Software Dependability report of JPacman

Francesco De Simone, Luca Di Luccio, M. Tanveer Jan

Università degli Studi di Salerno
Software Dependability course
Prof. Fabio Palomba

April 26, 2019

Abstract

Different activities are being conducted out on JPacman Framework. These activities consist of two main roles. The first role is about code smells and how to refactor them. Detecting code smells in the code and consequently applying the right refactoring steps is important to improve the quality of the code. Different types of code smells are being discussed that we found in the JPacman Framework with the help of JDeodrant along with an interpretation on how to encounter or refactor these smells. The second role is about the Prediction Models. Different types of tools like Pydriller with Weka are being used in order to get all the required parameter to generate a defect predicted model. Many Algorithms are tested based on the data that is generated from Github repository of JPacman Framework.

1 Introduction

A software defect is an error, bug, flaw, fault, malfunction or mistakes in software that causes it to create an erroneous or unpredicted outcome.Software flaws are programming errors which cause different performance compared with anticipation. In this report we try to analyze all the possible defects in jPacman. The analysis consist in three phases:

1. Code smells analysis
2. Study with static analysis technique the possible vulnerabilities
3. Extraction of production metrics and define a defect prediction model using Weka

We must consider that not necessarily all the code smells have to be removed, it depends on the system, sometimes the smell cannot be removed or some time remove a smell can not worth it for other paramether (performance for example).

2 Code Smells Detection

A code smell is any characteristics in a software that can possibly indicate a deeper problem. The definition of a code smell is subjective and varies by language, developer and development methodology. In the last few years there has been a number of research in the field that classified some objective code smells in order to give to the developers a sense of writing better code.

There are a lot of code smells that can be individuated and fixed in a programmatic way, we used the open-source tool JDeodorant to better understand the code smells of the code.

JDeodorant has been developed in a joint effort between the Software Refactoring Lab at the Department of Computer Science and Software Engineering, Concordia University, Canada and the Software Engineering Group at the Department of Applied Informatics, University of Macedonia, Thessaloniki, Greece.

In the following sections we will talk about smells we found in the JPacman framework and how to remove them.

2.1 Blob/God Class

This smell happen when there is a class in our repository that has too many responsibilities or a large number of attributes, operations and dependencies with data classes. The usual strategy to fix this kind of smell is to extract the behavior to a new/different class.

2.1.1 jpacman.board.Square

Inside this class there is a `List<Unit>` that should describe who is currently occupying that's `Square.java`. In order to fix it the tool suggest to extract the behavior, shifting the whole process to a different class. This way it is clearer to work with and we make better use of the strategies that the object oriented programming gives us.

2.1.2 jpacman.level.mapParser

In order to streamline the whole MapParser class, JDeodorant offers a way to split it into 2 different entities:

- MapParser which offer a way to manage `char[][]` elements
- MapParserProcessor that offers all the methods to actually change the map.

2.2 Type checking

This smell happens when the different types used in the project does not fully use the strategies given by the object oriented programming such as Inheritance and Abstract classes.

2.2.1 jpacman.level.LevelFactory

Because we have to create different kind of enemies (ghosts) it is suggested to use the design pattern called Strategy, this way we can simplify the creation of each ghost by making a single call to a generic function (e.g. `element.createGhost()`) to automatically generate the right one. This way we have to implement a new class for each case inside the switch statement, one for each kind of enemy. By doing it this way it will be easier to implement new enemies and we will never have to modify differents bit of code in order to make it work.

2.2.2 jpacman.level.PlayerCollision

The series of `if statements` that the current JPacman code does in order to check the collision of its entities is repetitive and error prone. This happen because the inheritance is used in the wrong way. To fix this error we must re-think the whole class/type hierarchy with 3 key changes:

1. Make all the class implement a `Collidable` interface
2. Implement the method `onCollide()` for each of them
3. Use the method `onCollide()` without checking the type of the element calling it

2.3 Feature Envy

This smell is usually detected when one object gets at the fields of another object in order to perform some sort of computation or make a decision, rather than asking the object to do the computation itself. The usual approach to fix it is to take the computation and put it inside the object in order to reduce the amount of computation done outside of the class itself.

