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

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.

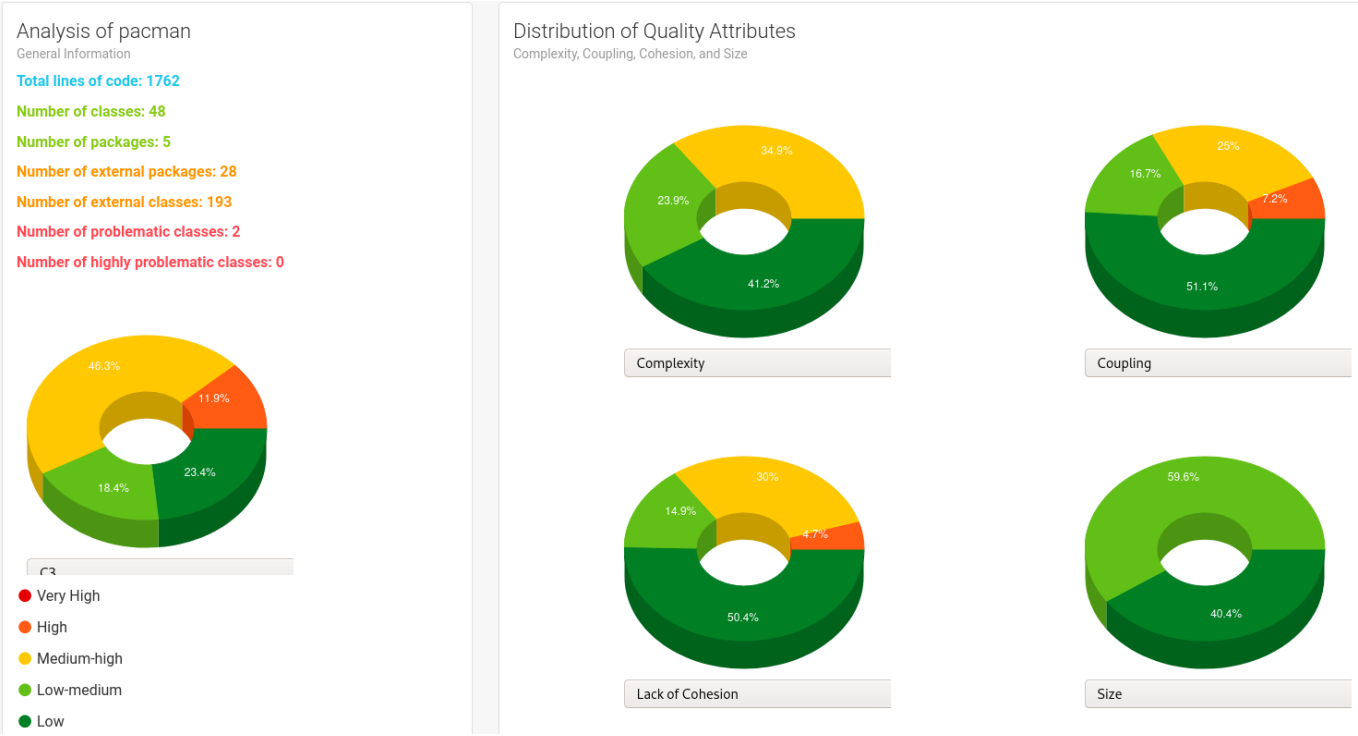


Figure 1: CodeMR dashboard summarizing health of system 2

[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.

