plus haut (Voir 2.2.4).

Caractère inutile

Ce sont des élément 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.

Typographie

Les problèmes de typographies reprennent les erreurs de formatage des différents éléments. Ce ne sont des erreurs que de conventions parce que elle ne change 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 permit 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 attention 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

2.2.13 Profilage

Ressources CPU

Comme visible sur la figure ?? (page ??), 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 chagenements 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 nécéssaire que ce score soit toujours à jour. Il n'est donc probablement pas possible d'accélérer l'application.

Consommation mémoire

Les ressources mémoires utilisées par l'application sont visible dans la seconde moitier de la figure ?? (page ??). Les observation 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.

3 Etape 2 : Ajout de tests unitaires

3.1 Enoncé

Votre première analyse a révélé la présence de certains problèmes de qualité du code de l'application. Avant d'envisager la correction de ces problèmes, il faut s'assurer que les modifications que vous apporterez au code source ne modifieront pas le comportement du logiciel. Etendez et complétez le jeu de tests unitaires fourni avec le code source afin de vous prémunir d'une telle modification. Effectuez également une analyse de couverture de tests. Quelle garantie avez-vous que vos futures modifications ne pourront pas casser le système ?

3.2 Résultat

Tests Eclipse -> Coverage As -> JUnit Test : La figure annexe D (page 47) montre que le projet est couvert à 68.3% avec en particulier, 63.5% pour la package *main* et 83.7% pour le package *test*. Il est donc important d'ajouter des tests sur les classes : FactoryException, Board, Sprite et PacmanKeyListener qui sont sous le seuil des 50% de couverture.

4 Etape 3 : Refactoring en vue d'améliorer la qualité

4.1 Enoncé

Avec les tests unitaires ajoutés dans l'étape précédente, vous pouvez vérifier automatiquement (jusqu'à un certain point) la préservation du comportement du logiciel. Réalisez les modifications nécessaires à l'amélioration de la qualité et la structure du logiciel. Vos ressources et votre temps étant limités, commencez par établir les modifications devant être réalisées en priorité. Sur base de quels critères réalisez-vous cette priorisation ? Refactorisez progressivement votre code, en vous assurant systématiquement que tous les tests déjà présents s'exécutent avec succès. Souvenez-vous que vos modifications doivent améliorer la qualité du code, et non étendre ou modifier le comportement du logiciel.

4.2 Résultat

5 Etape 4 : Analyse de la qualité du logiciel

5.1 Enoncé

Réalisez une étude similaire à celle décrite en Section ... La qualité du logiciel s'est-elle améliorée ? Les problèmes les plus critiques ont-ils été résolus ? Au vu de cette seconde analyse, quels sont les points qui devraient à présent être améliorés ?

5.2 Résultat

6 Etape 5 : Extensions

6.1 Enoncé

Il vous est demandé d'étendre le logiciel afin d'y ajouter certaines fonctionnalités ou d'en améliorer la qualité. Chaque équipe doit réaliser au moins deux extensions diférentes, décrites dans la section ... . Utilisez un processus de développement dirigé par les tests (test-driven development) : lors du développement des extensions, ajoutez de nouveaux tests unitaires pour tester le comportement prévu de l'extension. Effectuez également des tests de régression avec les tests unitaires déjà présents, afin de vous assurer que le comportement initial n'a pas été modifié.

6.2 Résultat

7 Etape 6 : Analyse de la qualité du logiciel

7.1 Enoncé

Pour chaque extension ajoutée, réalisez une analyse de qualité similaire à celle décrite en Section ... Au vu de cette analyse, quels sont les points qui devraient à présent être améliorés ?

7.2 Résultat

8 Etape 7 : Analyse de l'évolution de la qualité logicielle

8.1 Enoncé

Analysez l'évolution de la qualité du logiciel entre les diférentes versions, en utilisant les résultats d'analyse de qualité des sections ..., ... et ... . Montrez cette évolution graphiquement et interprètez-la.

