Software Dependability Report: A JPacman Framework Analysis

Mauro Borrazzo, Luigi D'Arco, Giulia Sellitto
Software Dependability Course, Università degli Studi di Salerno, April-May 2019

1 Introduction

JPacman Framework is a game developed at a software testing course at Delft University of Technology [1]. It is now mainly used as an exercise for those who want to learn testing on a non-trivial project.
In this paper we expose our own work on the framework. We managed to analyze the code in order to understand its weaknesses in terms of software dependability metrics.
We focused our attention onto three main tasks:

1) Detect any code smells affecting the system. We used static analysis tools such as *JDeodorant* and *SpotBugs*, both available as plug-ins into *Eclipse* IDE. Details on this phase are given in section 2.
2) Study the possible vulnerabilities of the code. We used *PMD* plug-in to detect defects in the code that could lead to any system failure. This phase is explained in section 3.
3) Define a defect predicition model basing on product metrics and historical information about the project. We used *PyDriller* and *Radon* to extract relevant data from the project and then *R* and *Weka* to build our model. Details are given in section 4.

2 Bad Code Smells

2.1 Blob

A *Blob*, also known as *God Class*, is a class implementing several responsibilities, having a large number of attributes, operations and dependencies with data classes.
An example of this bad code smell can be found in **nl.tundelft.jpacman.level.Level.java**. This class is affected by *God Class* smell because it has a high *LOC*, exactly 376 lines of code, and it has several responsibilities such as the registration of a player, the movement of a NPC (Non-Player-Character), checking for presence of any alive player, and it also measures the collision between units.
We analyzed the methods *registerPlayer* and *isAnyPlayerAlive*. This two methods use other classes that are not correlated to *Level* class.
In Listing 1 we show the method *registerPlayer*. It is called every time a player is added to the game. But this method doesn't have any correlation with the functionality of its class.

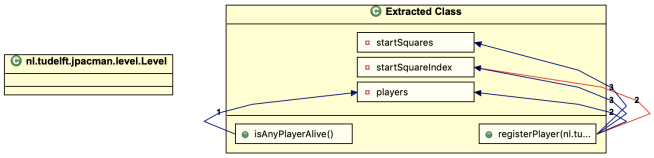


Fig. 1. Extract Method Refactoring as proposed by *JDeodorant*.

In the same way the method *isAnyPlayerAlive*, shown in Listing 2, doesn't have any correlation with the functionality of *Level* because it checks if there is still an alive player.

```
public void registerPlayer(Player player) {
  assert player != null;
  assert !startSquares.isEmpty();
  if (players.contains(player)) {
    return;
  }
  players.add(player);
  Square square=startSquares.get(startSquareIndex);
  player.occupy(square);
  startSquareIndex++;
  startSquareIndex %= startSquares.size();
}
```

Listing 1. Level.registerPlayer

```
public boolean isAnyPlayerAlive() {
  for (Player player : players) {
    if (player.isAlive()) {
      return true;
    }
  }
  return false;
}
```

Listing 2. Level.isAnyPlayerAlive

These methods can either be moved to another class or be splitted into a new class.

In Figure 1 we report three possible ways of performing **Extract Method Refactoring**, as suggested by *JDeodorant*. Different options are pointed out by the arrows labels.
We figured out that the methods can be extracted and moved to the *nl.tudelft.jpacman.level.Player* class, which better represents the concept they are interested into. Actually we suggest to perform a **Move Method Refactoring** operation, applicable when a method is more interested in a class rather than in the one it actually is in.

2.2 Feature Envy

THE *Feature Envy* code smell is generated when a method is more interested in another class rather than in the one it actually belongs to.
The class **nl.tundelft.jpacman.level.MapParser.java** seems to give us several examples of *Feature Envy* smell. Its responsibility is to build new *JPacman* levels basing on their textual representation. In order to do that, this class offers some methods that are more interested in other concepts. With a closer look at the source code of the class, we noticed that *MapParser* method *addSquare*, shown in Listing 3, often calls other *nl.tudelft.jpacman.board.BoardFactory.java* methods. The *makeGhostSquare* method, shown in Listing 4, operates on *nl.tudelft.jpacman.npc.Ghosts* objects and also has side effect such as the modification of a list. In order to solve these bad smells, we can simplify these methods in a more *Object Oriented* way, so each method is responsible to perform one and only task.

```
protected void addSquare(Square[][] grid, List<Ghost
    > ghosts,
List<Square> startPositions, int x, int y, char c) {
  switch (c) {
    case ' ':
      grid[x][y] = boardCreator.createGround();
      break;
    case '#':
      grid[x][y] = boardCreator.createWall();
      break;
    case '.':
      Square pelletSquare = boardCreator.
    createGround();
      grid[x][y] = pelletSquare;
      levelCreator.createPellet().occupy(
    pelletSquare);
      break;
    case 'G':
      Square ghostSquare = makeGhostSquare(ghosts,
    levelCreator.createGhost());
      grid[x][y] = ghostSquare;
      break;
    case 'P':
      Square playerSquare = boardCreator.
    createGround();
      grid[x][y] = playerSquare;
      startPositions.add(playerSquare);
      break;
    default:
      throw new PacmanConfigurationException("
    Invalid character at " + x + "," + y + ": " + c)
    ;
  }
}
```

