

# Software Analysis : pacman systems

## Project report for Software Evolution course

Group 3

BOOSKO Sam  
DECOCQ Rémy  
SCHERER Robin

# Contents

<b>1</b>	<b>Quality analysis of the initial versions</b>	<b>3</b>
1.1	System 1 (BOOSKO Sam)	3
1.1.1	Generalities	3
1.1.2	Static Analysis	3
1.1.3	Dynamic Analysis	8
1.2	System 2 (Rémy)	10
1.2.1	Generalities	10
1.2.2	Static Analysis	10
1.2.3	Dynamic Analysis	16
1.3	System 3 (Robin)	18
1.3.1	Generalities	18
1.3.2	Static Analysis	18
1.3.3	Dynamic Analysis	22
1.4	Comparaison of the 3 systems	23
<b>2</b>	<b>Quality improvement</b>	<b>24</b>
2.1	System 1 (BOOSKO Sam)	24
2.2	System 2 (Rémy)	25
2.2.1	Step 1	25
2.2.2	Step 2	25
2.2.3	Step 3	26
2.2.4	Step 4	26
2.3	System 3 (Robin)	27
<b>3</b>	<b>Adding basic functionalities</b>	<b>28</b>
3.1	System 1 (BOOSKO Sam)	28
3.2	System 2 (Rémy)	29
3.3	System 3 (Robin)	30
<b>4</b>	<b>Adding new features</b>	<b>31</b>
4.1	System 1 (Rémy)	31
4.1.1	New features discussion	31
4.2	System 2 (Robin Schérer)	33
4.3	System 3 (BOOSKO Sam)	35
4.3.1	Quality Improvement	35
4.3.2	Features	35
<b>5</b>	<b>Quality evolution analysis</b>	<b>40</b>
5.1	System 1 (Rémy)	40
5.2	System 2 (Robin)	43
5.2.1	Generalities	43
5.2.2	Static Analysis	43
5.2.3	Code Metrics (CodeMR)	43
5.2.4	Compliance bad smells (PMD, Designite)	43
5.2.5	Test coverage (IntelliJ built-in tool)	43
5.3	System 3 (BOOSKO Sam)	45
5.3.1	Generalities	45
5.3.2	Static Analysis	45
5.3.3	Dynamic Analysis	45

# 1 Quality analysis of the initial versions

## 1.1 System 1 (BOOSKO Sam)

### 1.1.1 Generalities

First of all, the structure of the project folder is classic with sub folders, such as:

- *src* containing two folders:
  1. *main*, all classes for the game (core, gui and movement controller), and
  2. *test* with all unit tests of Junit system.
- *doc*. In this folder, a file, *scenarios.md*, present the goal of the initial project and few scenarios of the game rules. Furthermore, there is another folder, *uml*, containing two uml class diagram files of *uxf* format. These two files describe a simply version of classes (see Figure 1 and Figure 2).

The building system, as demanded in directives, is provided with *Maven*.

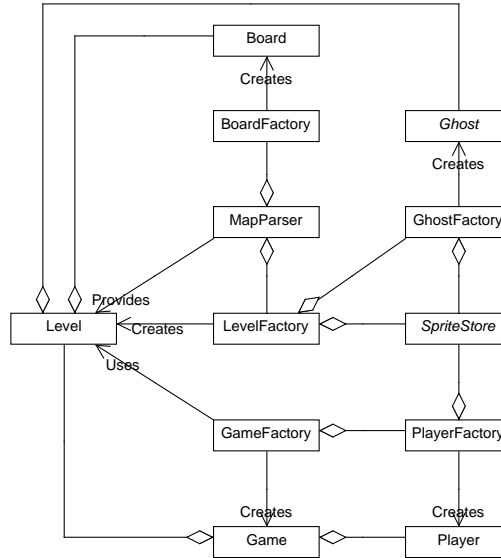


Figure 1: Factory Wiring Class Diagram

### 1.1.2 Static Analysis

#### 1.1.2.1 Bad Smells (Designite)

Designite is a code analyzer which can detect bad smells in a project. Firstly, made for C# implementation, another version for Java project is provided. Here, designite is used as an *IntelliJ* plugin to have a live visualization during the refactoring and as a "script" to get global information from the project as *csv* files (sample of used file see Table 1). Furthermore, the output of the "script" gives some information (see Figure 3), such as the number of cyclic dependency, here 5, the number of magic number, here 39 and also the number of long parameter list, here 8.

#### 1.1.2.2 Code Metrics (CodeMR)

CodeMR is a software which assess a project quality with a static code analysis to help developers or companies to develop better code, products quality. It generates an interactive *html* files to visualize all information assessed as a dashboard.

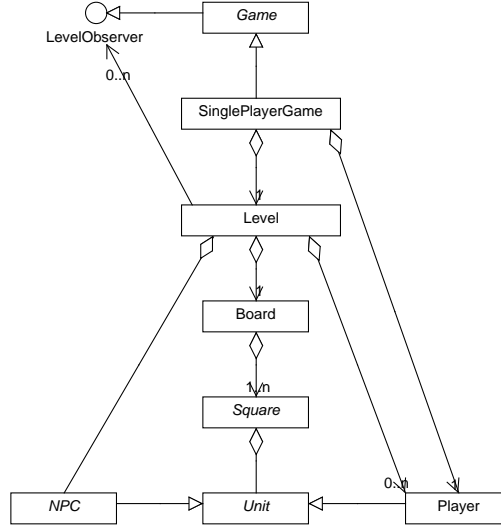


Figure 2: Single Player Game Class Diagram

28	jpacman-framework	nl.tudelft.jpacman.sprite	EmptySprite	draw	Long Parameter List	The method has 5 parameters.
29	jpacman-framework	nl.tudelft.jpacman.sprite	ImageSprite	draw	Long Parameter List	The method has 5 parameters.
30	jpacman-framework	nl.tudelft.jpacman.sprite	ImageSprite	newImage	Long Statement	The length of the statement "GraphicsConfigura...
31	jpacman-framework	nl.tudelft.jpacman.sprite	Sprite	draw	Long Parameter List	The method has 5 parameters.
32	jpacman-framework	nl.tudelft.jpacman.sprite	SpriteTest	animationWidth	Magic Number	The method contains a magic number: 4
33	jpacman-framework	nl.tudelft.jpacman.sprite	SpriteTest	animationWidth	Magic Number	The method contains a magic number: 16
34	jpacman-framework	nl.tudelft.jpacman.sprite	SpriteTest	animationHeight	Magic Number	The method contains a magic number: 4
35	jpacman-framework	nl.tudelft.jpacman.sprite	SpriteTest	animationHeight	Magic Number	The method contains a magic number: 64
36	jpacman-framework	nl.tudelft.jpacman.sprite	SpriteTest	splitWidth	Magic Number	The method contains a magic number: 10
37	jpacman-framework	nl.tudelft.jpacman.sprite	SpriteTest	splitWidth	Magic Number	The method contains a magic number: 11

Table 1: ImplementationSmells.csv

The main page (see Figure 4) informs that the project quality is fairly good. Only one problematic class and 9.9% of cohesion lack.

By coupling the Figures 5 and 4, we notice that the class *Level* impacts the project quality. The problematic metric, visible from the Figure 6, is the lack of cohesion<sup>1</sup> defined in the *CodeMR* as "Measure how well the methods of a class are related to each other. High cohesion (low lack of cohesion) tend to be preferable, because high cohesion is associated with several desirable traits of software including robustness, reliability, reusability, and understandability. In contrast, low cohesion is associated with undesirable traits such as being difficult to maintain, test, reuse, or even understand."

### 1.1.2.3 Dependencies (IntelliJ analyzer)

IntelliJ Ultimate version has an analyzer to assess a dependencies matrix (see Figure 7). We observe, as seen before with Designite analysis, few cyclic dependencies.

### 1.1.2.4 Javadoc Coverage (MetricsReloaded)

With the plugin *MetricsReloaded* of *IntelliJ*, we can assess few metrics like the Javadoc coverage (see Figure 2). We observe that the project is fairly covered by the javadoc.

<sup>1</sup><https://www.codemr.co.uk/documents/>

```

Searching classpath folders ...
Parsing the source code ...
Resolving symbols...
Computing metrics...
Detecting code smells...
Error - License validation unsuccessful. Status code: 406
Exporting analysis results...
--Analysis summary--
    Total LOC analyzed: 3657      Number of packages: 10
    Number of classes: 61      Number of methods: 311
-Total architecture smell instances detected-
    Cyclic dependency: 5      God component: 0
    Ambiguous interface: 0      Feature concentration: 1
    Unstable dependency: 3      Scattered functionality: 1
    Dense structure: 0
-Total design smell instances detected-
    Imperative abstraction: 0      Multifaceted abstraction: 0
    Unnecessary abstraction: 1      Unutilized abstraction: 4
    Feature envy: 0      Deficient encapsulation: 2
    Unexploited encapsulation: 1      Broken modularization: 0
    Cyclically-dependent modularization: 0      Hub-like modularization: 0
    Insufficient modularization: 0      Broken hierarchy: 0
    Cyclic hierarchy: 0      Deep hierarchy: 0
    Missing hierarchy: 1      Multipath hierarchy: 0
    Rebellious hierarchy: 0      Wide hierarchy: 0
-Total implementation smell instances detected-
    Abstract function call from constructor: 2      Complex conditional: 1
    Complex method: 0      Empty catch clause: 0
    Long identifier: 0      Long method: 0
    Long parameter list: 8      Long statement: 2
    Magic number: 39      Missing default: 2
----
Done.

```

Figure 3: Designite Output

#### Analysis of jpacman-framework

General Information

Total lines of code: 1241

Number of classes: 50

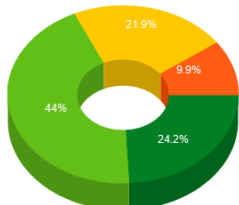
Number of packages: 8

Number of external packages: 28

Number of external classes: 182

Number of problematic classes: 1

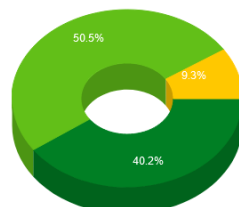
Number of highly problematic classes: 0



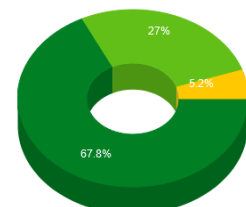
● Very High  
 ● High  
 ● Medium-high  
 ● Low-medium  
 ● Low

#### Distribution of Quality Attributes

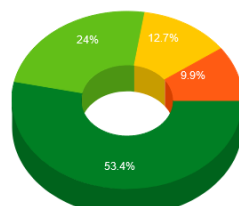
Complexity, Coupling, Cohesion, and Size



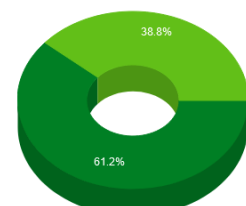
Complexity



Coupling



Lack of Cohesion



Size

Figure 4: CodeMR dashboard summarizing health of the system 1

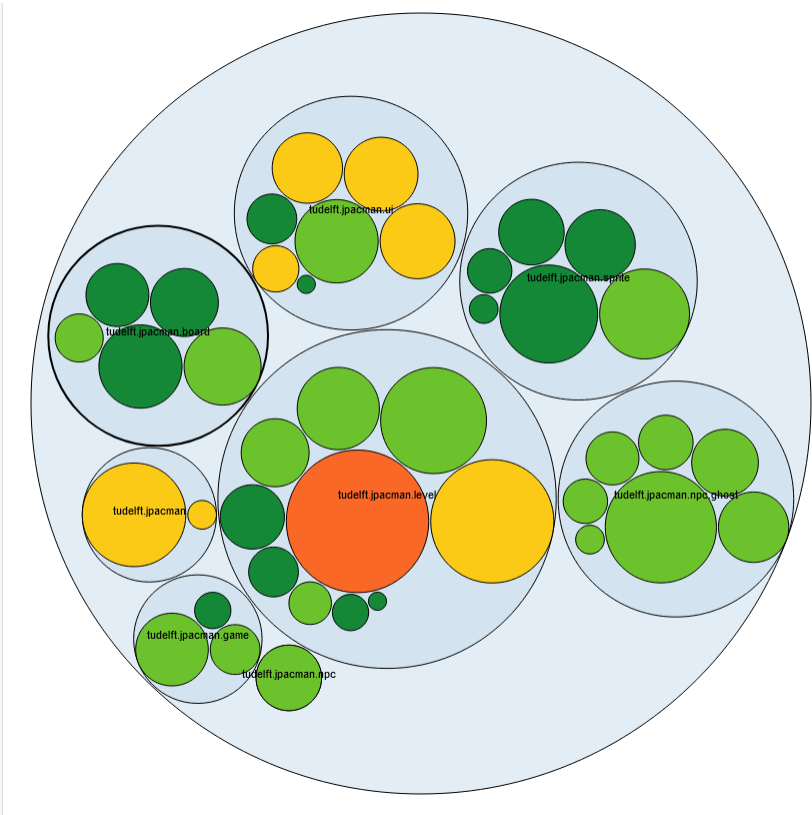


Figure 5: C3 Metric by package for the system 1

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
4	Level	■	■	■	■	123	low-medium	low-medium	high	low-medium