8.2 Résultat

Ressources CPU

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

Ressources CPU avec filtre "org.jpacman"

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

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

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

9 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 reppresentant tous les blocs identifié se trouve à la sous-section 2.2.1

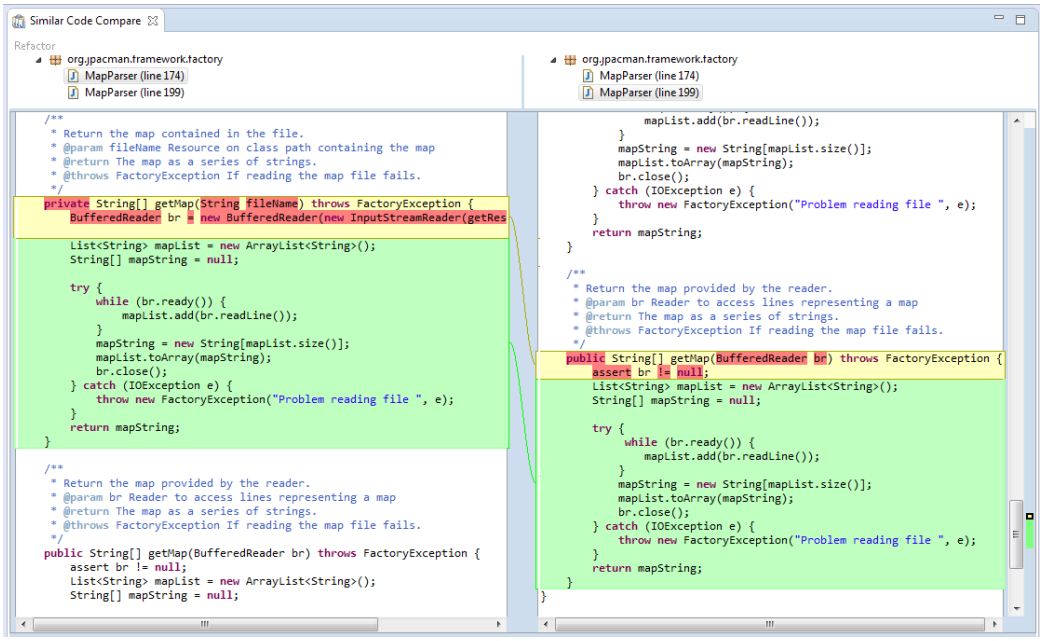


FIGURE 18 – Détail de l'analyse de code redondant par CodePro

