

Software Analysis : pacman systems

Project report for Software Evolution course

Group 3

BOOSKO Sam
DECOCQ Rémy
SCHERER Robin

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
2.1	System 2 (Rémy)	11
2.1.1	Step 1	11
2.1.2	Step 2	11
3	Adding basic functionalities	13
4	Adding new features	14
5	Quality evolution analysis	15
6	Conclusion	16

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 modeled under `.tmx` format, that is a popular way to deal with board games¹. 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.

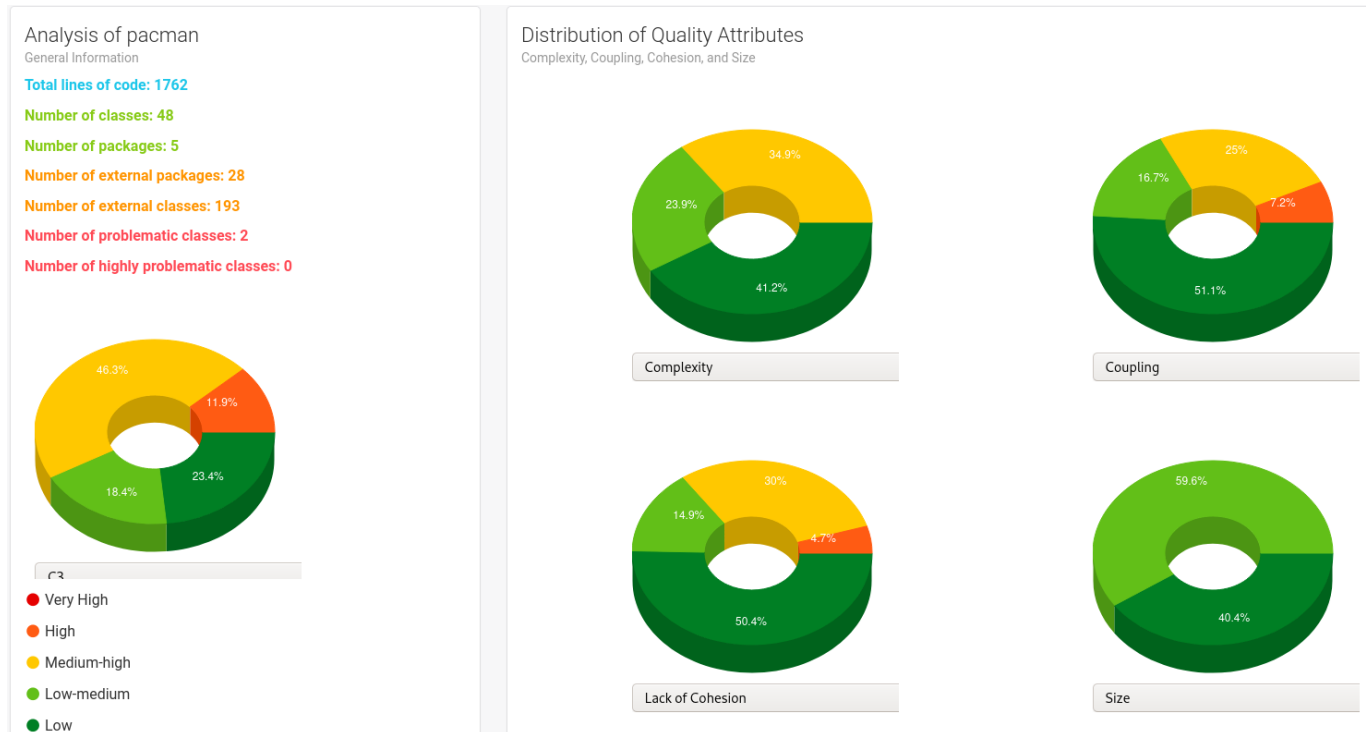


Figure 1: CodeMR dashboard summarizing health of system 2

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 documentation². We notice that, following the dashboard overview, two classes are impacting the software quality from the point of view of C3 metric.