Figure 6: Metrics of the class Level

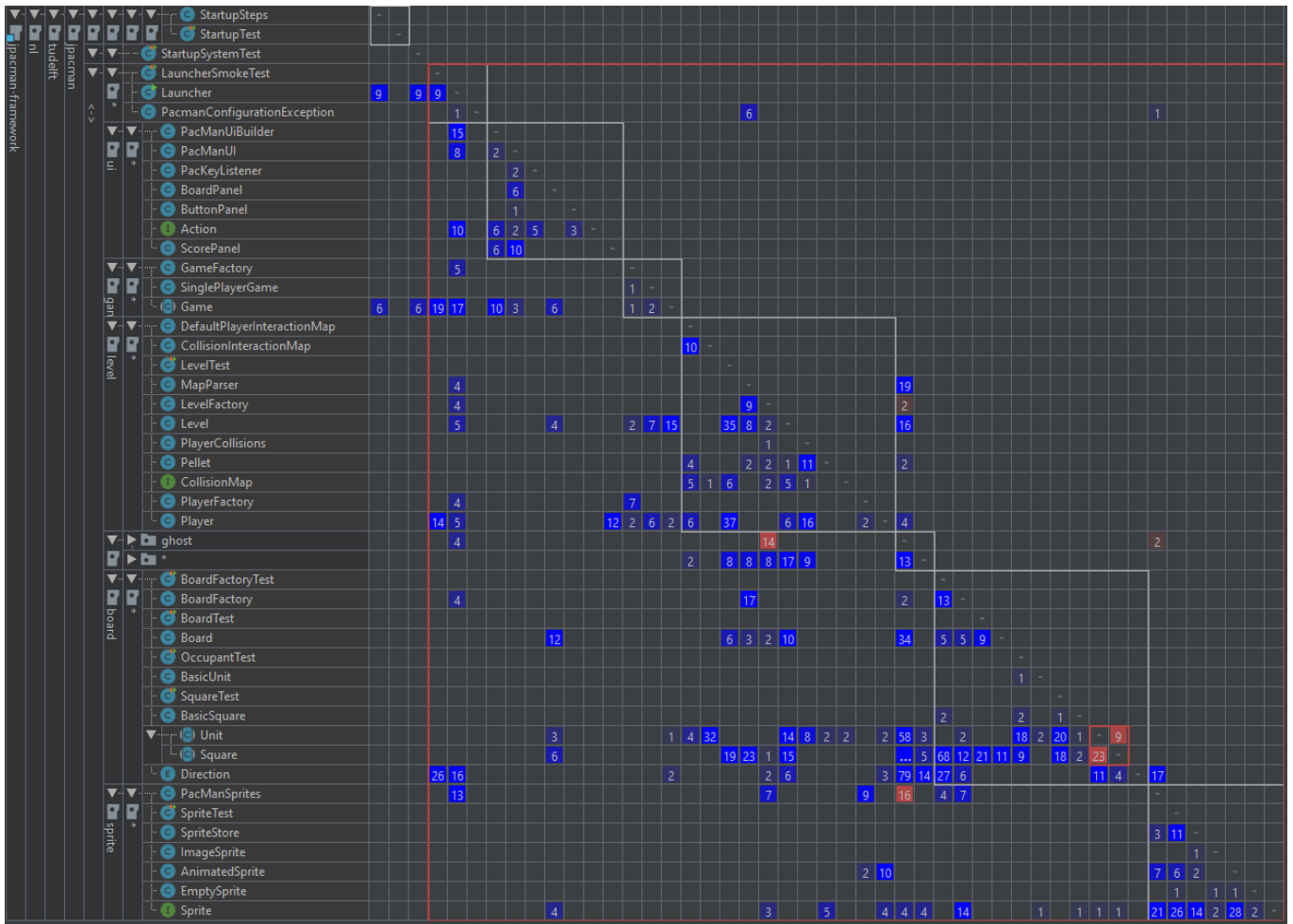


Figure 7: Matrix of Dependencies

package ▲	Jc	Jf	JLOC	Jm
nl.tudelft.jpacman	100.00%	0.00%	132	91.67%
nl.tudelft.jpacman.board	100.00%	78.26%	391	88.33%
nl.tudelft.jpacman.e2e.framework.startup	100.00%	0.00%	26	80.00%
nl.tudelft.jpacman.game	100.00%	100.00%	82	73.33%
nl.tudelft.jpacman.integration	100.00%	0.00%	13	75.00%
nl.tudelft.jpacman.level	100.00%	85.00%	651	86.49%
nl.tudelft.jpacman.npc	100.00%	100.00%	47	83.33%
nl.tudelft.jpacman.npc.ghost	100.00%	90.48%	446	91.18%
nl.tudelft.jpacman.sprite	100.00%	83.33%	286	71.74%
nl.tudelft.jpacman.ui	100.00%	100.00%	252	84.00%
<b>Total</b>			<b>2,326</b>	
<b>Average</b>	<b>100.00%</b>	<b>82.61%</b>	<b>232.60</b>	<b>84.30%</b>

Table 2: Coverage of the javadoc for the system 1

### 1.1.3 Dynamic Analysis

#### 1.1.3.1 Running tests & Test Coverage (IntelliJ Build-Run)

By running the tests provided with the project, 45 of 45 tests passed. Furthermore, IntelliJ provides a tool to assess the coverage of tests (see Table 3). Looking at the *Method, %* column, it's fairly well covered. We see that the Package *level* is covered at 66%.

In the Table 4, we get more detailed information and observe that two classes are not tested, *CollisionInteractionMap* and *DefaultPlayerInteractionmap*. Furthermore, another class, *LevelFactory* is covered at 50%. Some enhancement can be done for it.



Element	Class, %	Method, %	Line, %
board	100% (7/7)	100% (41/41)	98% (108/110)
game	100% (3/3)	85% (12/14)	90% (39/43)
level	66% (8/12)	71% (46/64)	73% (235/321)
npc	100% (9/9)	94% (35/37)	90% (147/163)
sprite	100% (5/5)	86% (31/36)	90% (103/114)
ui	100% (6/6)	77% (24/31)	85% (123/144)
Launcher	100% (1/1)	80% (16/20)	70% (33/47)
PacmanConfigurationException	0% (0/1)	0% (0/2)	0% (0/4)

Table 3: Test Coverage for the system 1

Element	Class, %	Method, %	Line, %
CollisionInteractionMap	0% (0/2)	0% (0/9)	0% (0/51)
DefaultPlayerInteractionMap	0% (0/1)	0% (0/4)	0% (0/10)
Level	100% (2/2)	94% (16/17)	94% (109/115)
LevelFactory	50% (1/2)	66% (4/6)	78% (15/19)
MapParser	100% (1/1)	90% (9/10)	90% (64/71)
Pellet	100% (1/1)	100% (3/3)	100% (6/6)
Player	100% (1/1)	100% (6/6)	90% (18/20)
PlayerCollisions	100% (1/1)	83% (5/6)	75% (18/24)
PlayerFactory	100% (1/1)	100% (3/3)	100% (5/5)

Table 4: Test Coverage Package level for the system 1

## 1.2 System 2 (Rémy)

### 1.2.1 Generalities

First of all, this is noticeable that authors provide some documents coupled with the implementation, even if it is not mentioned in the README file. This additional material is available under the `out/` directory at project roots and comprises :

- A `.pdf` file describing shortly the game, the controls and the multiplayer (2 players) mode available
- A complete class diagram covering the whole implementation
- A sequence diagram stating the execution flow when Pacman arrives on a cell and so “eat” what is at this place
- A graph of the mathematical function used to correlate difficulty with player’s progression

We also observe that in this Pacman implementation maps are modeled under `.tmx` format, that is a popular way to deal with board games<sup>2</sup>. Only one single basic map is provided<sup>3</sup>.

The project structure is classic, we have `main` and `test` separation under the `src` directory, each containing packaged sources. The building system provided with the implementation is hold by Gradle. So a switch to Maven will be required to comply with directives.

### 1.2.2 Static Analysis

#### 1.2.2.1 Code metrics (CodeMR)

CodeMR allows to get an overall idea of the actual health of the system considering several metrics. The dashboard illustrated by Figure 8 informs this software is doing quite good.

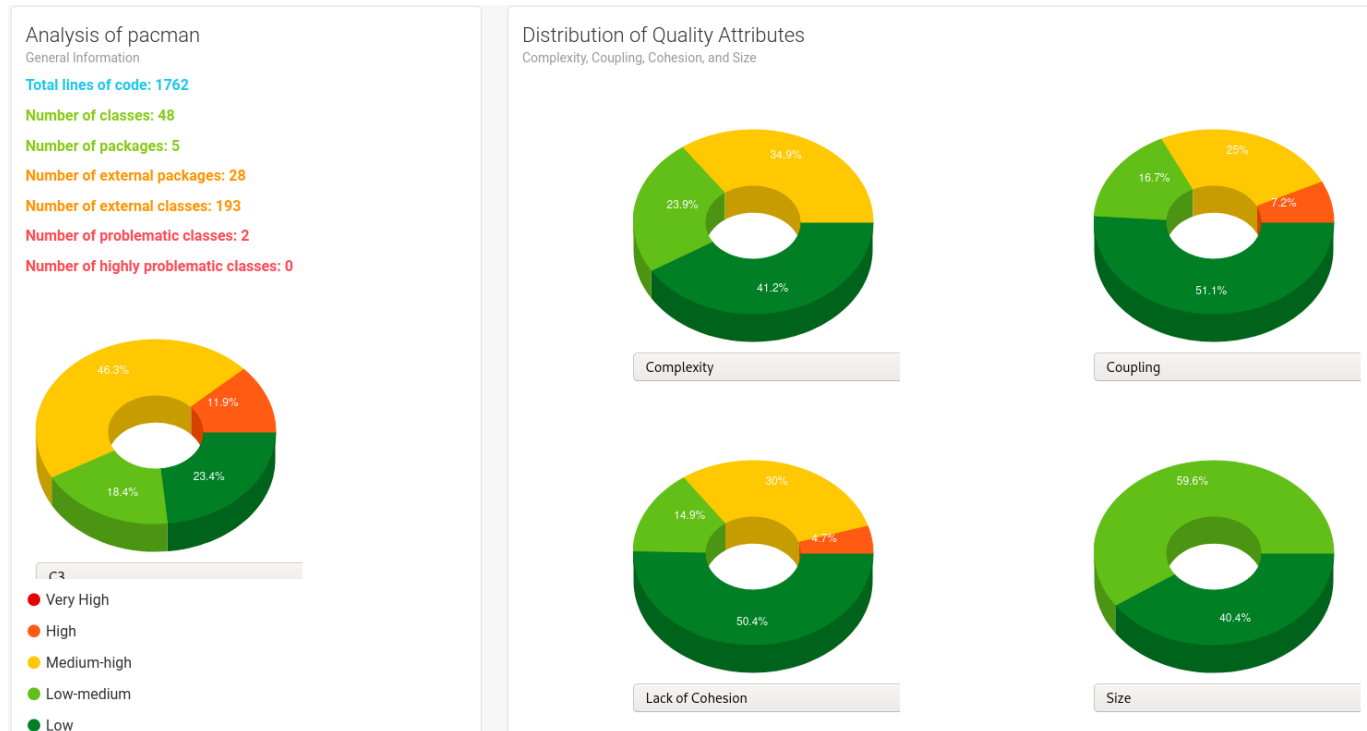


Figure 8: CodeMR dashboard summarizing health of system 2

<sup>2</sup><https://doc.mapeditor.org/en/stable/reference/support-for-tmx-maps/>

<sup>3</sup>Actually the TMX file is never used, the map is rewritten in code as-is

The Figure 9 illustrates also the C3 metric but coupled with detailed packages view. We notice authors apparently tried to follow some Model-View-Controller pattern to design their application. The C3 metric is defined as the maximum between 3 other well representative metrics : *Coupling*, *Cohesion* and *Complexity*. These are defined in the codeMR documentation<sup>4</sup>. We notice that, following the dashboard overview, two classes are impacting the software quality from the point of view of C3 metric.

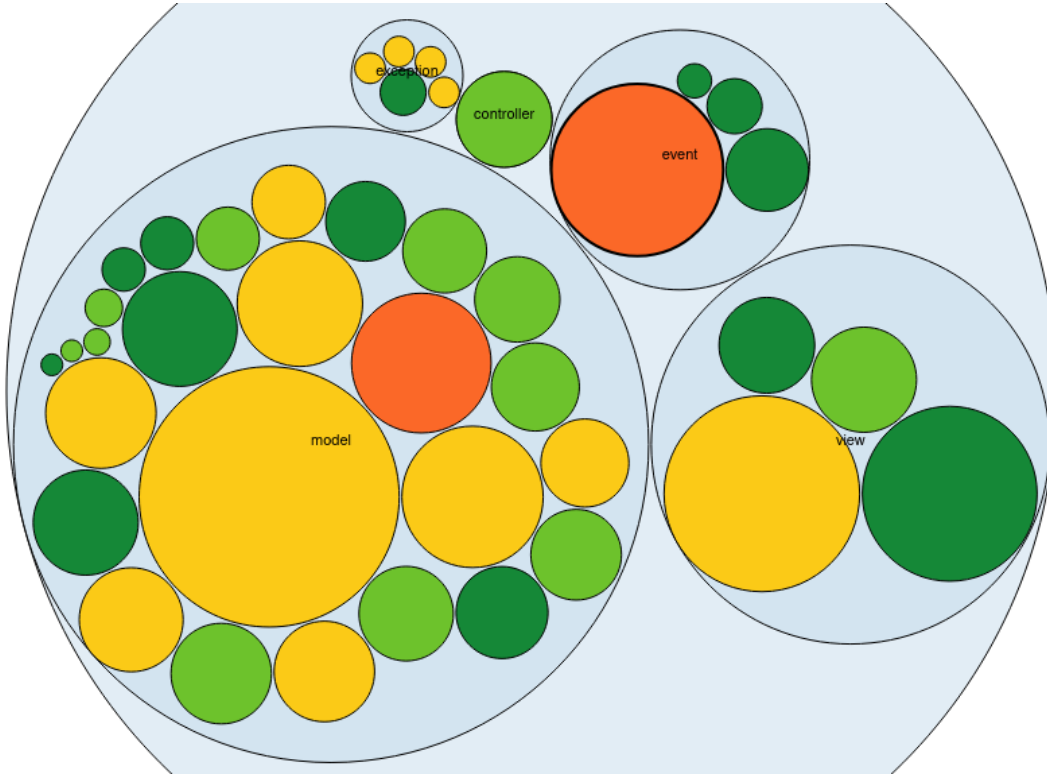


Figure 9: C3 Metric by package for system 2

The details of the measurements on the two more problematic classes are given by Figure 10 and Figure 11, for respectively *event.WorkerProcess* and *model.Ghost*. For both, two metrics are considered as high value, the meaning described by CodeMR is

- LTCC : The Lack of Tight Class Cohesion metric measures the lack cohesion between the public methods of a class. That is the relative number of directly connected public methods in the class. Classes having a high lack of cohesion indicate errors in the design.
- LCOM : Measure how methods of a class are related to each other. Low cohesion means that the class implements more than one responsibility. A change request by either a bug or a new feature, on one of these responsibilities will result change of that class. Lack of cohesion also influences understandability and implies classes should probably be split into two or more subclasses.

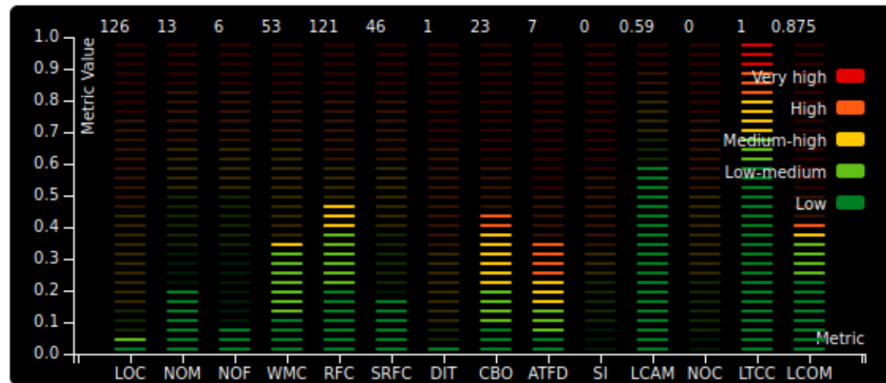
In addition, for *event.WorkerProcess* we have :

- CBO : The number of classes that a class is coupled to. It is calculated by counting other classes whose attributes or methods are used by a class, plus those that use the attributes or methods of the given class.
- AFTD : Access to Foreign Data is the number of classes whose attributes are directly or indirectly reachable from the investigated class. Classes with a high AFTD value rely strongly on data of other classes and that can be the sign of the God Class.

