Software Dependability Report

JPacman Framework

SOFTWARE DEPENDABILITY # UNISA 2019

23 aprile 2019

Authors: Orlando Napoli, Giovanni Cammarano, Salvatore Arienzo, Armando Ferrara

Software Dependability Report

JPacman Framework

Index

1. Report overview

JPacman is a Pacman-like game used for teaching software testing. Parts of the code are well tested, while others are left untested intentionally. This software dependability report has been realized for the Software Dependability course taught in UniSa in 2019. The report is based on the latest version of JPacman, 8.1.1. In the first chapter of the report, there is a short introduction about the work and about the tested software, in the second chapter there is an overview of the code smells affecting the system, shown by JDeodorant. In the third chapter there are some possible vulnerabilities affecting the system, detected with some static analysis tools. In the fourth chapter will be defined a defect prediction model based on the commit history of the project extracted with PyDriller. In the fifth and last chapter we will see CodeCity, a tool in which the classes of the projects are described as buildings.

2. Code Smells affecting the system

2.1 JPacman overview

JPacman is intentionally affected by lots of code smell. How we can discover and fix them? With a (J)Deodorant of course! JDeodorant is an Eclipse plug-in that identifies design problems and try to resolve them by applying appropriate refactorings. For JPacman, JDeodorant shows A LOT of code smell situated in A LOT classes. Let we see some examples:

Refactoring Type	Source Method	Variable Criterion
▶	nl.tudelft.jpacman.level.LevelTest::void registerThirdPlayer()	p3
▶	nl.tudelft.jpacman.ui.ScorePanel::protected void refresh()	score
▶	nl.tudelft.jpacman.level.LevelTest::void registerSecondPlayer()	p2
▶	nl.tudelft.jpacman.level.Player::public void setAlive(boolean)	deathSprite
▶	nl.tudelft.jpacman.sprite.SpriteStore::public nl.tudelft.jpacman.sprite.AnimatedSp...	animation
▶	nl.tudelft.jpacman.level.LevelTest::void registerPlayerTwice()	p
▶	nl.tudelft.jpacman.sprite.PacManSprites::private Map<nl.tudelft.jpacman.board.D...	animation
▶	nl.tudelft.jpacman.LauncherSmokeTest::public static void move(nl.tudelft.jpacma...	player
▶	nl.tudelft.jpacman.board.BoardFactoryTest::void setUp()	factory
▶	nl.tudelft.jpacman.level.Level::public void registerPlayer(nl.tudelft.jpacman.level....	player
▶	nl.tudelft.jpacman.level.LevelTest::void startStop()	level
▶	nl.tudelft.jpacman.level.LevelTest::void registerThirdPlayer()	level
▶	nl.tudelft.jpacman.level.LevelTest::void registerThirdPlayer()	p1
▶	nl.tudelft.jpacman.level.LevelTest::void registerThirdPlayer()	p2
▶	nl.tudelft.jpacman.level.LevelTest::void registerSecondPlayer()	level
▶	nl.tudelft.jpacman.level.LevelTest::void registerSecondPlayer()	p1
▶	nl.tudelft.jpacman.level.LevelTest::void registerPlayerTwice()	level
▶	nl.tudelft.jpacman.level.LevelTest::void registerPlayer()	level
▶	nl.tudelft.jpacman.level.MapParser::protected nl.tudelft.jpacman.board.Square m...	ghost
▶	nl.tudelft.jpacman.LauncherSmokeTest::void smokeTest() throws java.lang.Interru...	player

Long Method code smell

As we can see for long method code smell there are many results, for most of them the suggested refactoring type is to extract the method.

