
JPACMAN PROJECT ANALYSIS REPORT

A PREPRINT

Simona Santoro
s.santoro24@studenti.unisa.it

Marialisa Trerè
m.trere@studenti.unisa.it

Rosanna Coccaro
r.coccaro@studenti.unisa.it

April 26, 2019

1 Introduction

In this report we will analyze the JPacman project, to find various problems, such code smells, bad implementation choices, bugs, vulnerabilities and much more. After discovering the reasons and causes of these problems, we will try to give valid solutions for good refactoring choices.

The refactoring is really important for the software. Because you can find and resolve complexity problem, maintainability problems, and you can make your software easier to extend and more portable.

In the following sections, we will explain the code smells found and analyzed, and the related and possible solutions for remove them; vulnerabilities from which the code is affected; and, finally we will extract and analyze product metrics and a defect prediction model relating to them.

2 Code Smells

To detect the code smells in the code we used JDeodorant. JDeodorant is an Eclipse plug in. It's used in order to identify bad smells and resolve them by applying an appropriate refactoring. This plug-in is able to identifies five kinds of bad smells: **Feature Envy**, **Type Checking**, **Long Method**, **God Class**, **Duplicated code**. Our goal is: detect bad smells in JPacman by using JDeodorant. Mainly, our attention and analysis are concentrated in **Feature Envy**, **Long Method** and **God Class**, because this code smells are those that can create more problems in the code, increasing the complexity, the dependencies and maintainability costs.

2.1 Feature Envy

Feature envy occur when an method access to the data of another object more than its own data. Basically if things change together in the same time, you should take them in the same place. There are three way to refactor this type of smell:

- **Move Method**: if a method should be moved to another place
- **Extract Method** if only a part of method access the data of another object. If a method uses functions from different classes, first of all is necessary to determine which class contain most of data used and after move the method in that class with the other data. There are some case where we aren't able to determine which class contain most of data used, in this case we use the *Extract Method* in order to split the method into different part and move this part in the different classes.



Figure 1: Example of Feature Envy

This is an example of **Feature Envy** bad smell in JPacman. JDeodorant suggest us to move the method `getSinglePlayer()` from Launcher class to Game Class. This is because the method call the Game's method `getPlayers()` and doesn't use something of the Launcher's class, for this reason is better to use *Move method* refactoring then *Extract Method*.

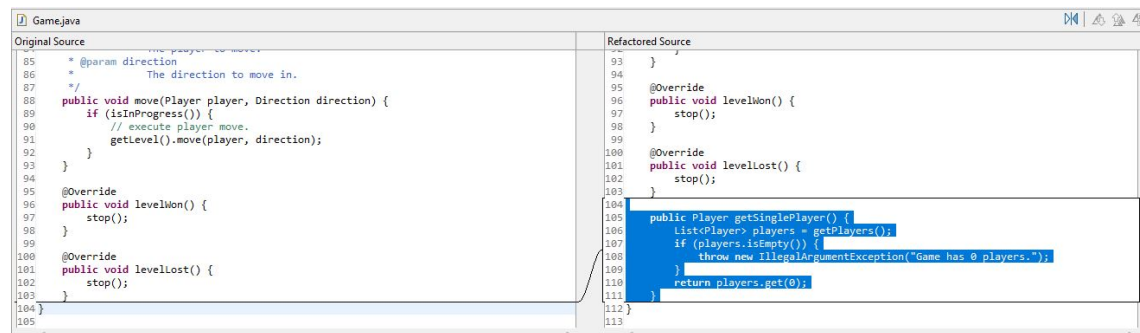


Figure 2: Example of Refactoring for Feature Envy

Is import to refactor Feature Envy smell, because this means having less duplicate code and also a better code organization.

2.2 Long Method

Long Method is a method that contains too many lines of code. This type of smell occur because is more easy to add new lines of code to an existing method than to create a new one. In the other hand, is more easy to write code than to read it. How we can refactor this type of smell? There are different way for different cases. The most used one is the Extract Method but in same case is used also, *Replace Temp with Query*, *Replace Method with Method Object*, *Decompose Conditional*. JDeodorant suggest us to resolve the *Long method* by using *Extract Method* refactoring. The idea is to create a new method in order to have one responsibility for method.