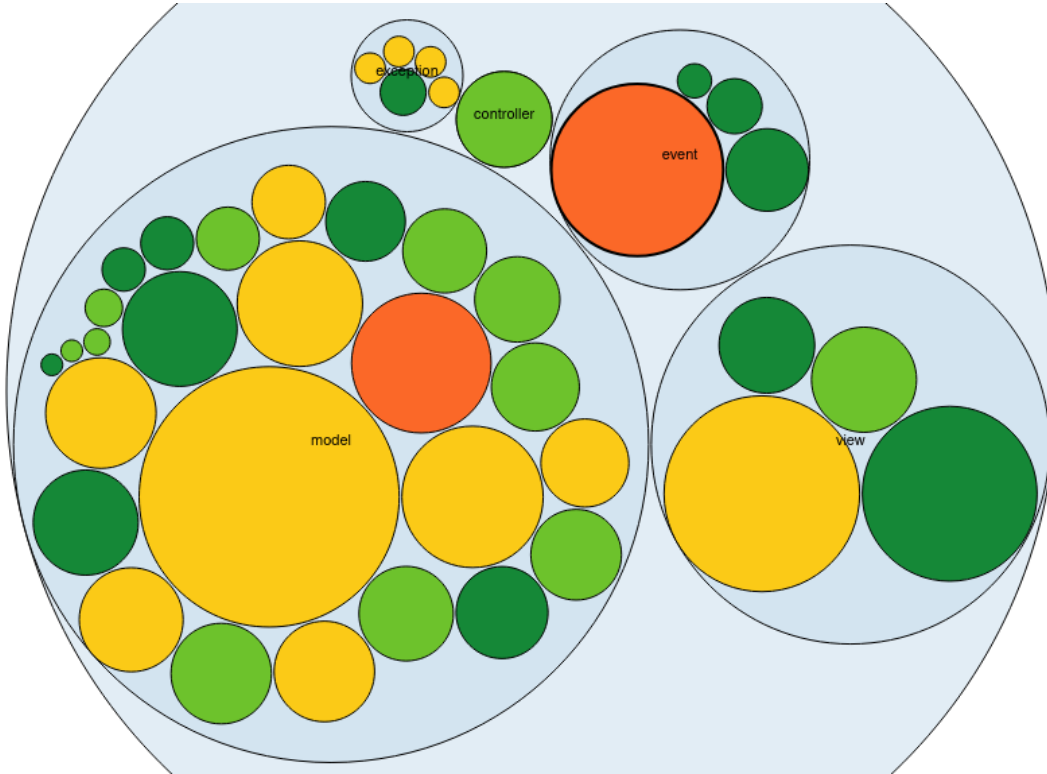


Figure 2: C3 Metric by package for system 2

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.

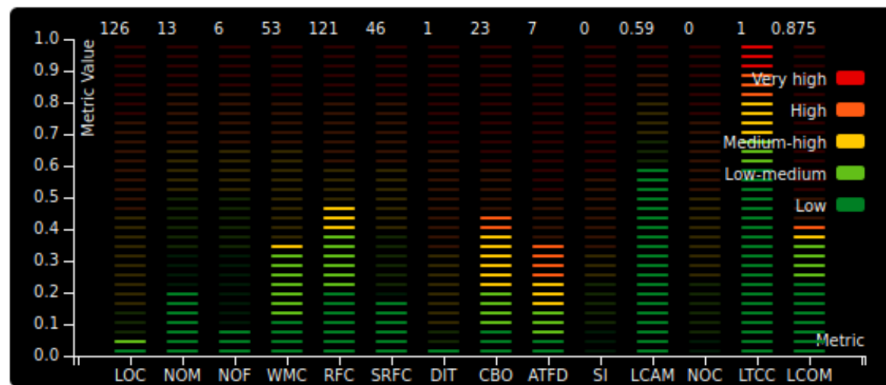
In addition, for *event.WorkerProcess* we have :

- 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 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 revealate relevant problems in the implementation.