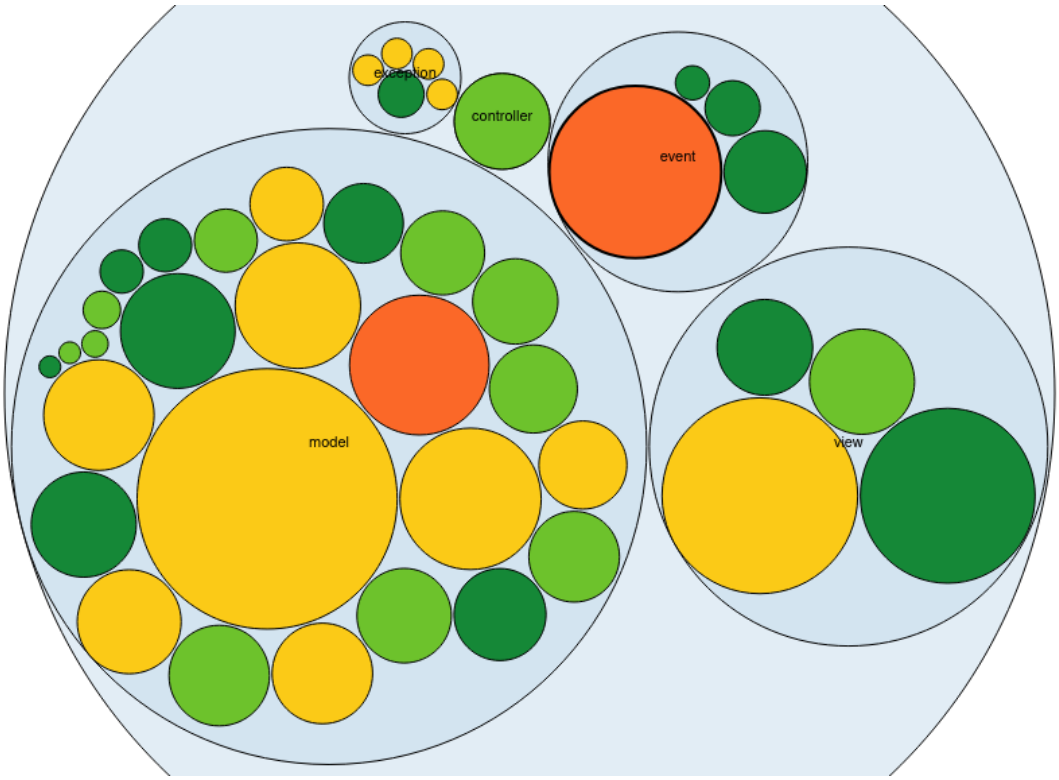


Figure 2: C3 Metric by package for system 2

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.

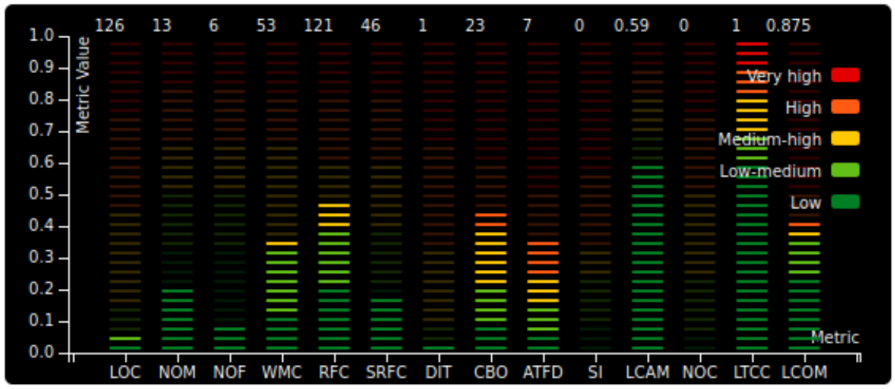
In addition, for *event.WorkerProcess* we have :

- 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

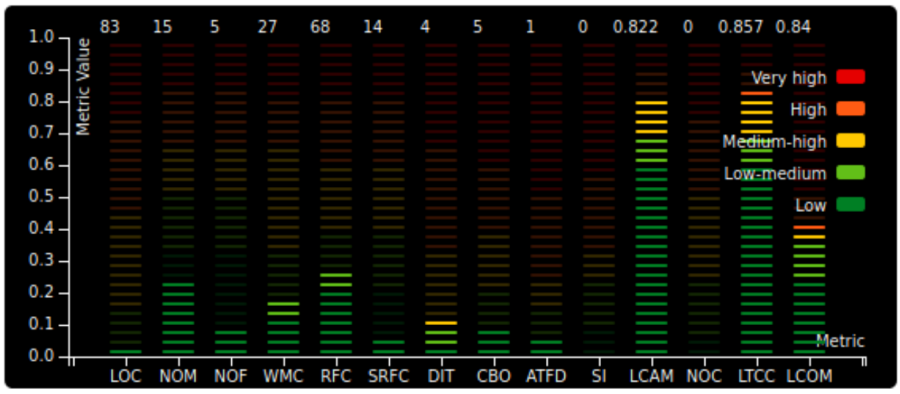
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	CBO	CBO APP	CBO LIB	RFC
1	WorkerProcess					126	23	23	0	121



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).