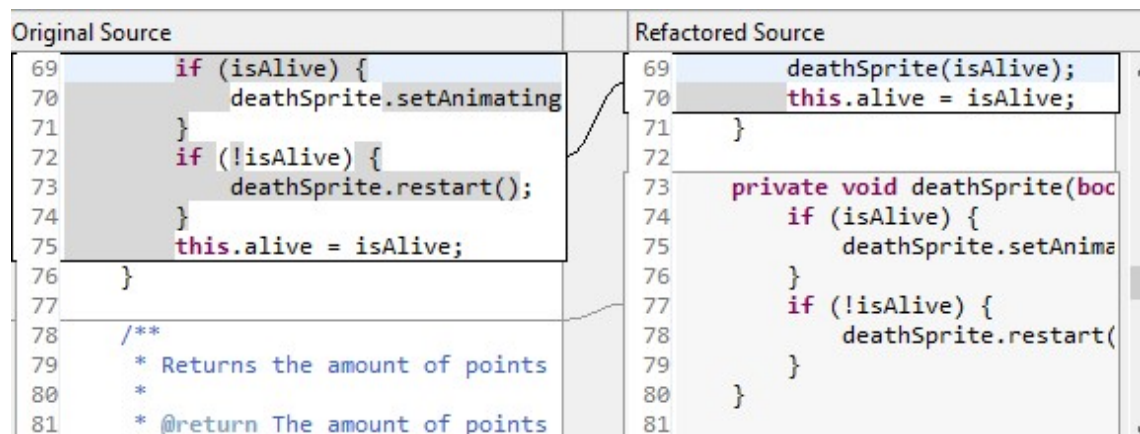


Figure 3: Example of Refactoring for Long Method

The figure represent the method `setAlive()`, this method implement also the responsibility to control if a players is alive or not, this isn't correct because the method is a set method. JDeodorante suggest to create a new method call `deathSprite` that implement some control about the life. Why is important to refactor this type of smell? Is important because longer a method is harder is to understand and maintain. Also, if a method implment different responsibilities the pobability of duplicated code improve.

2.3 God Class

A **God Class**(Blob) is a very important code smell. A class is defined a God class when implement several responsibilities, have a large number of attributes, operations and dependencies. This code smell makes it difficult to

understand the code and its features, and it increases maintenance.

There are several God classes in JPacman, such as the Level class, Unit, Ghost, MapParser, etc. There are several very significant classes that present this problem, and it is important to solve it.

The example in the figure, shows the Level class, it is one of the most obvious god classes, and it is one of the classes with multiple refactoring operations.

Refactoring Type	Source Class/General Concept	Extractable Concept	Source/Extracted at
▼	nl.tudelft.jpacman.level.Level		0/3
▼			
Extract Class		[player]	0/3
Extract Class		[cs, np]	0/1
Extract Class		[start]	1/13
Extract Class		[stop, start]	1/7
Extract Class		[stop, start]	1/5
Extract Class		[squar, start]	1/2
Extract Class		[stop, start, observ]	2/9
▼	nl.tudelft.jpacman.level.MapParser		1/3
▶			

Figure 4: Example of God Class

With this type of code smell, JDeodorant suggest us a refactoring method called *Extract Class*, with this solution different methods within the God class are moved to a new class.

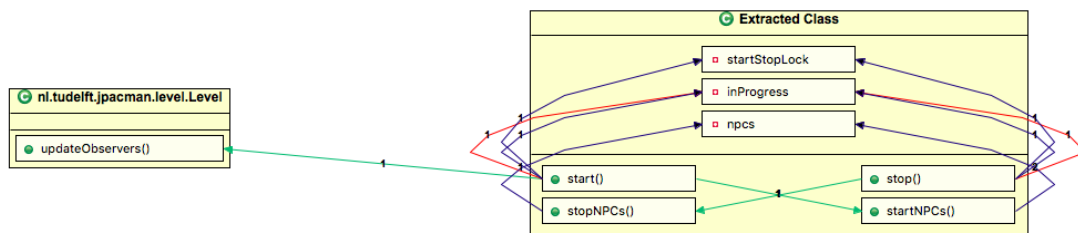


Figure 5: Example of refactoring solution

In this example, from the Level class, are moved methods concerning the start and stop of both player and NPCs, and the progress. These operations are moved to a new class, created for this purpose. This creates a new hierarchy between the original class and the one just created.

Original Source	Refactored Source
<pre> 199 * NPCs. 200 */ 201 public void start() { 202 synchronized (startStopLock) { 203 if (isInProgress()) { 204 return; 205 } 206 startNPCs(); 207 inProgress = true; 208 updateObservers(); 209 } 210 } 211 212 /** 213 * Stops or pauses this level, no longer 214 * and stopping all NPCs </pre>	<pre> 184 * NPCs. 185 */ 186 public void start() { 187 levelProduct.start(this); 188 } 189 190 /** 191 * Stops or pauses this level, no 192 * and stopping all NPCs. 193 */ 194 public void stop() { 195 levelProduct.stop(); 196 } 197 198 /** 199 * Returns whether this level is i </pre>

Figure 6: Moving the start() method

3 Vulnerabilities

3.1 Introduction

A **vulnerability** is a weakness which can be exploited by a threat actor, such as an attacker, to perform unauthorized actions within a computer system. To exploit a vulnerability, an attacker must have at least one applicable tool or technique that can connect to a system weakness.