Refactoring Type	Source Method	Variable Criterion	Block-Based Region	Duplicated/Extracted
▼	nl.tudelft.jpacman.level.LevelTest::void registerThirdPlayer()	p3		
Extract Method			B1	0/6

There are some methods that should be moved because they cause a Feature Envy code smell:

Refactoring Type	Source Entity	Target Class
Move Method	nl.tudelft.jpacman.level.PlayerCollisions::playerVersusPellet(nl.tudelft.jpacman.level.Player, nl.tudelft.jpacman.level.Pellet):void	nl.tudelft.jpacman.level.Pellet
Move Method	nl.tudelft.jpacman.ui.BoardPanel::render(nl.tudelft.jpacman.board.Square, java.awt.Graphics, int, int, int, int):void	nl.tudelft.jpacman.board.Square
Move Method	nl.tudelft.jpacman.Launcher::getSinglePlayer(nl.tudelft.jpacman.game.Game):nl.tudelft.jpacman.level.Player	nl.tudelft.jpacman.game.Game
Move Method	nl.tudelft.jpacman.level.PlayerCollisions::ghostColliding(nl.tudelft.jpacman.npc.Ghost, nl.tudelft.jpacman.board.Unit):void	nl.tudelft.jpacman.level.Player
Move Method	nl.tudelft.jpacman.level.PlayerCollisions::playerVersusPellet(nl.tudelft.jpacman.level.Player, nl.tudelft.jpacman.level.Pellet):void	nl.tudelft.jpacman.level.Player
Move Method	nl.tudelft.jpacman.ui.BoardPanel::render(nl.tudelft.jpacman.board.Board, java.awt.Graphics, java.awt.Dimension):void	nl.tudelft.jpacman.board.Board

Feature Envy code smell

Some other code smells are about the inheritance hierarchy and some variables declared as constant:

Refactoring Type	Type Checking Method	Abstract Method Name
▼	inheritance hierarchy: [nl.tudelft.jpacman.board.Unit]	
Replace Conditional with Polymorphism	nl.tudelft.jpacman.level.PlayerCollisions::public void collide(nl.tudelft.jpacman.board.Unit, nl.t...	collide
▼	constant variables: [BLINKY, INKY, PINKY, CLYDE]	
Replace Type Code with State/Strategy	nl.tudelft.jpacman.level.LevelFactory::nl.tudelft.jpacman.npc.Ghost createGhost()	createGhost

Type Checking code smell

2.2 The God Class

We will focus about the God Class, called also, Blob! What is a Blob?

No, we're not talking about a monster that

incorporates everything it meets. (Or maybe yes?)

A blob class is a class that implements lots of responsabilities,

lots of attribute, lots of operations, in short, lots of problems.

We decided to focus on the Blobs, because of all the code smell reported by JDeodorant, we believe that the Blobs are the ones that, more than anyone else, condition the design, the reliability, and the readability of the code. We will see the proposed solution by JDeodorant and another solution proposed by us that uses a design pattern to improve the structure of the code.

JDeodorant shows many different possible blobs:

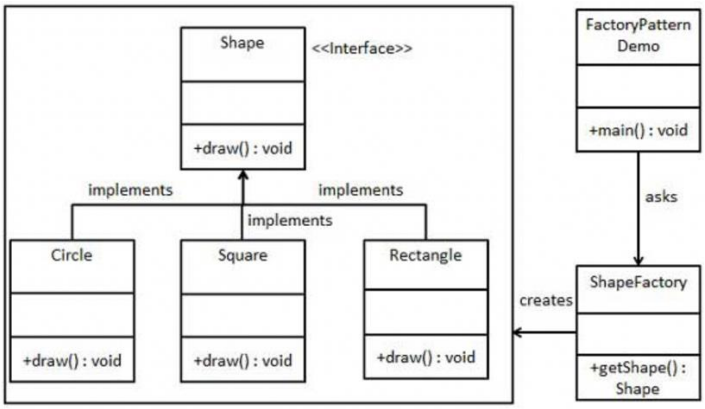
Refactoring Type	Source Class/General Concept	Extractable Concept
▼	nl.tudelft.jpacman.Launcher	
▼	[factori]	
Extract Class		[factori]
Extract Class		[factori]
▼	[pac, ui, dispos, main, man]	
Extract Class		[pac, ui, dispos, main, man]
▼	nl.tudelft.jpacman.level.Level	
▼	[start]	
Extract Class		[player]
Extract Class		[cs, np]
Extract Class		[start]
Extract Class		[stop, start]
Extract Class		[stop, start]
Extract Class		[squar, start]
Extract Class		[stop, start, observ]
▶	nl.tudelft.jpacman.level.LevelTest	
▶	nl.tudelft.jpacman.board.Unit	
▶	nl.tudelft.jpacman.npc.Ghost	
▶	nl.tudelft.jpacman.board.BoardTest	
▶	nl.tudelft.jpacman.board.Square	
▶	nl.tudelft.jpacman.ui.PacManUiBuilder	
▶	nl.tudelft.jpacman.level.MapParser	

We will analyze the first two: Launcher class and Level class.