²<https://www.codemr.co.uk/documents>

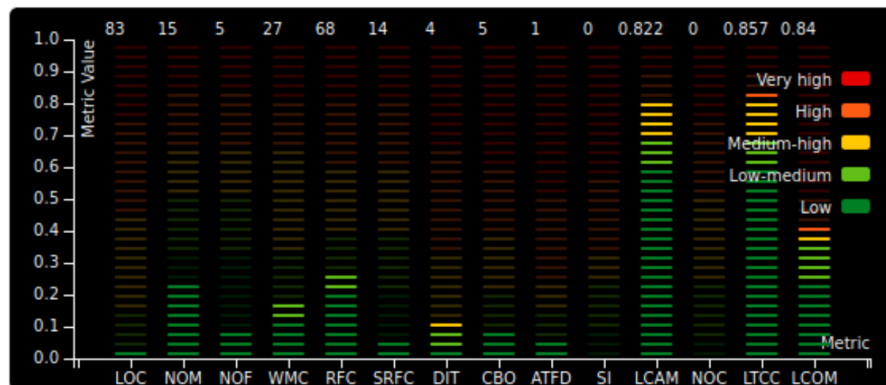
ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC	CBO	CBO APP	CBO LIB	RFC
1	WorkerProcess	High	Medium-high	Low	Low	126	23	23	0	121



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	Low	Medium-high	High	Low	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 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 6).

