

## 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

# Table des matières

<b>Résumé</b>	<b>3</b>
<b>1 Introduction</b>	<b>4</b>
1.1 Problème posé . . . . .	4
1.2 Etapes clés . . . . .	4
<b>2 Etape 1 : Première analyse de la qualité du logiciel</b>	<b>5</b>
2.1 Enoncé . . . . .	5
2.2 Résultat . . . . .	6
<b>3 Etape 2 : Ajout de tests unitaires</b>	<b>7</b>
3.1 Enoncé . . . . .	7
3.2 Résultat . . . . .	8
<b>4 Etape 3 : Refactoring en vue d'améliorer la qualité</b>	<b>8</b>
4.1 Enoncé . . . . .	8
4.2 Résultat . . . . .	8
<b>5 Etape 4 : Analyse de la qualité du logiciel</b>	<b>8</b>
5.1 Enoncé . . . . .	8
5.2 Résultat . . . . .	8
<b>6 Etape 5 : Extensions</b>	<b>8</b>
6.1 Enoncé . . . . .	8
6.2 Résultat . . . . .	9
<b>7 Etape 6 : Analyse de la qualité du logiciel</b>	<b>9</b>
7.1 Enoncé . . . . .	9
7.2 Résultat . . . . .	9
<b>8 Etape 7 : Analyse de l'évolution de la qualité logicielle</b>	<b>9</b>
8.1 Enoncé . . . . .	9
8.2 Résultat . . . . .	9
<b>9 Annexes</b>	<b>11</b>
<b>A Annexe : 1</b>	<b>11</b>
<b>B Annexe : .....</b>	<b>15</b>
<b>C Annexe : .....</b>	<b>17</b>

## TABLE DES MATIÈRES

---

<b>D Annexe : .....</b>	<b>22</b>
<b>E Annexe : .....</b>	<b>24</b>

## 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.

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

## 2 Etape 1 : Première analyse de la qualité du logiciel

### 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. Calculer la valeur de plusieurs métriques logicielles permettant d'estimer la qualité du logiciel, d'interpréter les résultats des métriques, et de mettre en évidence les modules (packages, classes ou méthodes) qui devraient être traités prioritairement afin d'améliorer leur qualité ainsi que la raison pour laquelle ils sont prioritaires.
2. 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.
3. Répertorier les portions de code qui ne sont pas utilisées et qui pourraient donc être supprimées sans altérer le comportement du système.
4. Répertorier les portions de code qui sont redondantes (code dupliqué) et qui pourraient donc être éliminées par une restructuration du système sans altérer son comportement.
5. Détecter la présence de bad smells. En trouvez-vous une plus forte concentration dans certains modules ?
6. 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 débouclage 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

Voici le tableau des résultats de l'analyse des différentes métriques :

Métrique	Outil utilisé + Résultat Interprétation des résultats Modules à traiter prioritairement + justification	Technique
Code dupliqué	Eclipse -> CodePro Tools -> Find Similar Code : 10 blocs de code (visible sur la figure A (page 11) et les suivantes) sont semblable. Ce résultat n'est pas alarmant mais mérite d'être refactorisé. Les classes Tile.java, GameTest.java, ButtonPanel.java, MainUITest.java, MapParser.java, PacmanKeyListener.java, PacmanKeyListener.java, PacmanKeyListener.java, GameTest.java, Board sont à traiter puisque c'est elle qui contiennent du code dupliqué.	
Respect des conventions de codage		
Dépendences cycliques	Eclipse -> CodePro Tools -> Analyse Dependencies : il y a des dépendances	
Performances		
Structure du code		
Style du code		
Design flaws		
Antipattern		
Test		
Dataflow		
Documentation		
		Outil
Profilage		
Couverture du code	Eclipse -> CodePro Tools -> Find dead code : Le résultat (visible à l'annexe B)	
Tests	Eclipse -> Coverage As -> JUnit Test : La figure figure E (page 24) montre la couverture des tests. Il est donc important d'ajouter des tests sur les classes : FactoryException, GameTest, Board, Tile, ButtonPanel, MainUITest, MapParser, PacmanKeyListener.	

Il est important de remarquer aussi, que sur base des warnings détectés par Eclipse, tout warning confondu :

- 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

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		MainUIFocusTest.java	1

NB : - slide 4 : outils

Images : - *Mtrique<sub>0</sub>.png* -

## 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

## 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 différentes, décrites dans la section ... . Utilisez un processus de développement dirigé par les tests (test-driven development) : lors du

## 8 ETAPE 7 : ANALYSE DE L'ÉVOLUTION DE LA QUALITÉ LOGICIELLE

---

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 différentes versions, en utilisant les résultats d'analyse de qualité des sections ..., ... et ... . Montrez cette évolution graphiquement et interprétez-la.

### 8.2 Résultat

Quelque exemple d'utilisation. *"Un peu d'italique"* **Du Gras**. Pour séparer deux paragraphes il suffit de mettre deux enter (une ligne blanche en gros)

Et voilà un nouveau paragraphe :)

On peut également simplement revenir en arrière comme ça

Une petite liste à puces ? numéroté ?

- $V$  est un ensemble fini de noeuds
- $E$  est un ensemble d'arcs reliant deux noeuds

1. Remplacer la valeur de la racine par celle du dernier noeud, celui qui sera le plus à droite de la dernière ligne (le "1" dans l'exemple de la figure ??(a)).

2. Supprimer ce dernier noeud

3. Faire redescendre tant que nécessaire la nouvelle racine.

Une image ? Avec une référence dans un texte ? no probleme :p

Un graphe orienté est défini par un couple :  $G = (V, E)$ . La figure ?? illustre un graphe.

La complexité c'est pas un simple  $O(\lg n)$  mais  $\mathcal{O}(\log_2 n)$ .

**un petit titre qui n'apparaît pas dans la table des matières**

blablabla

encore un plus petit titre

blablabla

Un algorithme ? ouille ouille ouille :p mais on si habitue. Le caption sera le titre visible et le label sera ce que tu devra utiliser pour le référencer comme ca Algo ??