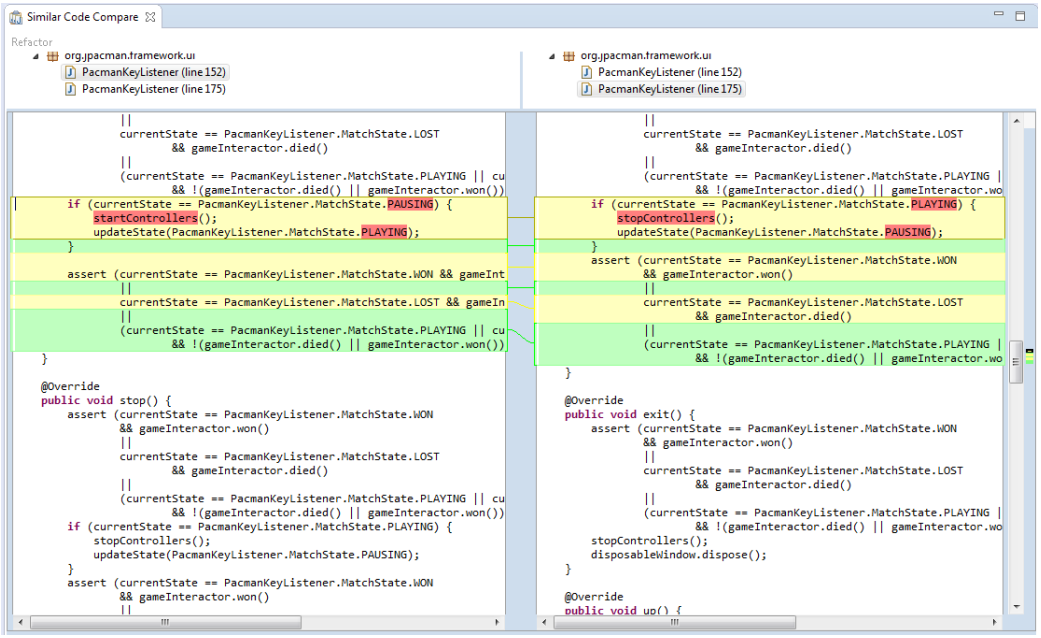


FIGURE 19 – Détail de l'analyse de code redondant par CodePro

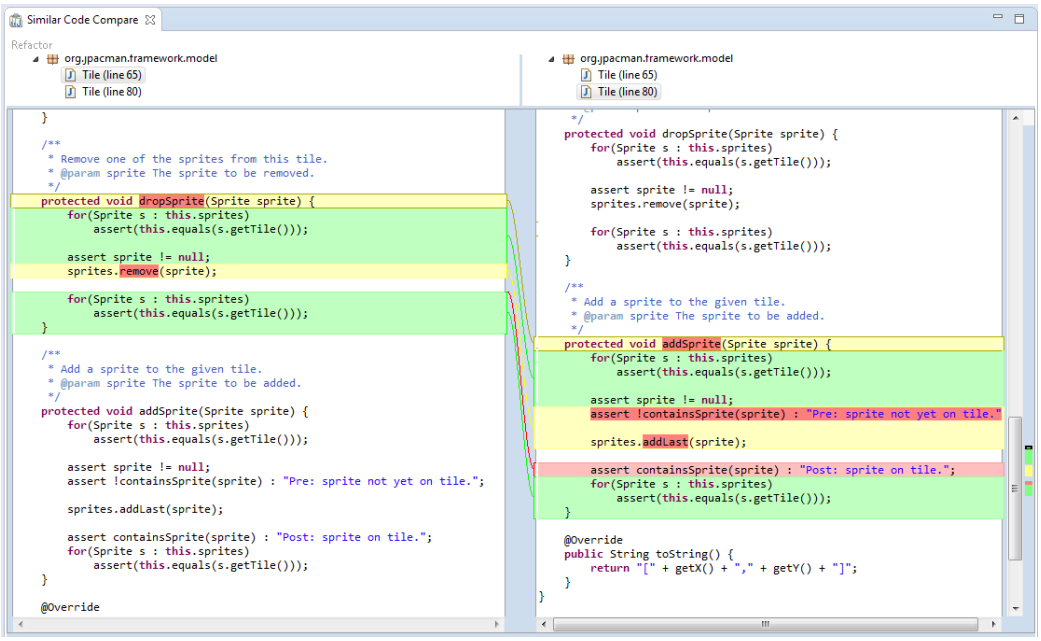


FIGURE 20 – Détail de l'analyse de code redondant par CodePro

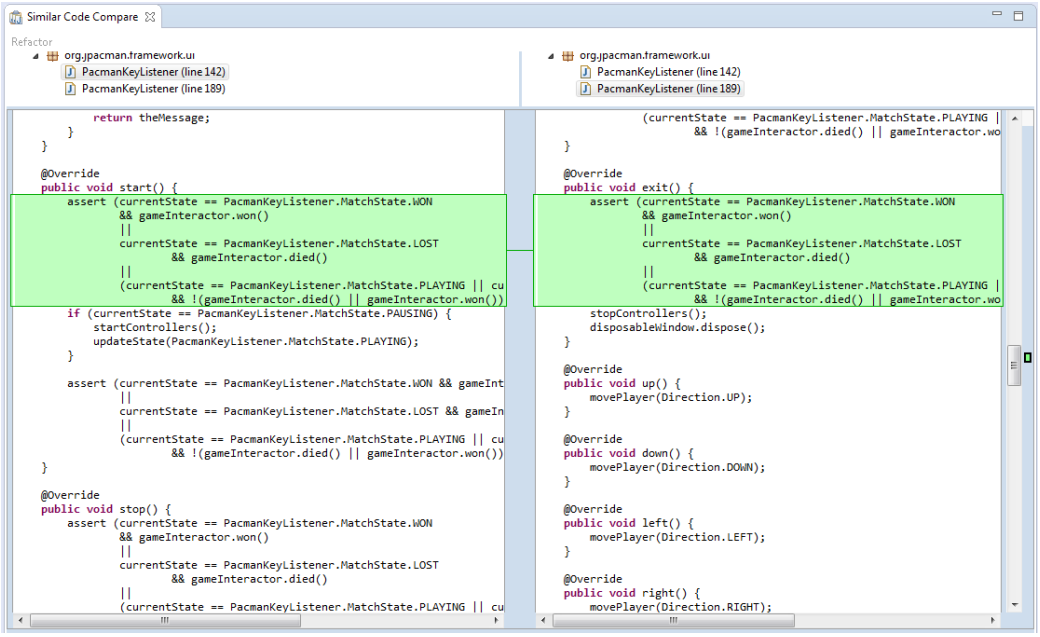


FIGURE 21 – Détail de l'analyse de code redondant par CodePro

FIGURE 22 – Détail de l'analyse de code redondant par CodePro (élément dont la visualisation n'est pas possible dans Eclipse)

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 trouve à la sous-section 2.2.2.