2.3.1 jpacman.Launcher

In this class we have found a method that uses a Game object to calculate the players. Its a clear indication of a feature envy because we can shift the computation to the game class in order to have a more cohesive class interface.

2.4 Long Method

These smells are found when a method contains too many lines of code. As a rule of thumb, any method longer than 10/15 lines should be split into multiple ones. This smell can be easily refactored by applying one of the following strategies:

- Extract the body into a new Method
- Introduce a ParameterObject
- Decompose the if/switch statements
- Simply keep the Method

2.4.1 jpacman.ui.ScrorePanel

This method is just 7 lines long but JDeodorant still treats it as a long method. The tool suggest to split it up by making a new method just to calculate the score. Even if after this refactoring the code will be prettier to look at, we don't think that it will improve the actual functionality or will make the code more understandable.

2.5 Other smells

The smells that we've looked into are not the only ones that are found during the refactoring process. Unfortunately the tools at our disposal were not able to pick up those smells and we must acknowledge only the presence of the ones we've found as we can't prove that our system is exempt from them. Some of the other smells that we could have found are:

- Middle man:
  As a project grows in size, delegation problems starts to appear inside the codebases. The usual symptom is a class that only performs one action which is delegating work to another class.

- Primitive Obsession:
  Sometimes developer uses primitives instead of small objects for simple tasks, this is not a good practice as they are forced to use too many constants for coding information.

- Inappropriate Intimacy:
  One class uses the internal fields and methods of another class. This behavior limits the re-usability and maintainability of the code.

3 Analysis of possible vulnerabilities

The static analysis is the analysis of computer software that is performed without actually executing programs. The term is usually applied to the analysis performed by an automated tool such as: PMD, SpotBugs and Checkstyle.

The main tools used by our team has been SpotBugs. It is a quick and easy tool that help the developer to better understand and predict problems with the system even before the actual running time.

SpotBugs can checks for more than 400 bug patterns, the most important ones are:

- Performance
- Security
- Correctness
- Bad Practise
- Internationalization

The default settings of the tool do not detect any severe bug or bad practice. Because of this we decided to reduce the minimum confidence of report to ”Low” and to include all the possibles categories of bugs into the analysis.

The results obtained shows that the code is not subject to dangerous vulnerabilities and all the bugs individuated are position 19 and 20 in a scale from 0 (severe) to 20 (meaningless). The SpotBug's report shows two vulnerability:

- `Use of non-localized toUpperCase()[..]`
  This happens because a String is being converted to upper or lowercase, using the platform's default encoding.

- `"method()"is never called`
  This is self explanatory.

- `Classes is not Serializable[..]`
  This is not a mistake because the developer never intended the class to be serializable in the fist place, its a suggestion that the tools give us to make the code more future-proof.

As we can see from the report the vulnerabilities those errors are can be easily solved with some precautions and does not interfere with the application execution itself.

We also checked the JPacman framework with the Checkstyle tool using both Google's and Sun's coding convention. The tool don't find any vulnerability in the code style as they are compliant with the specification.

4 Defect Prediction Model

Software Defect Prediction identifies the modules that are defective. It requires a wide range of testing. A prediction on possible error leads to effective allocation of resources, reduces the time and the cost to develope a software and to improve the quality of the product. For this reason, a software defect prediction model plays a vital role in understanding, evaluating and improving the quality of a software system. Weka is open source software that implements a large collection of machine learning algorithms and it's widely used in data mining tasks.

Weka give us some algorithms for classification and some filter for cleaning data that are useful to manage the data-set and build the model. The algorithms output data as precision and recall based on four values, namely True Positive, False Positive, False Negative and True Negative. We could take data of JPacman-framework with PyDrill and analyzes risks of software defects predicts.