Des theorme, lemme, propriete ? sa marche aussi :). De nouveaux tu peux reference le label :) Lemme 1

**Lemme 1** (Propriété de borne supérieur). *Nous avons à tout moment  $v.d \geq \delta(s, v) \forall v \in V$ , et une fois que  $v.d$  atteints  $\delta(s, v)$  il ne changera plus.*

**Corollaire 1** (Propriété d'absence de chemin). *Si il n'existe pas de chemin allant de  $s$  à  $v$  alors nous avons à tout moment  $v.d = \delta(s, v) = \infty$ .*

**Propriété 1** (Propriété de convergence). *Si  $s \rightsquigarrow u \rightarrow v$  est un plus court chemin dans  $G$  pour  $u, v \in V$  donné et si  $u.d = \delta(s, u)$  avant que l'arc  $(u, v)$  ne soit relaxé alors  $v.d = \delta(s, v)$  après la relaxation.*

**Théorème 1** (Thm de convergence). *Si  $s \rightsquigarrow u \rightarrow v$  est un plus court chemin dans  $G$  pour  $u, v \in V$  donné et si  $u.d = \delta(s, u)$  avant que l'arc  $(u, v)$  ne soit relaxé alors  $v.d = \delta(s, v)$  après la relaxation.*

des symbole grec, mathematique  $\delta(a), \sigma(a), \neq, \leq, \geq, \dots$

Ca doit etre entre dollar. Pareil pour toutes les variables, formule, ...

on n'ecrit pas l'index i mais l'index  $i$

## 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
<a href="#">13..10</a>	<a href="#">2</a>	<a href="#">Tile</a>
<a href="#">12..9</a>	<a href="#">2</a>	<a href="#">GameTest</a>
<a href="#">10..8</a>	<a href="#">2</a>	<a href="#">ButtonPanel</a>
<a href="#">11..6</a>	<a href="#">2</a>	<a href="#">MainUITest</a>
<a href="#">18..17</a>	<a href="#">2</a>	<a href="#">MapParser</a>
<a href="#">12..11</a>	<a href="#">2</a>	<a href="#">PacmanKeyListener</a>
<a href="#">10</a>	<a href="#">2</a>	<a href="#">PacmanKeyListener</a>
<a href="#">8</a>	<a href="#">2</a>	<a href="#">PacmanKeyListener</a>
<a href="#">9..7</a>	<a href="#">2</a>	<a href="#">GameTest</a>
<a href="#">3</a>	<a href="#">2</a>	<a href="#">Board</a>

FIGURE 1 – Résultat de l'analyse de code redondant par CodePro

## 9 Annexes

### A Annexe : 1

Dans cette section se trouve les différentes annexes qui permettent d'identifier les blocs de code dupliqués détecté par CodePro. La première image répertorie les 10 blocs. De la seconde à l'avant dernière, les images illustres les blocs de code. La dernière image illustres les 6 derniers blocs de code et est issue du rapport généré par CodePro parce que Eclipse les masque.

