UMONS

Faculté des Sciences

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 Quality analysis of the initial versions **4**
1.1 System 2 (Rémy) 4
1.1.1 Generalities 4
1.1.2 Static Analysis 4
1.1.3 Dynamic Analysis 9

2 Quality improvement **11**

3 Adding basic functionalities **12**

4 Adding new features **13**

5 Quality evolution analysis **14**

6 Conclusion **15**

Introduction

1 Quality analysis of the initial versions

1.1 System 2 (Rémy)

1.1.1 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 modelized under `.tmx` format, that is a popular way to deal with board games[1]. Only one single basic map is provided.

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.

1.1.2 Static Analysis

1.1.2.1 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 1 informs this software is doing quite good.

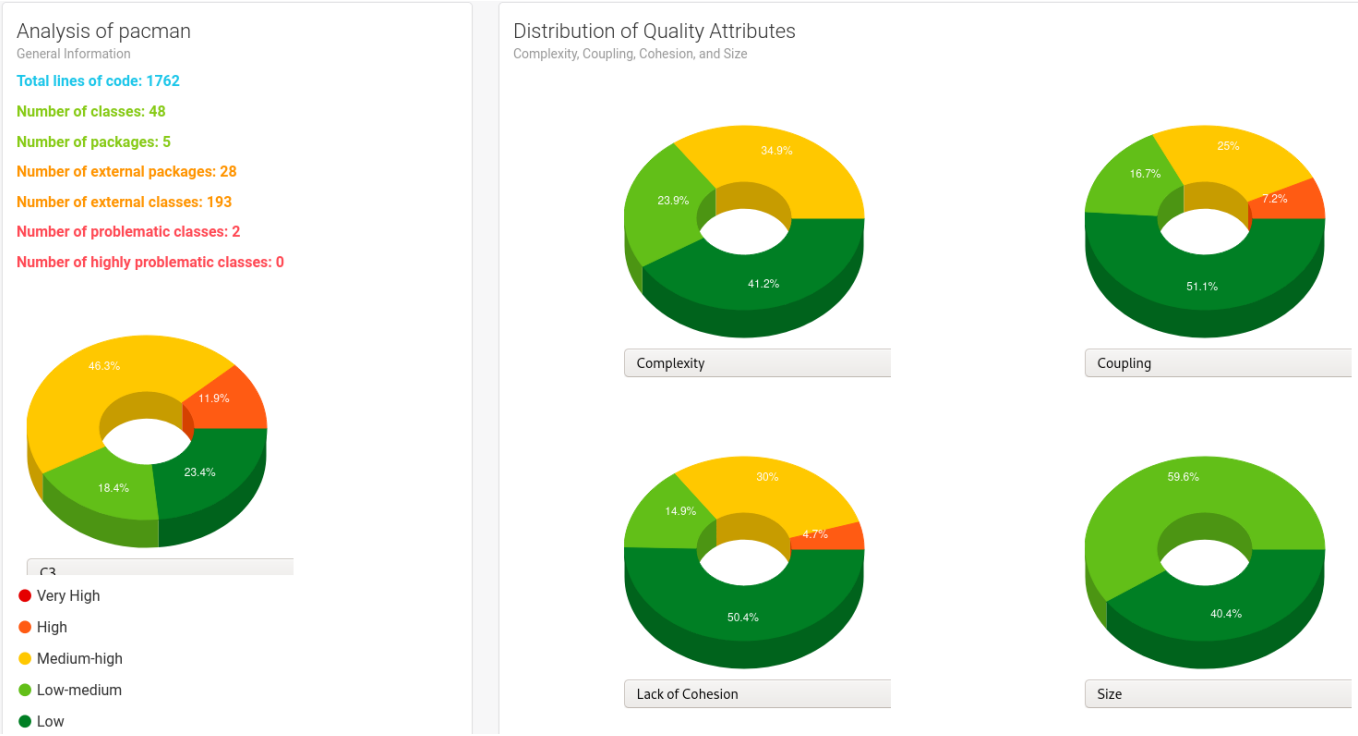


Figure 1: CodeMR dashboard summarizing health of system 2

[1] https://doc.mapeditor.org/en/stable/reference/support-for-tmx-maps/

The Figure 2 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[2]. We notice that, following the dashboard overview, two classes are impacting the software quality from the point of view of C3 metric.