Listing 3. MapFactory.addSquare

```
protected Square makeGhostSquare(List<Ghost> ghosts,
     Ghost ghost) {
  Square ghostSquare = boardCreator.createGround();
  ghosts.add(ghost);
  ghost.occupy(ghostSquare);
  return ghostSquare;
}
```

Listing 4. MapParser.makeGhostSquare

2.3 Functional Decomposition

THIS term describes a class where inheritance and polymorphism are poorly used, declaring many fields and implementing few methods.
By analyzing the code we have found a class that seems to be affected by *Functional Decomposition* smell, which implements few methods, in each of them polymorphism is poorly used, avoided with the use of some *if-else* statements.
The considered class is **nl.tudelft.jpacman.level.PlayerCollision.java**. The first method we have analyzed is *collide*, shown in Listing 5, which checks which type of character is colliding onto any other else and, basing on the types of the characters, it calls another method of the same class to complete actions such as kill the player, add points, and so on.

```
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);
  }
}
```

Listing 5. PlayerCollision.collide

This method uses a lot of *if-else* statements. This could be avoided by using **polymorphism**.
Same thing happens with the *playerColliding* method, shown in Listing 6, which is invoked when needed by the previous one and, as the previous one, it performs a check on the type of the object *nl.tudelft.jpacman.board.Unit*, checking if it is a *nl.tudelft.jpacman.npc.Ghost* specialization or a *nl.tudelft.jpacman.level.Pellet* specialization. By this, it calls a method of the same class rather than another, that means already to method calls performed.

```
private void playerColliding(Player player, Unit
    collidedOn) {
  if (collidedOn instanceof Ghost) {
    playerVersusGhost(player, (Ghost) collidedOn);
  }
  if (collidedOn instanceof Pellet) {
    playerVersusPellet(player, (Pellet) collidedOn);
  }
}
```

Listing 6. PlayerCollision.playerColliding

As said before, the use of **polymorphism** is a better choice to avoid all these controls.
Finally, there are the last two methods ending the set of methods calls. With these two methods, shown in Listings 7 and 8 there is the final choice for the right action to do. As the previous ones, there isn't any use of polymorphism, instead there are *if-else* statements, which represent a very poor choice.

```
private void ghostColliding(Ghost ghost, Unit
    collidedOn) {
  if (collidedOn instanceof Player) {
    playerVersusGhost((Player) collidedOn, ghost);
  }
}
```

Listing 7. PlayerCollision.ghostColliding

```
private void pelletColliding(Pellet pellet, Unit
    collidedOn) {
  if (collidedOn instanceof Player) {
    playerVersusPellet((Player) collidedOn, pellet);
  }
}
```

Listing 8. PlayerCollision.pelletColliding

As a solution we can improve the adherence to *Object Oriented Programming* principles by replacing conditionals with **polymorphism**.

2.4 Long Method

A *Long Method* is a method with a huge size. Through a static analysis of the code we have found a method in the **nl.tudelft.jpacman.LauncherSmokeTest.java** class that seems to be a long method because it has a high *LOC*, and it is shown in Listing 9.

```
void smokeTest() throws InterruptedException {
  Game game = launcher.getGame();
  Player player = game.getPlayers().get(0);

  // start cleanly.
  assertThat(game.isInProgress()).isFalse();
  game.start();
  assertThat(game.isInProgress()).isTrue();
  assertThat(player.getScore()).isZero();

  // get points
  game.move(player, Direction.EAST);
  assertThat(player.getScore()).isEqualTo(10);

  // now moving back does not change the score
  game.move(player, Direction.WEST);
  assertThat(player.getScore()).isEqualTo(10);

  // try to move as far as we can
  move(game, Direction.EAST, 7);
  assertThat(player.getScore()).isEqualTo(60);

  // move towards the monsters
  move(game, Direction.NORTH, 6);
  assertThat(player.getScore()).isEqualTo(120);

  // no more points to earn here.
  move(game, Direction.WEST, 2);
  assertThat(player.getScore()).isEqualTo(120);

  move(game, Direction.NORTH, 2);

  // Sleeping in tests is generally a bad idea.
  // Here we do it just to let the monsters move.
  Thread.sleep(500L);

  // we're close to monsters, this will get us
    killed.
  move(game, Direction.WEST, 10);
  move(game, Direction.EAST, 10);
  assertThat(player.isAlive()).isFalse();

  game.stop();
  assertThat(game.isInProgress()).isFalse();
}
```

Listing 9. LauncherSmokeTest.smokeTest

Why to refactor? As we can see it has a lot of lines of code, and it hasn't got a specific goal. An **Extract Method Refactoring** can solve this problem, because the more lines found in a method, the harder it is to figure out what the method does. This is the main reason for this refactoring. However, this is a test method and it can be considered not a problem to the project.