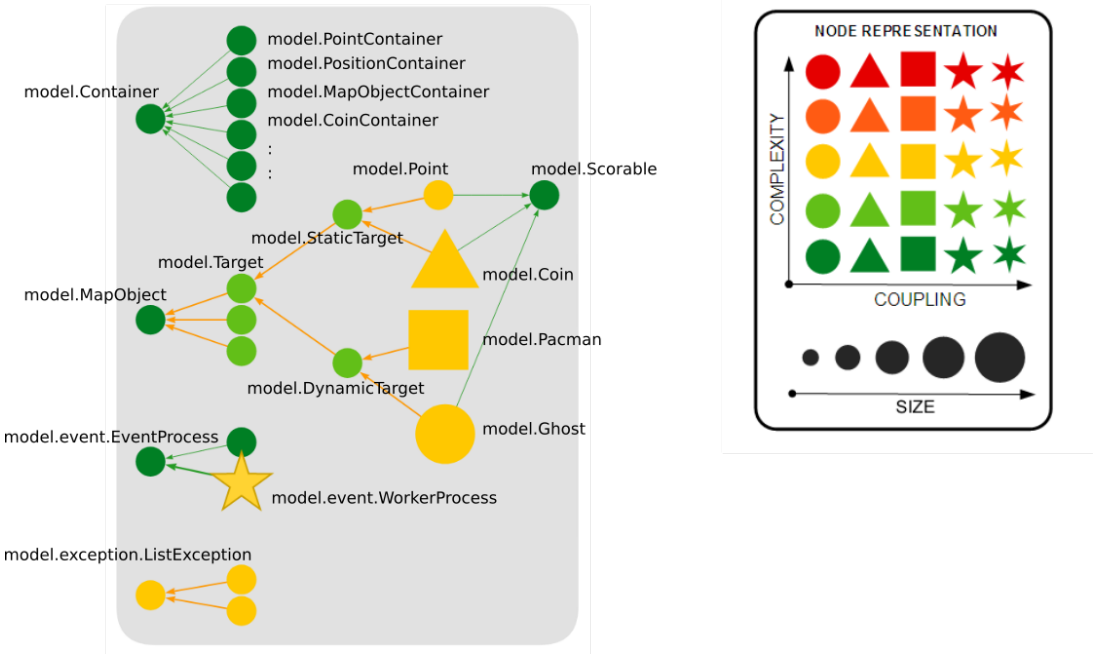


Figure 5: Inheritance relations between classes in system 2

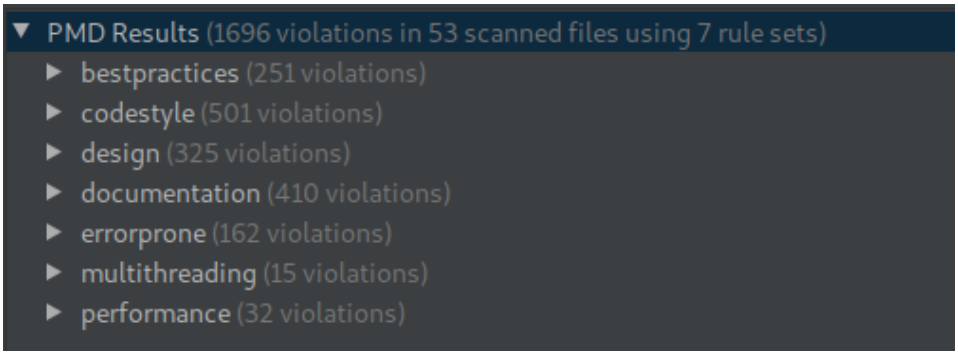


Figure 6: Violations found by PMD in system 2

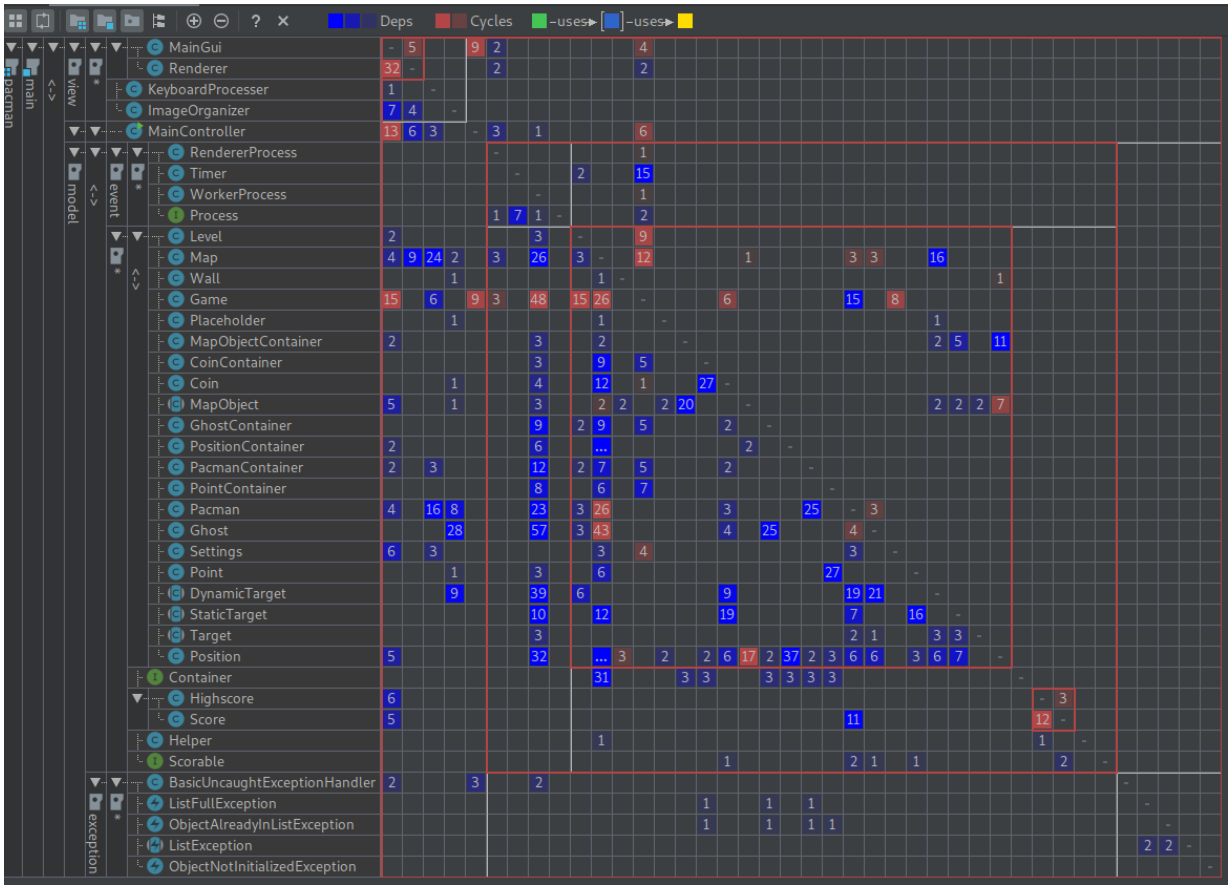


Figure 7: Dependency matrix for system 2

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

2.1 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 refractorings 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 refractoring will occurr 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 leas 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.

2.1.1 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.

2.1.2 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 overlayer 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 ineleguant, leading to code duplication and increased number of classes. Keeping in mind what were each class written for, we redisigned 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 11. 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 11. 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 dont 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.