Other codeMR metrics did not reveal relevant problems in the implementation.

<sup>4</sup><https://www.codemr.co.uk/documents>

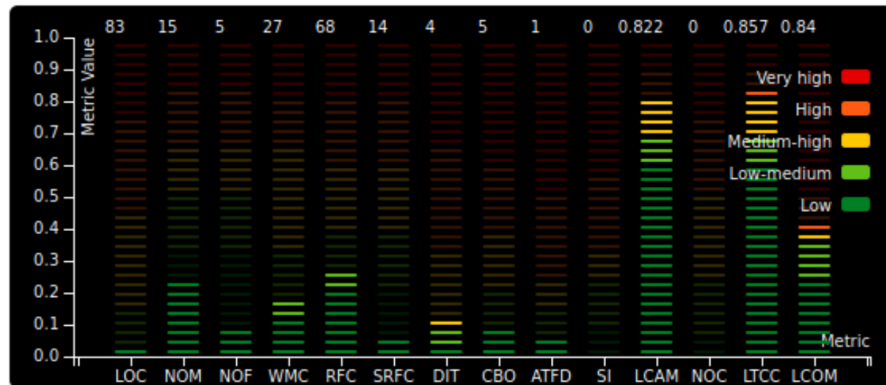
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	CBO	CBO APP	CBO LIB	RFC
1	WorkerProcess	<span style="color: orange;">■</span>	<span style="color: yellow;">■</span>	<span style="color: green;">■</span>	<span style="color: green;">■</span>	126	23	23	0	121



model.event.WorkerProcess  
Coupling: high  
Complexity: medium-high  
Lack of Cohesion: low

Figure 10: *event.WorkerProcess* class main metrics measurements

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
9	Ghost	<span style="color: green;">■</span>	<span style="color: yellow;">■</span>	<span style="color: orange;">■</span>	<span style="color: green;">■</span>	83	medium-high	low	high	low-medium



model.Ghost  
Coupling: low  
Complexity: medium-high  
Lack of Cohesion: high

Figure 11: *model.Ghost* class main metrics measurements

### 1.2.2.2 Dependencies (CodeMR, IntelliJ analyzer)

CodeMR allows also to inspect dependency relations between classes coupled with the metrics measured for each. We observe in the Figure 12 the same structure that in the class diagram. Once again the class *event.WorkerProcess* is displayed as problematic, being too complex and coupled with other classes.

We use the standard built-in tool of IntelliJ IDEA to instantiate the dependency matrix, illustrated by Figure 14. We clearly see reading 8th column that *event.WorkerProcess* depends on a lot of other classes from package *model*. This is also the case for *model.Map* that presents a lot of cyclic dependencies (red marked).

### 1.2.2.3 Compliance & bad smells (PMD, Designite)

PMD is a static analyzer that checks for problems of several natures in the code. It detected more than 1500 violations in the system 2, related to various topics (see Figure 13).

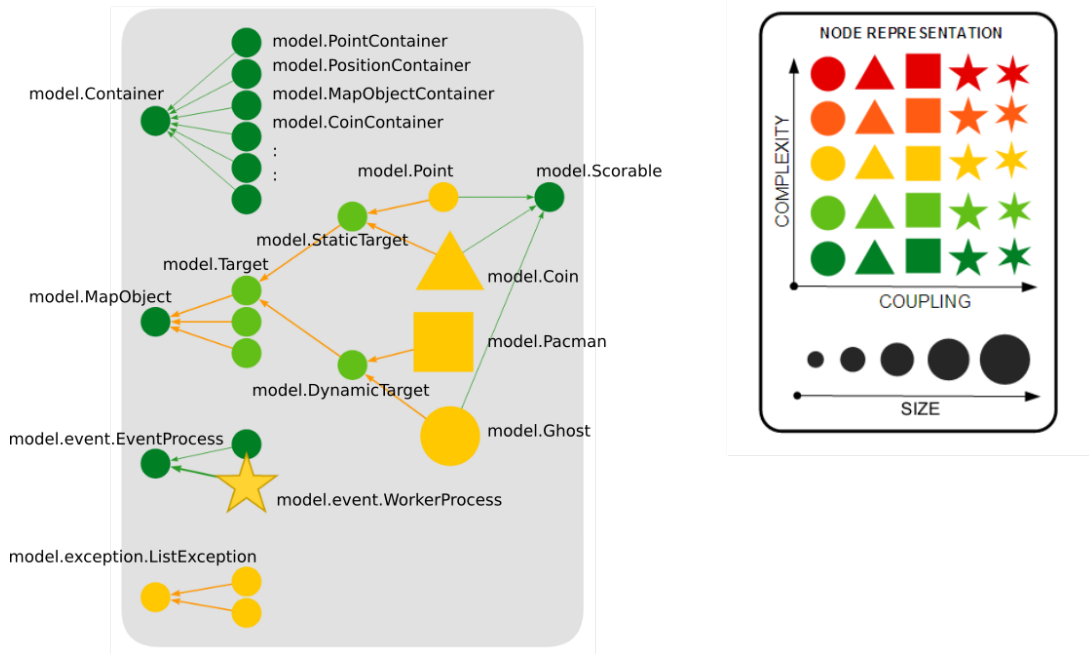


Figure 12: Inheritance relations between classes in system 2

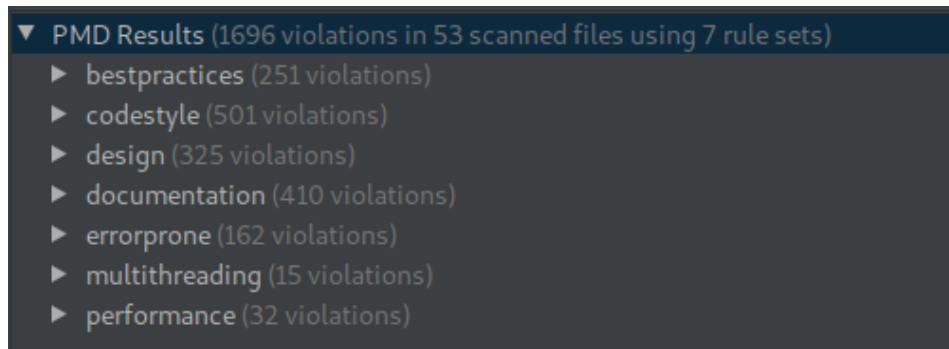


Figure 13: Violations found by PMD in system 2

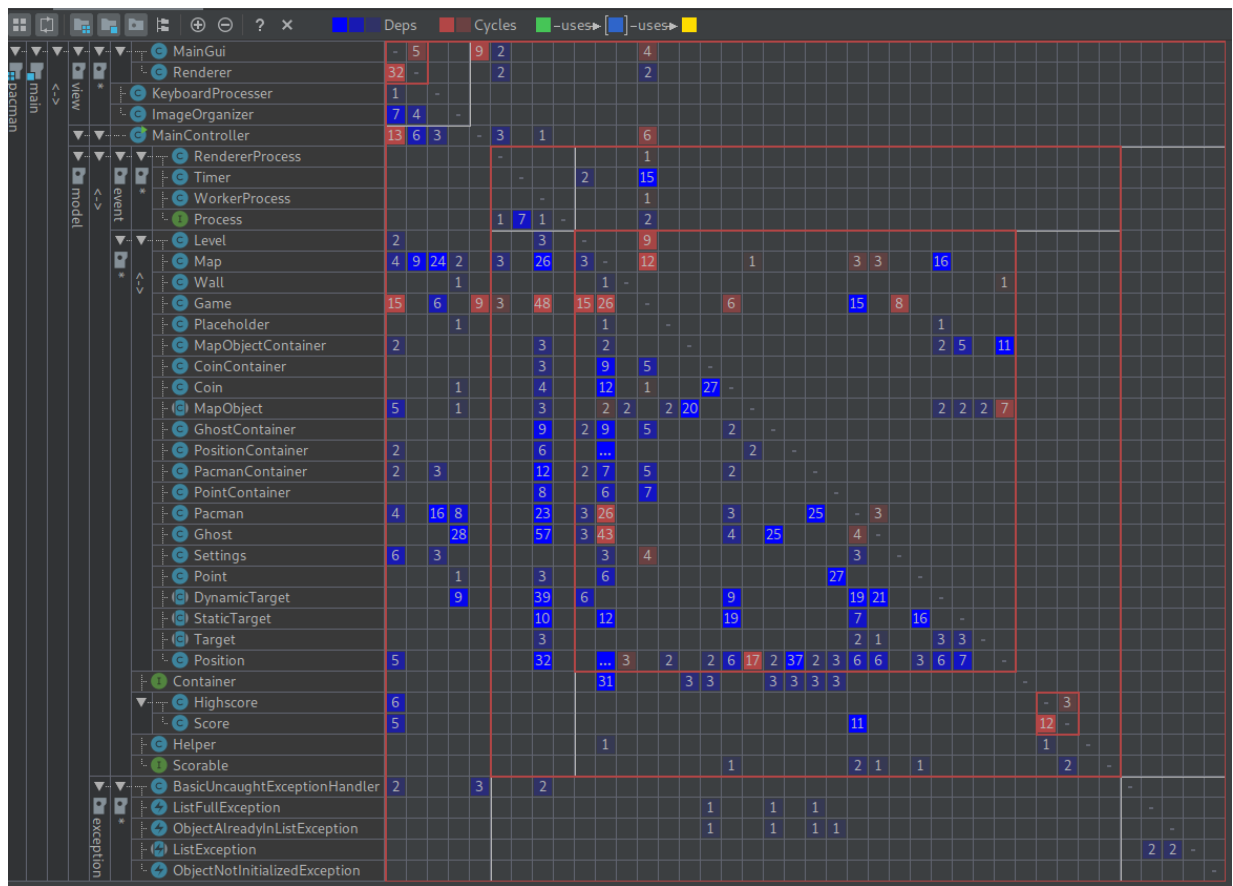


Figure 14: Dependency matrix for system 2

Designite is used to detect bad smells, a summary of the analyse is presented by Figure 15. The number of lines of code and classes is higher than the ones returned by CodeMR because the test sources were considered in the analysis.

```
--Analysis summary--
  Total LOC analyzed: 3181    Number of packages: 5
  Number of classes: 55    Number of methods: 361
-Total architecture smell instances detected-
  Cyclic dependency: 6    God component: 1
  Ambiguous interface: 0    Feature concentration: 1
  Unstable dependency: 2    Scattered functionality: 0
  Dense structure: 0
-Total design smell instances detected-
  Imperative abstraction: 0    Multifaceted abstraction: 0
  Unnecessary abstraction: 0    Unutilized abstraction: 7
  Feature envy: 4    Deficient encapsulation: 10
  Unexploited encapsulation: 1    Broken modularization: 1
  Cyclically-dependent modularization: 4    Hub-like modularization: 0
  Insufficient modularization: 1    Broken hierarchy: 7
  Cyclic hierarchy: 0    Deep hierarchy: 0
  Missing hierarchy: 1    Multipath hierarchy: 0
  Rebellious hierarchy: 0    Wide hierarchy: 0
-Total implementation smell instances detected-
  Abstract function call from constructor: 0    Complex conditional: 1
  Complex method: 6    Empty catch clause: 0
  Long identifier: 0    Long method: 0
  Long parameter list: 0    Long statement: 6
  Magic number: 165    Missing default: 3
```

Figure 15: Designite in-line use results for system 2

#### 1.2.2.4 Javadoc coverage (MetricsReloaded)

We use the IntelliJ MetricsReloaded plugin and its metric "Javadoc coverage" to get an overview of how complete is the initial javadoc. As shown by Figure 16, this is the case.

Package	Jc	Jf	JLOC	Jm
controller	100.00%	40.00%	42	13.33%
model	83.33%	19.82%	888	23.59%
model.event	100.00%	0.00%	48	0.00%
model.exception	100.00%	0.00%	145	50.00%
view	60.00%	0.00%	64	15.00%
Module	Jc	Jf	JLOC	Jm
pacman.main	81.25%	18.18%	1005	28.00%
pacman.test	100.00%	0.00%	182	0.00%
Project	Jc	Jf	JLOC	Jm
project	98.11%	15.58%	1187	21.69%

Figure 16: Javadoc coverage for system 2

### 1.2.3 Dynamic Analysis

#### 1.2.3.1 Running tests

Among the already written tests, 3 out of 67 failed at running time. The three tests are in *model.LevelTest* (*testGetLevel*, *testSecondsForCoin* and *testNextLevel*).

#### 1.2.3.2 Test coverage (IntelliJ built-in tool)

IntelliJ provides run configurations to dynamically analyze what are the parts of source codes covered by launched tests. Considering whole test packages, summary of the results are given by Figure 17. We observe the implementation benefits of a good test coverage, the most lacking part is package *view* but it makes sense by its nature.



## Overall Coverage Summary

Package	Class, %	Method, %	Line, %
all classes	94.4% (51/ 54)	80.8% (248/ 307)	73.8% (1033/ 1399)

## Coverage Breakdown

Package ▲	Class, %	Method, %	Line, %
controller	100% (2/ 2)	100% (14/ 14)	100% (32/ 32)
model	97.1% (34/ 35)	79.2% (183/ 231)	74.8% (676/ 904)
model.event	75% (3/ 4)	96% (24/ 25)	72.5% (111/ 153)
model.exception	100% (5/ 5)	87.5% (7/ 8)	81.2% (13/ 16)
view	87.5% (7/ 8)	69% (20/ 29)	68.4% (201/ 294)

Figure 17: Test coverage for system 2

## 1.3 System 3 (Robin)

### 1.3.1 Generalities

For this project the author provides no documents.

The Pacman maps are modeled under a `.txt` format, where each type of case are attributed a certain letter.

We also observe that in this Pacman implementation maps are modeled under `.tmx` format, that is a popular way to deal with board games<sup>5</sup>. Only one single basic map is provided.

In the project structure, there is `test` and `src` directory. In the first one there are the test classes and in the latter there are two directories, `Ressources` and `pacman_inf.d`. In `Ressources` we have the Pacman maps that are modelled under a `.txt` format, where each type of case are attributed a certain letter, and also the sound files in `.wav` format. `pacman_inf.d` contains all Java classes.

The building system provided with the implementation is hold by Ant. So a switch to Maven will be required to comply with directives.

### 1.3.2 Static Analysis

#### 1.3.2.1 Code metrics (CodeMR)

The dashboard illustrated by Figure 18 informs this software is doing really good.



Figure 18: CodeMR dashboard summarizing health of system 3

<sup>5</sup><https://doc.mapeditor.org/en/stable/reference/support-for-tmx-maps/>

The Figure 19 illustrates also the C3 metric but coupled with detailed packages view. We can see that every classes is doing good.

