Software Analysis : pacman systems

Project report for Software Evolution course

Group 3

BOOSKO Sam
DECOCQ Rémy
SCHERER Robin

Academic Year 2019-2020
Master Computers Science, block 2
Faculté des Sciences, Université de Mons

Contents

1 Introduction

2 Quality analysis of the initial versions

2.1 System 1 (BOOSKO Sam)

2.1.1 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 scenatios 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*.

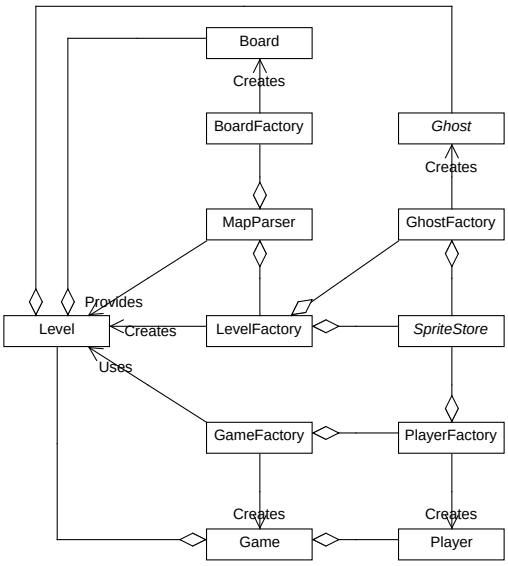

Figure 1: Factory Wiring Class Diagram

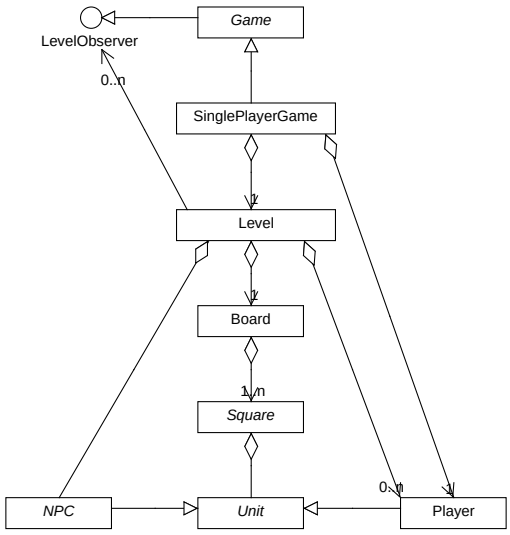

Figure 2: Single Player Game Class Diagram

2.1.2 Static Analysis

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.