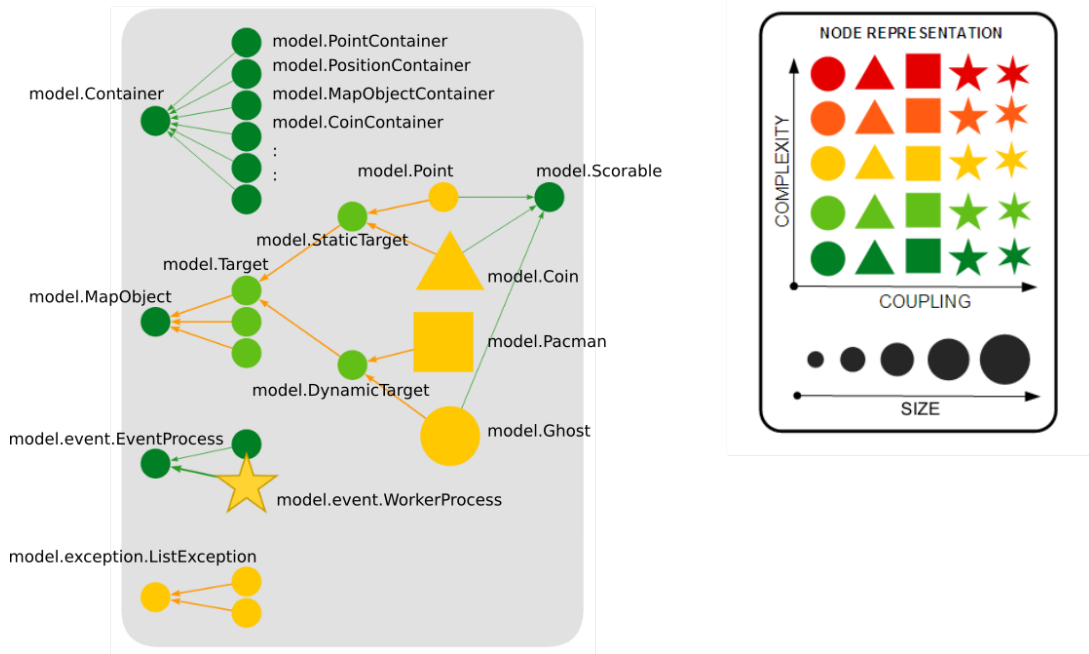


Figure 5: Inheritance relations between classes in system 2

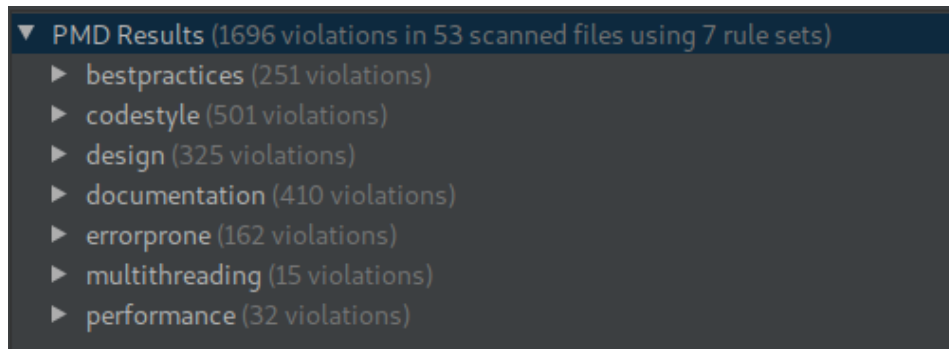


Figure 6: Violations found by PMD in system 2

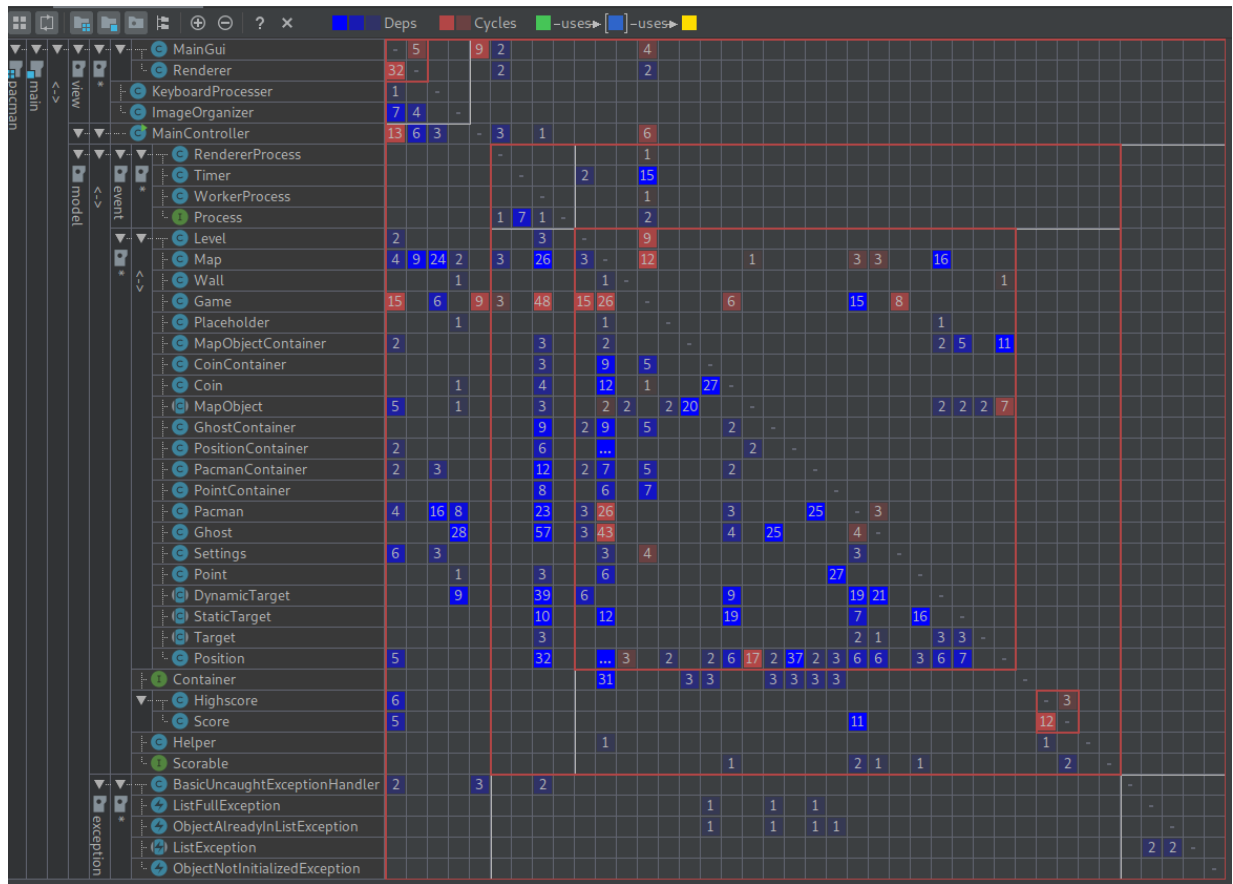


Figure 7: Dependency matrix for system 2

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 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 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 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).
5. Reuse all the same tools than in Section 1 to generate new reports and observe how quality varied.