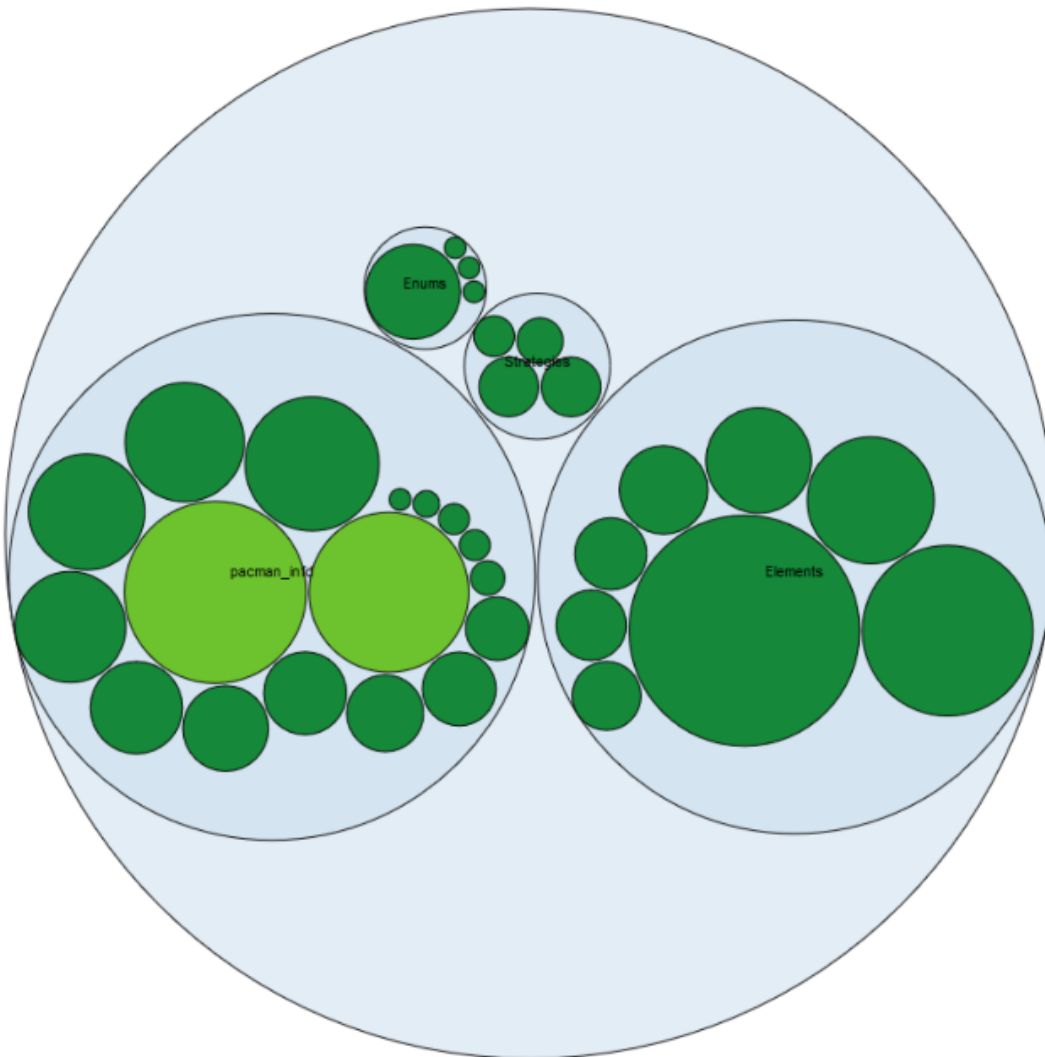


Figure 19: C3 Metric by package for system 3

There are no problems in the classes for almost all the metrics, only for two of them, we can see a problem: the Lack of Tight Class Cohesion and the lack of Cohesion of Methods. For the first one, nine classes have High or Very-High risks like the GameController or the View. And for the latter, three classes have High risks: GameController, ScorePanel and GameWorld.

1.3.2.2 Dependencies (CodeMR, IntelliJ analyzer)

We use the standard built-in tool of IntelliJ IDEA to analyze the dependency matrix, the result is shown on Figure 20.

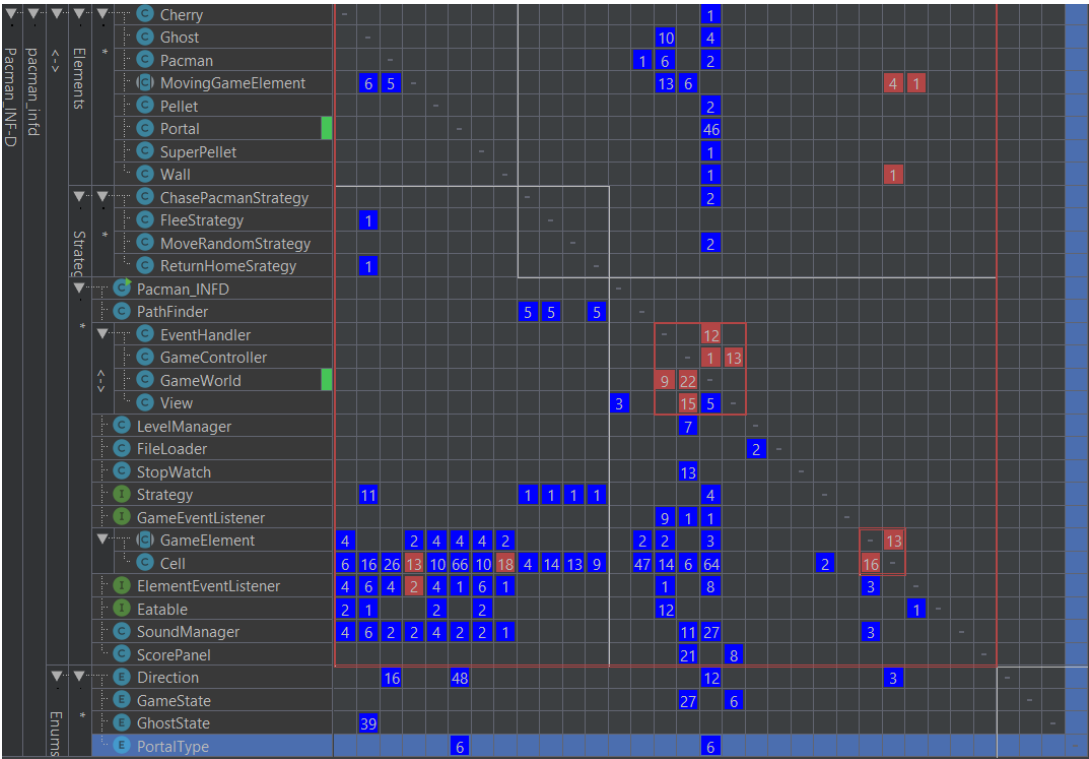


Figure 20: Dependency matrix for system 3

### 1.3.2.3 Compliance & bad smells (PMD, Designite)

We first used PMD, it detected more than 1343 violations in the system 3, here is the result:

- 1343 violations:
  - best practice: 54
  - code style: 414
    - \* 125: method arg could be final
    - \* 98: local var could be final
    - \* 60: short variable name
  - design: 503
    - \* 449: most is law of demeter 'only talk to friends'
    - \* 31: immutable field: private field values never change once object init could be final
  - documentation: 192
  - error prone: 167
    - \* 70: Bean member should serialize: make variable transient or static
    - \* 21: avoid literal in if condition
  - performance: 13

The result of Designite analysis is on Figure 21.

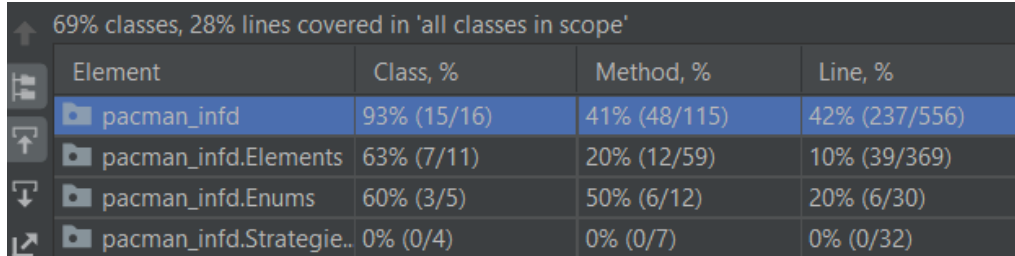
```
--Analysis summary--
Total LOC analyzed: 1844      Number of packages: 4
Number of classes: 31      Number of methods: 210
-Total architecture smell instances detected-
Cyclic dependency: 2      God component: 0
Ambiguous interface: 0      Feature concentration: 2
Unstable dependency: 1      Scattered functionality: 0
Dense structure: 0
-Total design smell instances detected-
Imperative abstraction: 0      Multifaceted abstraction: 0
Unnecessary abstraction: 0      Unutilized abstraction: 1
Feature envy: 0      Deficient encapsulation: 0
Unexploited encapsulation: 1      Broken modularization: 0
Cyclically-dependent modularization: 0      Hub-like modularization: 0
Insufficient modularization: 0      Broken hierarchy: 1
Cyclic hierarchy: 0      Deep hierarchy: 0
Missing hierarchy: 1      Multipath hierarchy: 0
Rebellious hierarchy: 0      Wide hierarchy: 0
-Total implementation smell instances detected-
Abstract function call from constructor: 1      Complex conditional: 3
Complex method: 3      Empty catch clause: 0
Long identifier: 0      Long method: 0
Long parameter list: 2      Long statement: 15
Magic number: 260      Missing default: 0
```

Figure 21: Designite in-line use results for system 3

### 1.3.3 Dynamic Analysis

#### 1.3.3.1 Test coverage (IntelliJ built-in tool)

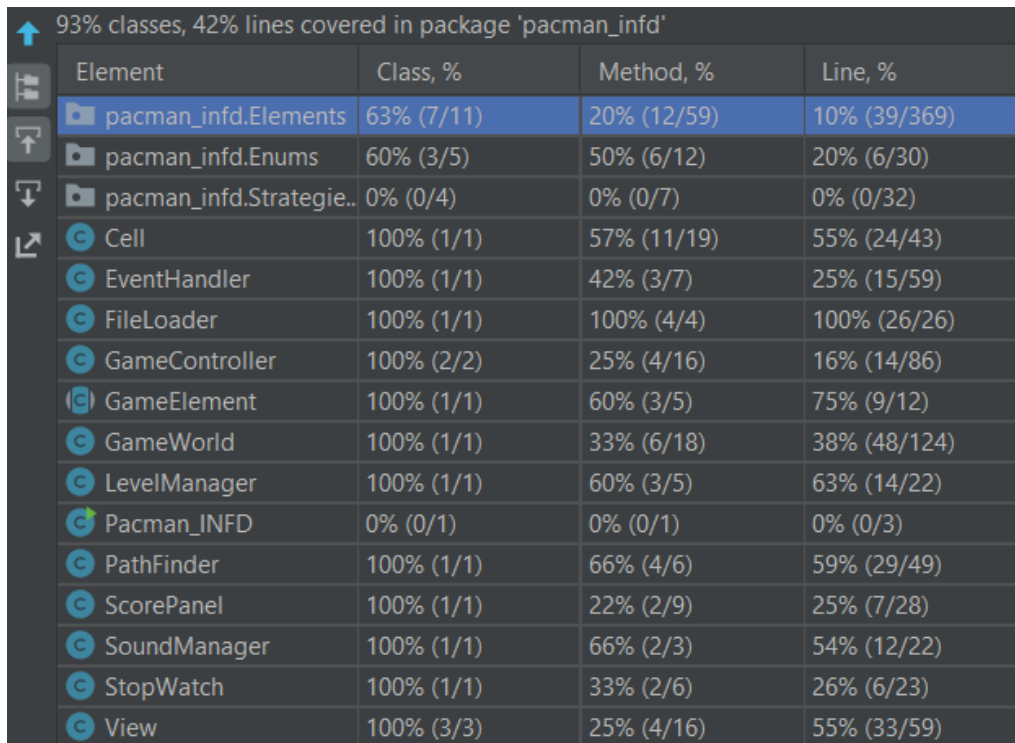
The test coverage of the given tests was analyzed, and the three following figures 22, 23 and 24 give the results. We can see that only 28% of the code is tested and 69% of the classes. We can see that neither of the strategies in the Strategies packages is tested, and in the Elements package only 12 methods is tested on the 59 methods that are available.



69% classes, 28% lines covered in 'all classes in scope'

Element	Class, %	Method, %	Line, %
pacman_infd	93% (15/16)	41% (48/115)	42% (237/556)
pacman_infd.Elements	63% (7/11)	20% (12/59)	10% (39/369)
pacman_infd.Enums	60% (3/5)	50% (6/12)	20% (6/30)
pacman_infd.Strategie..	0% (0/4)	0% (0/7)	0% (0/32)

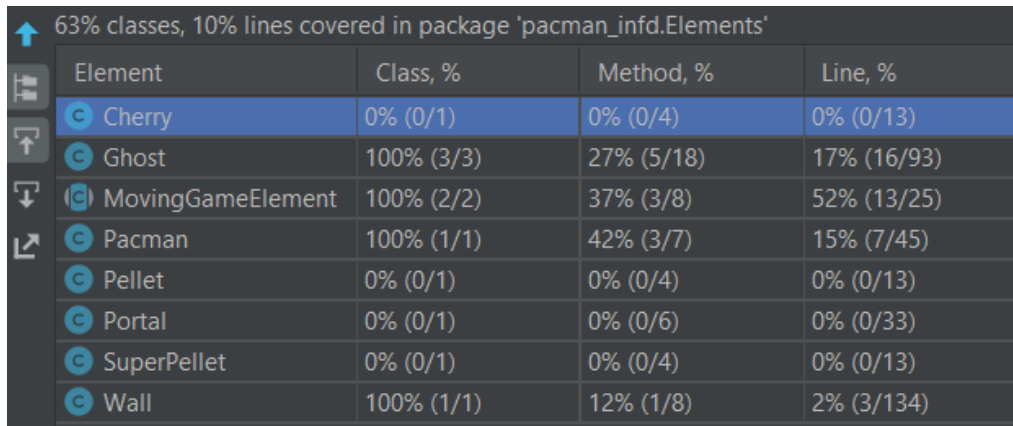
Figure 22: Test coverage for system 3



93% classes, 42% lines covered in package 'pacman\_infd'

Element	Class, %	Method, %	Line, %
pacman_infd.Elements	63% (7/11)	20% (12/59)	10% (39/369)
pacman_infd.Enums	60% (3/5)	50% (6/12)	20% (6/30)
pacman_infd.Strategie..	0% (0/4)	0% (0/7)	0% (0/32)
Cell	100% (1/1)	57% (11/19)	55% (24/43)
EventHandler	100% (1/1)	42% (3/7)	25% (15/59)
FileLoader	100% (1/1)	100% (4/4)	100% (26/26)
GameController	100% (2/2)	25% (4/16)	16% (14/86)
GameElement	100% (1/1)	60% (3/5)	75% (9/12)
GameWorld	100% (1/1)	33% (6/18)	38% (48/124)
LevelManager	100% (1/1)	60% (3/5)	63% (14/22)
Pacman_INFD	0% (0/1)	0% (0/1)	0% (0/3)
PathFinder	100% (1/1)	66% (4/6)	59% (29/49)
ScorePanel	100% (1/1)	22% (2/9)	25% (7/28)
SoundManager	100% (1/1)	66% (2/3)	54% (12/22)
StopWatch	100% (1/1)	33% (2/6)	26% (6/23)
View	100% (3/3)	25% (4/16)	55% (33/59)

Figure 23: Test coverage of the pacman\_infd package



63% classes, 10% lines covered in package 'pacman\_inf.d.Elements'

Element	Class, %	Method, %	Line, %
Cherry	0% (0/1)	0% (0/4)	0% (0/13)
Ghost	100% (3/3)	27% (5/18)	17% (16/93)
MovingGameElement	100% (2/2)	37% (3/8)	52% (13/25)
Pacman	100% (1/1)	42% (3/7)	15% (7/45)
Pellet	0% (0/1)	0% (0/4)	0% (0/13)
Portal	0% (0/1)	0% (0/6)	0% (0/33)
SuperPellet	0% (0/1)	0% (0/4)	0% (0/13)
Wall	100% (1/1)	12% (1/8)	2% (3/134)

Figure 24: Test coverage for the Elements package

## 1.4 Comparaison of the 3 systems

From what we discussed, it appears that the system 1 is definitively the best. It is already fully documented and well tested, so robust (for example using `assert` Java statement). Even if it doesn't appear clearly in the analysis, the structure and the patterns used are better as well. For example, system 2 overuses the Singleton design pattern, leading to a non structured code. This also leads to some bugs the authors apparently didn't correct.

