Universitá degli Studi di Salerno

Dipartimento di Informatica

Hands-on: a software dependability report of JPacman

Assignment components
Carmine D'Alessandro
Lucio Giordano
Simone Faiella

Anno Accademico 2018-2019

Code smells affecting the system

The analysis regards the search for code smells within JPacman and is performed using the JDeodorant tool. JDeodorant defines code smells with the name of bad smells and divides them into five different categories:

God Class. A class implementing several responsibilities, having a large number of attributes, operations and dependencies with data classes.

Duplicated Code. A sequence of source code that occurs more than once, either within a program or across different programs owned or maintained by the same entity.

Long Method. A method having a huge size.

Type Checking. Mainly manifested as complicated conditional statements that make the code difficult to understand and maintain.

Feature Envy. A method that is more interested in a class other than the one it actually is in.

The project was analyzed for packages in which we searched for these five categories of bad smells. After having found the bad smells, we proceeded with the proposal of a refactoring solution, examining the ones indicated by the tool taken into consideration.

Package nl.tudelft.jpacman

The first problem analyzed in this package is a feature envy problem, whose code snippet is represented there:

Listing 1: Feature envy 1a

```
1
2 private Player getSinglePlayer(final Game game) {
3         List<Player> players = game.getPlayers();
4         if (players.isEmpty()) {
5                 throw new IllegalArgumentException("Game has
                      0 players.");
6         }
7         return players.get(0);
8 }
```

Based on the aforementioned definition of feature envy, the best refactoring solution to solve the problem is to move the analyzed method into the class that best fits its behaviour.

In particular, analyzing the refactoring proposed by JDeodorant, it is appropriate to move the *"getSinglePlayer"* method to the Game class. The proposed solution allows to increase the cohesion of the Game class, as well as to decrease the coupling between the classes, thus, guaranteeing an improvement of the code.

Another problem that occurs in this package is the presence of a God Class. The Launcher class in fact has a multitude of responsibilities whose task could be divided into new classes.

It is obvious that the refactoring of a God Class can take place in different ways, but each with a common basis, i.e. the extraction of responsibilities that can make the class difficult to understand and maintain. In particular, JDeodorant offers two possible solutions.

The most appropriate solution to solve the problem is the creation of a LauncherProduct class, which acquires multiple responsibilities previously associated with the Launcher class. In particular, the creation of the level, the game, the player and the ghosts is associated with the new class, while the Launcher class will guarantee the layout of the graphic interface, as well as the actual start of the game.

Package nl.tudelft.jpacman.board

The analysis of this package focuses in particular on two classes: Square and Unit, both of which represent God Classes that can be separated, making their management easier.

Initially performing the analysis on the Square class, which represents the actual square that is occupied by the units, also presents a management of the elements present within the square. It is therefore possible to associate this task to a new class, which JDeodorant defines as SquareProduct. A similar evaluation can be performed for the Unit class, a new UnitProduct class will be correctly associated with the class.

Package nl.tudelft.jpacman.level

This package also contains type-checking bad smells, which can be distinguished in two cases.

If the conditional code fragment is an *if/else if* structure, the static attributes should be compared for equality with the type field (or an invocation of its getter method) in all conditional expressions. (Type checking 1a)

If the conditional code fragment is a switch statement, the type field (or an invocation of its getter method) should appear in the switch expression, while the static attributes representing the different values that the type field may obtain should appear in all case expressions.(Type checking 1b)

It should be noted that a switch/if statement should contain more than one case to be considered as a valid type-checking candidate since a single case is usually not regarded as a sign of possible future changes.

Listing 2: Type checking 1a

```
1
2  public void collide(Unit mover, Unit collidedOn) {
3          if (mover instanceof Player) {
4                  playerColliding((Player) mover, collidedOn);
5          }
6          else if (mover instanceof Ghost) {
7                  ghostColliding((Ghost) mover, collidedOn);
8          }
9          else if (mover instanceof Pellet) {
10                 pelletColliding((Pellet) mover, collidedOn);
11         }
12 }
```

Such a program involves the usage of conditional statements to simulate dynamic dispatch and late binding, instead of taking advantage of polymorphism.