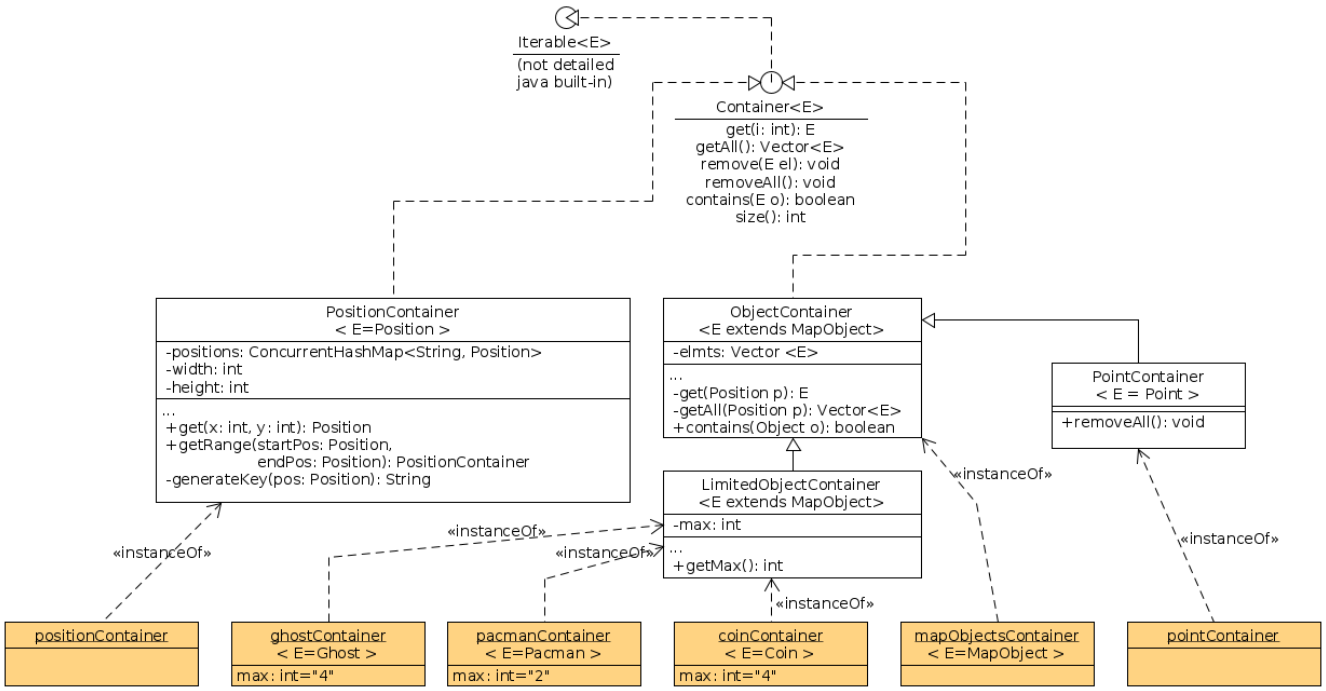


Figure 11: New class structure for `Containers` part of the implementation

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.).

2.1.3 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 reseting not properly. Also, the usage of `static` elements lead to inconsistencies. Leveraging some adjustements in the source code, we managed to get the system more consistent and the tests passing in a determenistic behaviour.

2.1.4 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.

3 Adding basic functionalities

3.1 System 2

The following functionnalities 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 functionnalities 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 frequence. 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 12.

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 12: New timer integrated in the interface (below the map)

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 cumulutative 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 sublcassed `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 13 (illustrating actual effect is quite difficult for most, it should be played instead).

