

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	3
2	Quality analysis of the initial versions	4
2.1	System 1 (Sam)	4
2.1.1	Generalities	4
2.1.2	Static Analysis	5
2.1.3	Dynamic Analysis	10
3	Quality improvement	11
4	Adding basic functionalities	11
5	Adding new features	11
6	Quality evolution analysis	11
7	Conclusion	11

1 Introduction

2 Quality analysis of the initial versions

2.1 System 1 (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*.

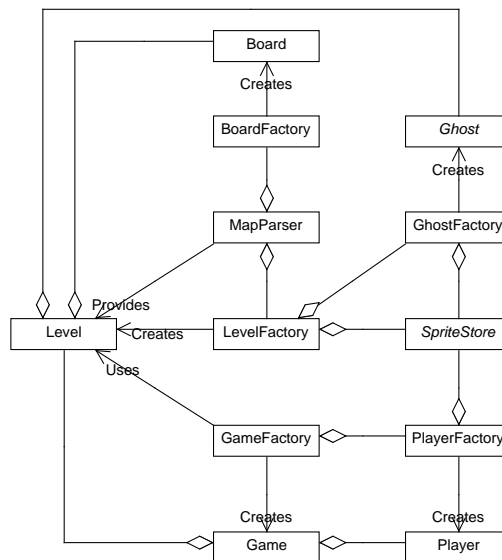


Figure 1: Factory Wiring Class Diagram

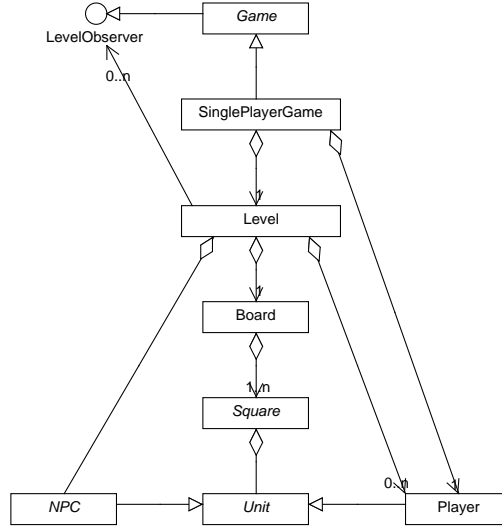


Figure 2: Single Player Game Class Diagram

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 cohesion¹ 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/>

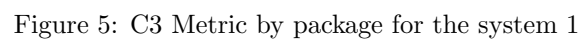


Figure 6: Metrics of the class Level

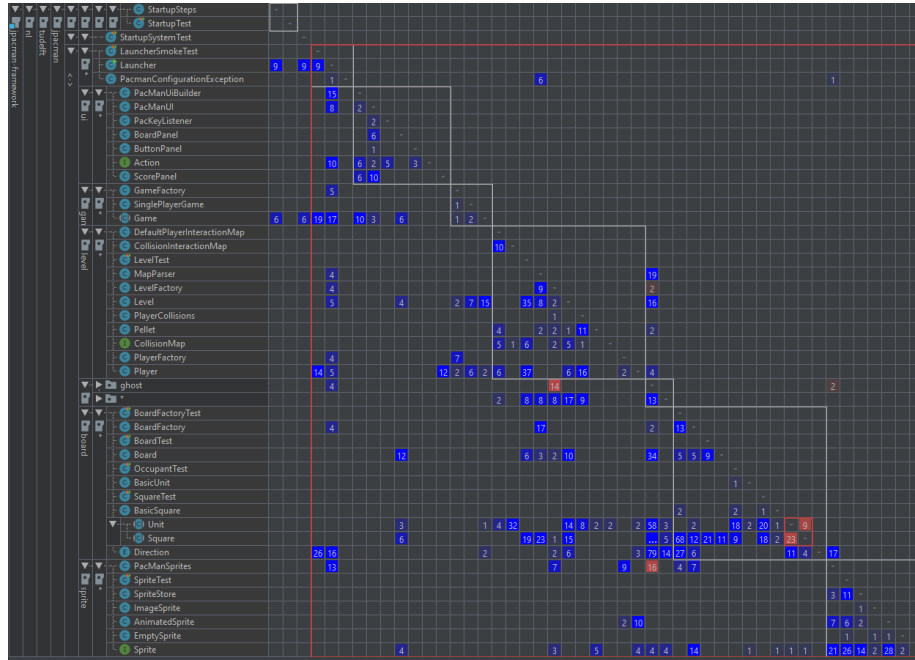


Figure 7: Matrix of Dependencies

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
- 4 Adding basic functionalities
- 5 Adding new features
- 6 Quality evolution analysis
- 7 Conclusion