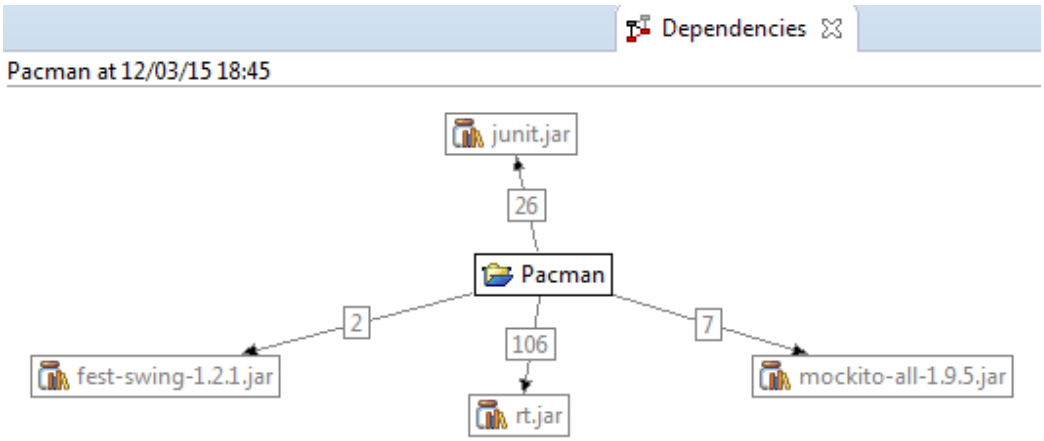


FIGURE 23 – Détail de l'analyse des dépendences cycliques du projet.

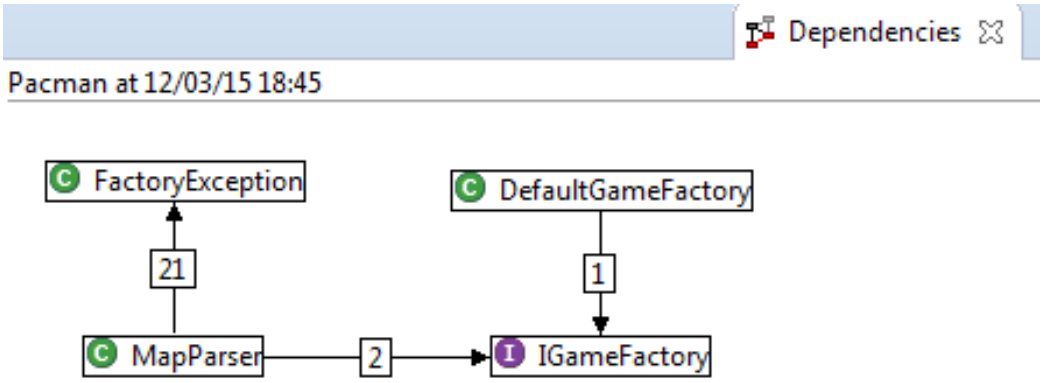


FIGURE 24 – Détail de l'analyse des dépendences cycliques du package Factory.