2.5 Type Code

TYPE Code occurs when we have a set of numbers or strings that form a list of allowable values for some entity, instead of a separate data type. In this specific case, we have found a *Type Code* smell in **nl.tudelft.jpacman.level.LevelFactory.java** class, into the method *createGhost*, which is reported in Listing 10.

```
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));
  }
}
```

Listing 10. LevelFactory.createGhost

In this case, the *ghostIndex* variable can assume four different values that are specified by four constants, in order to give them more understable names. This type of code is usually encountered and used and it affects system behaviour. In this case, the aim is to create different ghosts in such a way that the types are balanced. We could replace this code with a random generation.

3 VULNERABILITIES

CODE *Vulnerabilities* expose the program to several errors and malfunctionings. In order to find any vulnerability in *JPacman* project, we used *PMD*. It is an automatic tool for performing static analysis on source code. By feeding the project to *PMD*, we found out the following vulnerabilities:

- Several classes do not declare a *serializationUid*. This is a bad programming practice in Java, because we may want to send data in and out of the program and not declaring a proper *serializationUid* can lead into *Deserialization Exceptions*.
  In particular, *JPacman* now uses text files to store data about the level map and how it should be showed. In upcoming versions we may want to store the map object rather than its textual representation, in order to avoid the parsing phase, which is quite complex and generates *Feature Envy* bad smell too, as seen before.
- There is a massive use of *copy-and-paste* between classes in the *nl.tudelft.jpacman.npc.ghost* package. The *nextAiMove* method is implemented in similar ways in both *Blinky, Clyde, Inky* and *Pinky* classes. As

shown in Listings 11, 12, 13 and 14, they actually perform the same tasks and the method could be moved to the *Ghost* parent class, with some variations.

```
public Optional<Direction> nextAiMove() {
  assert hasSquare();

  Unit nearest = Navigation.findNearest(Player.class
    , getSquare());
  if (nearest == null) {
    return Optional.empty();
  }
  assert nearest.hasSquare();
  Square target = nearest.getSquare();

  List<Direction> path = Navigation.shortestPath(
    getSquare(), target, this);
  if (path != null && !path.isEmpty()) {
    return Optional.ofNullable(path.get(0));
  }
  return Optional.empty();
}
```

Listing 11. Blinky.nextAiMove

```
public Optional<Direction> nextAiMove() {
  assert hasSquare();

  Unit nearest = Navigation.findNearest(Player.class
    , getSquare());
  if (nearest == null) {
    return Optional.empty();
  }
  assert nearest.hasSquare();
  Square target = nearest.getSquare();

  List<Direction> path = Navigation.shortestPath(
    getSquare(), target, this);
  if (path != null && !path.isEmpty()) {
    Direction direction = path.get(0);
    if (path.size() <= SHYNESS) {
      return Optional.ofNullable(OPPOSITES.get(
    direction));
    }
    return Optional.of(direction);
  }
  return Optional.empty();
}
```

Listing 12. Clyde.nextAiMove

```
public Optional<Direction> nextAiMove() {
  assert hasSquare();
  Unit blinky = Navigation.findNearest(Blinky.class,
     getSquare());
  Unit player = Navigation.findNearest(Player.class,
     getSquare());

  if (blinky == null || player == null) {
    return Optional.empty();
  }

  assert player.hasSquare();
  Square playerDestination = player.squaresAheadOf(
    SQUARES_AHEAD);

  List<Direction> firstHalf = Navigation.
    shortestPath(blinky.getSquare(),
  playerDestination, null);

  if (firstHalf == null) {
    return Optional.empty();
  }

  Square destination = followPath(firstHalf,
    playerDestination);
  List<Direction> path = Navigation.shortestPath(
    getSquare(),
  destination, this);

  if (path != null && !path.isEmpty()) {
    return Optional.ofNullable(path.get(0));
  }
  return Optional.empty();
}
```

Listing 13. Inky.nextAiMove

```
public Optional<Direction> nextAiMove() {
  assert hasSquare();

  Unit player = Navigation.findNearest(Player.class,
     getSquare());
  if (player == null) {
    return Optional.empty();
  }
  assert player.hasSquare();
  Square destination = player.squaresAheadOf(
    SQUARES_AHEAD);

  List<Direction> path = Navigation.shortestPath(
    getSquare(), destination, this);
  if (path != null && !path.isEmpty()) {
    return Optional.ofNullable(path.get(0));
  }
  return Optional.empty();
}
```

Listing 14. Pinky.nextAiMove

4 Product Metric Analysis

Long software may contains several bugs, which can be dangerous for the system itself and for the users.In order to improve code quality without adding or changing functionalities of the software, discovering these bugs and correcting them is needed. These operations need to spend money, effort and time to be performed, however we can use a machine learner that predicts if a part of a given software is *buggy* or not.