4.1 PyDriller

Developers and researchers might need a quick and easy way to obtain metrics about a specific project. That's where PyDriller comes in.

PyDriller is a Python framework that can help the mining process on software repositories. With this tool we can easily extract information and metrics from any public Git repository, making the mining process as easy as writing python scripts. With data on developers, history of commits, lines of code changed, complexity of a specific change and much more, we can build metrics to better understand and predict potential bugs and vulnerabilities induced by programmers during the development cycle.

The software is distributed for free under the Apache License 2.0 through the GitHub page of the project or the pip repository. It is available for free and commercial uses and suits perfectly our needs.

During the creation of the Defect Prediction Model we used PyDriller extensively is order to obtain data about the repository. With the tool to get a list of all the buggy commits we have to write less than 10 lines of code while getting metrics about the commits is another 10-15 lines. Through the use of Python's own csv library, exporting all the data in such file format was an easy task to accomplish.

The kind of data collected from each `.java` files into the csv is:

- filename
- numchanges
- lineadded
- lineremoved
- linesofcode
- complexity
- isBuggy

4.2 Classification Machine Learning Algorithms

After extracting data with PyDriller from the jPacman-framework repo, we can analyze the result with Weka. With PyDriller we have extracted 61 instances with 6 attributes. Weka plot for us the univariate attribute distributions (Figure 1). From the plot we observe that the data is unbalanced, for example in figure 2 we notes that the instances with bugs (drawn in blue) are less than the instances without. We use the `SMOTE` algorithm to balance the dataset by creating

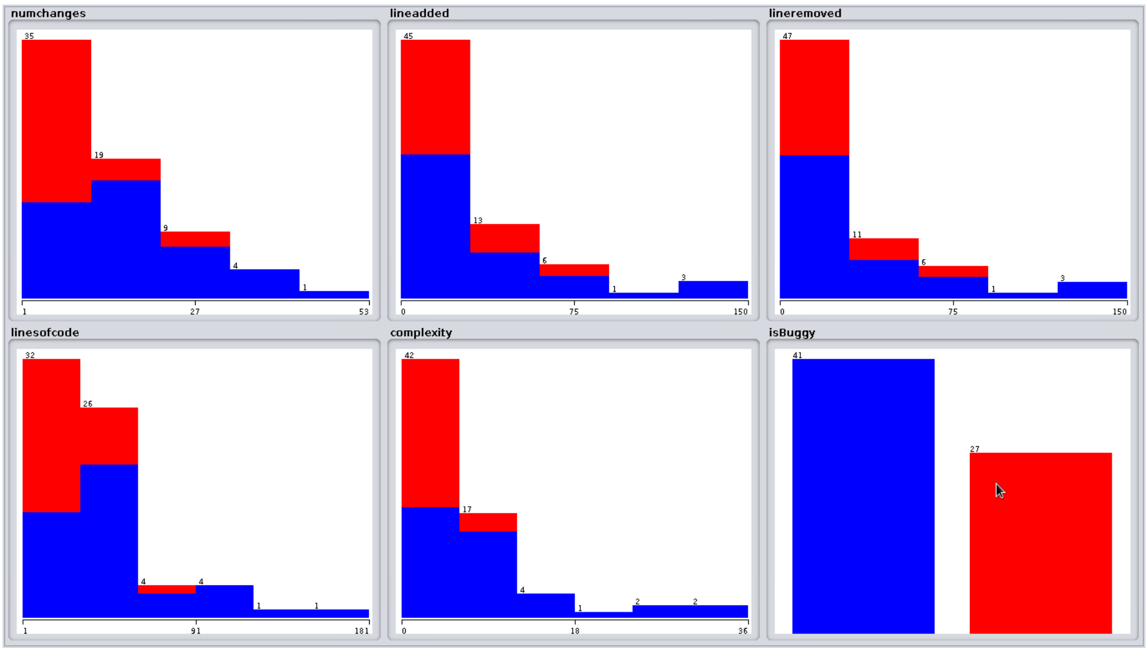

Figure 1: The plot of the raw univariate attribute distributions

synthetic instances of the minority class using statistical methods and analysis of the distribution. After the application of `SMOTE` we randomize the dataset with the filter `Randomize` because `SMOTE` generatet the same instances at the end of the file. With SMOTE we increase our instances to 88: in figure 4 we can see the difference and in the figure 3 all the final results.