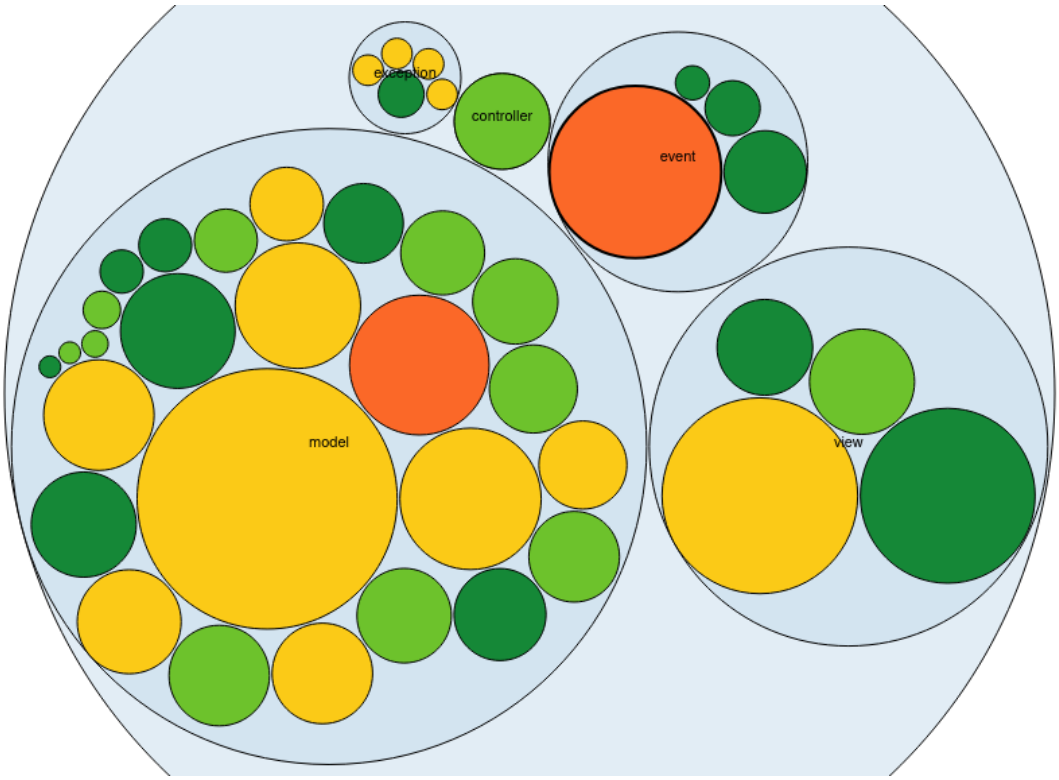


Figure 2: C3 Metric by package for system 2

The details of the measurements on the two more problematic classes are given by Figure 3 and Figure 4, 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.

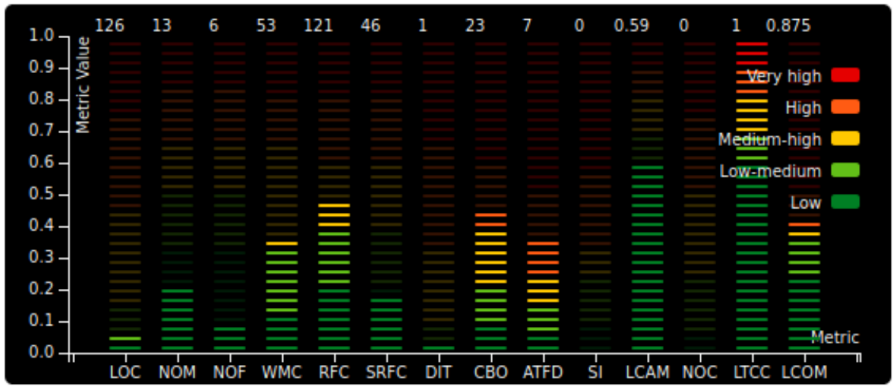
In addition, for *event.WorkerProcess* we have :

- 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 investiggated class. Classes with a high ATFD value rely strongly on data of other classes and that can be the sign of the God Class.