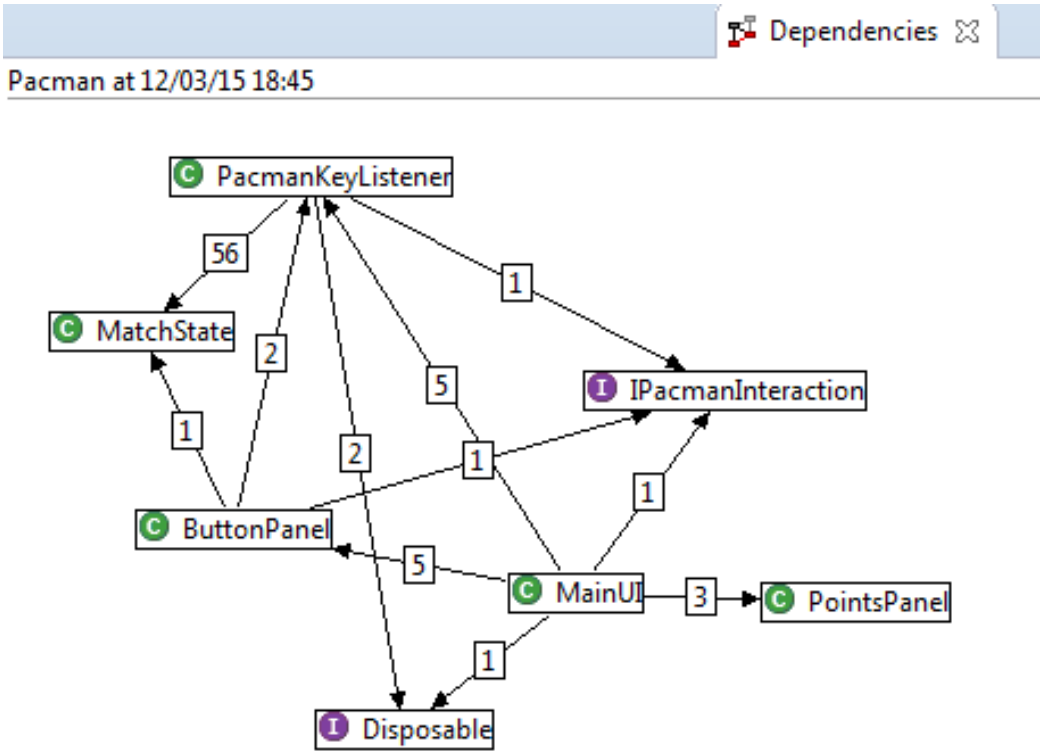


FIGURE 25 – Détail de l'analyse des dépendences cycliques du package UI.

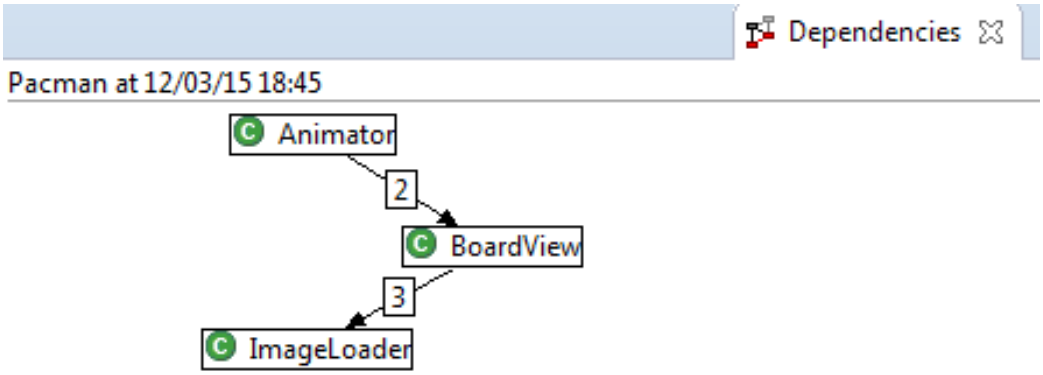


FIGURE 26 – Détail de l'analyse des dépendences cycliques du package Vieuw.

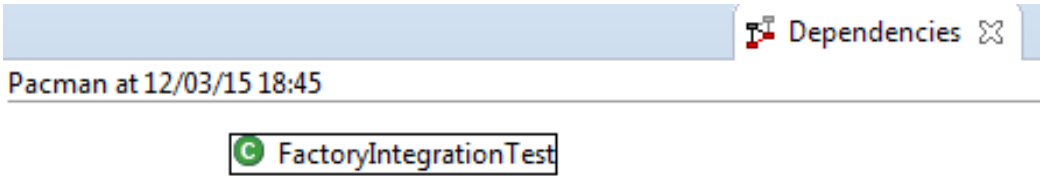


FIGURE 27 – Détail de l'analyse des dépendences cycliques du package Factory (Test).