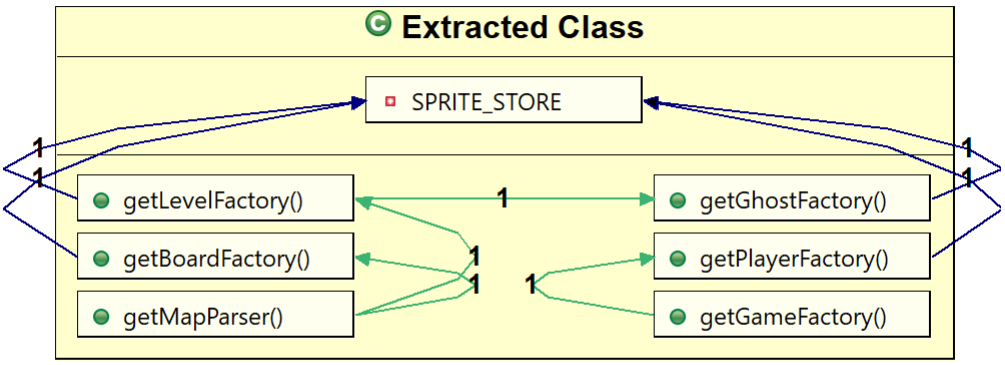
2.2.1 Launcher.java

The first one, the Launcher class, contains a lot of methods that build the UI, the Levels, the Map, the Ghosts, etc and can be considered a blob also in a future perspective. JDeodorant suggest to split this class into many classes. For instance each time a Factory object is needed, like ghostFactory, a getGhostFactory() method will be instantiated. getGhostFactory will create a ghostFactory object and will return it. This operation tends to increase as the difficulty of the program increases, for instance, let's suppose that in a future others objects, or enemies should be added beyond the ghosts: every time we have to go to add a method in the classes that use this object. The code will become unmanageable.

Our solution is to apply the Factory design pattern that allows all the classes they have such object need to not have to create an ad hoc method every time, allowing a better abstraction and more readability of the code. This is the suggested design pattern:



While JDeodorant suggests to extract this class:



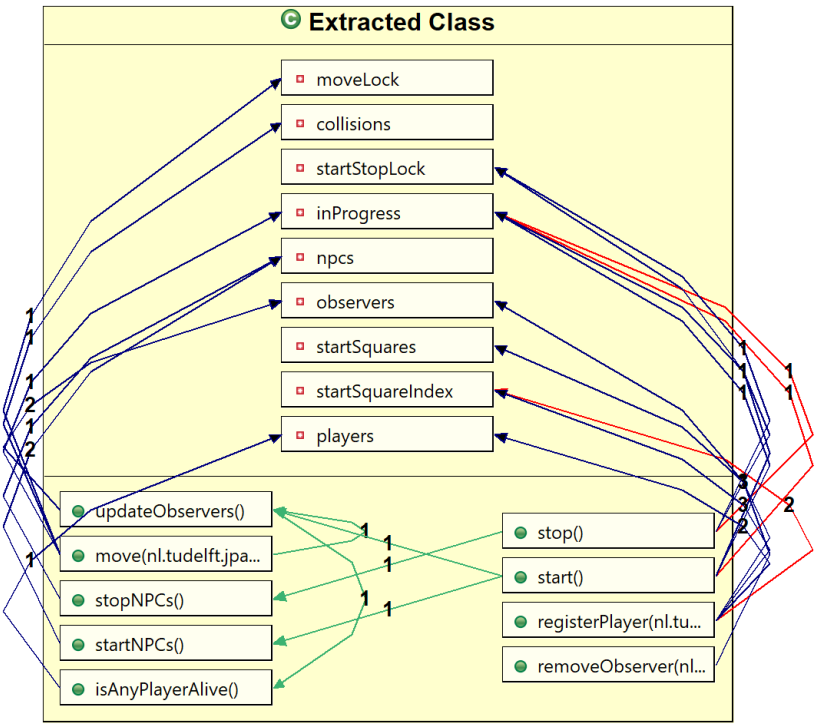
Anyway we suppose that adopting a design pattern is the best solution to the problem, because it is not the only class in which there is this problem (we also know that would not be possible for an automated tool to detect a design pattern, because it is a NP-COMPLETE problem).