Before using a machine learner we need to train the model with a dataset. The dataset can be created using different metrics, such as structural, complexity, process, developer-based and textual. These metrics are extracted by using *Python* programming language, with the support of a dedicated library named *PyDriller*, as shown in Listing 15. We analyzed all the commits in the *JPacman* repository and every time we found a commit labeled with the string *fix* all files inside it have been categorized as *buggy* classes.

```
def calculate_metrics(bugs):
  files = {}
  authors = {}

    for commit in RepositoryMining(repo).
    traverse_commits():
      for mod in commit.modifications:
        noc = 0
        if mod.filename.endswith('.java') and mod.
    change_type is not ModificationType.DELETE:
    buggy = True if commit.hash in bugs else False

        try:
          anal = analyze(mod.source_code)
          lloc = anal.lloc
          blank = anal.blank
          sloc = anal.sloc
```

```
        except SyntaxError as inst:
          print(inst.args)

        source = str(mod.source_code)
        if source.find('extends') > 0:
          noc = noc+1

        num_of_bugs = 4.86 + 0.019 * mod.nloc
        process_metrics = {'change': mod.change_type
    , 'num_of_bugs': num_of_bugs,'added': mod.added,
     'removed': mod.removed, 'lloc': lloc, 'sloc':
    sloc, 'blank': blank, 'noc': noc, 'loc': mod.
    nloc, 'comp': mod.complexity, 'buggy': buggy}
        path = mod.new_path
        if path not in files:
          files[path] = []
        files.get(path, []).append(process_metrics)
        if path not in authors:
          authors[path] = {}
        authors[path][commit.author.email] = 1
  return files, authors
```

Listing 15. Metrics extraction.

The extracted metrics are:

- n-changes (Number of Changes)
- num_of_bugs (Number of Bugs)
- added (Number of row added)
- removed (Number of row added)
- lloc (Logical Lines of Code)
- sloc (Source Lines of Code)
- blank (Number of blank lines)
- noc (Number of Children)
- loc (Lines of Code)
- complexity (Cyclomatic complexity)
- num_of_authors (Number of authors)
- buggy (bool: Class contains a bug)

To train a machine learner *R* programming language was used.
The first thing to do is to import the dataset into the *R* environment, as shown in Listing 16.

```
# Importing the dataset
dataset <- read.csv("../output.csv", header=TRUE,
    sep=",")
```

Listing 16. Importing the dataset.

Then we need to apply a data clean operation to improve the quality of the dataset. We applied a *Recursive Feature Elimination* algorithm (RFE). Its goal is removing non-relevant features by recursively consider and evaluate smaller and smaller sets of features. The result of this operation tells us that num_of_authors is not a useful feature, so then it will be removed from the dataset.

```
# Removing non-relevant features from the dataset
    using the Recursive Feature Elimination
    algorithm (RFE):
#it has the goal of remove non-relevant features by
    recursively consider and evaluate smaller and
    smaller sets of features.
control <- rfeControl(functions=rfFuncs, method="cv"
    , number=10)
# Run the RFE algorithm
results <- rfe(dataset[,1:11], dataset[,12], sizes=c
    (1:12), rfeControl=control)

results
```

```
# Rebuilding the dataset - Dropping the non-relevant
    features
drops <- c("num_of_authors")
dataset <- dataset[ , !(names(dataset) %in% drops)]
```

Listing 17. Dataset cleaning.

In Listing 18 we show how we used the *trainControl* function to define the training options. Our model has been assessed by using the *10-Fold Cross validation* strategy, where the data is randomly partitioned in ten folds and nine of them are used as training set, while one is retained as test set. The same process is then repeated ten times, so that each fold is used as test once. However a problem occurs when using this strategy, because the process may lead to over-estimate or under-estimate the real performance of a model, it all depends on the initial random splitting. In order to solve this problem, the validation process is performed multiple times, so that the random effect is mitigated.
We also used the *Smote* algorithm, which creates synthetic instances of the minority class using statistical methods. Since our classifier hasn't got enough data to classify the response variable, we used *Data Balancing* in order to distort the relative proportions in the data to better aid model building and to train models that are better capable of addressing the underlying problem.

```
# Defining the training options - A repeated 10-fold
    cross validation will be applied
train_control<- trainControl(method="repeatedcv",
    number=10, search="grid", sampling="smote",
    repeats=10, savePredictions = TRUE)
```

Listing 18. Training and buolding the model.