Some vulnerabilities allow information leakage or elevate user privileges or grant otherwise unauthorized access. These are security vulnerabilities. If all software has bugs and it is inevitable that some bugs will be security vulnerabilities, all software will have security vulnerabilities.

To find vulnerabilities in the JPacman project, we have used an Eclipse pug-in called PMD.

3.2 About PMD

PMD is an open source static source code analyzer that reports on issues found within application code. PMD includes built-in rule sets and supports the ability to write custom rule. PMD does not report compilation errors, as it only can process well-formed source files. Issues reported by PMD are rather inefficient code, or bad programming habits, which can reduce the performance and maintainability of the program if they accumulate. So, It finds common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and so forth. It's mainly concerned with Java and Apex, but supports six other languages.

While PMD does not officially stand for anything, it has several unofficial names, the most appropriate probably being **Programming Mistake Detector**.

3.3 Analysis of Vulnerabilities

Since the system we are analyzing has been written in Java, we focus on all built-in rules available for Java. In Java, there are nine categories of built-in rules:

- **Best Practices:** rules which enforce generally accepted best practices.
- **Code Style:** rules which enforce a specific coding style.
- **Design:** rules that help you discover design issues.
- **Documentation:** rules that are related to code documentation.
- **Error Prone:** rules to detect constructs that are either broken, extremely confusing or prone to runtime errors.
- **Multithreading:** rules that flag issues when dealing with multiple threads of execution.
- **Performance:** rules that flag suboptimal code.
- **Security:** rules that flag potential security flaws.
- **Additional rulesets:** rules that contain all those deprecated.

In order to identify the most important defects in JPacman, we utilize critical levels that PMD assigns to each rule.

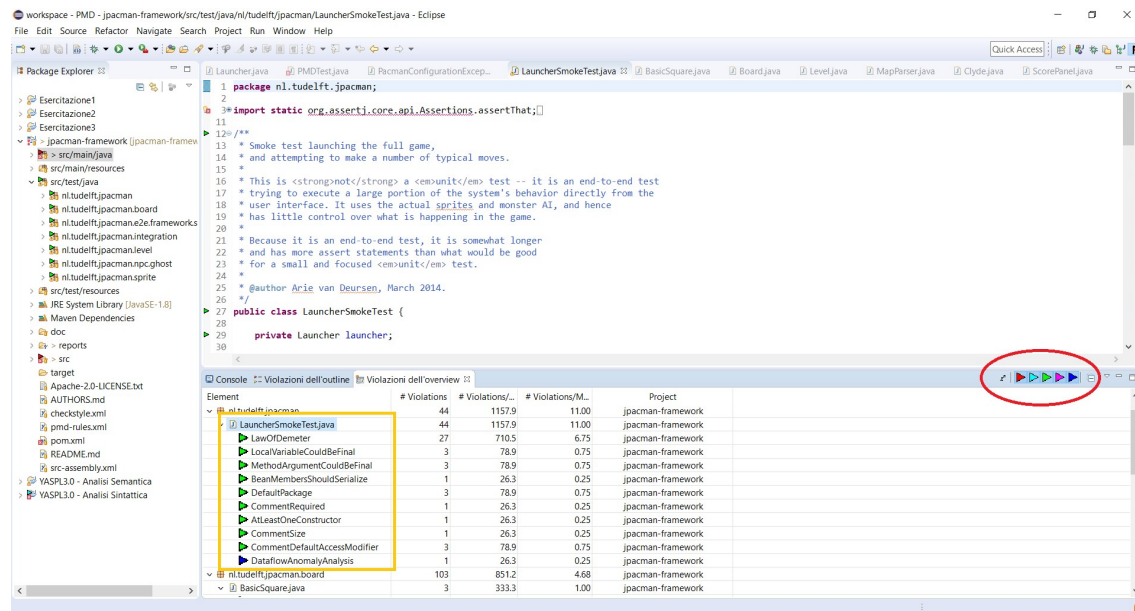


Figure 7: PMD defect classification

In the figure, the levels of importance of each defect found in the code have been highlighted in different colors:

- ▶ blocker violation
- ▶ critical violation
- ▶ urgent violation
- ▶ important violation
- ▶ warning violation

The figure also shows the violations that occurs for each class in different packages (highlighted through a yellow rectangle).

The **Launcher.java** class contains a series of violations, including:

- BeanMemberShouldBeSerialize
- LocalVariableCouldBeFinal
- AtLeastOneConstructor

Now we're going to analyze each of them in detail and we will illustrate some solutions to overcome these problems. First of all, we have **BeanMemberShoulBeSerialize**. PMD identifies this violation through a green flag that shows a certain urgency to solve it. So the priority associated to it is **average**. This violation belongs to **Error Prone Category**, so it concerns constructs that are faulty.