Going into more detail, the proposed solution is to exploit the use of polymorphism, rather than an analysis using instanceof, thus resulting in a replace from Conditional to Polymorphism.

The type checking code fragment should be replaced with an invocation of the abstract method of the top level class through the superclass type reference.

Listing 3: Type checking 1b

```
1
2  switch (ghostIndex) {
3          case BLINKY:
4                  return ghostFact.createBlinky();
5          case INKY:
6                  return ghostFact.createInky();
7          case PINKY:
8                  return ghostFact.createPinky();
9          case CLYDE:
10                 return ghostFact.createClyde();
11         default:
12                 return new RandomGhost(sprites.getGhostSprite
                       (GhostColor.RED));
13 }
```

In this case (Type checking 1b) there is an opportunity for applying the "Replace Type Code with State/Strategy" refactoring. Particularly, the class containing the type field will play the Context role in the State/Strategy pattern . The conditional branches of the type checking code will be moved as separate methods to the subclasses of a newly created State/Strategy inheritance hierarchy.

Concerning the construction of the State/Strategy inheritance hierarchy, an abstract class should be created that will play the role of the State/Strategy. The name of the abstract class will be the name of the type field. An abstract method having the same name and return type with the method containing the typechecking code fragment should be added to the abstract class, while the number of the

concrete State/Strategy subclasses that should be created is equal to the number of the conditional branches inside the typechecking code. The names of the concrete subclasses will be the names of the static attributes corresponding to each case.

Finally, the type checking code fragment should be replaced with an invocation of the State/Strategy abstract method through the type field reference.

It should be emphasized that, through the analysis with JDeodorant, other types of code smell have also been identified in this package, in particular three of these refer to "Long method". The choice not to expose a solution is based on the evaluation that the latter, as structured within the program, do not seem to be real long methods, and the refactoring of the latter as proposed seems only to make the code analyzed more complex.

Package nl.tudelft.jpacman.ui

Finally, the package related to the user interface was analyzed, within which a God Class was identified, relative to the PacManUiBuilder class. The evaluation that has been carried out, in accordance with JDeodorant's analysis, is the possibility of separating this class, creating a so-called "PacManUiBuilderProduct" in order to reduce complexity and increase comprehensibility in reading it.

Analysis of possible vulnerabilities affecting the system

This part focuses on the analysis, through static analysis techniques, of vulnerabilities that may be present within JPacman.

The presence of multiple tools for static analysis of software is obviously known, in particular, before performing the analysis, PMD, SpotBugs and Checkstyle were evaluated. The main choice for the analysis was directed to the use of SpotBugs for two reasons in particular: the first is based on the study of the papers that distinguish the various analysis tools, showing how the analysis capacity of this tool is very high, it is highly efficient as it carries out both a Lexical analysis and a Dataflow Analysis. More specifically it works on the byte code.

The second reason is the possibility of being able to carry out a very thorough setting for each project, through which is possible to evaluate the categories of bugs to report, the minimum rank to be reported and so on.

More in detail, the categories of bugs that are provided in the settings are:

- Bad Practise
- Malicious code vulnerability
- Correctness
- Performance
- Security
- Dodgy code
- Multithreaded correctness
- Internationalization

The default settings provided by the tools include the analysis of Bad practice, Correctness, Performance, Dodgy code and Multithreaded correctness. However, whatever the "Minimum rank to report", no vulnerabilities are identified in the JPacman code with a "Minimum confidence of report" set to "Medium".

Consequently to this evaluation, it was decided to reduce the Minimum confidence of report to "Low", including all the possible categories of bugs, in order to be able to carry out a more in-depth analysis.

The results that have been obtained are relatively few, showing how the analyzed code, with the aforementioned settings, is not subject to dangerous vulnerabilities:

- For the *Internationalization* category, the only snippet of code that is considered "vulnerable" contains a bug defined as follows: *"Use of non-localized String.toUpperCase() or String.toLowerCase() in getGhostSprite(GhostColor). A String is being converted to upper or lowercase, using the platform's default encoding. This may result in improper conversions when used with international characters. Use the "String.toLowerCase(Locale.ROOT)" versions instead".*

- As for the *Performance* category, a method called *"getParent()"* is never called, within the Navigation class.