## A ANNEXE : 1

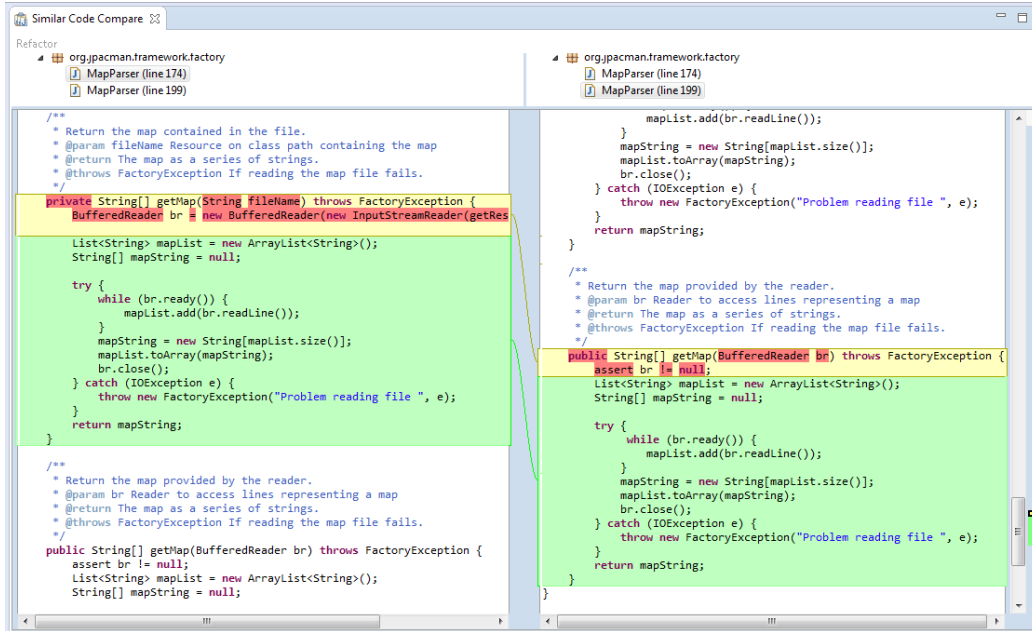


FIGURE 2 – Détail de l'analyse de code redondant par CodePro

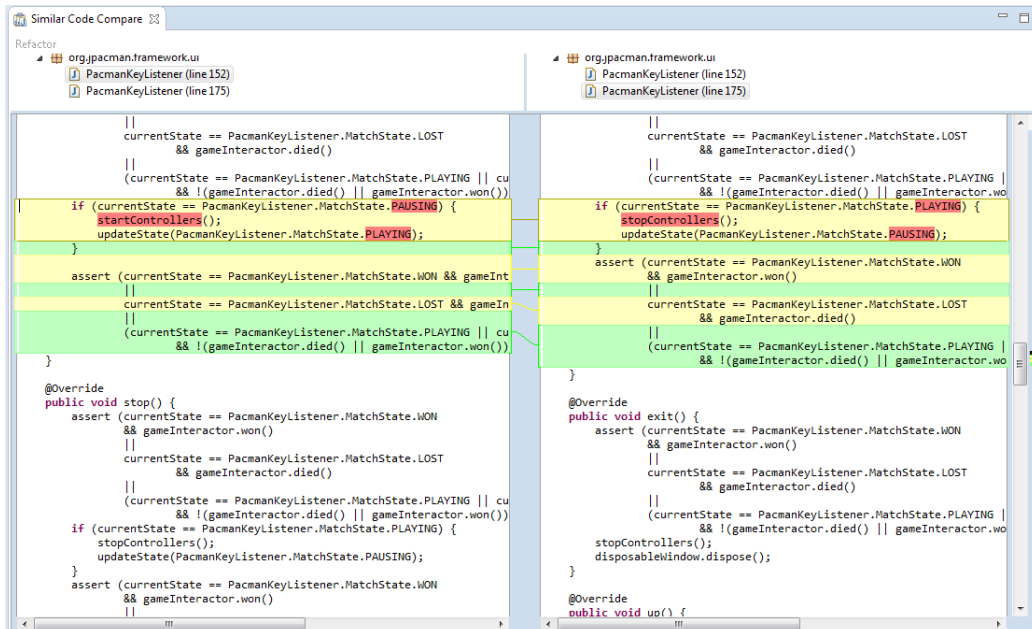


FIGURE 3 – Détail de l'analyse de code redondant par CodePro

## A ANNEXE : 1

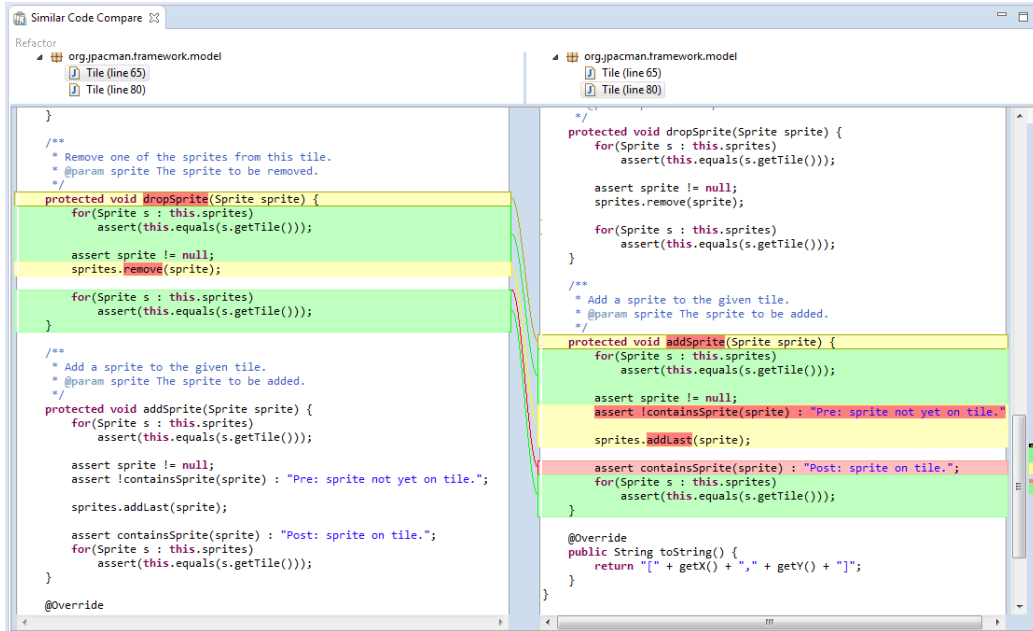


FIGURE 4 – Détail de l'analyse de code redondant par CodePro

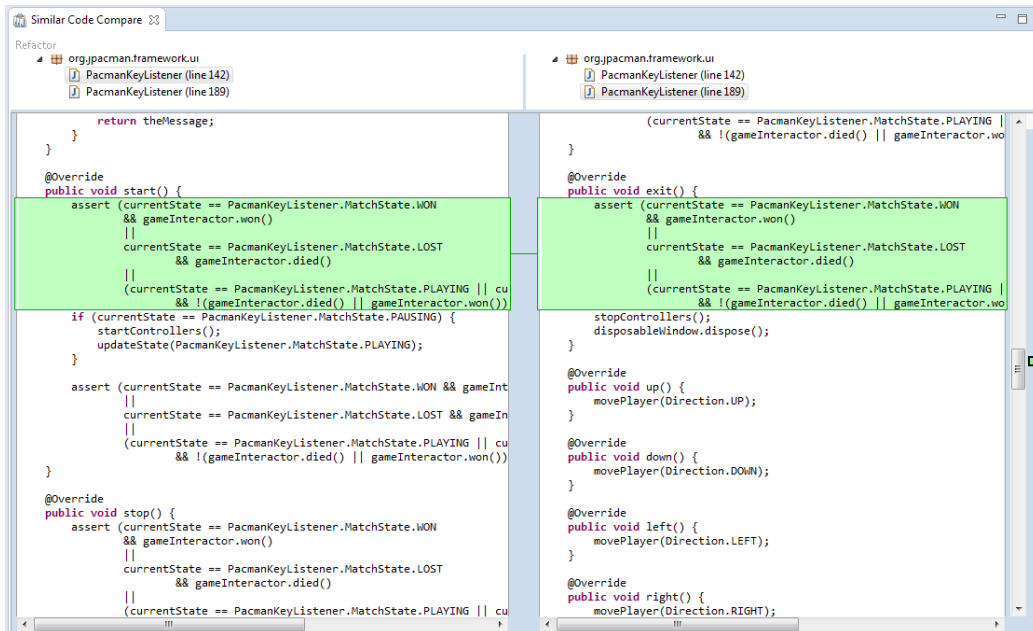


FIGURE 5 – Détail de l'analyse de code redondant par CodePro

9/03/15 21:10 Powered by CodePro Server

## B ANNEXE : .....

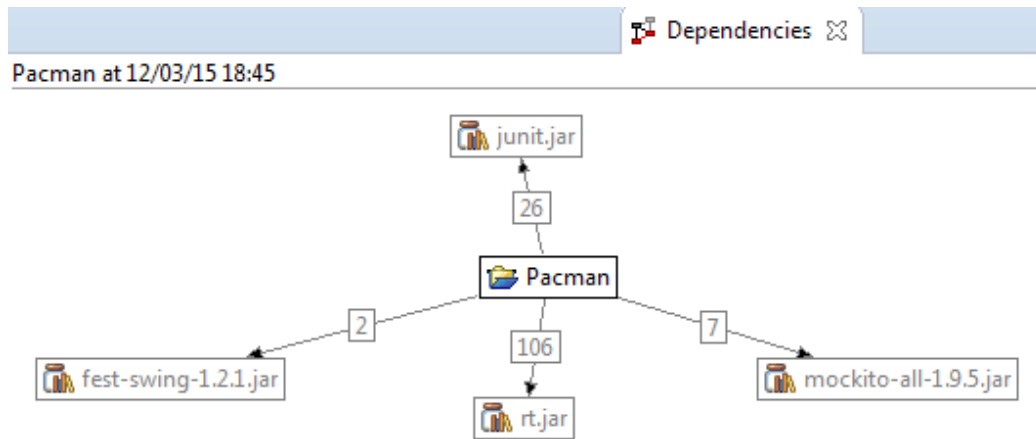


FIGURE 7 – Détail de l'analyse des dépendances cycliques du projet.

## B Annexe : .....



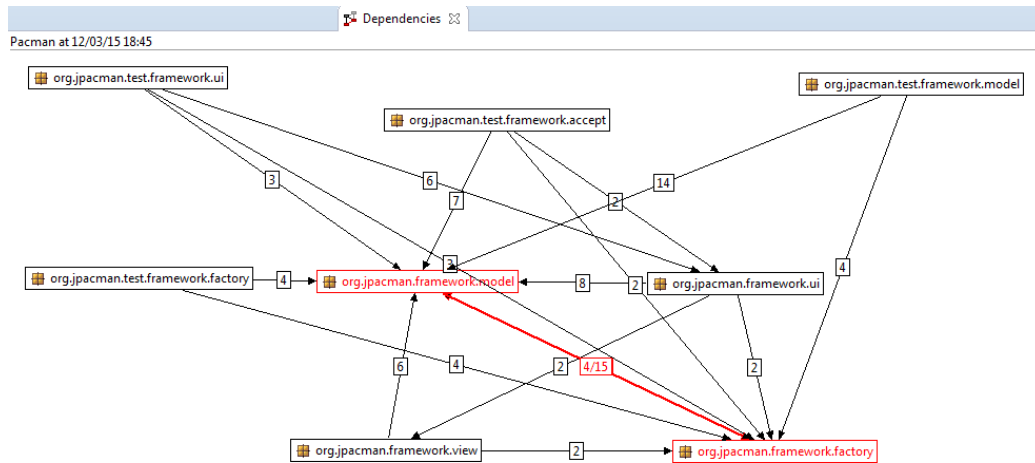


FIGURE 8 – Détail de l’analyse des dépendances cycliques des packages du projet.