BeanMemberShoulBeSerialize indicates that if a class is a bean, or is referenced by a bean directly or indirectly it needs to be serializable. Member variables need to be marked as transient, static, or have accessor methods in the class.

Marking variables as transient is the safest and easiest modification. Accessor methods should follow the Java naming conventions. PMD recommends to mark the fields that constitute the problem, with **transient** keyword.

Transient is a Java Keyword which marks a member variable not to be serialized when it is persisted to streams of

bytes. So, member variables marked by the java transient keyword are not transferred: they are lost intentionally. After making the necessary transformations to the Launcher.java class, the situation is shown in the following image:

The screenshot shows the Launcher.java file in an IDE. The code is as follows:

```

23 /**
24  * Creates and launches the JPacMan UI.
25  *
26  * @author Jeroen Roosen
27  */
28 @SuppressWarnings("PMD.TooManyMethods")
29 public class Launcher {
30
31     private static final PacManSprites SPRITE_STORE = new PacManSprites();
32
33     public static final transient String DEFAULT_MAP = "/board.txt";
34     private transient String levelMap = DEFAULT_MAP;
35
36     private transient PacManUI pacManUI;
37     private transient Game game;
38
39     /**
40      * @return The game object this launcher will start when {@link #launch()}
41      *         is called.
42      */
43     public Game getGame() {
44         return game;
45     }
46

```

Below the code, the PMD violations overview table is displayed:

Element	# Violations	# Violations/KLOC	# Violations/Method	Project
> (default package)	1328	N/A	N/A	YASPL3.0 - Analisi Sintattica
> nl.tudelft.jpacman	24	358.2	1.20	jpacman-framework
▼ Launcher.java	20	317.5	1.11	jpacman-framework
▶ LawOfDemeter	6	95.2	0.33	jpacman-framework
▶ LocalVariableCouldBeFinal	4	63.5	0.22	jpacman-framework
▶ MethodArgumentCouldBeFinal	3	47.6	0.17	jpacman-framework
▶ CommentRequired	5	79.4	0.28	jpacman-framework
▶ AtLeastOneConstructor	1	15.9	0.06	jpacman-framework
▶ ShortVariable	1	15.9	0.06	jpacman-framework

Figure 8: Situation after fixing BeanMemberShouldBeSerialize violation

Before, we had three instances of the BeanMemberShouldBeSerialize violation, now this violation in the Launcher.java class does not occur.

But as we can see, there are still other important violations to be resolved. As already mentioned we focus only on the most critical ones.

One of these is the **LocalVariableCouldBeFinal** violation. In the Launcher.java class there are four instances of this kind of violation. PMD identifies all the LocalVariableCouldBeFinal violations through a green flag. The priority associated to it, isn't very high but it is average. This violation belongs to **Code Style**, so it concerns some rules that guarantee a specific coding style which allow you to avoid open ports that can be maliciously used by others to perform attacks.

To avoid this kind of violations, it's necessary to declare **final** a local variable assigned only once.

Finally, in Launcher.java class we have also another important violation: **AtLeastOneConstructor**. The name already suggest us that it is a class without constructor. One of the Java principles is that each class must have at least one constructor that allows us to instantiate a new object. This is necessary even if they are "abstract" classes from which it is not possible to invoke the constructor to instantiate new objects. This means that each non-static class should declare at least one constructor.

