Università degli Studi di Salerno
Software Dependability
Prof. Fabbio Palomba
Report
JPacman Framework Vulnerabilities
Ruslana Lishchynska Aadm Ziro Abylay Salimzhanov

Abstract.
This work is associated with analyzing the code smells, vulnerabilities, defect prediction models for JPacman. In this work is represented a code smells affecting the system, using JDeodorant. In this report we will define God Classes, Feature Envies and Long Methods, because they often occur in the development process. Defining of possible vulnerabilities on basis of SonarLint. Describing extracted product metrics from project, with PyDriller. Defect prediction models defined using Weka, reporting the methodology of the model.

1. Detected Code Smells.

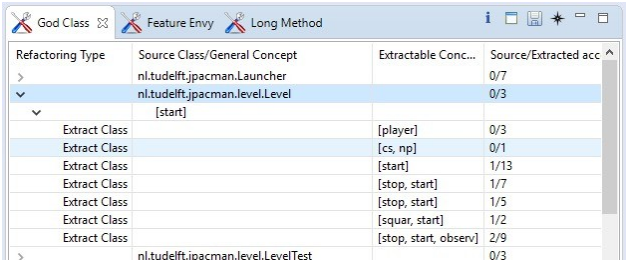
Blob - 9 files, refactoring - breaking code in more logical pieces - extract class refactoring.

Feature Envy - 6, refactoring - improving location of code - move method refactoring.

Long Method - 20, refactoring - extract method.

Type Checking - 2, refactoring - replace conditional with polymorphism, replace type code with state/strategy.

1.1. Blob(also named God Class) is a ”class implementing several responsibilities, having a large number of attributes, operations and dependencies with data class. In the process of detecting a Code Smells, there were found 9 God Classes, by the JDeodorant, there will be described one of them(Level.java). A class consists of the 348 lines of code, with too many methods, comments, sub-classes. By the analysis of the project, there were provided solutions for refactoring, by JDeodorant, class extraction. This method belongs to the refactoring operations as ”Breaking Code in More Logical Pieces”. Representing the solution for 'Level.java' class in JDeodorant:



1.2. Feature Envy Bad Smell is a method, which is more interested in a class other than the one actually is in. In the project were found 6 Feature Envies in different classes, the one that is described below is 'PlayerCollisions.java' Provided refactoring solution for "Feature Envy" bad smell in 'PlayerCollisions.java' is "move method", improving location of code to 'Player.java' class for 'ghostColliding' method. See the diagram below, which illustrates high

coupling between 'ghostColliding' and proposed class for moving:

1.3. Type Checking, when the different types used in the project dœsn't fully use the strategies given by the Object Oriented Programming, such as Inheritance and Abstract Classes. In the given project, JDeodorant detected 2 Type Checking code smells, and proposed different methods of refacoring. Possible ways for refactoring this code smell:

a) Replace Conditional with Polymorphism

```
@Override
public void collide(Unit mover, Unit collidedOn) {
    if (mover instanceof Player) {
        playerColliding((Player) mover, collidedOn);
    }
    else if (mover instanceof Ghost) {
        ghostColliding((Ghost) mover, collidedOn);
    }
    else if (mover instanceof Pellet) {
        pelletColliding((Pellet) mover, collidedOn);
    }
}
```

b) Replace Type Code with State/Strategy

```
Ghost createGhost() {
    ghostIndex++;
    ghostIndex %= GHOSTS;
    switch (ghostIndex) {
        case BLINKY:
            return ghostFact.createBlinky();
        case INKY:
            return ghostFact.createInky();
        case PINKY:
            return ghostFact.createPinky();
        case CLYDE:
            return ghostFact.createClyde();
        default:
            return new RandomGhost(sprites.getGhostSprite(GhostColor.RED));
    }
}
```

2. Analysis of possible vulnerabilities

2.1. SonarLint

Vulnerabilities. JPacman was analyzed using SonarLint for finding possible vulnerabilities. SonarLint highlights code issues with markers on open files. It also provides an issues summary table for a selected component in the IDE, including the creation time of the issue. In the given project was detected 45 possible code issues/vulnerabilities. Let's take a look on some of them:

a) Unused private "getParent" method. Unused methods don't seem to cause vulnerabilities, but if to look closer, it is clear that such methods can cause many issues in the future development process, for example, for new workers these methods create confusion and lead to waste of time on a project, because they not only have to understand the working code, they have to understand unused material also. There is a danger that at sometime someone will make a change which involves wrong assumptions about unused code that can introduce bugs and, of course, it increases the confusion, potential misunderstanding and administrative overhead.