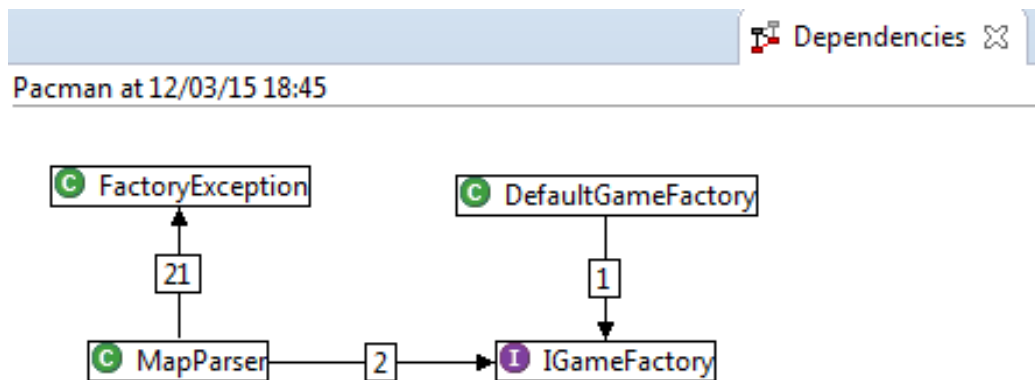


FIGURE 9 – Détail de l’analyse des dépendances cycliques du package Factory.

## C ANNEXE : .....

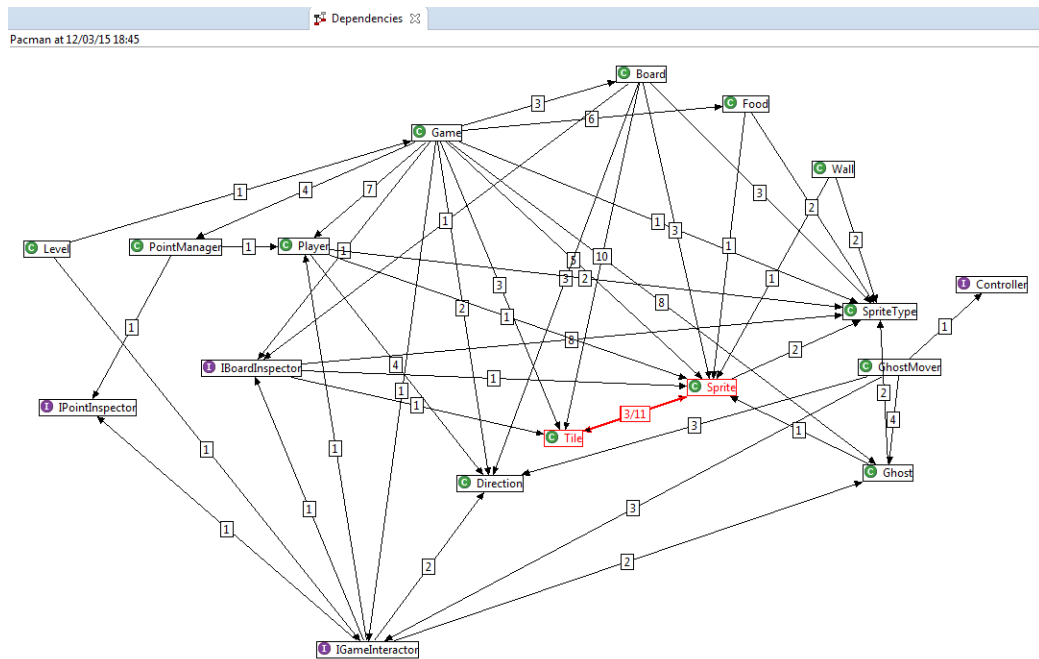


FIGURE 10 – Détail de l'analyse des dépendances cycliques du package Model.

## C Annexe : .....

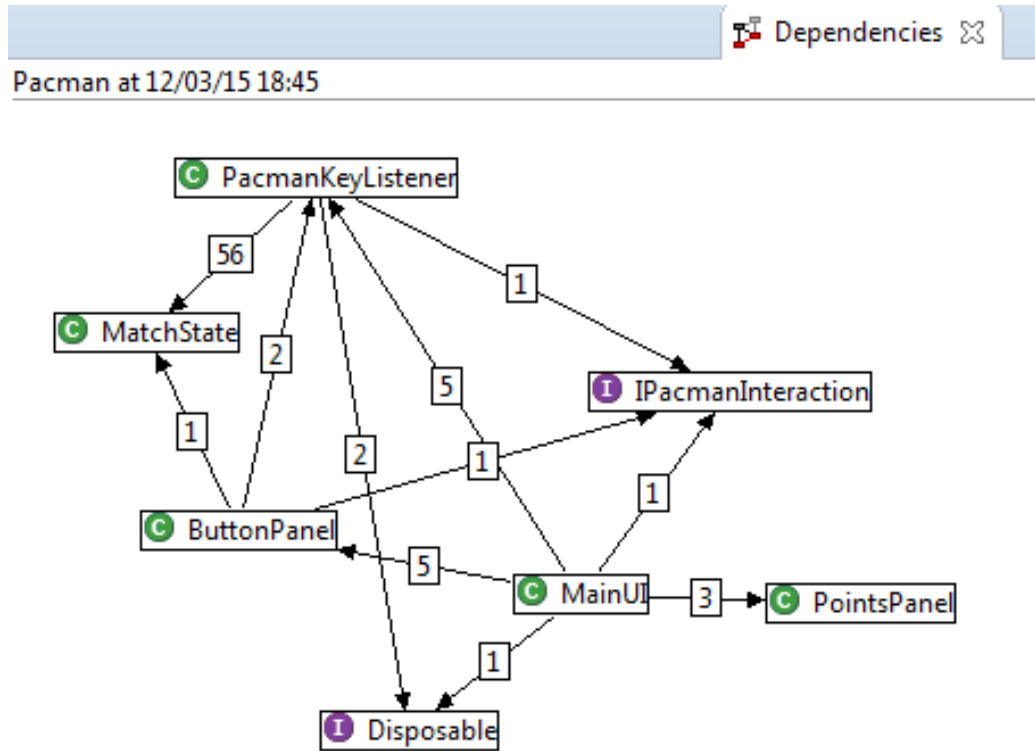


FIGURE 11 – Détail de l'analyse des dépendances cycliques du package UI.