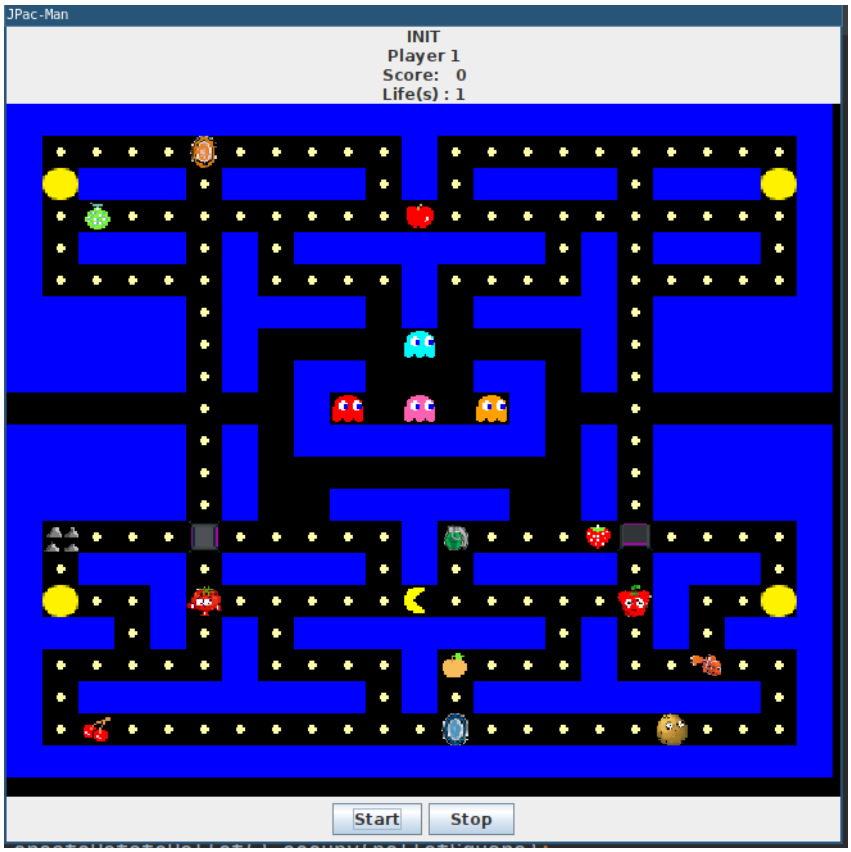


Figure 13: Illustrating all special units on the bottom of the map

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 `SpecialUnitySpawner`. 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 uniformally 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

4.1.2 Quality analysis

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 mentionned, 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 14: Results from Designite for final system 2

We observe in Figure 14 that the system has consequently increased in size, almost doubled in mulitple 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 unutulized 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 15. 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
nl.tudelft.jpacman.level	1.72	223
nl.tudelft.jpacman.board	1.49	139
nl.tudelft.jpacman.sprite	1.31	76
nl.tudelft.jpacman.npc.ghost	1.97	69
nl.tudelft.jpacman.ui	1.45	48
nl.tudelft.jpacman.level.specialpellet	1.46	41
nl.tudelft.jpacman	1.13	35
nl.tudelft.jpacman.level.specialbox	1.59	27
nl.tudelft.jpacman.game	1.38	18
nl.tudelft.jpacman.npc	1.55	17
nl.tudelft.jpacman.e2e.framework.startup	1.00	5
nl.tudelft.jpacman.integration	1.00	4
Total		**702**
Average	1.53	58.50

class	OCavg	WMC
nl.tudelft.jpacman.level.Level	2.19	57
nl.tudelft.jpacman.level.MapParser	1.39	43
nl.tudelft.jpacman.board.Board	2.42	29
nl.tudelft.jpacman.level.Player	1.22	28
nl.tudelft.jpacman.sprite.PacManSprites	1.12	27
nl.tudelft.jpacman.Launcher	1.05	23
nl.tudelft.jpacman.board.Square	2.00	22
nl.tudelft.jpacman.level.SpecialUnitySpawner	2.75	22
nl.tudelft.jpacman.level.LevelFactory	1.50	21
nl.tudelft.jpacman.npc.ghost.Navigation	3.33	20
nl.tudelft.jpacman.board.Unit	1.27	19
nl.tudelft.jpacman.level.CollisionInteractionMa	2.57	18
nl.tudelft.jpacman.npc.Ghost	1.55	17
nl.tudelft.jpacman.sprite.AnimatedSprite	1.50	15

Figure 15: Complexity metrics at package and class levels for final system 2

The Figure 16 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 17, 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).