- Finally, three "bugs" are detected regarding the *BadPractise* category: *These Serializable classes defines a non-primitive instance field which is neither transient, Serializable, or java.lang.Object, and does not appear to implement the Externalizable interface or the readObject() and writeObject() methods. Objects of these classes will not be deserialized correctly if a non-Serializable object is stored in this field.*

As can be seen from the report of this analysis, the vulnerabilities present, which rank is 19 or 20 (considering a maximum of 20), are relatively of reduced confidence, their refactoring is intrinsic in the error itself, so they can be easily solved with some precautions.

For completeness of the work, the Checkstyle tool was finally used. As for the configuration, the Google style configuration it was that checks the Google coding conventions from Google Java Style that can be found at https://google.github.io/styleguide/javaguide.html. The resulting analysis is the absence of any vulnerabilities regarding the conventions provided by Google.

Repository Analysis

Product metrics extraction

After having evaluated the JPacman project using static analysis tools, we have used PyDriller, a python library, to mine the repository. The work has been divided into several phases:

- Extract the set of defect-fixing commits
  To find the set of defect-fixing commits we decided to mine the repository using PyDriller and look for all the commits including 'fix' in their message. After having extracted all the defect-fixing commits, we proceeded to find the defect inducing commits. We had to look for the commits whose lines had been modified in the fixing-ones.
  Following, part of the used code:

Listing 4: Buggy commits

```
1  #repo is the local link to JPacman
2  for commit in RepositoryMining(repo,
       only_modifications_with_file_types=['.java']).
       traverse_commits():
3  fix_commits = []
4  #find fixing commits
5      if 'fix' in commit.msg:
6          fix_commits.append(commit)
7  gr = GitRepository(repo)
8  #find defect-inducing commits
9  for fix_commit in fix_commits:
10     bug_commits = gr.get_commits_last_modified_lines(
           fix_commit)
```

  In this way, we can iterate over all the commits who induced the defects that have been fixed and work with them.

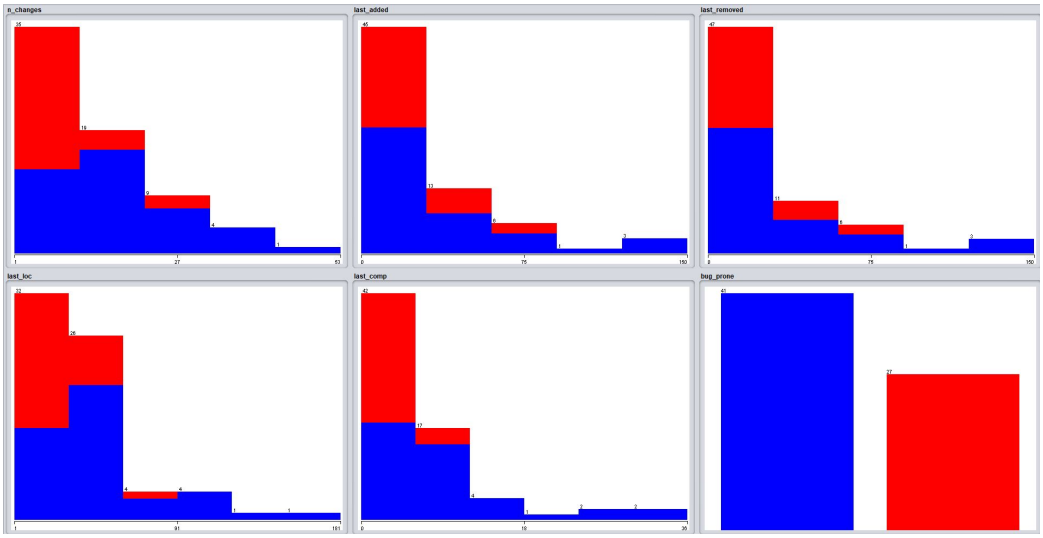
- Find defect prone and not defect prone classes
  After having identified the fixing commits, we have also been able to find the classes that have been modified in those commits and understand, then, whether a class is defect prone or not. In fact, we consider a class defect prone if it has been 'fixed' in a fixing commit, not defect prone, otherwise.