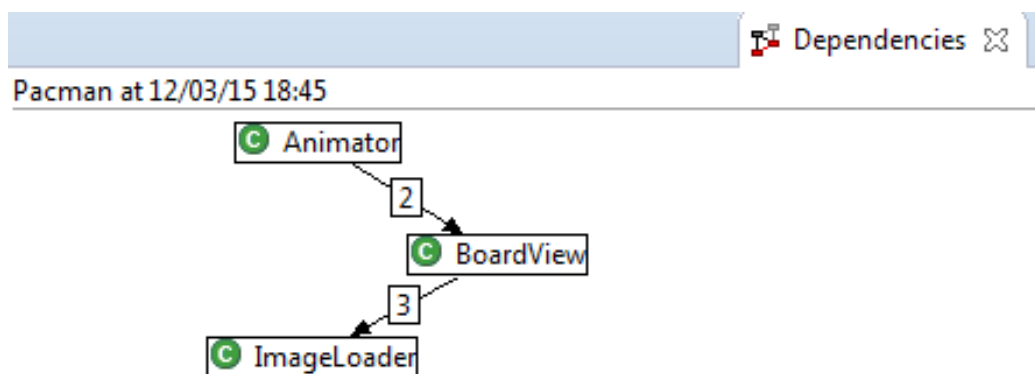


FIGURE 12 – Détail de l'analyse des dépendances cycliques du package View.

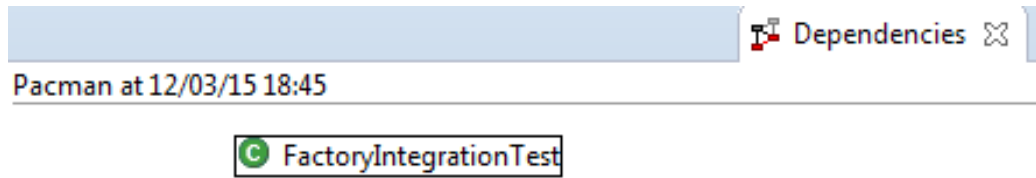


FIGURE 13 – Détail de l’analyse des dépendences cycliques du package Factory (Test).

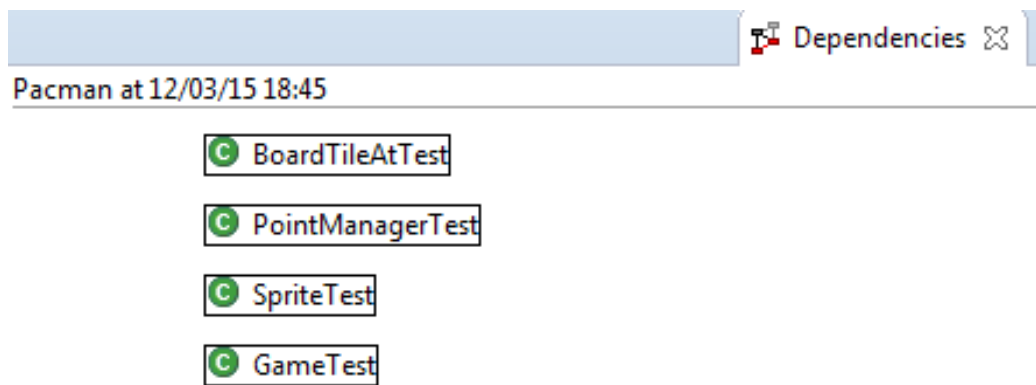


FIGURE 14 – Détail de l’analyse des dépendences cycliques du package Model (Test).

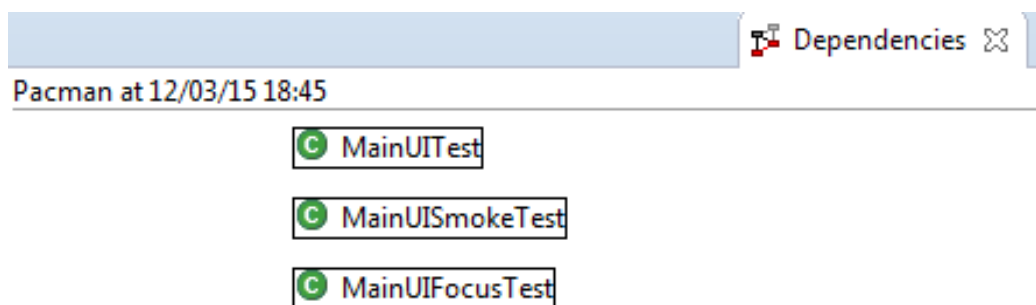


FIGURE 15 – Détail de l’analyse des dépendences cycliques du package UI (Test).

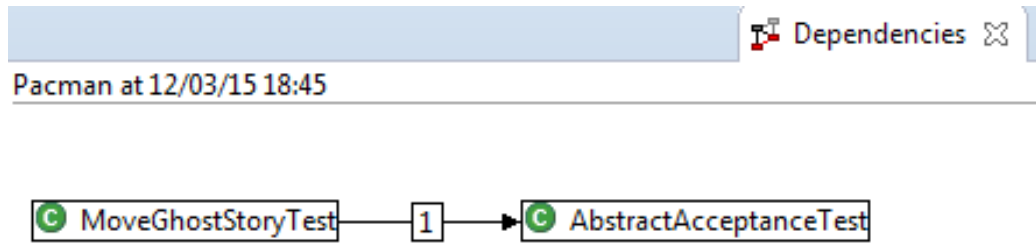


FIGURE 16 – Détail de l’analyse des dépendances cycliques du package Accept (Test).

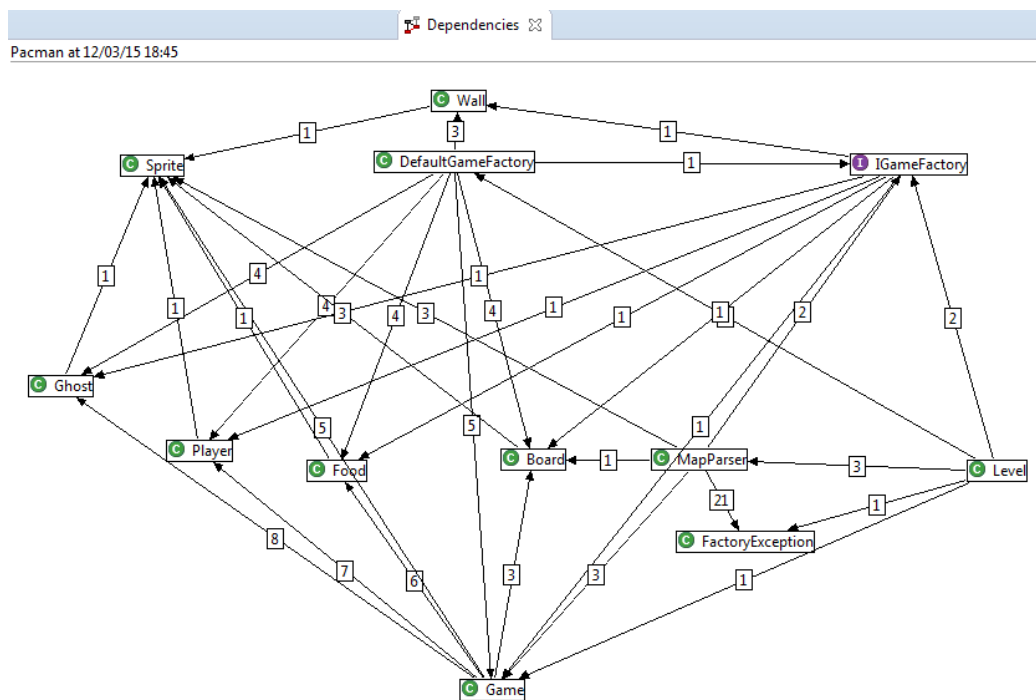


FIGURE 17 – Détail de l’analyse des dépendances cycliques entre le package Model et la package Factory.