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

```
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

Figure 4: CodeMR dashboard summarizing health of the system 1

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.

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[1] 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.".

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.

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.

[1] https://www.codemr.co.uk/documents/

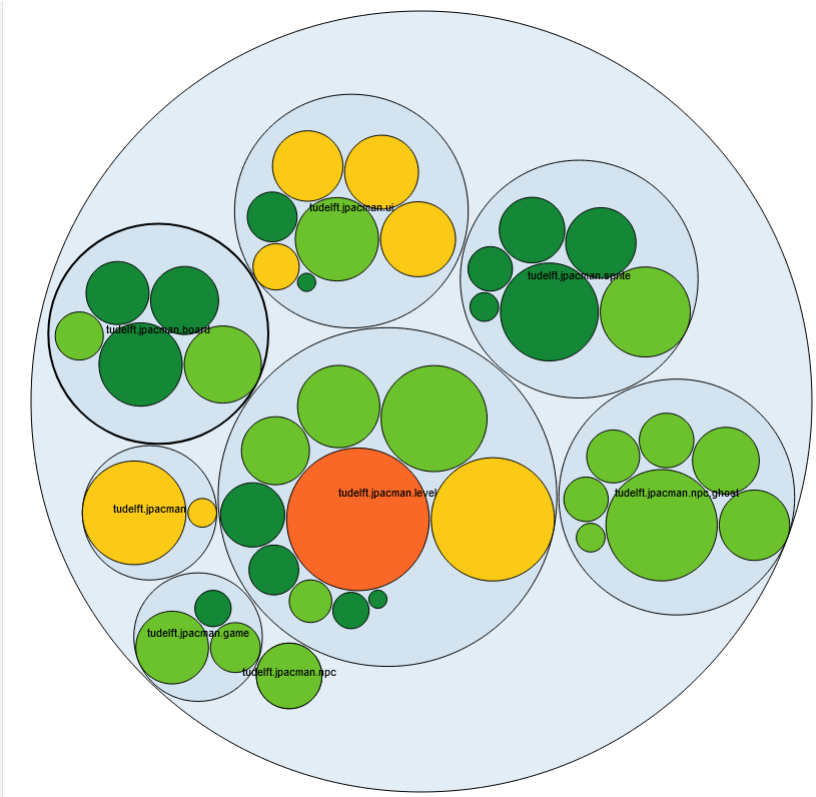

Figure 5: C3 Metric by package for the system 1

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

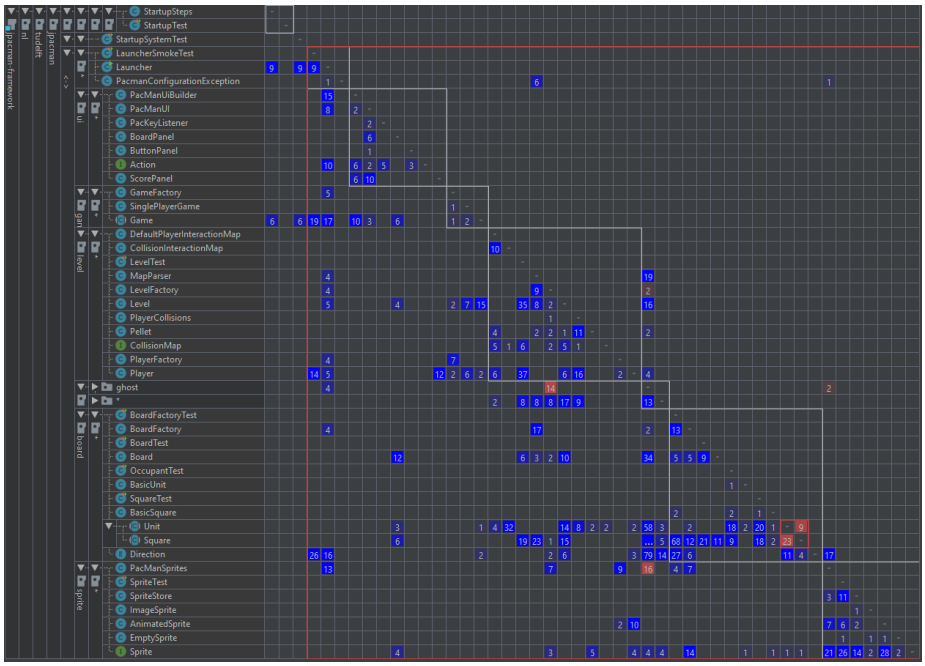

Figure 7: Matrix of Dependencies

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%
Total			**2,326**	
Average	100.00%	82.61%	232.60	84.30%

Table 2: Coverage of the javadoc for the system 1

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

2.1.3 Dynamic Analysis

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.

3 Quality improvement

3.1 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.

4 Adding basic functionalities

4.1 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 ot, *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.

5 Adding new features

5.1 System 3 (BOOSKO Sam)

5.1.1 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*.

5.1.2 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

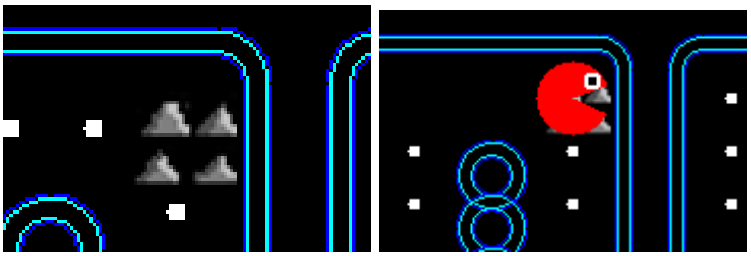

Figure 8: System 3 - Traps

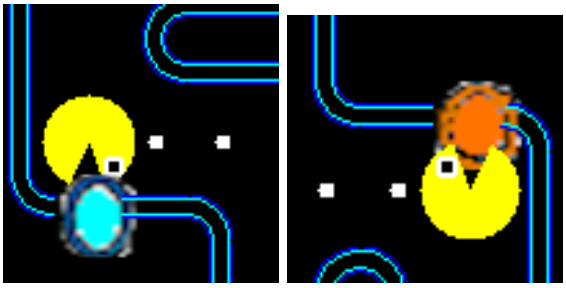

Figure 9: System 3 - Portals

the Pacman color to have a visualisation about his state.

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_infd.Elements.ExtensionElements.TrapElement* (see Figure 8).

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 9). This feature is implemented in *pacman_infd.Elements.ExtensionElements.TeleporterElement*.

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

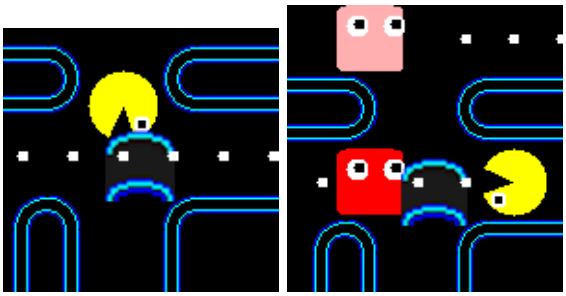

Figure 10: System 3 - Bridges

Figure 11: System 3 - Grenades