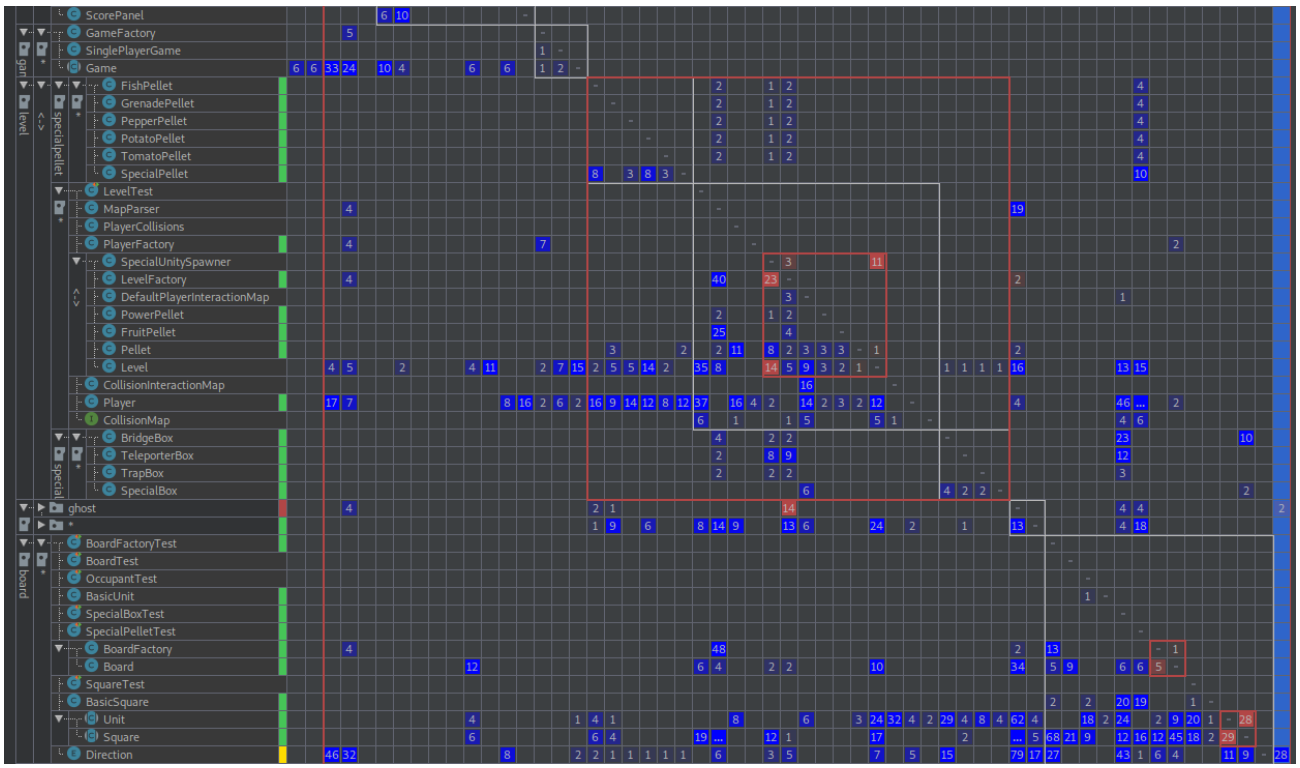


Figure 16: Dependency matrix for final system 2

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%
Total			**3,166**	
Average	90.91%	59.06%	263.83	76.68%

Figure 17: 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 18. 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.

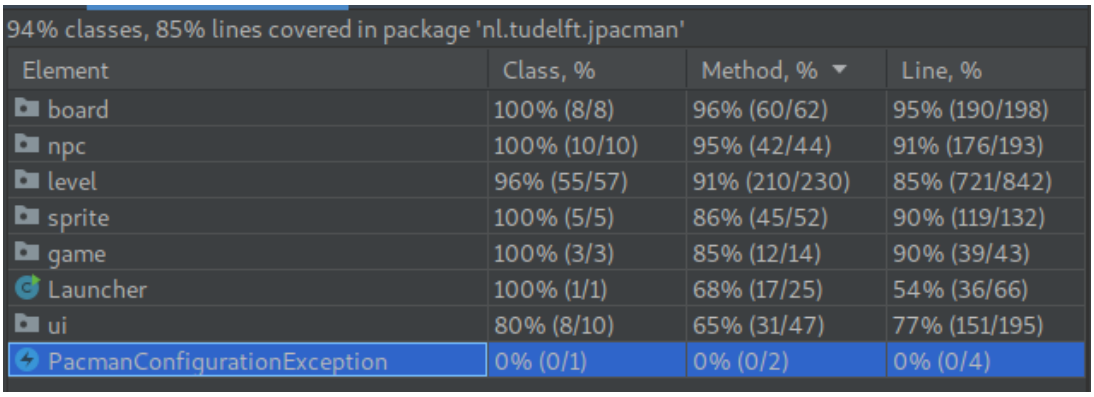


Figure 18: Results for test coverage for final system 2

5 Quality evolution analysis

6 Conclusion