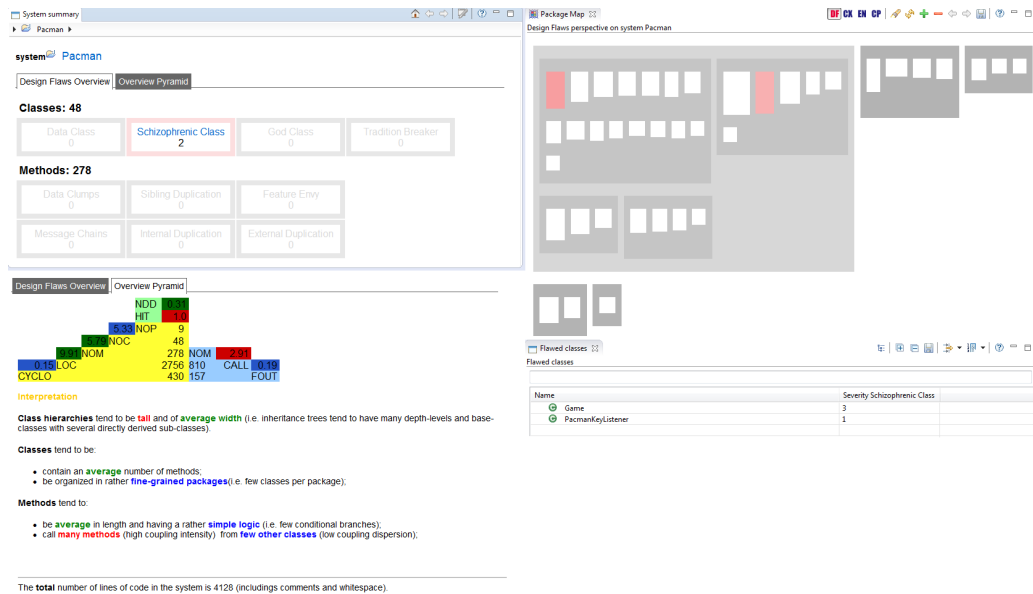
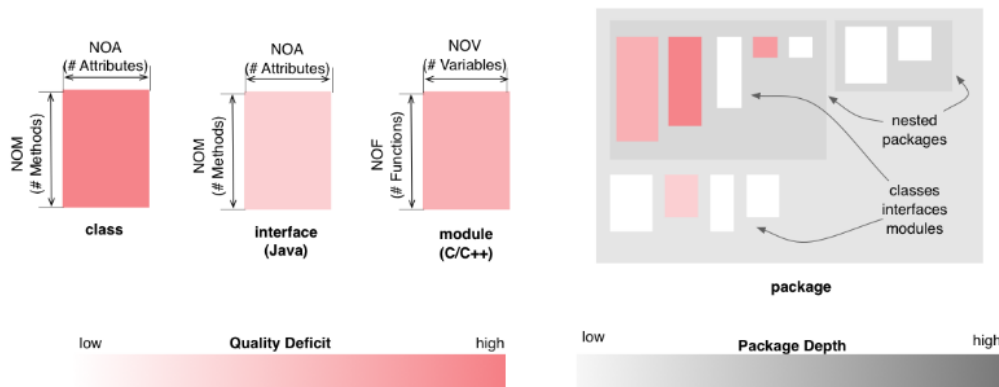


FIGURE 18 – Détail de l'analyse .....

## 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 19 – Légende de l'analyse .....

## D ANNEXE : .....

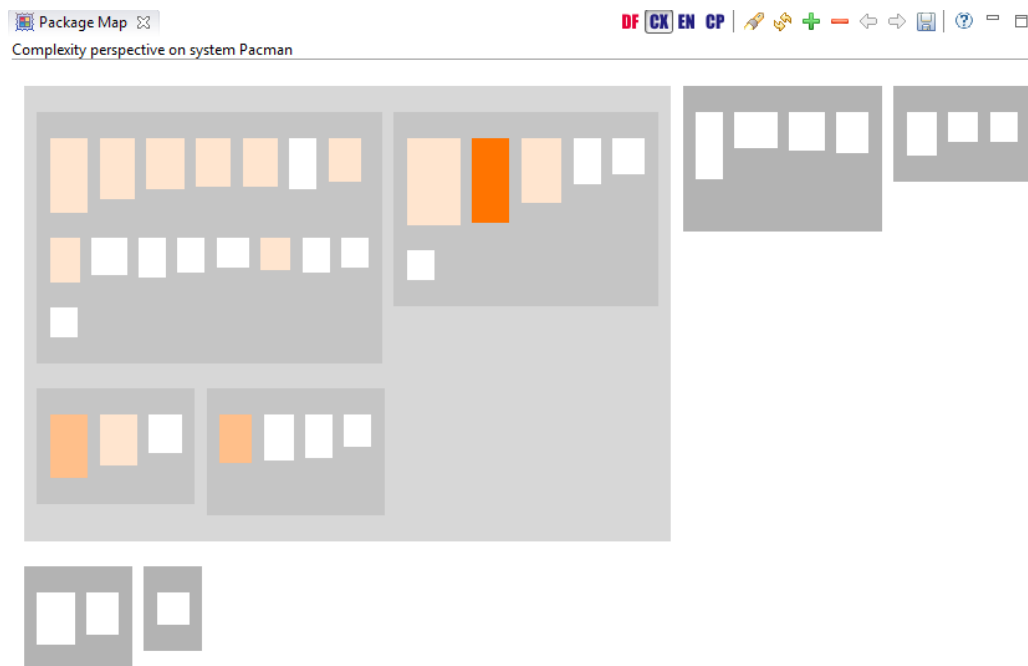
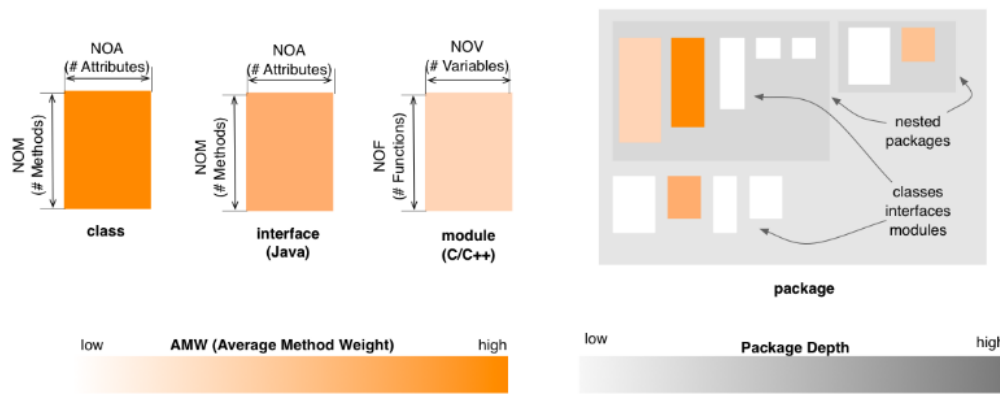


FIGURE 20 – Détail de l'analyse .....