Finally we have to train the model through the *train* function. This function evaluates the effect of model tuning parameters on performance, chooses the optimal model across these parameters, and estimates model performance from a training set. The parameter method with value *rf* specifies that we are using the *Random Forest* algorithm.

```
# Building the model - We will use a Random Forest
    classifier (rf)
simple_model <- train(buggy~., dataset, trControl=
    train_control, method="rf")
```

Listing 19. Training and buolding the model.

In the final Listing 20 the entire output of the computation is reported and, as a final result, the accuracy of the model is showed.

```
summary(dataset$buggy)
False  True
62     6
# Removing non-relevant features from the dataset
    using the Recursive Feature Elimination
    algorithm (RFE):
control <- rfeControl(functions=rfFuncs, method="cv"
    , number=10)
# Run the RFE algorithm
results <- rfe(dataset[,1:11], dataset[,12], sizes=c
    (1:12), rfeControl=control)
results

Recursive feature selection
```

```
Outer resampling method: Cross-Validated (10 fold)

Resampling performance over subset size:

Variables Accuracy  Kappa AccuracySD KappaSD
    Selected
1    0.9714 0.7647    0.06023  0.4092
2    0.9714 0.7647    0.06023  0.4092
3    0.9714 0.7647    0.06023  0.4092
4    0.9714 0.7647    0.06023  0.4092
5    0.9714 0.7647    0.06023  0.4092
6    0.9714 0.7647    0.06023  0.4092
7    0.9714 0.7647    0.06023  0.4092
8    0.9857 0.8333    0.04518  0.4082        *
9    0.9714 0.7647    0.06023  0.4092
10   0.9714 0.7647    0.06023  0.4092
11   0.9714 0.7647    0.06023  0.4092

The top 5 variables (out of 8):
n.changes, num_of_authors, lloc, loc, num_of_bugs

# Rebuilding the dataset - Dropping the non-relevant
    features
drops <- c("num_of_authors")
dataset <- dataset[ , !(names(dataset) %in% drops)]
# Defining the training options - A repeated 10-fold
    cross validation will be applied
train_control<- trainControl(method="repeatedcv",
    number=10, search="grid", sampling="smote",
    repeats=10, savePredictions = TRUE)
# Building the model - We will use a Random Forest
    classifier (rf)
simple_model <- train(buggy~., dataset, trControl=
    train_control, method="rf")
There were 50 or more warnings (use warnings() to
    see the first 50)

# Training accuracy
simple_model
Random Forest

68 samples
10 predictors
2 classes: 'False', 'True'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 10
    times)
Summary of sample sizes: 61, 62, 62, 61, 60, 61, ...
Addtional sampling using SMOTE

Resampling results across tuning parameters:

mtry  Accuracy   Kappa
2     0.9136243  0.11263721
6     0.9047619  0.16320856
10    0.9138804  0.09803922

Accuracy was used to select the optimal model using
    the largest value.
The final value used for the model was mtry = 10.
```

Listing 20. Summary.

5 Conclusion

In conclusion, we can say that *JPacman* is a *quite-buggy* project, and this can be resulting by the fact that it is quite big. In our work, we focused our attention on what we have covered in our *Software Dependability* course at *Università degli Studi di Salerno*, Italy [2], but there is always room for further investigation and deeper study on this project. Our next goal is to learn new topics in the *Software Dependability* field and apply them on *JPacman* as an exercise.

References

[1] *JPacman Framework Project on GitHub:*
https://github.com/SERG-Delft/jpacman-framework

[2] *Software Dependability Course Site on GitHub:*
https://github.com/fpalomba/SWDependability-Unisa2019

Further Analysis on JPacman Framework

1 Introduction

In this additional document we discuss the results of an additional analysis we performed on JPacman Framework project.
Our analysis was made using the *CodeMR* Software Quality Tool. *CodeMr* is an architectural quality and static code analysis tool that helps software companies in developing better code and delivering better products.
CodeMR computes software metrics and high-level quality attributes such as coupling, cohesion, complexity, size. It shows projects in different views, such as *Package Structure*, *TreeMap*, *Sunburst*, *Dependency* and *Graph Views*. In the following, we will show some examples of these views of JPacman project.