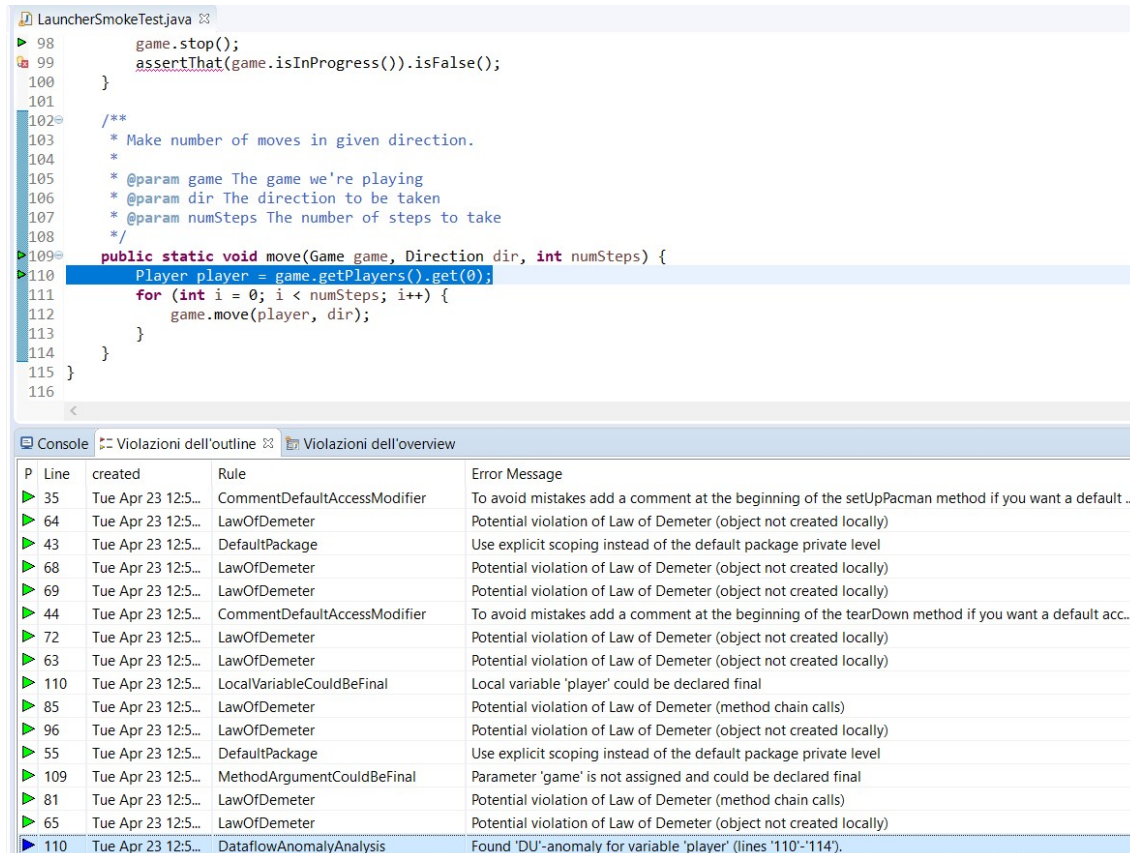
Therefore, a good rule would be to add to our class that violates this principle, an empty constructor, or that in any case initializes the fields in order to obtain future objects whose data are in a consistent state. This kind of violation is also an urgent violations for PMD that assign to it a medium priority.

Our analysis continues in search of other violations that seem to be critical. So, we come across the **PacmanConfigurationException.java** class. A violations that seems to be very important is **MissingSerialVersionUID** that is included in the **Error Prone** rules. The message shown in console says: "Classes implementing Serializable should set a serialVersionUID". Serializable classes should provide a serialVersionUID field. The serialVersionUID field is

also needed for abstract base classes. Each individual class in the inheritance chain needs an own serialVersionUID field. The class PacmanConfigurationException.java extends RuntimeException: so, why do we need serialVersionUID when extending RuntimeException? Is RuntimeException a serializable class? RuntimeException extends Exception, Exception extends Throwable. Finally, Throwable implements Serializable. So PacmanConfigurationException.java is Serializable too.

PMD assigns to this kind of violation an **high priority** because making an exception serializable means it can be transferred across the network between tiers of a distributed application.

Finally, in the **LauncherSmokeTest.java** class, an instance of **DataflowAnomalyAnalysis** violation occurs.



```

LauncherSmokeTest.java
98     game.stop();
99     assertThat(game.isInProgress()).isFalse();
100 }
101
102 /**
103  * Make number of moves in given direction.
104  *
105  * @param game The game we're playing
106  * @param dir The direction to be taken
107  * @param numSteps The number of steps to take
108  */
109 public static void move(Game game, Direction dir, int numSteps) {
110     Player player = game.getPlayers().get(0);
111     for (int i = 0; i < numSteps; i++) {
112         game.move(player, dir);
113     }
114 }
115 }
116