## D Annexe : .....

## 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 21 – Légende de l'analyse .....



## E ANNEXE : .....

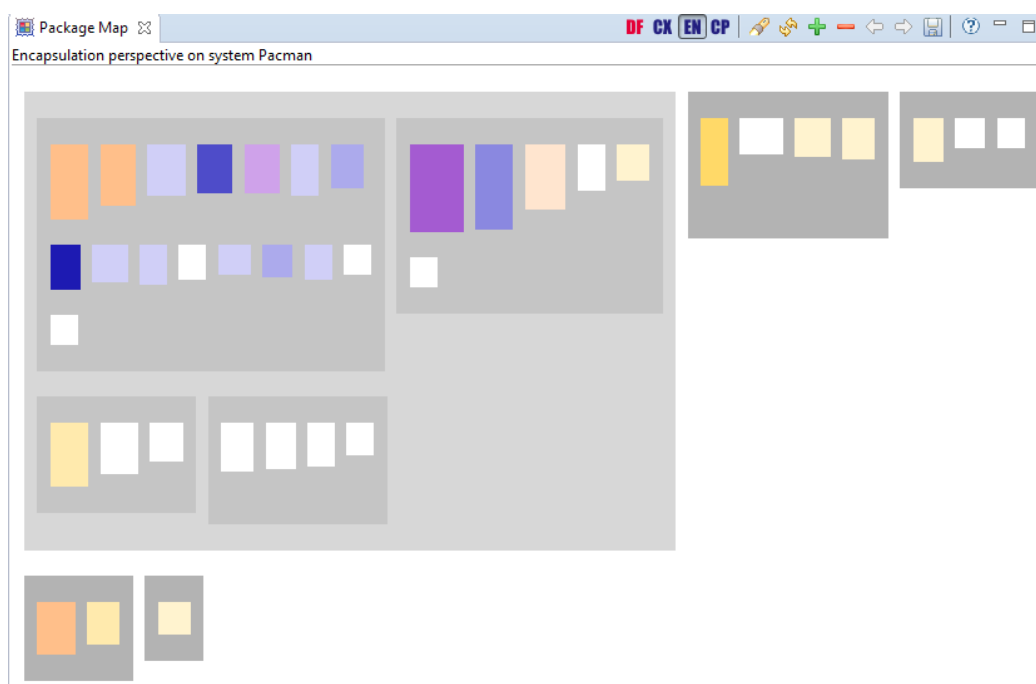


FIGURE 22 – Détail de l'analyse .....

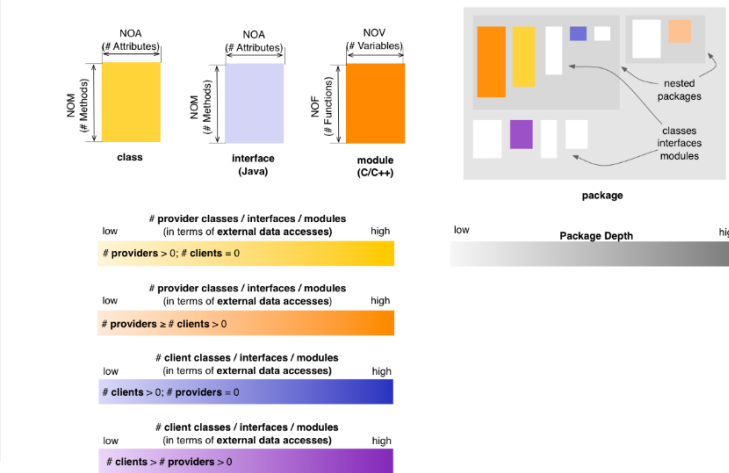
## E Annexe : .....

## 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 selected 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 Encapsulation 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.

		collaborator class / interface / module			
selected	class	# provider data > 0; # client data = 0 (in terms of external data accesses related to the selected class)	# provider data ≥ # client data > 0 (in terms of external data accesses related to the selected class)	# client data > # provider data > 0 (in terms of external data accesses related to the selected class)	# client data > 0; # provider data = 0 (in terms of external data accesses related to the selected class)
	interface	# provider data > 0 (in terms of external data accesses related to the selected interface)	# provider data = 0 (in terms of external data accesses related to the selected interface)		
	module	# provider data > 0; # client data = 0 (in terms of external data accesses related to the selected module)	# provider data ≥ # client data > 0 (in terms of external data accesses related to the selected module)	# client data > # provider data > 0 (in terms of external data accesses related to the selected module)	# client data > 0; # provider data = 0 (in terms of external data accesses related to the selected module)

FIGURE 23 – Légende de l'analyse .....

## RÉFÉRENCES

---

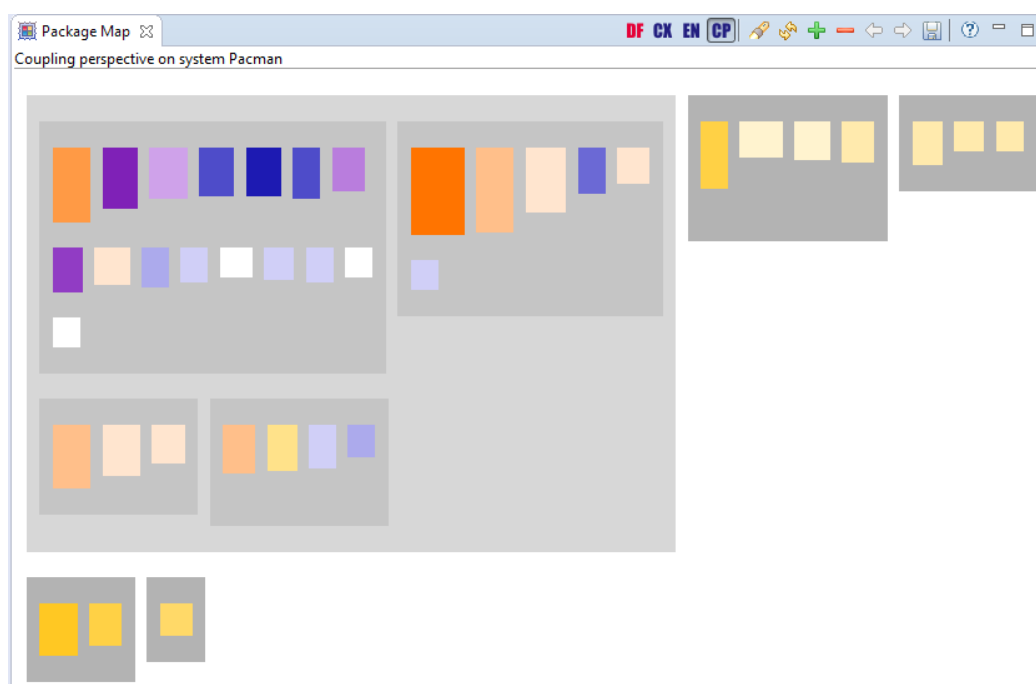


FIGURE 24 – Détail de l'analyse .....