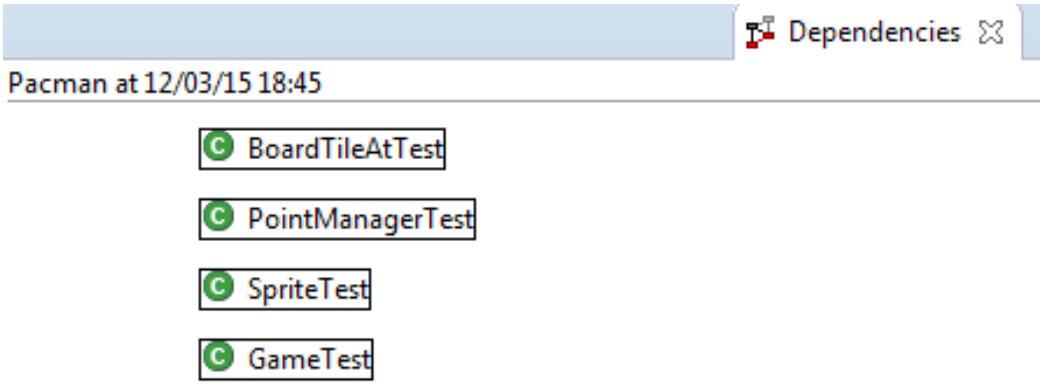


FIGURE 28 – Détail de l'analyse des dépendences cycliques du package Model (Test).

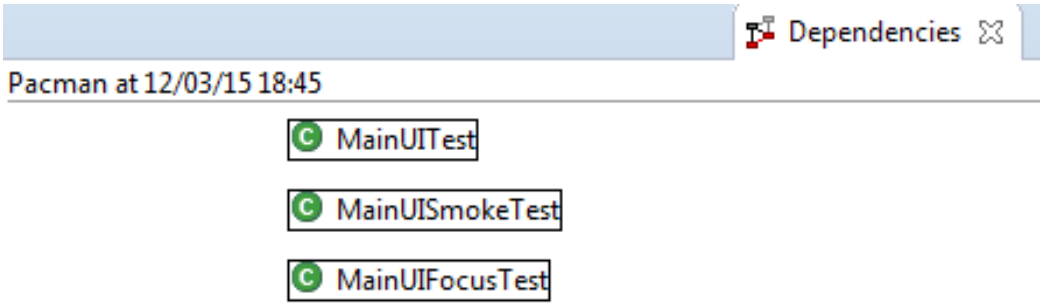


FIGURE 29 – Détail de l'analyse des dépendences cycliques du package UI (Test).

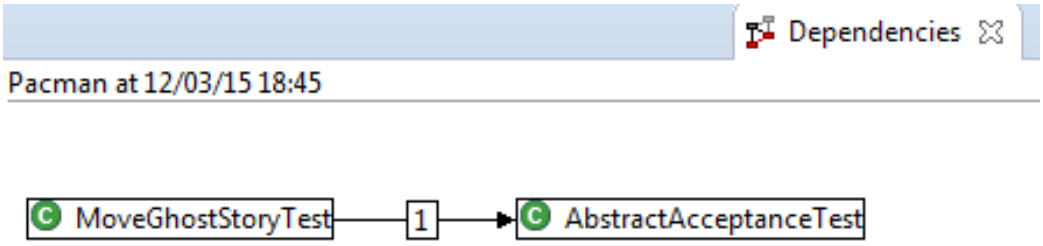


FIGURE 30 – Détail de l'analyse des dépendences cycliques du package Accept (Test).