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

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

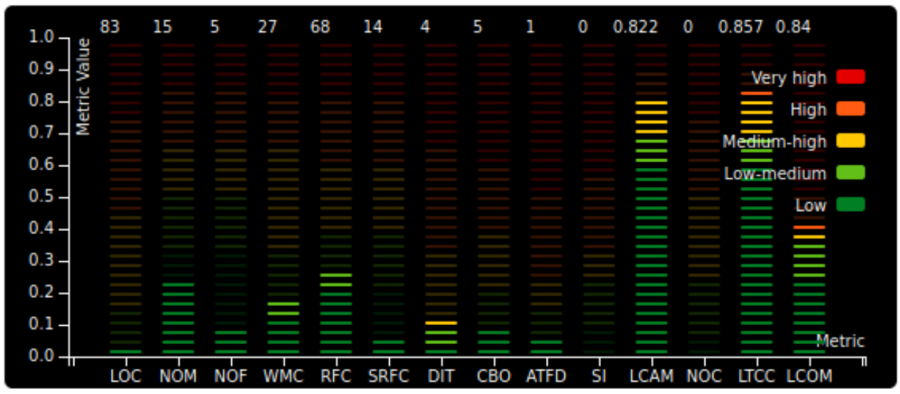
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	CBO	CBO APP	CBO LIB	RFC
1	WorkerProcess					126	23	23	0	121



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

Figure 3: *event.WorkerProcess* class main metrics measurements

ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	COMPLEXITY	COUPLING	LACK OF COHESION	SIZE
9	Ghost					83	medium-high	low	high	low-medium



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

Figure 4: *model.Ghost* class main metrics measurements

1.1.2.2 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 5 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 7. 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).

1.1.2.3 Compliance & bad smells (PMD, Designite)

PMD is a statical 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 6).