```

P	Line	created	Rule	Error Message
▶	35	Tue Apr 23 12:5...	CommentDefaultAccessModifier	To avoid mistakes add a comment at the beginning of the setUpPacman method if you want a default ...
▶	64	Tue Apr 23 12:5...	LawOfDemeter	Potential violation of Law of Demeter (object not created locally)
▶	43	Tue Apr 23 12:5...	DefaultPackage	Use explicit scoping instead of the default package private level
▶	68	Tue Apr 23 12:5...	LawOfDemeter	Potential violation of Law of Demeter (object not created locally)
▶	69	Tue Apr 23 12:5...	LawOfDemeter	Potential violation of Law of Demeter (object not created locally)
▶	44	Tue Apr 23 12:5...	CommentDefaultAccessModifier	To avoid mistakes add a comment at the beginning of the tearDown method if you want a default acc...
▶	72	Tue Apr 23 12:5...	LawOfDemeter	Potential violation of Law of Demeter (object not created locally)
▶	63	Tue Apr 23 12:5...	LawOfDemeter	Potential violation of Law of Demeter (object not created locally)
▶	110	Tue Apr 23 12:5...	LocalVariableCouldBeFinal	Local variable 'player' could be declared final
▶	85	Tue Apr 23 12:5...	LawOfDemeter	Potential violation of Law of Demeter (method chain calls)
▶	96	Tue Apr 23 12:5...	LawOfDemeter	Potential violation of Law of Demeter (object not created locally)
▶	55	Tue Apr 23 12:5...	DefaultPackage	Use explicit scoping instead of the default package private level
▶	109	Tue Apr 23 12:5...	MethodArgumentCouldBeFinal	Parameter 'game' is not assigned and could be declared final
▶	81	Tue Apr 23 12:5...	LawOfDemeter	Potential violation of Law of Demeter (method chain calls)
▶	65	Tue Apr 23 12:5...	LawOfDemeter	Potential violation of Law of Demeter (object not created locally)
▶	110	Tue Apr 23 12:5...	DataflowAnomalyAnalysis	Found 'DU'-anomaly for variable 'player' (lines '110'-'114').

Figure 9: Example of DataFlowAnomalyAnalysis violation

PMD indicates a DataflowAnomalyAnalysis vulnerability through a blue flag. **Blue flag** shows warnings that can occur during the implementation of a program on eclipse. The DataFlowAnomalyAnalysis is also a violation that is included in the **Error Prone** class. The dataflow analysis tracks local definitions, undefinitions and references to variables on different paths on the data flow. From those information there can be found various problems:

1. **UR - Anomaly:** there is a reference to a variable that was not defined before. This is a bug and leads to an error.
2. **DU - Anomaly:** a recently defined variable is undefined. These anomalies may appear in normal source text.
3. **DD - Anomaly:** a recently defined variable is redefined. This is ominous but don't have to be a bug.

In our case, it is a DU-Anomaly for variable 'player'. The solution to this anomaly is to utilize a field Player for the LauncherSmokeTest.java class instead of creating two different 'player' variables in the two methods that use it.

4 Product metrics and defects prediction

Product and process metrics are really useful and important for the analysis of a software.

These metrics can be used to improve the development and maintenance activities of the software, and describe the characteristics of the product such as size, complexity, design features, performance, and quality level.

It is important to be aware of this information, because we can analyze them and thus be able to predict possible faults and bugs.

To extract information about these metrics we have used PyDriller, it is a Python framework that helps to extract information from any Git repository. With PyDriller, we have extracted, mainly, each commit with information about changes, additions or removals, complexity, and if each file in a commit is buggy or not. We can build metrics to better understand and predict potential bugs and vulnerabilities induced by programmers during the development cycle.

First of all, we have extracted the number of commits with bugs and the number of commits with fixes. This because, past faults and number of changes are the better indicators of bug presence. Moreover, we have to know that files, which are changed often, have a higher probability of being defective due to the fact that any change has a certain probability of introducing new defects. Again, files, which are involved in bug fixing activities, have a higher probability of containing defects. We expect such finding as files, which are fixed, already are defect prone and fixes bring the risk of introducing new defects. Finally, the maximum number of files committed to the repository within a short time window seems also to contain some information about its defect proneness.

As more files are touched and modified in a short time, in a single refactoring session, the probability of introducing bugs increases. Generally, we have a fix commit after bug inducing commits. As we can see, we have more bug

```
number of commits with bugs: 48
number of commits with fixes: 12
```

Figure 10: Number of bug commits and fix commits

inducing commits that fix commit, this indicates that multiple files with bugs have been fixed in the same fix commit even after long periods. In fact, as we can see in the figure 2, we have also 28 commits with bugs before a fix commit.

```
number of commits with bugs: 1
number of commits with bugs: 3
number of commits with bugs: 2
number of commits with bugs: 2
number of commits with bugs: 15
number of commits with bugs: 6
number of commits with bugs: 4
number of commits with bugs: 28
number of commits with bugs: 3
number of commits with bugs: 3
number of commits with bugs: 5
number of commits with bugs: 1
number of commits with fixes: 12
```

Figure 11: Number of bug commits before each fix commit.

For the analysis, we have considered all commits with fixes, and the corresponding bug inducing commits.