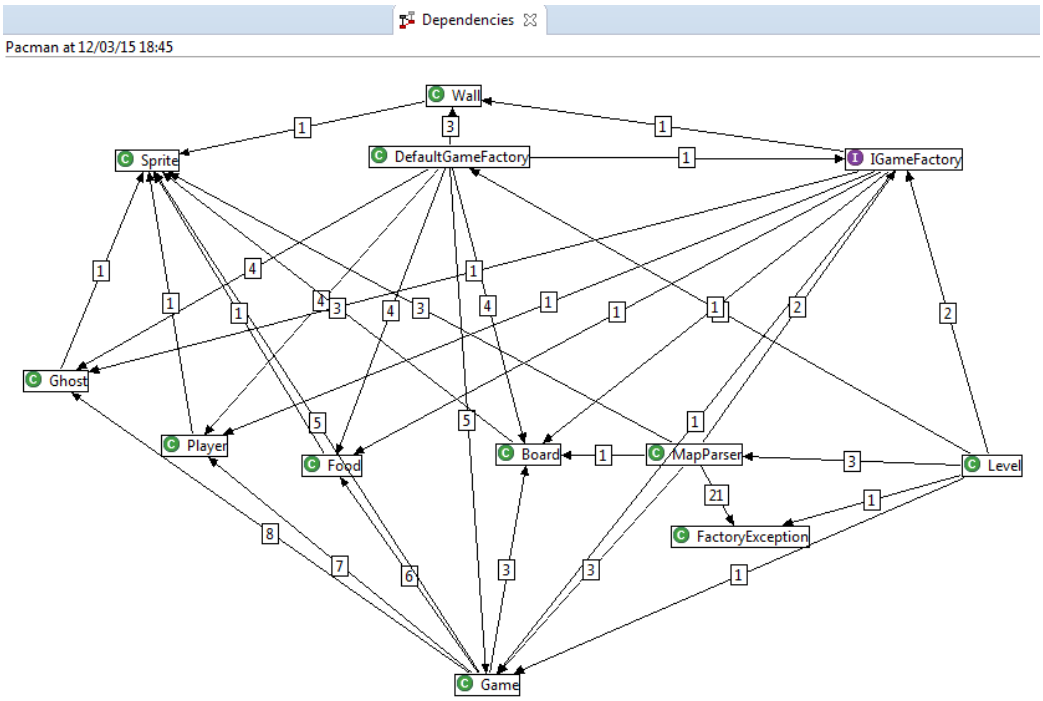
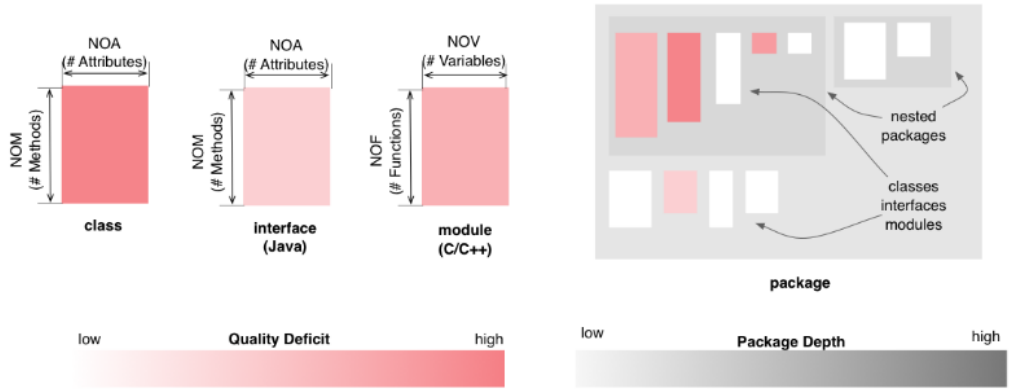


FIGURE 31 – Détail de l'analyse des dépendences cycliques entre le package Model et la package Factory.

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.



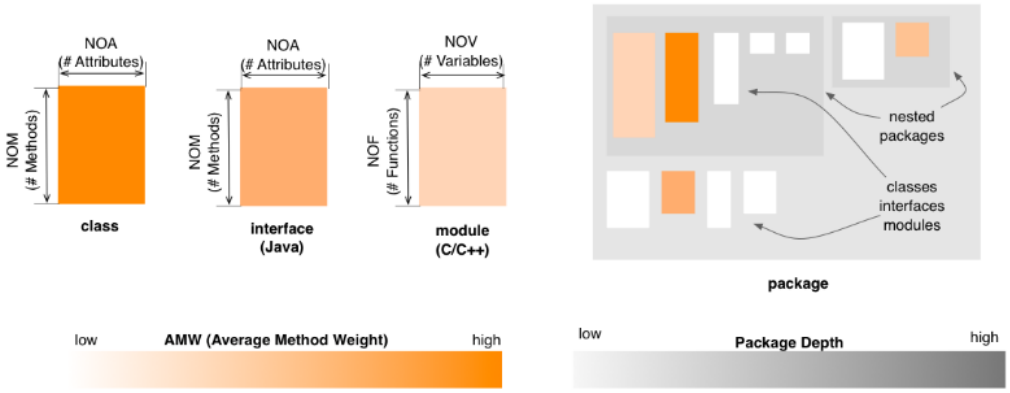
Entity selection

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 32 – Légende de l'outil InCode d'analyse de conception

Package Map - Complexity Perspective

The Complexity Perspective of the Package Map colors the classes, interfaces (Java) and modules (C and C++) based on their AMW (Average Method Weight) or respectively AFW (Average Function Weight) metric values. This coloring uses a white to orange gradient, with darker shades of orange for higher AMW values.