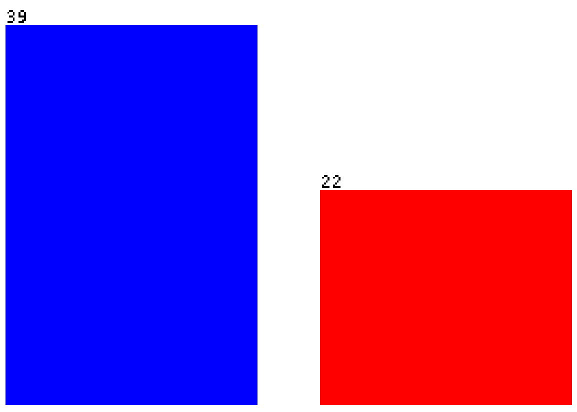

Figure 2: Bug-prone classes (blue) and bug-free classes (red) - unbalaced dataset

Analyzed the various algorithm given by weka we focused on the ensemble based algorithms. Ensemble algorithms differ from stochastic training algorithm because the latters are sensitive to the specifics of the training data and may find a different set of weights each time they are trained. Ensemble learning combine the predictions from multiple models instead. This reduce the variance of prediction and improve the results.

For the analysis we use the `Random Forest` model. `Random Forest` is an extension of bagging for decision trees that can be used for classification or regression. A down side of bagged decision trees is that are constructed using a greedy algorithm that selects the best split point at each step in the tree building process. As such, the resulting tree end up looking very similar and this process reduces the variance of the predictions.

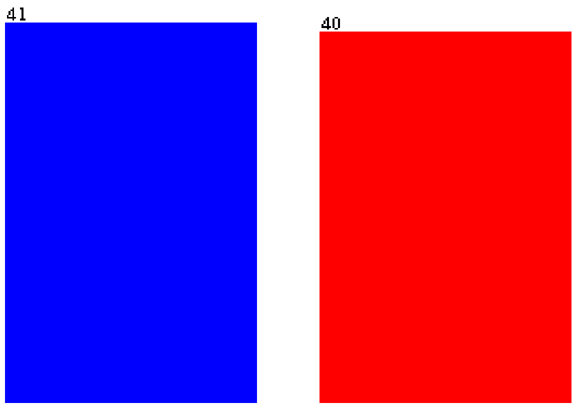

Figure 4: Bug-prone classes (blue) and bug-free classes (red) - balanced dataset

`Random Forest` is an improvement upon bagged decision trees that disrupts the greedy splitting algorithm during tree creation so that split points can only be selected from a random subset of the input attributes. This simple change can have a big effect decreasing the similarity between the bagged trees and in turn the resulting predictions.

For validation we use the 10-fold cross validation strategy. It means 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