After this, we got, always thanks to PyDriller, a set of information about the repository. The kind of data collected from each *.java* files are:

- File
- numChanges
- linesAdded
- linesRemoved
- LOC(lines of code)
- complexity
- isBuggy
- isTestFile


```

File: Launcher.java. Changes: 53. Added: 2. Removed: 8. LOC: 104. Comp.: 20. Buggy:False. isTestFile:False.
File: Board.java. Changes: 32. Added: 1. Removed: 5. LOC: 35. Comp.: 12. Buggy:False. isTestFile:False.
File: Square.java. Changes: 20. Added: 1. Removed: 4. LOC: 45. Comp.: 10. Buggy:False. isTestFile:False.
File: Unit.java. Changes: 26. Added: 3. Removed: 3. LOC: 51. Comp.: 13. Buggy:False. isTestFile:False.
File: Game.java. Changes: 15. Added: 80. Removed: 80. LOC: 52. Comp.: 12. Buggy:False. isTestFile:False.
File: Level.java. Changes: 1. Added: 9. Removed: 0. LOC: 5. Comp.: 0. Buggy:True. isTestFile:False.
File: Player.java. Changes: 4. Added: 50. Removed: 5. LOC: 41. Comp.: 8. Buggy:True. isTestFile:False.
File: Sprite.java. Changes: 8. Added: 43. Removed: 43. LOC: 8. Comp.: 0. Buggy:False. isTestFile:False.
File: Action.java. Changes: 5. Added: 6. Removed: 6. LOC: 4. Comp.: 0. Buggy:False. isTestFile:False.
File: BoardPanel.java. Changes: 9. Added: 101. Removed: 101. LOC: 51. Comp.: 7. Buggy:False. isTestFile:False.
File: ButtonPanel.java. Changes: 7. Added: 23. Removed: 23. LOC: 21. Comp.: 2. Buggy:False. isTestFile:False.
File: PacKeyListener.java. Changes: 7. Added: 33. Removed: 33. LOC: 25. Comp.: 5. Buggy:False. isTestFile:False.
File: PacManUI.java. Changes: 19. Added: 1. Removed: 3. LOC: 50. Comp.: 4. Buggy:False. isTestFile:False.
File: PacManUiBuilder.java. Changes: 20. Added: 1. Removed: 2. LOC: 58. Comp.: 9. Buggy:False. isTestFile:False.
File: ScorePanel.java. Changes: 18. Added: 76. Removed: 76. LOC: 47. Comp.: 7. Buggy:False. isTestFile:False.
File: Direction.java. Changes: 10. Added: 56. Removed: 56. LOC: 19. Comp.: 3. Buggy:False. isTestFile:False.

```

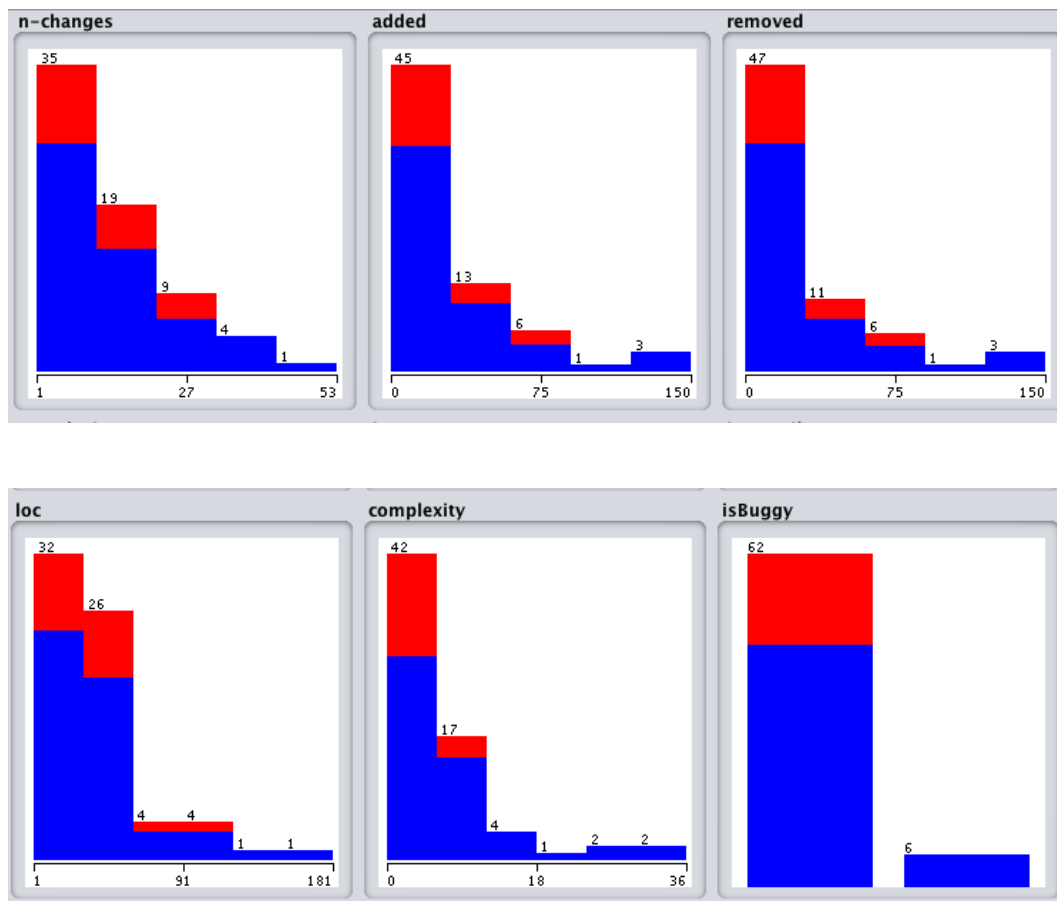
Figure 12: Example of the extracted data

This data are stored in a csv file, and we have used this file for the defect prediction model.