The number of test doesn't make it all, as system 2 proves. There are a lot of tests, but some just don't pass or are empty. Even if apparently good using CodeMR, system presents some catastrophic classes like `Wall`, and some bad programming practice as very long `else-if` statements that gives an idea about how much authors don't apply well OOP programming guidelines.

## 2 Quality improvement

### 2.1 System 1 (BOOSKO Sam)

In general, the provided implementation didn't need improvement but few were done like making an object as a parameters for method with too many parameters. Furthermore, few magic numbers were fixed in unit test implementation. Mainly, magic numbers were in testing codes, then it's normal to have them there. It's more readable to understand them and write them.

The part of the implementation in *MapParser* which manage the creation of element of the game from a text file were modify to increase the readability and help new developers to add new elements in the project. At first, it was done with a switch case on a *Char*. To make it more modular, an Interface, called *ISquareBuilder*, was create. This class take as parameter and object, *AddSquareParameters*, which contains all information of the game to create easily and new element. An abstract class, *ADefaultSquareBuilder*, implementing this interface, is also created. This abstract class helps to reduce the duplication code. Finally, it's pretty easy to add a new element on the grid by adding the *Char* key and the responding *ISquareBuilder*.



## 2.2 System 2 (Rémy)

We employ the following methodology to improve the system in its current state :

1. Reviewing the whole code and correct problems of form. It allows to acquire a global overview of the implementation to lead next steps and improve code quality on a per-class basis. These refactorings comprise, among others :
  - completing the Javadoc
  - getting rid of forgot/useless artifacts
  - detecting and correcting code smells

The main tool used during this step will be the IDE (IntelliJ) and plugins associated with like Designite, which allow on-the-fly analysis and pointing out problems in the code itself.

2. Reviewing the system structure and correct structural design problems. From the acquired global overview, it is possible to have an idea of drawbacks implied by the system design. The refactoring will occur at a class-to-class relations level, and will then impact package structure level. Some tools and metrics, for example from CodeMR, can be used to lead this step : a class reported too long may be splitted into more than 1 class, inheritance should be used better, etc.
3. If some tests are not passing, find the reason and correct them.
4. Complete the tests based on test coverage reports generated (from IntelliJ).

Of course, after each of these steps, the current yet written tests must be launched to control the consistence of the implementation.

### 2.2.1 Step 1

We read through the whole code and corrected what had to be in a first time. Some smells are straightforward, like avoiding Magic Numbers detected by Designite. The help of IntelliJ is precious to get rid of some deprecated/forgot code artifacts the authors left. Some problems reported by Designite are not regarding the actual usage and left as-is.

### 2.2.2 Step 2

It was figured out the current implementation presents some drawbacks. It is mainly related to code duplication for already written objects for **Container** purposes (can be found in provided class diagram). The fact is that authors wanted to write an overlay to describe different collections of other objects in the implementation (**Coin** (= pills), **Point**, etc.). But they wrote a specific class for every type of object to contain, albeit implementing the common **Container** interface, this design is very poor and inelegant, leading to code duplication and increased number of classes. Keeping in mind what were each class written for, we redesigned this part of the implementation in a better way. It also brought the occasion to cluster **Container** class concerned in a new package to improve the project structure readability.

The resulting structure is depicted by Figure 25. We now consider to pass through a **Containers** class to construct any **Container** needed in the rest of the implementation. Its static methods instantiate the right container with adapted type of content from generic classes. These instances are shown in orange in the Figure 25. The restricted typing **E extends MapObject** allows getting element(s) considering a **Position** object. A **PositionContainer** is a specialized container to hold coordinates (already present in original code), so we don't use an index but form a key from the couple  $(x, y)$ . The method **getRange(...)** allows to get a subset of contained positions, considering a rectangle selection formed from two positions given in parameters. For some object types (**Point**), an overload is necessary to comply with the rest of the implementation. Anyway, this is somehow masked because all containers instances are retrieved from the class **Containers** mentioned above, not represented in the figure.

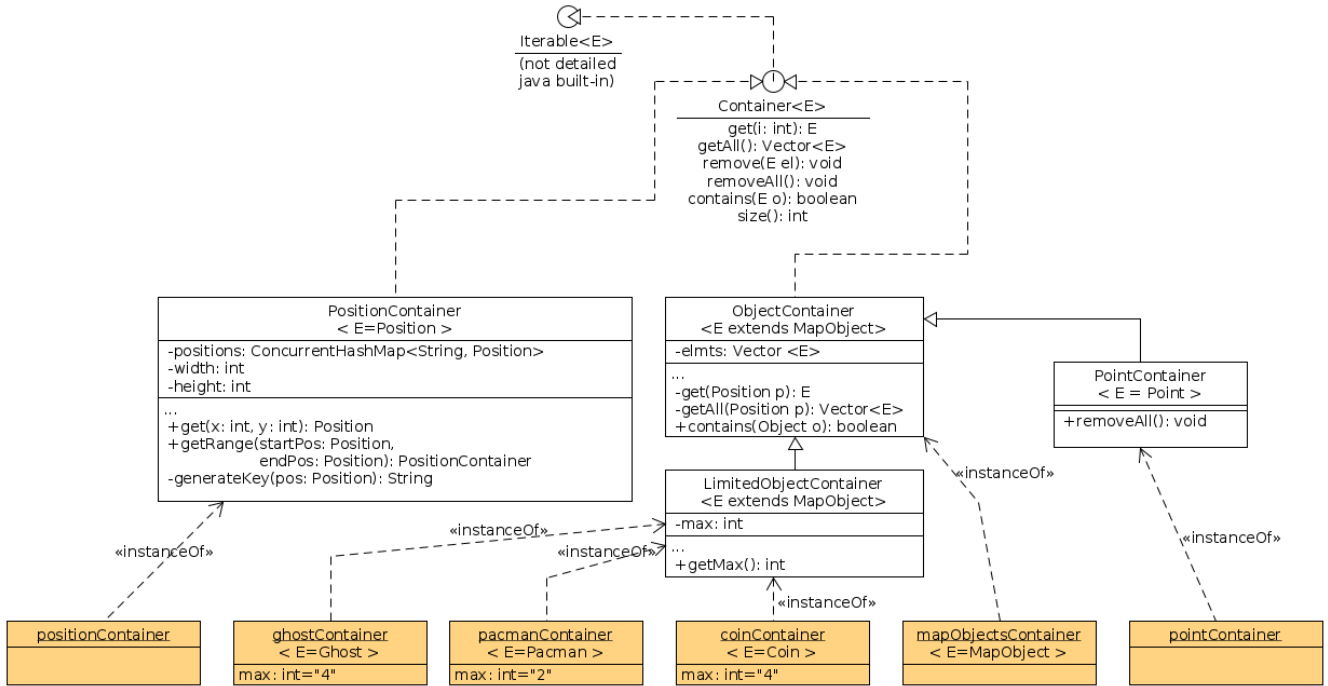


Figure 25: New class structure for Containers part of the implementation

An other problem is related to the `Map` class. We observe the `Map` itself and the objects that will represent it in the application are instantiated in raw. Coordinates for each element are given in the code itself, that is a bad practice. However, due to the lack of time and considering the picked extension doesn't relate to maps, a trade-off is taken. A new class `MapPlacer` is written to hold the placement of `MapObjects` on the declared possible `Positions` of the unique considered map. It cleans up the code of the original class `Map` that should only hold dynamic operations such as reinitializing the content of the map when the player passes a level.

Concerning package structure, some changements were also needed because the `model` package had no sub-level (grouping almost 30 classes). We added a subpackage `model.container` that holds all the hierarchy depicted above and another `model.mapobject` to group all classes standing for the actors/components of the game (`Wall`, `Pacman`, `Coin`, etc.).

### 2.2.3 Step 3

Some already written tests were not passing, or passing but throwing an exception. That was mainly due to the management of multithreading and game resetting not properly. Also, the usage of `static` elements lead to inconsistencies. Leveraging some adjustments in the source code, we managed to get the system more consistent and the tests passing in a deterministic behaviour.

### 2.2.4 Step 4

The provided existing tests are numerous but not relevant for some. Firstly, there are empty tests, whose only signature is written. They were filled in the right way to evaluate what they're expected to. Some others focus only on get/accessors and methods that are even never used in the rest of the implementation. So, tests oriented to behavior evaluation are mainly missing. We added some for `Map`, `Ghost`, `Pacman`, etc. They aim to ensure the game is running as expected. More diverse additions were done all over to improving testing quality.

As some new classes have been written to improve the system quality, the associated tests were also written. We have `PositionContainerTest` and `TimerTest` to evaluate the new implementation parts described above.

## 2.3 System 3 (Robin)

The first change was changing the building system from Ant to Maven.

After that, we used PMD and Designite to find problems and refactor them. The access of classes, methods and variables was changed depending of the need, it was set to protected or private if it was possible. Some variables were set to final. We found magic number, and tried to remove them if it was possible and usefull.

Then some long methods were divided into multiple methods for more readability. Some switch-case were also added instead of long if-else. Some new method were also added.

The Direction Enumerator class was changed to make it smaller and not just a succession of switch-case.

The structure of the classes was also changed. The source code of the project is now divided in 5 packages: Elements, Enums, Fileloader, Game and Strategies. Elements contains all the moving elements of the game, like Pacman or the ghosts, and the other game elements like the cherry or the pellet. The Enums packages contains all the enumeration that are used in the project. The FileLoader package contains all the classes that are used to manage the loading of a level. Strategies contains all the strategies and pathfinders that are used by the ghosts. And the Game package contains everything else that is used for the game, the GameWorld, the Controller or the Listener for exemple.

Then the GUI was changed for Pacman, before changing the code Pacman wasn't changing his orientation on the application, now we can see Pacman change it's orientation. There is also a little animation for the mouth of Pacman, he opens and shut his mouth to simulate eating.

Some new test where also created on the test package to cover more code. The different strategies of the Strategy package was tested. New tests for the eating of pellet, cherry and super pellet was also created.

## 3 Adding basic functionalities

### 3.1 System 1 (BOOSKO Sam)

The provided project missed basic functionalities, such as the continuous PacMan Moving, fruit elements, power pellet...

The first feature added is the continuous PacMan moving, a new class is created for it, *PlayerController*. This class contains information from the game and have a scheduled action. This action is simply to move the pacman to the register position. This direction can be changed by a key listener. The speed of the pacman is managed here with two attributes, 1) *Speed* and 2) *SpeedModifier*. With them, the final speed is compute as  $speed * speedModifier$ . Where the speed unit is the number of tiles per second. The speed modifier is thought to help the implementation of extensions.

Secondly, to be able to add new elements that pacman can eat on the board, the class *Pellet* is edited to add an action *onEat(Level level, Player player)* where the player is the pacman who ate the element. This action is called when a player have a collision with a pellet element. With this method, it was easy to implement the feature of scared ghosts (Pacman can eat a scared ghost) and the feature of new life by eating fruits.

Everything was thought to help another person to add new features from the sectioned extension. Adding a new element on the board take 1 minute. The longest time being the implementation of the action performed of this new element.

### 3.2 System 2(Rémy)

The following fonctionnalités were pointed out as missing in this system :

- The last two pills eaten in a Level must give an invicibility of 5 seconds instead of 7 seconds
- A ghost must disappear when munched and respawn in the ghost base after 5 seconds
- Consecutive eaten ghosts must give more points (200-400-800-1600)
- There is a timer ruling each level (stoped when pacman is hunting)

The commit corresponding to the final version of the basic game is

<https://github.com/RemDec/pacman-system2/commit/cb5e36143d6ab84a9c02d80cd4c58ae56ff0e8aa>

These fonctionnalités are somehow straightforward to implement in this system. Of course some unit tests are written to verify the new behaviors. The only remark would go to the timer. As this system is intended to strengthen the difficulty increasing the “refresh rate”, this rate govern the display frequency. This is why the displayed timer increments step-by-step (we avoid using new display threads to keep consistency and good integration in the actual system). As this timer feature was not present, new classes `Timer` and `TimerProcess` were written to be easily integrated with the already existing `Scheduler` (renamed, previously `Timer`). The timer has been integrated to the interface as depicted by Figure 26.

It was also necessary to correct some hidden bugs related to the behavior of ghosts. Sometime when eaten, they stay at their position instead of respawning in their base. The code was modified to obtain the right behavior and some tests were added to ensure this.



Figure 26: New timer integrated in the interface (below the map)

### 3.3 System 3 (Robin)

In this implementation, we couldn't advance to the next level once we finished eating all the pellets. We first changed that. Now a level can be succeeded and when it's done we advance to the next level, there is 3 level in total but the first 2 are the same.

The levels were also re-written, in the levels there were some disconnected regions, so we removed that.

After that the behavior for the super pellet/power pill was also changed. The clock was not set in pause when the pill was activated. The score gained by eating the ghosts was also not the one set in the rules, so it was also changed. The time for the pills was also changed, the first two lasts 7 seconds and the last two lasts 5 seconds.

Eating a cherry on the initial implementation wasn't making pacman gain an additional live, so that was also added.