b) Replace this assert with a proper check. Can be a problem for execution phase. Not proper check can exploit vulnerabilities and can lead to the crushing of software. Need to be reviewed.

c) Make "class" serializable or transient. This warning is not actually vulnerability, more like purpose for future development, because if developer made class not serializable, it means that he/she intended to do it from the beginning.

d) Parameters to "addHandler" have the same names but not the same order as the method arguments. Not vulnerability in this case, because arguments have the same type, this detected code issue can be a problem in case, when arguments has completely different type(String and int). In case with this function wrong assumption about vulnerability, Java can deal with this problem through overloading. Overloading methods offers no specific benefit to the JVM but it is useful to the programmer to have several methods do the same things but with different parameters.

e) Complete the task associated with TODO comment. Only suggestions for future development in case with JPacman.

3. Defect Prediction Model

3.1. Pydriller is a wrapper around GitPython that eases the extraction

of information from Git repositories. The most significant difference between the two tools is that GitPython offers many features (almost all the features of Git), while PyDriller offers only features
that are important when performing MSR tasks, thus hiding the
underlying complexity to the end user. In this section, we explain
the design of Pydriller, as well as its main APIs.

3.2. Weka 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.

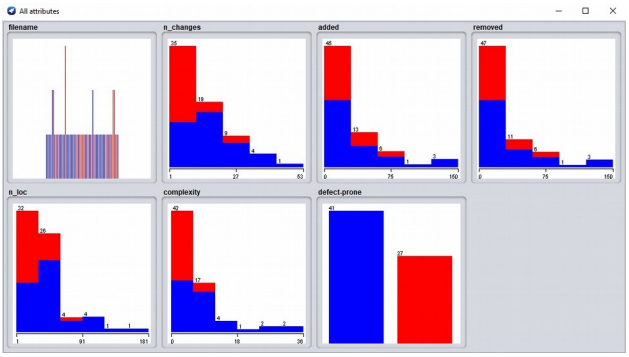
3.3. Results

All – 68
Good – 27
Defect – 41
For classification have been chosen logistic regression. Logistic regression is the classification counterpart to linear regression. Predictions are mapped to be between 0 and 1 through the logistic function, which means that predictions can be interpreted as class probabilities.

filename	numchanges	lineadded	lineremoved	linesofcode	complexity	isBuggy
Launcher.java	53	2	8	104	20	True
Board.java	32	1	5	35	12	True
Square.java	20	1	4	45	10	True
Unit.java	26	3	3	51	13	True
Game.java	15	80	80	52	12	True
Level.java	1	9	0	5	0	True
Player.java	4	50	5	41	8	False
Sprite.java	8	43	43	8	0	True
Action.java	5	6	6	4	0	False
BoardPanel.java	9	101	101	51	7	True
ButtonPanel.java	7	23	23	21	2	False
PacKeyListener.java	7	33	33	25	5	True
PacManUI.java	19	1	3	50	4	True
PacManUiBuilder.java	20	1	2	58	9	True
ScorePanel.java	18	76	76	47	7	True
Direction.java	10	56	56	19	3	True

Outputs have a nice probabilistic interpretation, and the algorithm can be regularized to avoid overfitting. Logistic models can be updated easily with new data using stochastic gradient descent.

```
=== Summary ===

Correctly Classified Instances          49               72.0588 %
Incorrectly Classified Instances        19               27.9412 %
Kappa statistic                          0.4412
Mean absolute error                      0.2562
Root mean squared error                  0.4842
Relative absolute error                 53.3745 %
Root relative squared error             98.8373 %
Total Number of Instances               68

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure  MCC      ROC Area  PRC Area  Class
                 0,683     0,222     0,824      0,683     0,747     0,451     0,831     0,891     True
                 0,778     0,317     0,618      0,778     0,689     0,451     0,831     0,762     False
Weighted Avg.    0,721     0,260     0,742      0,721     0,724     0,451     0,831     0,840

=== Confusion Matrix ===

  a  b   <-- classified as
 28 13 |  a = True
  6 21 |  b = False
```



Random Forest is a flexible, easy to use machine learning algorithm that produces, even without hyper-parameter tuning, a great result most of the time.
Classification was made by two classification algorithms, Logistic Regression and Random Forest. Results seems to be nearly the same. As we can see, from not high precision and recall, predicted model is not so accurate. We couldn't reach good results, because of the small dataset and the initial partitioning on classes wasn't made properly, because we relied on the commits of developers.

4. Conclusion

All the possible code smells and vulnerabilities were detected using different tools, but some of them were analyzed wrong. As we can see from the quantity of code smells and vulnerabilities, and not accurate predicted model JPacman is not very dependable project and need to be enhanced by the developers community.