2 Analysis

We computed several metrics in JPacman project by using the tool described previously.
In particular, we analyzed:

- Total lines of code: 1241
- Number of classes: 50
- Number of packages: 8
- Number of external packages: 27
- Number of external classes: 175

The first step of our analysis was discovering the distribution of values of quality attributes among classes.
CodeMR generated donut charts in which dark green represents a good value of the considered metric, light green means a quite good value, yellow represents a medium value, orange represents a quite bad value of the metric and red means a bad value.
In Figure 1 we can notice that only about 5% of classes have a medium level of coupling. The remaining have a low and low-medium value of coupling.
Figure 2 shows cohesion metric values distribution. Half of the classes have a good cohesion, but nearly 10% of classes have a quite bad cohesion, that is, they have a high *Lack of Cohesion of Methods*.
In Figure 3 it is shown that half of the classes have a medium-low value of cyclomatic complexity metric. The majority of the remaining classes have a low complexity and fewer than 10% of classes have a medium complexity.
Figure 4 shows size of classes. This chart is quite encouraging, because there are no yellow classes, that means that there are no extremely big classes.
In the following sections we will discuss the analysis done, with special attention to problems found and we will provide some solutions to those problems.

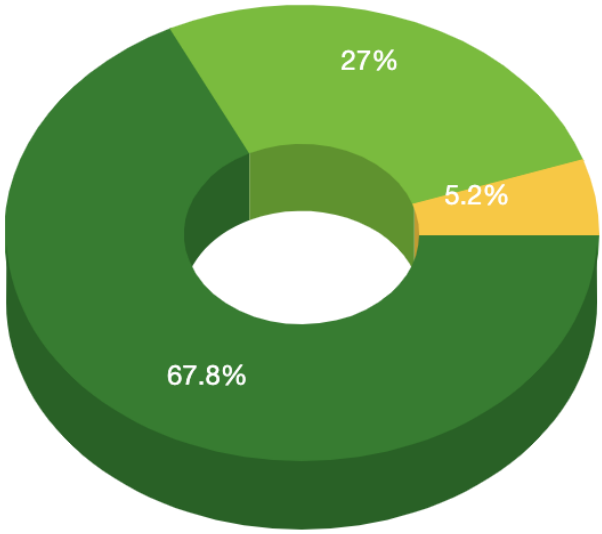


Fig. 1. Distribution of Coupling metric values.

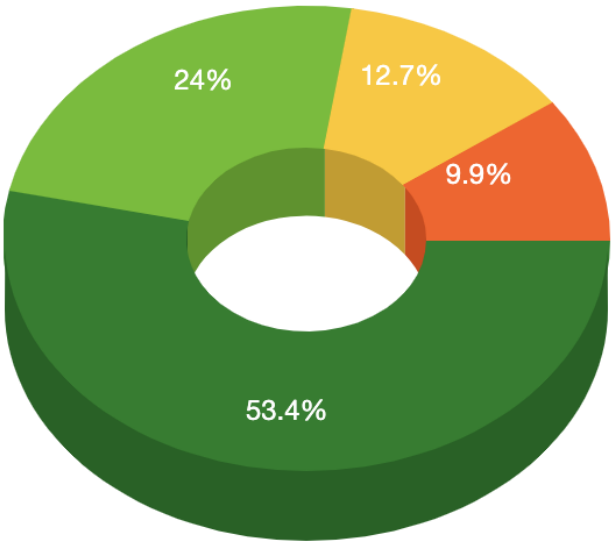


Fig. 2. Distribution of Cohesion metric values.

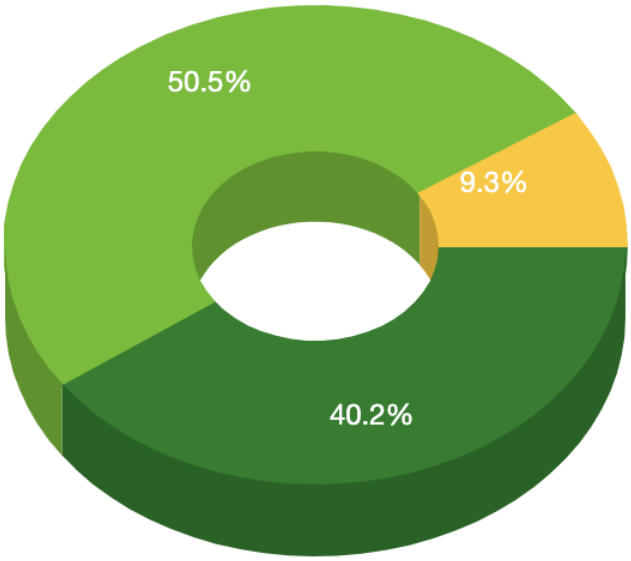


Fig. 3. Distribution of Cyclomatic Complexity metric values.