The final commit,for the implementations can be found here :

<https://github.com/irikay/pacman-system3/commit/c34569f5095c552876e3a12a91c8d73975448cbe>

## 4 Adding new features

### 4.1 System 1 (Rémy)

### 4.1.1 New features discussion

The integration of special fruits/boxes in this system is quite facilitated by the quality of the implementation, which allows to define new unities and their collision map in a generic way. It is done by subclassing `Unity` class or one of its already existing subclasses. For example, for new fruits that apply an effect within a given duration, a generic class `SpecialPellet` is written, extending `Pellet`. It handles the duration and resetting original state once exhausted. Each fruit is a short subclass of it (`PotatoPellet` for example) which implements its logic depending the associated effect. We managed to give a variation in each effect that makes sense with the current game state. For `PotatoPellet`, the more pacman has score, the juicier he looks for ghost so they are buffered for an increased duration. Note that we made the choice that duration effects are not cumulative to keep the game clear and the current Pacman state always indicated to the player (thanks to the Pacman's skin). Though Pacman can still become a hunter during a special effect (indicated by ghosts skin).

Implementing special boxes is also somehow straightforward : we subclassed `Unity` with `SpecialBox` and each box is a subclass of the latter. These boxes have an impact on the game logic and are not especially coupled with a duration. It lead to modifications on the collisions logic. For instance, `BridgeBox` implements bridges that required an introduction of the vertical level notion (`DOWN` and `UP`), a state associated with each unit. We reconsidered collisions to be handled occurring only between units in the same vertical level.

The detail about each special unit and the way effects vary depending current game state can be found in the corresponding class documentation. An illustration of the implemented special units is provided by Figure 27 (illustrating actual effect is quite difficult for most, it should be played instead).

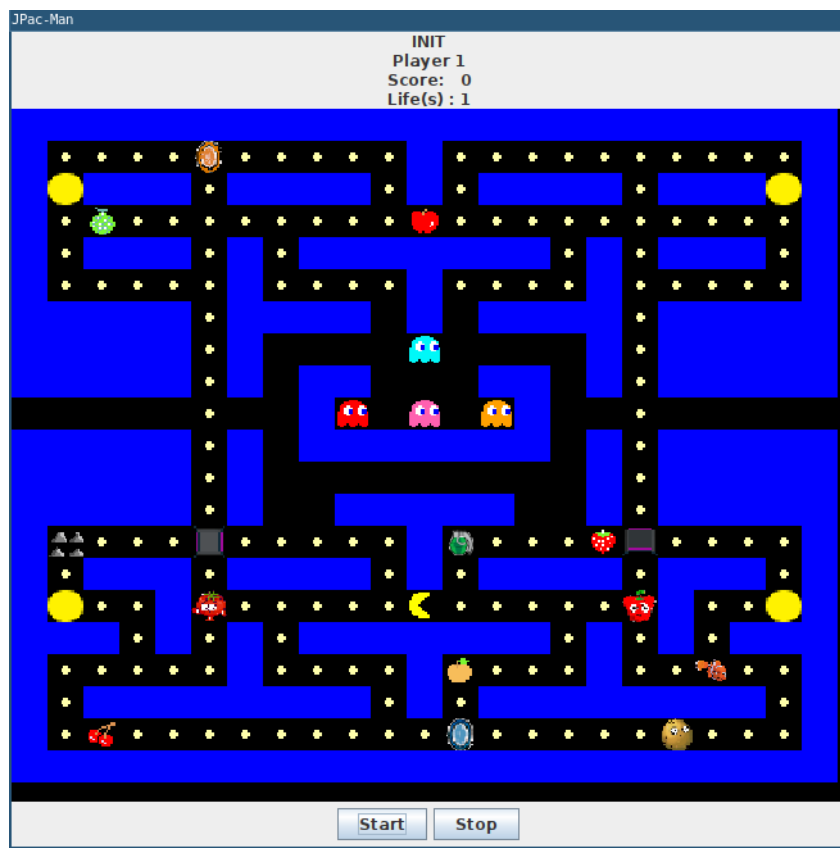


Figure 27: Illustrating all special units on the bottom of the map

Apart from the effect of special units, it was also asked to make them dynamically spawn taking account the current state of the level. This new mechanic is handled by the class **SpecialUnitSpawner**. At a fixed time interval (7 seconds), a try is done to spawn a special unit on the board : this decision is made in a probabilistic fashion. The process accounts for the following guidelines :

- if the board is nearly empty (ie. there are a few pellets compared to the number initially present), it could be good to fill it a bit to change the destiny of this game that seems nearly over. So chances to spawn a new unit are higher in this configuration (exact formula in code).
- as pellets are eatable, that is not the case for boxes that are persistent. To avoid overloading the board, we set a probability to choose a pellet at 0.7.
- in the case of a pellet to spawn, we have possible bonuses and penalties. The decision is made looking to the current player score. The higher it is, the closer he should be to the win and it also indicates the player is good. So we make his job harder, giving more probability to spawn a penalty pellet.
- the decision of which box/pellet will spawn among available is uniformly random.

Finally, we restructured the packages for special units to group pellets and boxes. The final commit, with a playable and balanced game and all tests passing is accessible here :

<https://github.com/Lroemon/pacman-system1/commit/e7a404331bb5577c5e8d6145059f7c6379b029e5>



## 4.2 System 2 (Robin Sch  rer)

Working on this system to implement new feature wasn't easy because almost all the logic of the collisions was done in the WorkerProcess class, in this class there is a lot of Feature Envy, this class also have a lot of condition that test the Ghost or Pacman state to see what can happen.

This system also doesn't provide a way to change the speed of the DynamicTargets so I had to implement that. The problem is that the run method in WorkerProcess is calling the move function of Pacman on each run, so doing a proper way to handle Target's speed would have needed a lot of refactoring. So I choose to change the REFRESH\_RATE of the run method to 0.1 second and the Target has a speed between 0 and 10. At a speed of 0 the target never moves, at 10 the target moves on each run call, at 5 the target moves every 5 call, etc. It's not the best method, but it's a easy way to create that fonctionnality.

In this system the map is really small and adding new element on the map isn't an easy task, as the map is created in the MapPlacer class and not by parsing a file. I choose that every type of fruits is available at the beginning of the game, just so you can try them, also some more fruit will randomly pop in the board if Pacman has enough score.

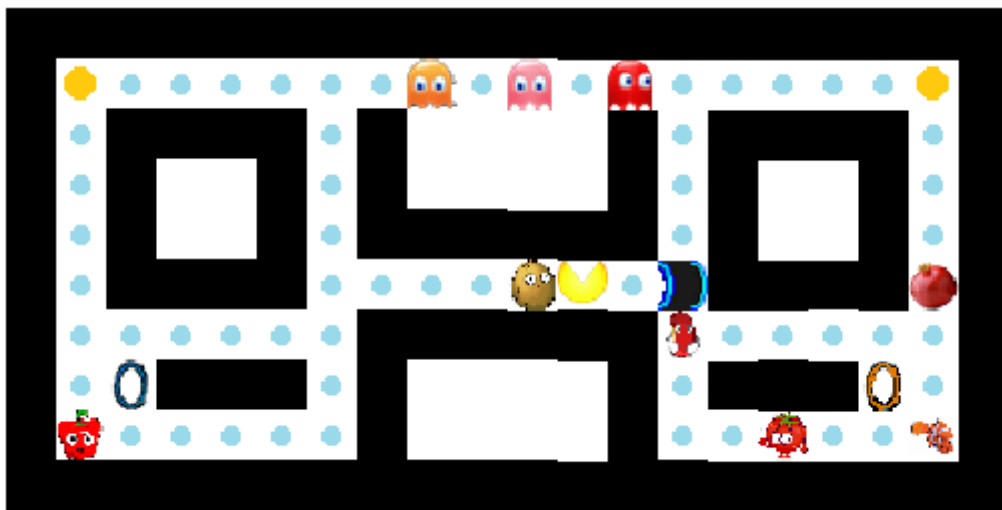


Figure 28: The system 2 map with the new blocs/fruits

The fruits were created using the StaticTarget abstract class as eating a fruit will behave like eating a coin, except that the eaten fruit will trigger a new action. The action will, depending on the fruit, change the way Pacman behave for a certain period of time. Some of the fruits will change Pacman state, like the Bean or the Tomato.

**Fish** This 'fruit' will make Pacman stops for 3 seconds. It will just change the speed of pacman to 0 for these 3 seconds.

**Grenade** I chose for this fruit to kill every ghosts in a range of 4 blocs and it doesn't care of walls.

**Pepper** The pepper will increase Pacman's speed to the maximum when eaten, so it will be set to 10 for 10 seconds.

**Potato** When eaten, the potato will increase ghost's speed to 7 for 5 seconds.

**Tomato** This fruit will change Pacman's state to INVINSIBLE for 4 seconds, in this mode Pacman cannot be eaten by the ghosts. So as this systems handle collisions in the WorkerProcess I added a condition that the ghost won't eat pacman if he is in this mode. The figure 29 shows pacman when he eats this fruit.

**RedBean** This very hot fruit will transform Pacman into Fire mode for 5 seconds, while in this state, he will throw Fireball each time he moves. The fireball kills every Ghost it met and it can cross multiple ghosts. Fireball is

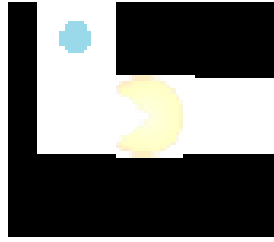


Figure 29: Pacman when in invinsible mode

implemented with the DynamicTarget abstract class. The figure 30 shows pacman when in fire mode.



Figure 30: Pacman when in fire mode

The new map objects were implemented using a new abstract class that extends the MapObject abstract class. These objects will trigger some action when a Target is on it.

**Trap** This box will make the ghost or pacman's speed to 0 for 3 seconds, it can only handle 1 target at a time, so if a ghost is stuck on the trap another ghost can cross it.

**Teleporter** There is 2 types of teleporter, an entry and an exit. When created a teleporter will be linked to another teleporter or linked to nothing, if the teleporter has a link it will move Pacman to the exit teleporter. On Figure 28 we can see the two types of teleporter, the blue one is the entry teleporter and the orange one is the exit.

**Bridge** The bridge can normally be used to make pacman cross crossroads, but in this systems there is none on the map, so in this case it will just block Pacman. For this block I created a new kind of state for DynamicTarget, the bridgeState, this state will have 3 main states: NOT\_ON, UNDER and ON. So it will be used to know if a target is not on a bridge, or under/on a bridge, collision between a ghost and pacman that are not on the same state won't happen. The Figure 31 shows Pacman on a bridge and figure Figure 32 'shows' pacman under a bridge.

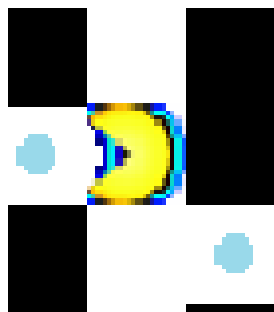


Figure 31: Pacman when on a bridge

The last commit for the implementation of the extension can be found here: <https://github.com/RemDec/pacman-system2/commit/5e6e4c46c404a25f3b98c9e9be61b890709eb71b>

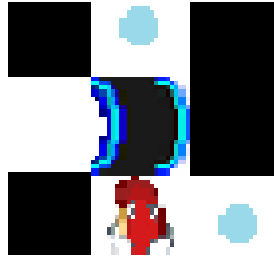


Figure 32: Pacman when under a bridge

## 4.3 System 3 (BOOSKO Sam)

### 4.3.1 Quality Improvement

The title sounds weird in this section because we had to improve the quality in the previous step. Unfortunately for me, after reading the implementation and listening Robin speech about his work, I could notice that, it wouldn't be easy to add easily new features as demanded. I decided to rework few part of the system 3 based on what I did and saw in System 1.

Firstly, I changed the method based on a *switch-case* in an object which creates a game element from a *character* by a *HashMap* where the keys are a *Character* and the value an object, *IElementBuilder*. The same way is used to create *Cell* of the board. It's useful to bridge because bridges are not a game element but cell.

Secondly, I implemented a new object to manage collision with my new game element. Because, it wasn't my job to rework this system. I didn't change the previous collision detection. I added mine just to manage new collisions. It's based on a double *HashMap* accesible by a couple of key (*Class object*) to get an *ICollisionAction* to perform the action on the game from this collision.

Finally, the last reworking is on *MovingGameElement*. This class doesn't have any documentation and have an attribute *speed*. It's quite difficult to understand the unit used for the speed and very diffult to **play** with it. Then, I decide to modify it to have a speed based on the unit, such as the number of cells traveled per second because usually, the speed unity follows the format such as  $x$  unit of distance per  $t$  unit of time. Also deleting code duplication from *Pacman class* and *Ghost class*, both extending *MovingGameElement*.

### 4.3.2 Features

To be able to add new game element, a new abstract class is created, *AExtensionElement*. This class provides a simply implementation to display an image for this game element. Also, to use this class such as a game element, it extends the main class *GameElement*. Another little feature is the possibility to change the Pacman color to have a visualisation about his state.

#### 4.3.2.1 Traps

When Pacman goes on this game element, he is trapped for a random time between 1 and 4 seconds and turns red. This feature is implemented in *pacman.inf.d.Elements.ExtensionElements.TrapElement* (see Figure 33).

#### 4.3.2.2 Teleportation

This game element is working by couple, because when the Pacman use a portal, he needs the output of this portal. In the rule chosen, only two portals can be created on a board. Also, a timer is used after the usage of a portal making the link broken during 2 seconds (see 34). This feature is implemented in *pacman.inf.d.Elements.ExtensionElements.TeleporterElement*.