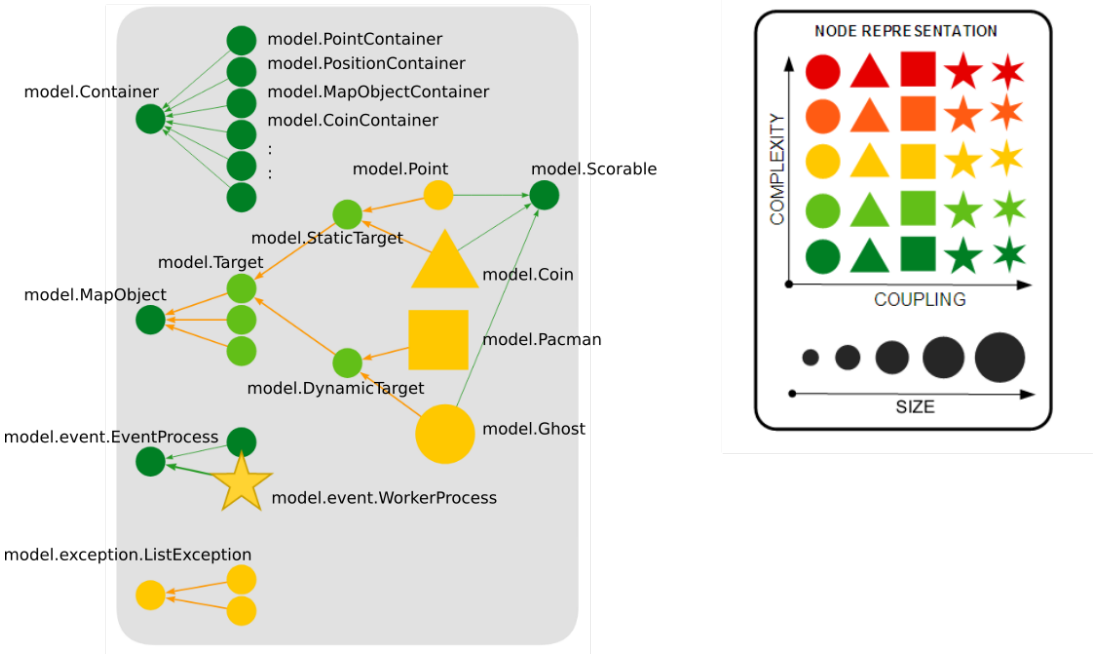


Figure 5: Inheritance relations between classes in system 2

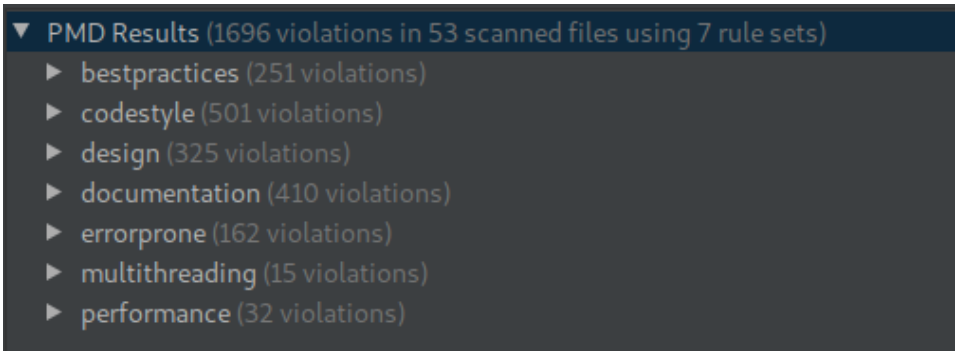


Figure 6: Violations found by PMD in system 2

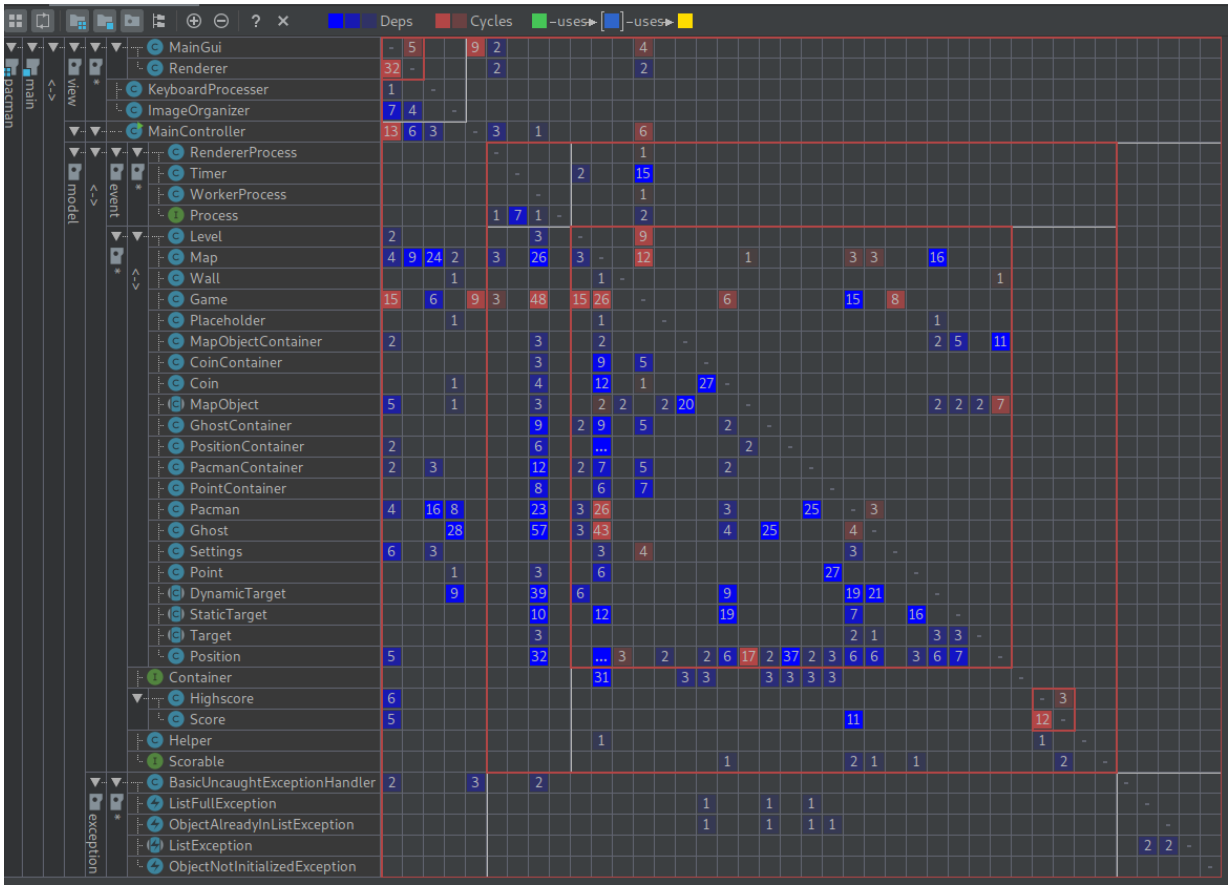


Figure 7: Dependency matrix for system 2

Designite is used to detect bad smells, a summary of the analyse is presented by Figure 8. 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 8: Designite in-line use results for system 2

1.1.2.4 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 **??**, 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 9: Javadoc coverage for system 2

1.1.3 Dynamic Analysis

1.1.3.1 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*).

1.1.3.2 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 10. We observe the implementation benefits of a good test coverage, the most laking 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 10: Test coverage for system 2

2 Quality improvement

3 Adding basic functionalities

4 Adding new features

5 Quality evolution analysis

6 Conclusion