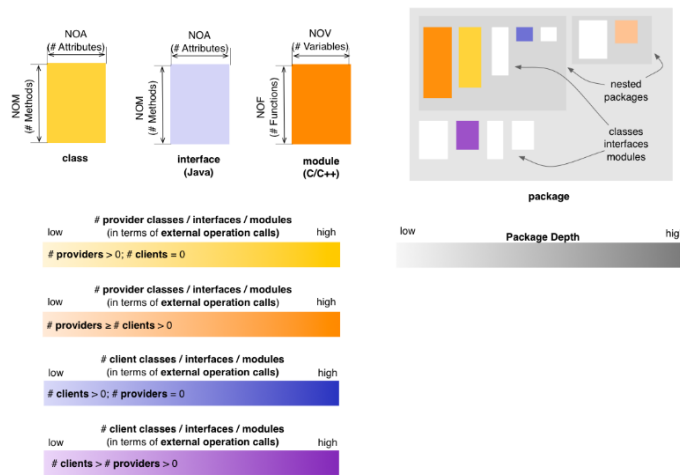
## Références

# RÉFÉRENCES

## 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 selected 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.

		collaborator class / interface / module			
selected	class	# provider operations > 0; # client operations = 0 (in terms of external operation calls related to the selected class)	# provider operations ≥ # client operations > 0 (in terms of external operation calls related to the selected class)	# client operations > # provider operations > 0 (in terms of external operation calls related to the selected class)	# client operations > 0; # provider operations = 0 (in terms of external operation calls related to the selected class)
	interface	# provider operations > 0 (in terms of external operation calls related to the selected interface)	# provider operations = 0 (in terms of external operation calls related to the selected interface)		
	module	# provider operations > 0; # client operations = 0 (in terms of external operation calls related to the selected module)	# provider operations ≥ # client operations > 0 (in terms of external operation calls related to the selected module)	# client operations > # provider operations > 0 (in terms of external operation calls related to the selected module)	# client operations > 0; # provider operations = 0 (in terms of external operation calls related to the selected module)

FIGURE 25 – Légende de l'analyse .....

## RÉFÉRENCES

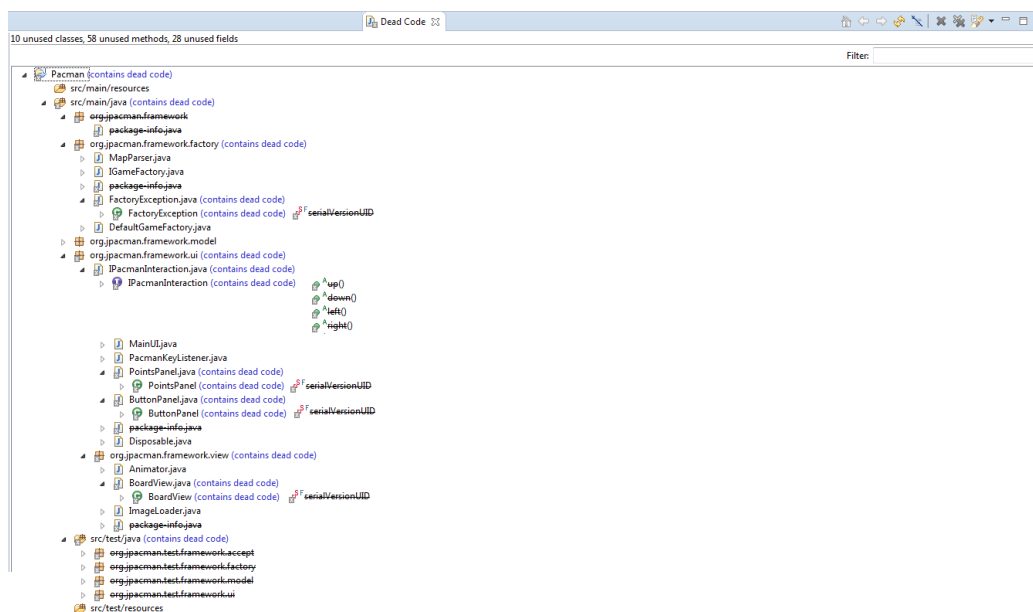


FIGURE 26 – Détail de l’analyse des parties de code non utilisé lors de l’exécution du programme.

## RÉFÉRENCES













































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

## RÉFÉRENCES

---

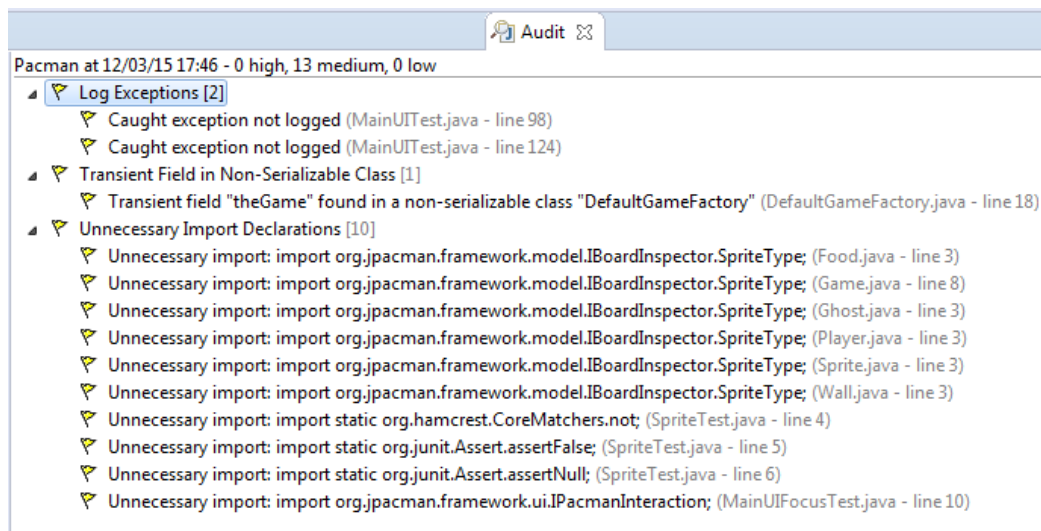


FIGURE 28 – Détail de l'analyse d'audit faite par Eclipse.