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 10).

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 11.

Red Beans The Red Bean, implemented in *pacman_infd.Elements.ExtensionElements.RedBeanElement*, is a special element. When Pacman eats this bean, Pacman is slown 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 12).

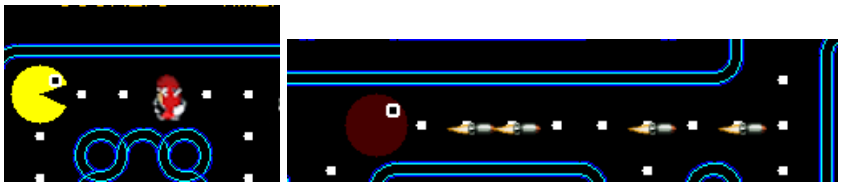

Figure 12: System 3 - Red Beans

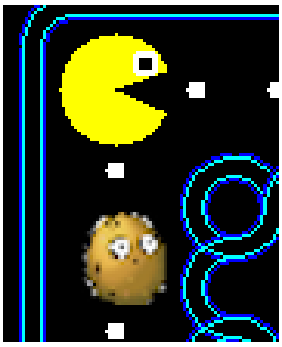

Figure 13: System 3 - Potato

Potato The Potato, implemented in *pacman_infd.Elements.ExtensionElements.PotatoElement*, is a 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 13).

Fish The fish is another simple element and the last, this one, implemented in *pacman_infd.Elements.ExtensionElements.PotatoElement*, slows down the Pacman to a very slow speed and he turns cyan (see Figure 14) .

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 pel-

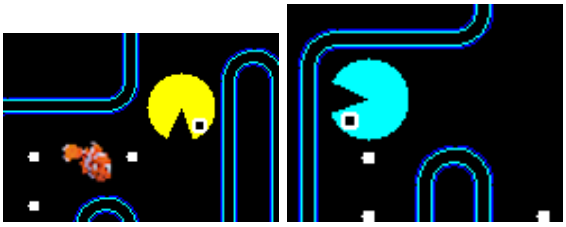

Figure 14: System 3 - Fish

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%
Total			**909**	
Average	90.32%	10.27%	129.86	38.42%

Table 5: System 3 - Javadoc Coverage

let ate, there is 10% that a random elements spawn on an empty *cell*. This system is implement in the object *GameWorld* on the method *placeCherryOnRandomEmptyCell* already provided.

6 Quality evolution analysis

6.1 System 3 (BOOSKO Sam)

6.1.1 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.

6.1.2 Static Analysis

Bad smells (Designite) With Designite, we observe that there are 229 magic numbers (see Figure 15), corresponding in the implemention of the default elements drawing method. In general, the system is doing well.

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

```
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 15: System 3 - Designite Output

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_INFD	0% (0/1)	0% (0/1)	0% (0/3)

Table 6: System 3 - Test Coverage

6.1.3 Dynamic Analysis

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.

7 Conclusion