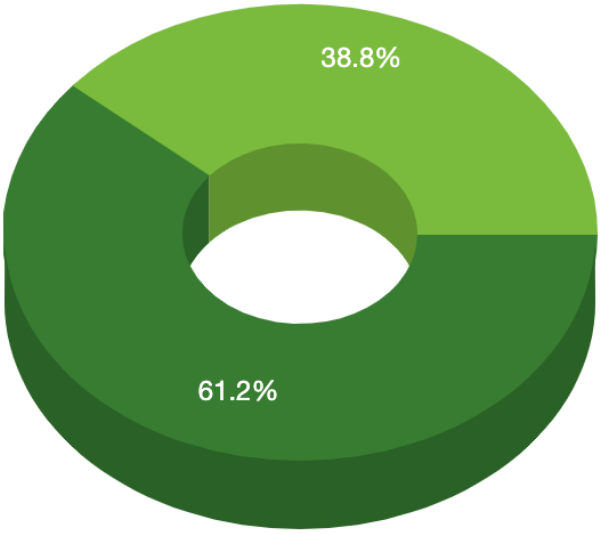


Fig. 4. Distribution of Size metric values.

3 Dependencies between Packages

As we can see in Figure 5 there is an high level of coupling between packages. This leads to several problems:

- a change in one class usually forces a ripple effect of changes in other classes
- maintenance requires more effort and/or time due to the increased dependency level
- it might be harder to reuse a class because dependent classes must be included.

The solution to this problem is to apply some refactoring operations, in particular location of code can be improved in such a way that similar functionalities are located in the same package. This operation can be performed by changing the class location, namely it can be called **Move Class Refactoring**.

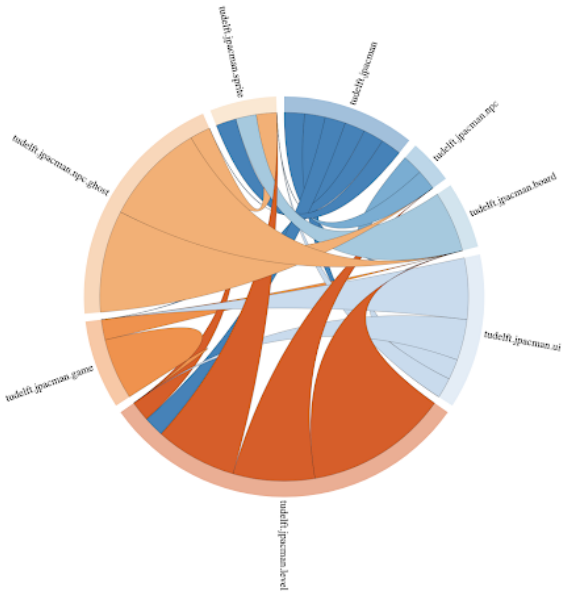


Fig. 5. Coupling between packages.

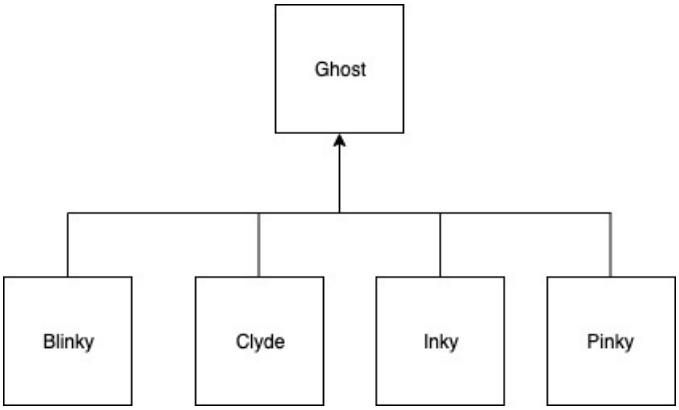
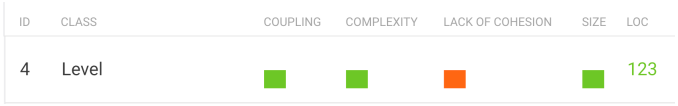


Fig. 6. Ghost class hierarchy tree.

In this example we focus on the *Ghost.java* class inside the package **nl.tudelft.jpacman.npc**, but the same operation can be performed in the same way onto other classes in the packages. As we can see in Figure 6, the *Ghost* class can be considered as the root class of all the ghosts of the game. The majority of classes in the package **nl.tudelft.jpacman.npc.ghost** extend this class. A solution can be applied by moving the class *Ghost* to the package **nl.tudelft.jpacman.npc.ghost** in such a way that coupling between packages is reduced.

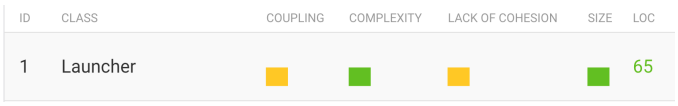
4 Maintainability of Classes

From our analysis it emerged that some classes in JPacman are poorly maintainable because of high coupling, high cyclomatic complexity or low cohesion. Some examples of such classes are discussed in the following.
The class *Level.java*, as shown in Figure 7, turned out to be poorly maintainable because of the high value of the *Lack of Cohesion of Methods* metric. This is due to the fact, as we have discussed in the bad smells detection part of the document, that *Level* class is a **Blob** class. This means



ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC
4	Level					123

Fig. 7. Metrics values for the class Level.



ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC
1	Launcher					65

Fig. 8. Metrics values for the class Launcher.

that it implements a lot of functionalities and it has a lot of dependencies with other data classes. All of this lows down the internal cohesion of the class and increases *LCOM* metric value. It is useful to detect a refactor opportunity, because high cohesion is desiderable, and our additional analysis confirmed that the best refactoring operation for this class is **Move Method Refactoring**.

Another example of a problematic class is *Launcher.java*, whose metrics are shown in Figure 8. It presents different weaknesses in terms of maintainability, due to its high coupling and low cohesion. So there is another refactoring opportunity for us to increase the software maintainability.
The Launcher class has two main problems that need to be solved in order to increase the quality of the code. In particular the class has high coupling with other classes: as we can see in Listing 1, this class uses several other classes.

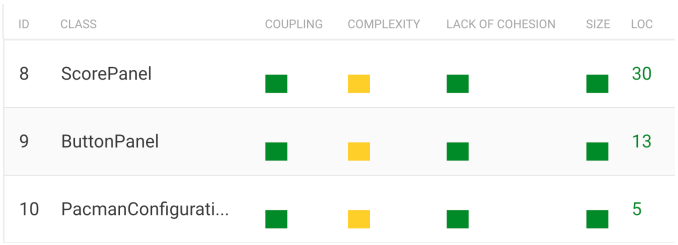
```
import nl.tudelft.jpacman.board.BoardFactory;
import nl.tudelft.jpacman.board.Direction;
import nl.tudelft.jpacman.game.Game;
import nl.tudelft.jpacman.game.GameFactory;
import nl.tudelft.jpacman.level.Level;
import nl.tudelft.jpacman.level.LevelFactory;
import nl.tudelft.jpacman.level.MapParser;
import nl.tudelft.jpacman.level.Player;
import nl.tudelft.jpacman.level.PlayerFactory;
import nl.tudelft.jpacman.npc.ghost.GhostFactory;
import nl.tudelft.jpacman.sprite.PacManSprites;
import nl.tudelft.jpacman.ui.Action;
import nl.tudelft.jpacman.ui.PacManUI;
import nl.tudelft.jpacman.ui.PacManUiBuilder;
```

Listing 1. Launcher

There are methods that increase the coupling of the class, because they instantiate and use functionalities of classes which are located in other packages. Furthermore, this contributes to make the class cohesion lower, because the *Launcher* class implements a lot of functionalities that are not correlated to each other.
Also in this case, a **Move Method Refactoring** can increase code quality because methods like the ones shown in Listings 2 and 3 are more interested in other classes rather than in the one they actually are in. This kind of refactoring operation aims to decrease coupling and increase cohesion in *Launcher* class, that is the main goal to increase code quality and then code maintainability.

```
protected GhostFactory getGhostFactory() {
return new GhostFactory(getSpriteStore());
}
```

Listing 2. Launcher.getGhostFactory



ID	CLASS	COUPLING	COMPLEXITY	LACK OF COHESION	SIZE	LOC
8	ScorePanel					30
9	ButtonPanel					13
10	PacmanConfigurati...					5

Fig. 9. Further metrics on classes.

```
public Level makeLevel() {
  try {
    return getMapParser().parseMap(getLevelMap());
  } catch (IOException e) {
    throw new PacmanConfigurationException("Unable
    to create level, name = " + getLevelMap(), e);
  }
}
```

Listing 3. Launcher.makeLevel

In Figure 9 we can see that we have further different problems. The *ScorePanel.java*, *ButtonPanel.java* and *PacmanConfigurationException.java* classes have high complexity. This is another issue that affects the maintainability of the system, and that can be eliminated using refactoring operations.
Let us focus our attention on the class *ScorePanel.java*. This class has a high value of cyclomatic complexity metric. There are a few operation that we can apply, one of these is to substitute the *for* cycle shown in Listing 4 which uses an iterator with a *for* cycle that uses index. This does not reduce the cyclomatic complexity of this class but improves the performance of the application. In fact, indexes are 23-40% faster than iterators.

```
for (Player player : players) {
  JLabel scoreLabel = new JLabel("0",JLabel.CENTER);
  scoreLabels.put(player, scoreLabel);
  add(scoreLabel);
}
```

Listing 4. ScorePanel cycle that uses iterator

In Listing 5 we propose a code replacement in order to solve this issue.

```
for (int i=0; i<players.size(); i++) {
  JLabel scoreLabel = new JLabel("0",JLabel.CENTER);
  scoreLabels.put(players.get(i), scoreLabel);
  add(scoreLabel);
}
```

Listing 5. Proposed solution for ScorePanel

5 Conclusion

From this additional analysis we learnt the usage of a new automatic tool for metrics computation and deeperly studied JPacman project, as we proposed to do in the first part of the document. We can say that we are fully satisfied by our work.