- Extract product metrics
  The last part of the extraction consists of creating a Machine Learning model for the defect prediction relating product metrics to defect-proneness.

A Metric is a measure of degree to which a system, component or process possesses a given attribute. We evaluated the project using product metrics, which are meant to measure the quality of the artifacts developed in the software development.
In this case, we have extracted the name of the class, the number of changes, the number of lines added and removed in the last commit, the last complexity and whether a class is bug prone or not, according to the output of the previous part. These product metrics extracted from the classes of the project are then given as input to Weka for the generation of the model.

Defect Prediction Model

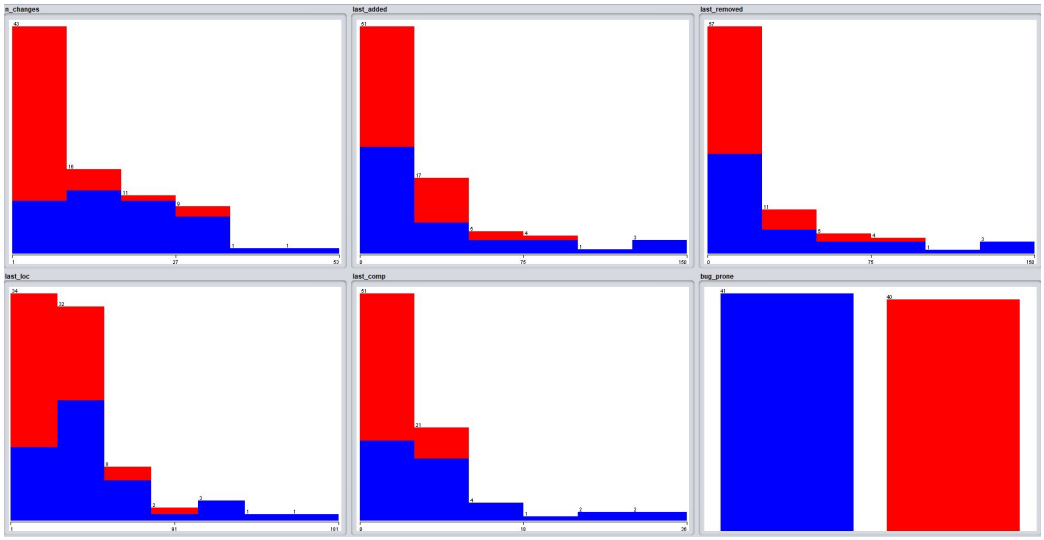
The last part of the work is about the definition of a model for the detection of defects using the product metrics previously extracted. It was decided to use the software Weka. It is a collection of machine learning algorithms for data mining tasks. It contains tools for data preparation, classification, regression, clustering, association rules mining, and visualization.
The unanalyzed data looks like this:



The interface of Weka makes the user able to perform many different preprocessing operations, including the review of the distribution of the attributes shown ahead. The original dataset in made of 68 different tuples and 7 attributes:

- class name (already removed)
- n changes: the number of changes in the defect fixing commit
- last removed: the number of removed lines in the defect fixing commit
- last loc: the number of current lines of code
- last comp: the current complexity of the class
- bug prone: true if the class is prone to bugs, false otherwise.