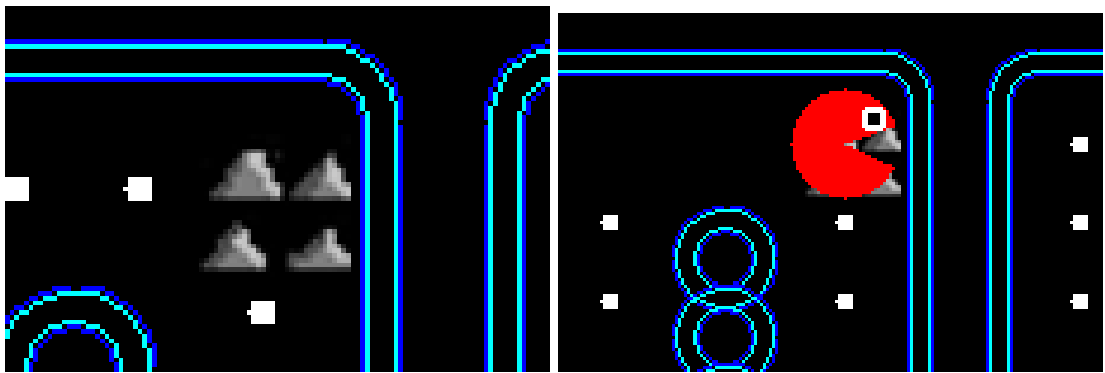


Figure 33: System 3 - Traps

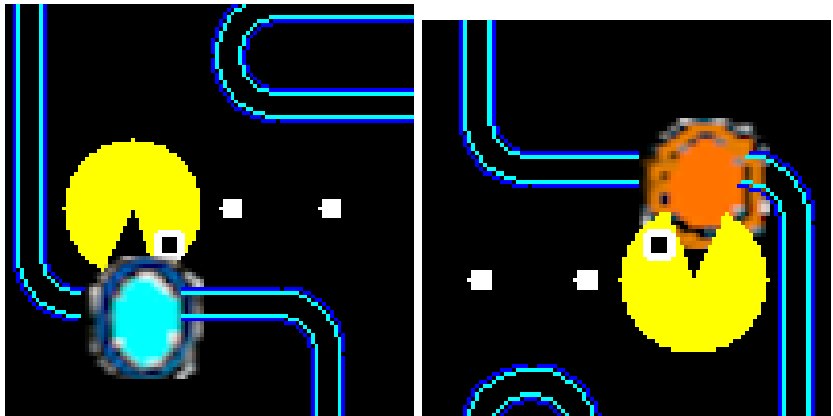


Figure 34: System 3 - Portals

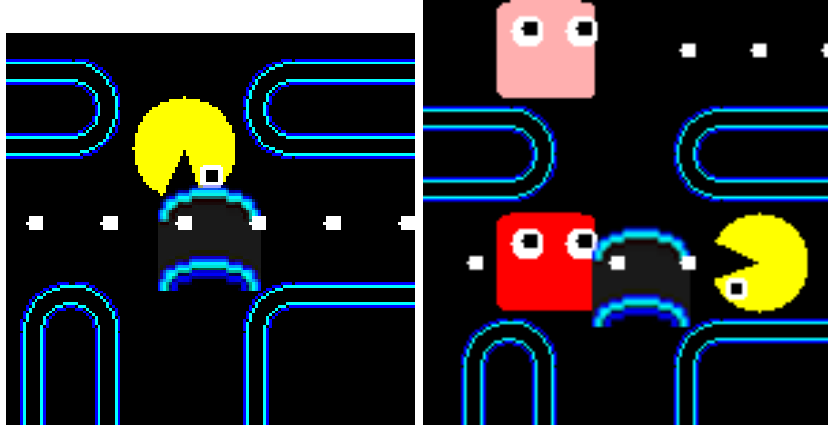


Figure 35: System 3 - Bridges

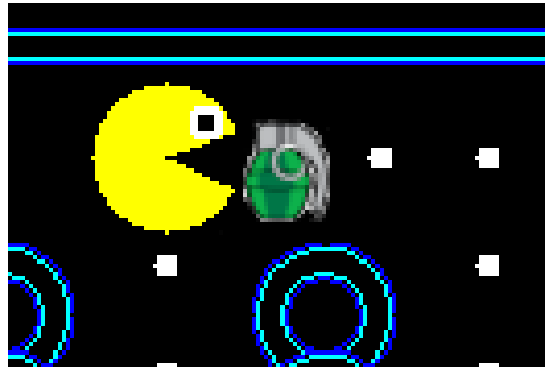


Figure 36: System 3 - Grenades

#### 4.3.2.3 Bridges

The bridges was the most complex feature to implement because it's not considered as a game element in this System. The best way to make this feature is to create a new kind of *Cell*. This special cell assesses where is the *MovingGameElement*, under or above the bridge. From this, it requires an override of default methods from *Cell*. Two new lists are used, one with *MovingGameElement* under the bridge and another one with *MovingGameElement* above the bridge. Also, updates are made on the *EventHandler* to let the collision checking assessing in the *Cell* object and on the *MovingGameElement* to be able to know the current direction of the element. Thanks to these updates, the new *Cell* kind can manage how it wants the collision between game elements on itself and assesses as said where is the game element depending on his direction. This feature is mainly implemented in *pacman\_infd.Game.BridgeCell* (see 35).

#### 4.3.2.4 Grenade

The Grenade is implemented in *pacman\_infd.Elements.ExtensionElements.GrenadeElement*. In the rule chose, the grenade kills all ghosts around in the maximum distance of 4 cases with a special specification. The grenade doesn't cross the walls. It means that a ghost can survive if he is protected by a wall. The main method for this is implemented on *Cell* where a *Breadth First Search* is implemented. The method takes two parameters, the Class (extends *GameElement*) searched in a maximum distance given as the second parameter (see 36).

#### 4.3.2.5 Red Beans

The Red Bean, implemented in *pacman\_infd.Elements.ExtensionElements.RedBeanElement*, is a special element. When Pacman eats this bean, Pacman is slowed down by 50% but he shoots 3 projectiles per second. These projectiles have the speed of *Pacman* + 10. Also, Pacman turns in dark red. The *MovingGameElement* is used to create these projectile easily and a new collision is added in the *CollisionMap* between *Ghost* and *Projectile* (see Figure 37).

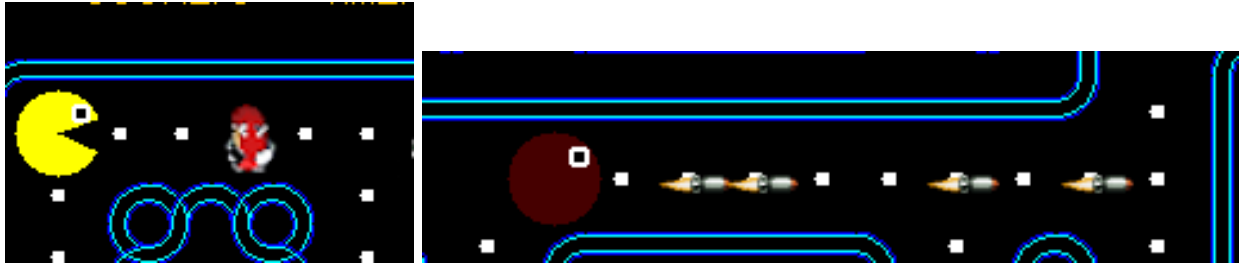


Figure 37: System 3 - Red Beans

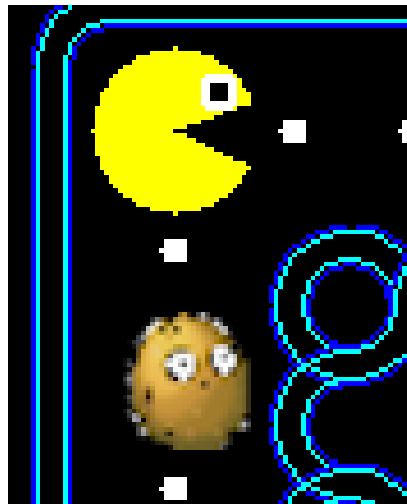


Figure 38: System 3 - Potato

#### 4.3.2.6 Potato

The Potato, implemented in *pacman\_inf.d.Elements.ExtensionElements.PotatoElement*, is a simple element which increase the speed of all ghost on the board by 2 during a time. For this element, the visual information is done by ghosts which are faster than Pacman (see Figure 38).

#### 4.3.2.7 Fish

The fish is another simple element and the last, this one, implemented in *pacman\_inf.d.Elements.ExtensionElements.FishElement*, slows down the Pacman to a very slow speed and he turns cyan (see Figure 39) .

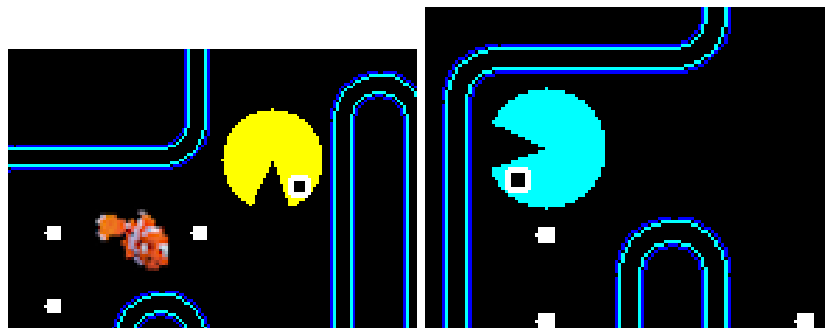


Figure 39: System 3 - Fish

#### 4.3.2.8 Elements Spawning System

All elements can be setup in the file map. During the game, two elements, the bridges and the teleporters don't spawn by the spawning system. This spawning system works such as for each pellet ate, there is 10% that a random elements spawn on an empty *cell*. This system is implement in the object *GameWorld* on the method *placeCherry-OnRandomEmptyCell* already provided.

## 5 Quality evolution analysis

### 5.1 System 1 (Rémy)

From the extension development point of view, this implementation was really nice and easy to extend. The code was already well documented, and all is structured in a way it is quickly understandable. As the quality analysis at the beginning of the project mentioned, this is a quite good implementation that suffers really few drawbacks. We lead firstly some statis analysis to get an overview of how the implementation evolved.

```
--Analysis summary--
  Total LOC analyzed: 6019    Number of packages: 12
  Number of classes: 93    Number of methods: 535
-Total architecture smell instances detected-
  Cyclic dependency: 1    God component: 0
  Ambiguous interface: 0    Feature concentration: 6
  Unstable dependency: 1    Scattered functionality: 0
  Dense structure: 0
-Total design smell instances detected-
  Imperative abstraction: 0    Multifaceted abstraction: 0
  Unnecessary abstraction: 1    Unutilized abstraction: 55
  Feature envy: 0    Deficient encapsulation: 16
  Unexploited encapsulation: 1    Broken modularization: 0
  Cyclically-dependent modularization: 1    Hub-like modularization: 0
  Insufficient modularization: 4    Broken hierarchy: 0
  Cyclic hierarchy: 0    Deep hierarchy: 0
  Missing hierarchy: 1    Multipath hierarchy: 0
  Rebellious hierarchy: 0    Wide hierarchy: 0
-Total implementation smell instances detected-
  Abstract function call from constructor: 0    Complex conditional: 1
  Complex method: 0    Empty catch clause: 0
  Long identifier: 0    Long method: 1
  Long parameter list: 11    Long statement: 4
  Magic number: 41    Missing default: 3
----
Done.
```

Figure 40: Results from Designite for final system 2



We observe in Figure 40 that the system has consequently increased in size, almost doubled in multiple counting metrics (LOC, number of classes/methods). The increasing size of the code lead to some new bad smells, but also avoided some previously present. Some metrics like magic numbers seem still (41 now, 39 previously), but actually this is not the case considering the implementation almost doubled in size. An metric that increased a lot is the unutilized abstraction, growing from 4 to 55. But some important ones also go better, like the number of cyclic dependencies that lowered from 5 to 1.

Wan can no more use CodeMR with this system because we need a license due to the increased size. It is hard to found an equivalent that retrieves the same metrics. However, we used some built-in IntelliJ to perform complexity analysis. Results are depicted by Figure 41. The average cyclomatic complexity ranges from 1 to 1.70, there is no strong variation from a package to another that could indicate some “super methods”. However, some classes have a high weighted method complexity, these outliers are in central classes like `Level` and `Board`.

package	v(G)avg ▼	v(G)tot	class	OCavg ▼	WMC
nl.tudelft.jpacman.level	1.72	223	nl.tudelft.jpacman.level.Level	2.19	57
nl.tudelft.jpacman.board	1.49	139	nl.tudelft.jpacman.level.MapParser	1.39	43
nl.tudelft.jpacman.sprite	1.31	76	nl.tudelft.jpacman.board.Board	2.42	29
nl.tudelft.jpacman.npc.ghost	1.97	69	nl.tudelft.jpacman.level.Player	1.22	28
nl.tudelft.jpacman.ui	1.45	48	nl.tudelft.jpacman.sprite.PacManSprites	1.12	27
nl.tudelft.jpacman.level.specialpellet	1.46	41	nl.tudelft.jpacman.Launcher	1.05	23
nl.tudelft.jpacman	1.13	35	nl.tudelft.jpacman.board.Square	2.00	22
nl.tudelft.jpacman.level.specialbox	1.59	27	nl.tudelft.jpacman.level.SpecialUnitySpawner	2.75	22
nl.tudelft.jpacman.game	1.38	18	nl.tudelft.jpacman.level.LevelFactory	1.50	21
nl.tudelft.jpacman.npc	1.55	17	nl.tudelft.jpacman.npc.ghost.Navigation	3.33	20
nl.tudelft.jpacman.e2e.framework.startup	1.00	5	nl.tudelft.jpacman.board.Unit	1.27	19
nl.tudelft.jpacman.integration	1.00	4	nl.tudelft.jpacman.level.CollisionInteractionMa	2.57	18
<b>Total</b>		<b>702</b>	nl.tudelft.jpacman.npc.Ghost	1.55	17
Average	1.53	58.50	nl.tudelft.jpacman.sprite.AnimatedSprite	1.50	15
			nl.tudelft.jpacman.level.SpecialBox	2.17	13

Figure 41: Complexity metrics at package and class levels for final system 2

The Figure 42 confirms what Designite told us : the new written classes under `specialpellet` and `specialbox` packages didn’t introduce new cycles in the system, preserving its design quality. There are still some big classes like `Level` implied in a lot of dependencies, but sometimes it’s impossible to avoid it.

Another aspect is the javadoc coverage, that was initially very good. For the similar tabular than Figure 43, we had in average 100% (class coverage), 82% (fields coverage), 232 lines of javadoc and 84% (methods coverage). It looks like the documentation quality lowered, but we have to consider that the package structure changed. We also notice that the average number of lines increased, meaning that what had absolutly to be documented is well documented (some non relevant fields may have been omitted leading to the observed lowering).