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 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 11. We now consider to pass through a **Containers** class to construct any **Container** needed in the rest of the implementation. Its methods instantiate the right container with adapted type from generic classes. These instances are shown in orange in the Figure 11. A **PositionContainer** is a special container to hold coordinates, so we don't use an index but form a key from the couple (x, y) . The method `getRange(..)` allow to get a subset of contained positions, considering a rectangle selection formed from two positions. 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, and not represented in the figure.

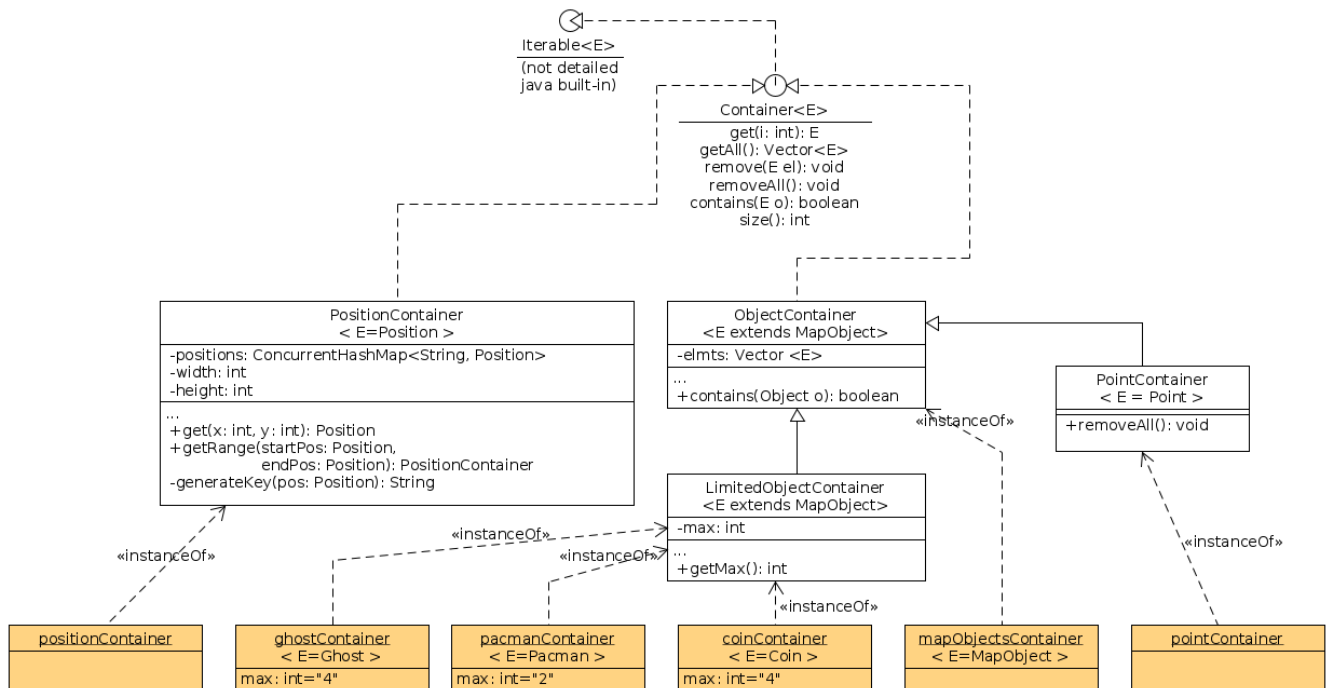


Figure 11: New class structure for Containers part of the implementation

3 Adding basic functionalities

4 Adding new features

5 Quality evolution analysis

6 Conclusion