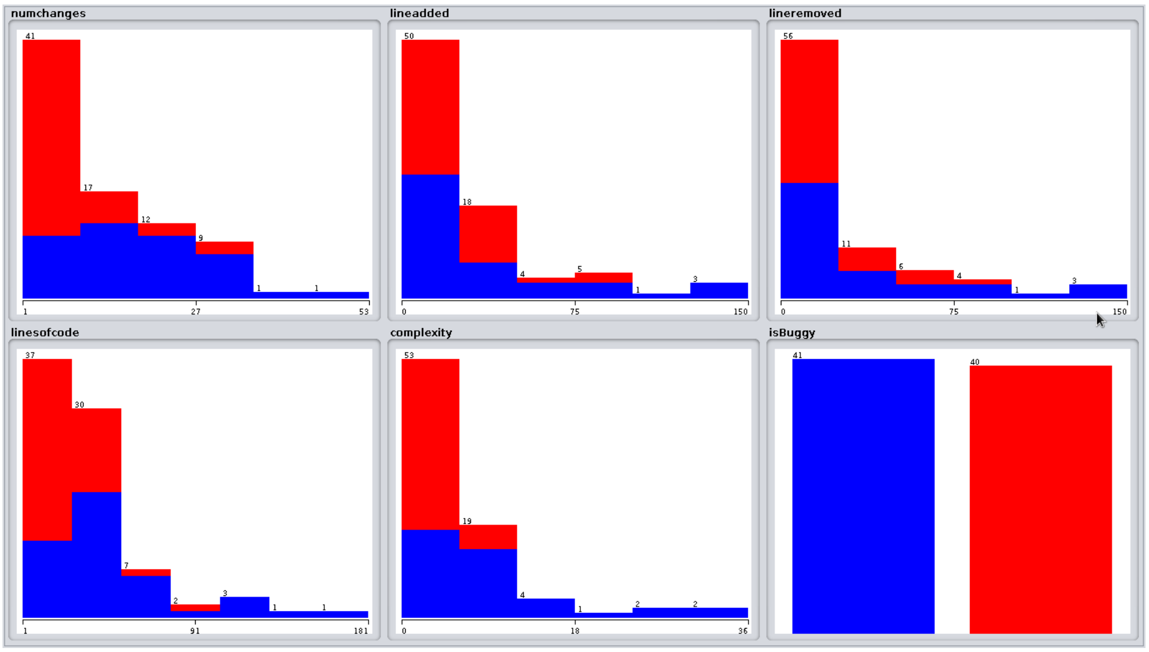

Figure 3: The plot of the univariate attribute distributions with SMOTE e Randomize

testing the model, the remain subsamples are used as training data. To avoid problems 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. From the output of the algorithm we can analyze the confusion matrix as shown in Table 1 for both bug-prone and bug-free classes. We calculate this values for the bug-prone classes:

- Precision
- Recall
- F-Measure

Precision is the fraction of relevant instances among the retrieved instances. With our data we calculate a precision of:

$$precision = \frac{TP}{TP + FP} = \frac{30}{30 + 12} \cong 0.714$$

Recall is the measure of the proportion of actual positives that are correctly identified as such

$$recall = \frac{TP}{TP + FN} = \frac{30}{30 + 11} \cong 0.732$$

We combine precision and recall to have the F-Measure which represent a measure of a test's accuracy and it is used for measuring the model performance.

$$\begin{aligned} F - Measure = 2 * \frac{precision * recall}{precision + recall} = \\ = 2 * \frac{0.714 * 0.732}{0.714 + 0.732} \cong 0.723 \end{aligned}$$

The F-Measure is near to 1, so we register an acceptable accuracy (about 71.60%)

Table 1: Confusion Matrix

	A	B
A	30	11
B	12	28

A = false, B = true

With the Table 1 we can calculate also the accuracy for the bug-free classes where result a:

$$precision = \frac{TP}{TP + FP} = \frac{28}{28 + 11} \cong 0.718$$

$$recall = \frac{TP}{TP + FN} = \frac{28}{28 + 12} \cong 0.7$$

$$\begin{aligned} F - Measure = 2 * \frac{precision * recall}{precision + recall} = \\ = 2 * \frac{0.718 * 0.7}{0.718 + 0.7} \cong 0.709 \end{aligned}$$

Also here the F-Measure is quite near to one.

5 Conclusion

The analysis result show how jPacman doesn't show important vulnerability. Several tools was used to check any kind of problem without spot a particular vulnerability. However, regarding code smell jDeodorant identify some possible treat which they are analyzed and describe early. The defect prediction model generate thought Weka give us good result, but we must consider that we had a small data-set to work on and balancing it with `SMOTE` means that we work on a significant quantity of synthetic data. We would like to refine these result by increase the data-set, so waiting for other modification to jPacman repository and repeat the analisys with weka.