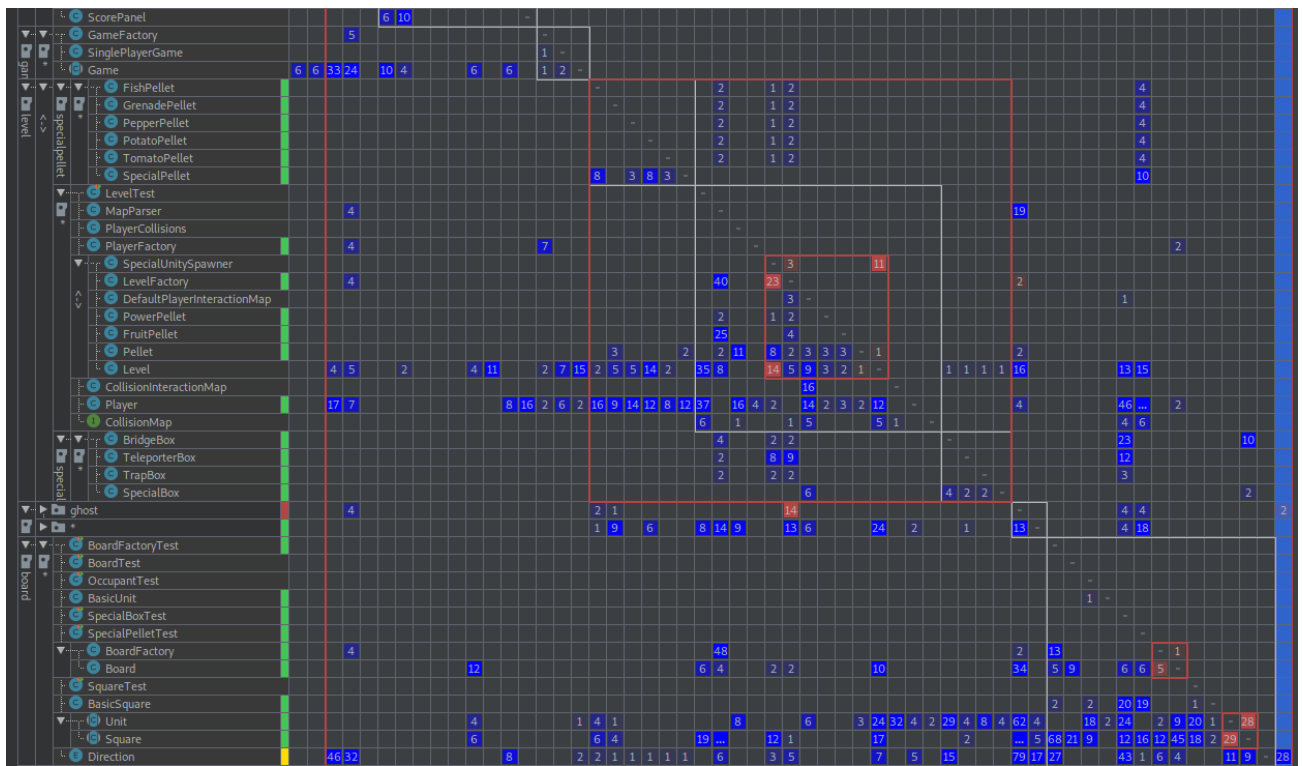
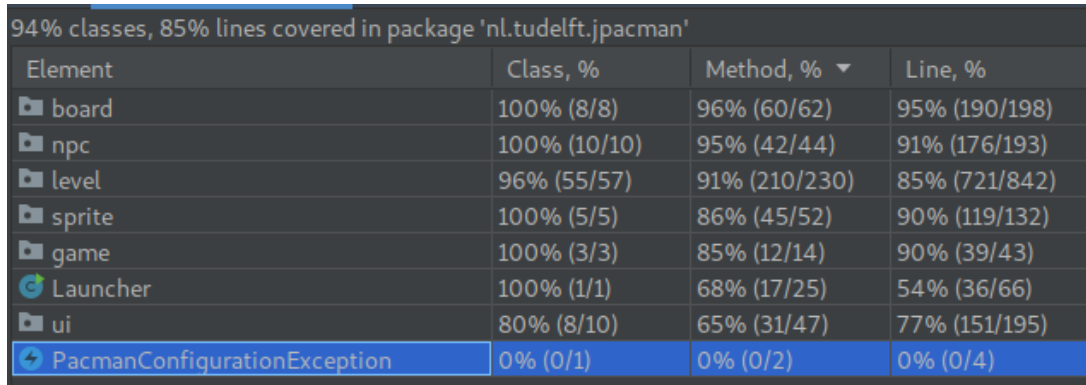


Figure 42: Dependency matrix for final system 2

package	Jc	Jf	JLOC	Jm
nl.tudelft.jpacman	75.00%	11.11%	157	83.87%
nl.tudelft.jpacman.board	100.00%	46.67%	482	70.83%
nl.tudelft.jpacman.e2e.framework.startup	100.00%	0.00%	26	80.00%
nl.tudelft.jpacman.game	100.00%	100.00%	82	73.33%
nl.tudelft.jpacman.integration	100.00%	0.00%	13	75.00%
nl.tudelft.jpacman.level	80.77%	45.19%	896	80.15%
nl.tudelft.jpacman.level.specialbox	100.00%	57.14%	126	100.00%
nl.tudelft.jpacman.level.specialpellet	88.89%	91.67%	242	82.14%
nl.tudelft.jpacman.npc	100.00%	100.00%	75	75.00%
nl.tudelft.jpacman.npc.ghost	90.00%	73.08%	446	88.57%
nl.tudelft.jpacman.sprite	100.00%	83.33%	294	56.45%
nl.tudelft.jpacman.ui	90.91%	100.00%	327	82.86%
<b>Total</b>			<b>3,166</b>	
<b>Average</b>	<b>90.91%</b>	<b>59.06%</b>	<b>263.83</b>	<b>76.68%</b>

Figure 43: Results for javadoc coverage for final system 2

We use the test coverage provided by IntelliJ to analyze which part of the code are not under test coverage. Global results are given by Figure 44. We observe that the coverage stayed good, as new tests have been written to test the extension features. The 57 tests available (45 previously) pass without any problem.



94% classes, 85% lines covered in package 'nl.tudelft.jpacman'

Element	Class, %	Method, % ▼	Line, %
board	100% (8/8)	96% (60/62)	95% (190/198)
npc	100% (10/10)	95% (42/44)	91% (176/193)
level	96% (55/57)	91% (210/230)	85% (721/842)
sprite	100% (5/5)	86% (45/52)	90% (119/132)
game	100% (3/3)	85% (12/14)	90% (39/43)
Launcher	100% (1/1)	68% (17/25)	54% (36/66)
ui	80% (8/10)	65% (31/47)	77% (151/195)
PacmanConfigurationException	0% (0/1)	0% (0/2)	0% (0/4)

Figure 44: Results for test coverage for final system 2

## 5.2 System 2 (Robin)

### 5.2.1 Generalities

This system is a mess, Rémy improved it but the system was too bad to begin with, everything could have been changed but it would have taken too much time. The way the collisions are handled is, I think, bad, it's not really clear how we have to create new collision type. Also creating or modify the map is dreadful, it was a pain to change the map, but of course changing that would have taken a lot of time. Creating new type of object is also not that easy because we have to create new Containers, ... And WorkerProcess is, in my mind, a mess in general, it doesn't really respect object oriented programming, everything is compared with else-if with a lot of Feature Envy, everything could have been reworked to make it clearer and simpler to update. But still tanks to Rémy this system is clearer than before.

### 5.2.2 Static Analysis

### 5.2.3 Code Metrics (CodeMR)

CodeMR was used before to anylise the system but now we cannot do it because of the free version, the system is now too big for the free trial. We couldn't afford to buy the full version.

### 5.2.4 Compliance bad smells (PMD, Designite)

**PMD** The Figure 45 shows the results of the PMD analysis, we can see that there is 1880 violations.

**Designite** The Figure 46 shows the results of the Designite analysis, there is still a lot of code smels, like 12 cyclic dependencies, 7 complex method, 1 Feature Envy and 162 magic numbers.

### 5.2.5 Test coverage (IntelliJ built-in tool)

The following figures 47, and 48 shows the test coverage. All the tests passed and almost all the code is covered. We can see that 96% of the classes were tested and 71% of the code line.

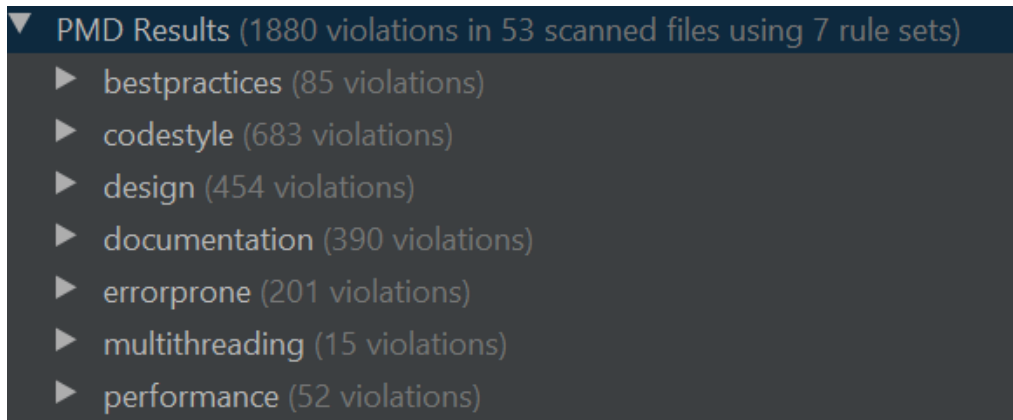


Figure 45: PMD results for system 2

```
--Analysis summary--
  Total LOC analyzed: 3823      Number of packages: 7
  Number of classes: 55      Number of methods: 389
-Total architecture smell instances detected-
  Cyclic dependency: 12      God component: 0
  Ambiguous interface: 0      Feature concentration: 1
  Unstable dependency: 4      Scattered functionality: 5
  Dense structure: 1
-Total design smell instances detected-
  Imperative abstraction: 0      Multifaceted abstraction: 0
  Unnecessary abstraction: 0      Unutilized abstraction: 1
  Feature envy: 1      Deficient encapsulation: 18
  Unexploited encapsulation: 8      Broken modularization: 0
  Cyclically-dependent modularization: 7      Hub-like modularization: 0
  Insufficient modularization: 2      Broken hierarchy: 7
  Cyclic hierarchy: 0      Deep hierarchy: 0
  Missing hierarchy: 8      Multipath hierarchy: 0
  Rebellious hierarchy: 1      Wide hierarchy: 0
-Total implementation smell instances detected-
  Abstract function call from constructor: 0      Complex conditional: 5
  Complex method: 7      Empty catch clause: 1
  Long identifier: 0      Long method: 0
  Long parameter list: 0      Long statement: 6
  Magic number: 162      Missing default: 7
----
```

Figure 46: Designite result for system 2

96% classes, 71% lines covered in 'all classes in scope'

Element	Class, %	Method, %	Line, %
controller	100% (2/2)	100% (14/14)	100% (36/36)
diagrams	100% (0/0)	100% (0/0)	100% (0/0)
documents	100% (0/0)	100% (0/0)	100% (0/0)
graphics	100% (0/0)	100% (0/0)	100% (0/0)
maps	100% (0/0)	100% (0/0)	100% (0/0)
model	96% (63/65)	78% (289/369)	71% (1141/1600)
view	87% (7/8)	64% (18/28)	67% (242/356)

Figure 47: System 2 test coverage

96% classes, 71% lines covered in package 'model'			
Element	Class, %	Method, %	Line, %
container	100% (5/5)	86% (39/45)	82% (92/111)
event	100% (4/4)	82% (28/34)	58% (146/248)
exception	100% (5/5)	50% (4/8)	50% (9/18)
mapobject	94% (34/36)	68% (113/166)	63% (410/645)
Game	100% (2/2)	90% (28/31)	80% (69/86)
Helper	100% (1/1)	100% (3/3)	100% (31/31)
Highscore	100% (2/2)	87% (7/8)	46% (22/47)
Level	100% (1/1)	100% (6/6)	90% (18/20)
Map	100% (2/2)	86% (19/22)	81% (78/96)
MapPlacer	100% (3/3)	100% (10/10)	97% (190/195)
Position	100% (1/1)	90% (10/11)	55% (24/43)
Score	100% (1/1)	100% (7/7)	90% (20/22)
Settings	100% (1/1)	62% (5/8)	61% (8/13)
Timer	100% (1/1)	100% (10/10)	96% (24/25)

Figure 48: System 2 test coverage

## 5.3 System 3 (BOOSKO Sam)

### 5.3.1 Generalities

The personal view is that a lot of improvements can be done, but it should mean, rework all the system. This system, is, according to me, not good. A lot of parts are not clear, not easy to modify, or to update without changing a huge part of it. It would be too long to enumerate all of these problems seen. But few of them are, 1) the merging between, the controller, the visual and the core (model), 2) using not logical value for attributes as the previous speed in the object *MovingGameElement*, increasing this value gave the effect of slowing down the *GameElement*, 3) the class *Wall*, don't need to explain, just opening it make us understand and many more problems could be said.

### 5.3.2 Static Analysis

#### 5.3.2.1 Bad smells (Designite)

With Designite, we observe that there are 229 magic numbers (see Figure 49), corresponding in the implementation of the default elements drawing method. In general, the system is doing well.

#### 5.3.2.2 Javadoc Coverage (MetricsReloaded)

We observe that the javadoc coverage should be improved (see Table 5).

### 5.3.3 Dynamic Analysis

#### 5.3.3.1 Running tests & Test Coverage (IntelliJ Build-Run)

By running the tests provided and new tests added, 28 tests of 28 passed. Furthermore, the project is quite well covered by tests (see Table 6). Anyway, some improvements could be done.

```

Searching classpath folders ...
Parsing the source code ...
Resolving symbols...
Computing metrics...
Detecting code smells...
Error - License validation unsuccessful. Status code: 406
Exporting analysis results...
--Analysis summary--
    Total LOC analyzed: 3644          Number of packages: 7
    Number of classes: 58   Number of methods: 405
-Total architecture smell instances detected-
    Cyclic dependency: 0   God component: 0
    Ambiguous interface: 0   Feature concentration: 2
    Unstable dependency: 0   Scattered functionality: 4
    Dense structure: 0
-Total design smell instances detected-
    Imperative abstraction: 0       Multifaceted abstraction: 0
    Unnecessary abstraction: 0       Unutilized abstraction: 1
    Feature envy: 0   Deficient encapsulation: 2
    Unexploited encapsulation: 1     Broken modularization: 0
    Cyclically-dependent modularization: 0   Hub-like modularization: 0
    Insufficient modularization: 2   Broken hierarchy: 10
    Cyclic hierarchy: 0       Deep hierarchy: 0
    Missing hierarchy: 1       Multipath hierarchy: 0
    Rebellious hierarchy: 1   Wide hierarchy: 0
-Total implementation smell instances detected-
    Abstract function call from constructor: 0       Complex conditional: 6
    Complex method: 1       Empty catch clause: 0
    Long identifier: 1       Long method: 1
    Long parameter list: 8   Long statement: 24
    Magic number: 299       Missing default: 1
----
Done.

```

Figure 49: System 3 - Designite Output

package ▲	Jc	Jf	JLOC	Jm
pacman_infd	50.00%	0.00%	11	0.00%
pacman_infd.Elements	100.00%	17.65%	210	37.36%
pacman_infd.Elements.ExtensionElements	92.31%	14.00%	144	41.18%
pacman_infd.Enums	100.00%	0.00%	16	0.00%
pacman_infd.Fileloader	100.00%	0.00%	26	33.33%
pacman_infd.Game	89.47%	7.78%	449	50.64%
pacman_infd.Strategies	100.00%	0.00%	53	26.67%
<b>Total</b>			<b>909</b>	
<b>Average</b>	<b>90.32%</b>	<b>10.27%</b>	<b>129.86</b>	<b>38.42%</b>

Table 5: System 3 - Javadoc Coverage

97% classes, 58% lines covered in package 'pacman\_infd'

Element	Class, %	Method, %	Line, %
Elements	100% (26/26)	65% (102/156)	48% (347/711)
Enums	100% (4/4)	100% (11/11)	100% (12/12)
Fileloader	100% (2/2)	77% (7/9)	85% (40/47)
Game	100% (34/34)	65% (129/197)	62% (484/769)
Strategies	80% (4/5)	92% (13/14)	84% (59/70)
Pacman_INFID	0% (0/1)	0% (0/1)	0% (0/3)

Table 6: System 3 - Test Coverage