The first operation done was the adjustment of the class distribution of the dataset. The algorithm chosen for this purpose was SMOTE (Synthetic Minority Over-sampling Technique). The aim of this step is to increase the number of the "not defect prone"-labeled (in red) to have a better balancing of the dataset and make the model more capable of analyzing new data correctly.
The size of the tuples identified as "not defect prone" was so increased by 50% to match the number of the defect prone tuples, the dataset was then randomized to avoid incorrect behaviours in future steps:

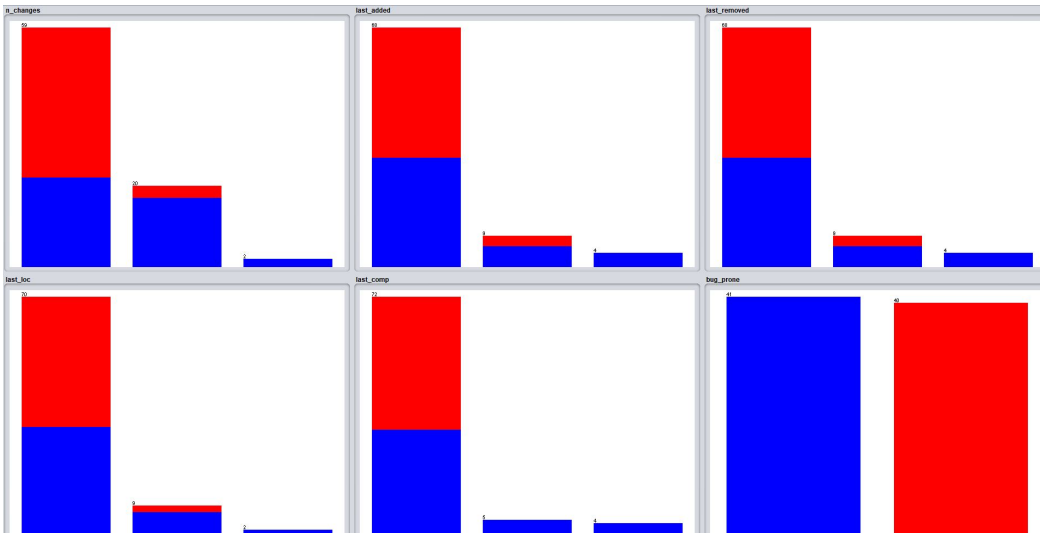


The classifier chosen to classify the data is "*Random forest*". It is an ensemble learning method for classification that operates by constructing several decision trees while training and giving in output their mode. It is an improvement of the decision tree strategy to improve their behaviour against overfitting.
A 10 folds cross validation was repeated 10 different times and in the end the average distribution was chosen as definitive model. To evaluate the goodness of the training were taken into consideration the accuracy of the validation, the recall, the confusion matrix and other parameters useful to estimate the performance of the model, as shown in the figure:

```
=== Summary ===

Correctly Classified Instances          62               76.5432 %
Incorrectly Classified Instances        19               23.4568 %
Kappa statistic                          0.5306
Mean absolute error                      0.3319
Root mean squared error                  0.4266
Relative absolute error                 66.3704 %
Root relative squared error             85.3173 %
Total Number of Instances               81

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure  MCC      ROC Area  PRC Area  Class
               0,780     0,250     0,762       0,780   0,771      0,531    0,799     0,801     1
               0,750     0,220     0,769       0,750   0,759      0,531    0,799     0,762     0
Weighted Avg.  0,765     0,235     0,766       0,765   0,765      0,531    0,799     0,782

=== Confusion Matrix ===

  a  b   <-- classified as
 32  9 |  a = 1
 10 30 |  b = 0
```