A **defect prediction model** identifies the modules that are defective. A prediction on possible error leads to effective allocation of resources, reduces the time and cost of developing a software and improve the quality of the product. For this reason, an software defect prediction model is very important for understanding, evaluating and improving the quality of a software system.

For the defect prediction model, we have used Weka. Weka(Waikato Environment for Knowledge Analysis) is open source software that contains a collection of visualization tools and algorithms for data analysis and predictive modeling, together with graphical user interfaces for easy access to these functions. Weka implements a large collection of machine learning algorithms and is widely used in data mining applications, it contains tools for data pre-processing, classification, regression, clustering, association rules, and visualization.

The data looks like this:



Some important types of classifiers are:

- Bayes classifiers
- Rules classifiers
- Tree classifiers

We have used different algorithms to analyze the data on csv file, such *NaiveBayes* and *RandomForest*, but finally we decided to use an algorithm called **Random Tree**. The Random Tree is an algorithm that is part of the decision tree algorithms, it is a supervised Classifier; it is an ensemble learning algorithm that generates lots of individual learners. Ensemble learning combine the predictions from multiple models. This reduce the variance of prediction and improve the results. RandomTree employs a bagging idea to construct a random set of data for constructing a decision tree. In standard tree every node is split using the best split among all variables. For validation we use the 10-fold cross validation strategy. It's mean that the dataset is randomly partitioned in ten equal size folds. Of the 10 subsamples, a single subsamples is retained as the validation data for testing the model, the remain subsamples are used as training data. For avoid problem due to the initial random splitting the cross-validation process is repeated for 10 times. The advantage of this method is that all observations are used for both training and validation, and each observation is used for validation exactly once.

In the figure below, we can see the confusion matrix that was produced by the RandomTree algorithm.

Observing the confusion matrix, we can calculate the *accuracy*. The **accuracy** is the most intuitive performance measure and it is simply a ratio of correctly predicted observation to the total observations; in our case, with RandomTree we have the accuracy of *0.9264*. The accuracy is a great measure but only when we have symmetric datasets where values of false positive and false negatives are almost same. Therefore, we have to look at other parameters to evaluate the performance of our model. So, we can consider two others parameters, called *precision* and *recall*.

Precision is the ratio of correctly predicted positive observations to the total predicted positive observations. **Recall** is the ratio of correctly predicted positive observations to the all observations in actual class - yes.

In our case, we have the precision of *0.667* and the recall of *0.333*. With precision and recall we can calculate another useful parameter called F-Measure, it is the ratio between the product and the sum of precision and recall, multiplied by 2; in our case we have the F-Measure of *0.45*.

```
a  b  <-- classified as
61  1 | a = False
  4  2 | b = True
```

Figure 13: Confusion matrix

To the Jpacman project, participated 18 members. But only 4/5 members have touched and modified all the files. Usually, the higher the number of developers that touch a file, the higher its bug-proneness, but it's not always like this, the problem is when the developers concentrate on the features in a scattered way.

A complex code base, the addition of a large number of features within a short period of time, or a large number of developers simultaneously changing the source code of a project, are some of the many reasons that could cause code modifications to be highly scattered. This scatter of modifications throughout the code, within a short time, makes it difficult for developers working on the project to keep track of its progress and the changes. A complex modification pattern will cause delays in releases, high bug rates, stress and anxiety to all the personnel involved in a project. And in this project we can find many classes modified several times in the same day with different commits, even by different authors, and this could lead to problems.

5 Conclusion

In conclusion, we have noticed that the most popular code smell in JPacman are God Class, consequently most refactoring operations are extract class. A lot of vulnerabilities, detect with PDM, are error prone. Also, the majority vulnerabilities detect by PMD are urgent (the green flag). The most frequent and important vulnerabilities are: *BeanMemberShouldBeSerialize*, *LocalVariableCoulBeFinal*, *AtLeastOneConstructor*.

About the defect prediction model, the RandomTree Algorithm used gives us the *0.92* accuracy, compare to RandomForest and NaiveBayes that respectively gives us the *0.91* and *0.89* of accuracy. So, in conclusion we can say that the RandomTree Algorithm is the best one.