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.

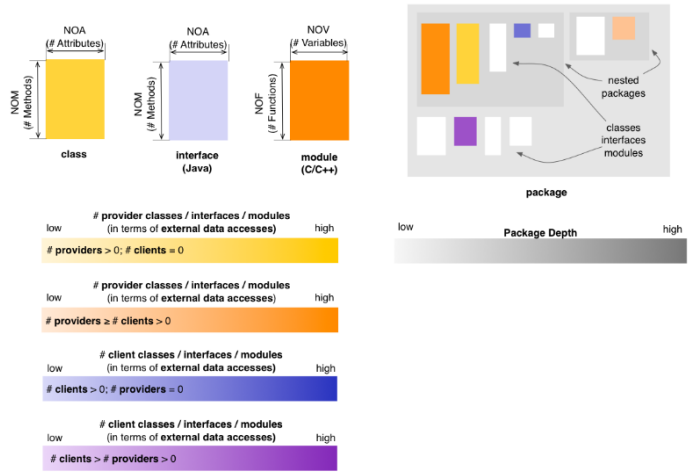
FIGURE 33 – Légende de l'outil InCode d'analyse de complexité

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.

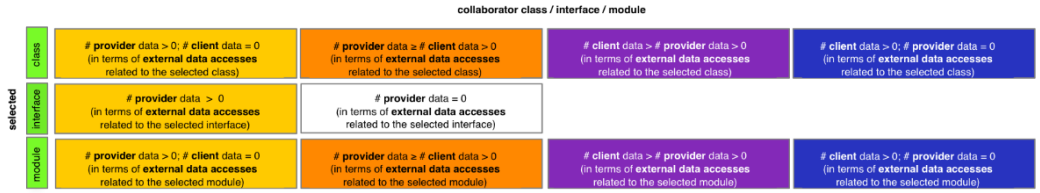
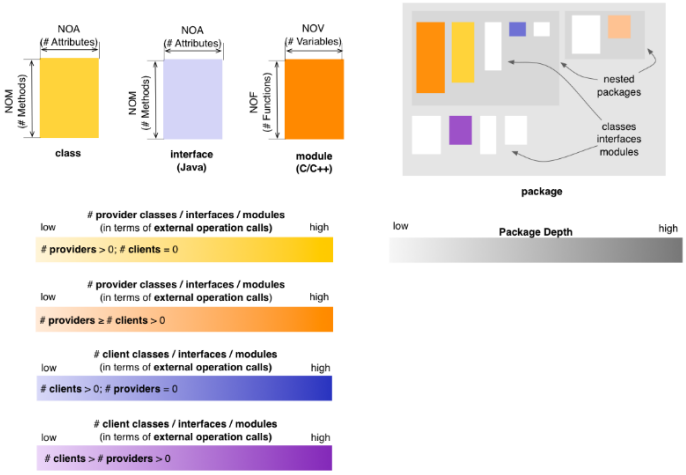


FIGURE 34 – Légende de l'outil InCode d'analyse d'encapsulation

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.

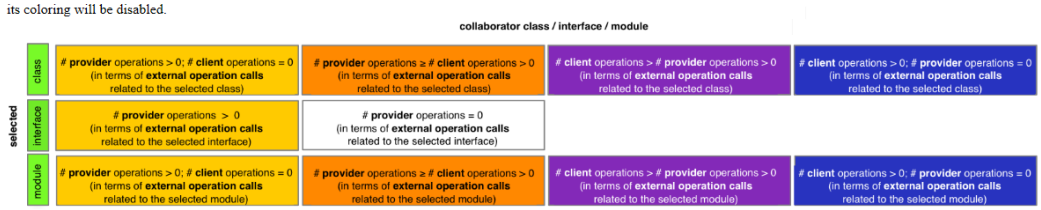


FIGURE 35 – Légende de l'outil InCode d'analyse de couplage

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

Références