Discretized data

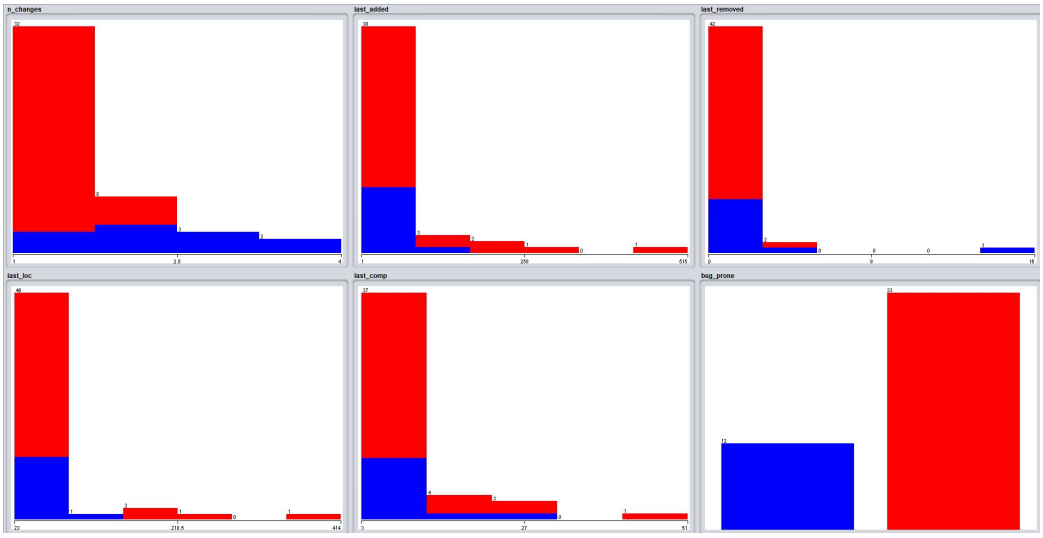
As possible other approach, we tried to discretize the data. As it could be seen, the data is distributed on a range of many different distinct values. The most of the times, the values are totally distinct and there are "holes" in the distribution. So, we discussed on a way to improve the input data and we thought discretization would improve the model. We tried to apply it and train again the model, obtaining these input data and these result of the training:



```
=== Summary ===

Correctly Classified Instances          61               75.3086 %
Incorrectly Classified Instances        20               24.6914 %
Kappa statistic                          0.5073
Mean absolute error                      0.3611
Root mean squared error                  0.428
Relative absolute error                 72.2175 %
Root relative squared error             85.5873 %
Total Number of Instances               81

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure  MCC      ROC Area  PRC Area  Class
               0,659     0,150     0,818       0,659   0,730      0,517    0,734     0,769     1
               0,850     0,341     0,708       0,850   0,773      0,517    0,734     0,634     0
Weighted Avg.  0,753     0,245     0,764       0,753   0,751      0,517    0,734     0,703

=== Confusion Matrix ===

  a  b   <-- classified as
 27 14 |  a = 1
  6 34 |  b = 0
```

Testing the model

As last part of the definition of the model, we wanted to evaluate it with totally new data. So, we repeated the extraction procedure of the metrics on another real project, getting this brand new data:



The data is extracted from an old project of the bachelor degree made before learning the basic GitHub good commit practices (For example some bug fixing commits also implementing functionalities). We could not access a better repository and this made the process of extraction of the test dataset very difficult.
However, we were very interested in testing the model on actual new data, so we decided to continue and extract the data we could trust.
This data was used to test the model with these results:

```
=== Evaluation on test set ===

Time taken to test model on supplied test set: 0.03 seconds

=== Summary ===

Correctly Classified Instances          28               62.2222 %
Incorrectly Classified Instances        17               37.7778 %
Kappa statistic                          0.0078
Mean absolute error                      0.4736
Root mean squared error                  0.5131
Relative absolute error                 94.1816 %
Root relative squared error            102.034  %
Total Number of Instances               45
```

These results are obtained with a discretization of the original data of the model:

```
=== Summary ===

Correctly Classified Instances          33               73.3333 %
Incorrectly Classified Instances        12               26.6667 %
Kappa statistic                          0
Mean absolute error                      0.4977
Root mean squared error                  0.4977
Relative absolute error                 98.9826 %
Root relative squared error             98.9808 %
Total Number of Instances               45
```

Results

As expected, the accuracy obtained during testing is inferior. This is due to the nature of testing and to the quality of the code from which we extracted test cases which also penalized this step.
On the other side, the model obtained with discretized values performs better in testing than the previous one. We think it might be due to the fact that the data used to train it is better prepared, so it is easier for the algorithm to identify the correct class.
Both the models were not able to identify correctly a subset of tuples. We think that this ones are placed in the wrong class by the worst practices of commit which made very difficult to identify them correctly.
As other result, we think this method of evaluation of defect proneness is good and can lead good results but only if the developer are aware of how their commits should be done, according to the metrics that will be extracted.