Pros of applying the design pattern:

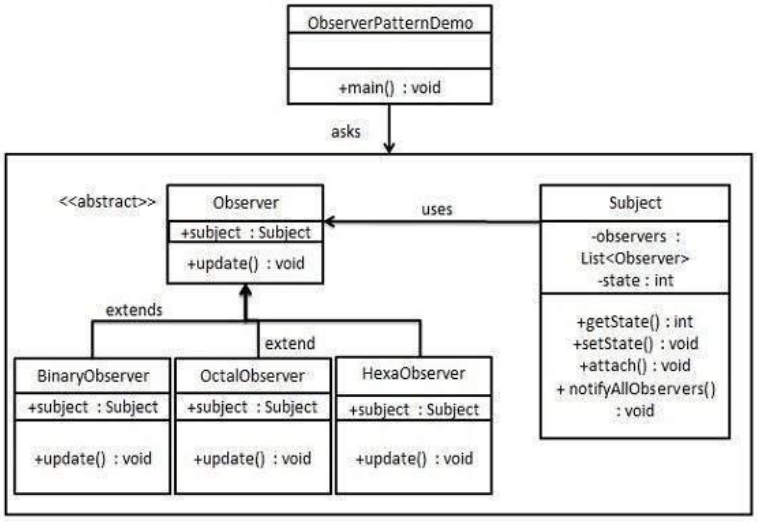
1. Loose coupling, which helps in changing the application design more readily. The application is separated from a family of classes.
2. It makes the application more customizable.

2.2.2 Level.java

The Level class is another huge class (with 400+ lines of code) and that implements lots of responsabilities, inherent to the changements of the states during the game. We notice a LevelObserver interface instantiated in the class. This interface can be a problem if we decide to add more observer. JDeodorant suggests to extract the class in this way:



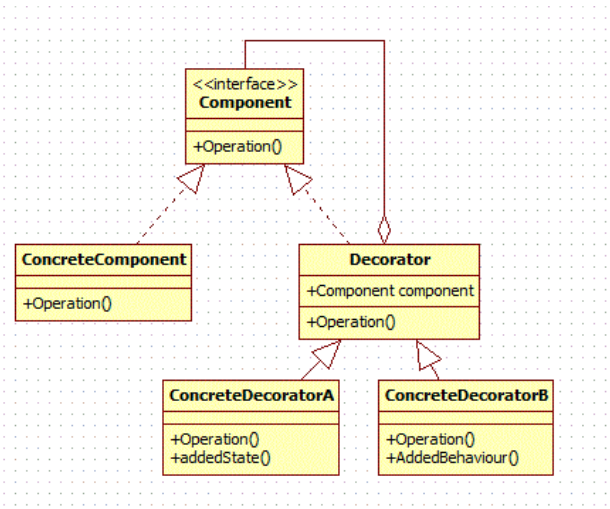
Otherwise, we suggest to adopt the Observer design pattern to improve the abstraction and the readability of the code.



- It supports the principle of loose coupling between objects that interact with each other
- It allows sending data to other objects effectively without any change in the Subject or Observer classes
- Observers can be added/removed at any point in time

2.2 Final considerations

An observation (or suggestion) that is not reported by the JDeodorant tool, is the absence of a concrete design pattern. In order to future improvements, more levels, more objects, more enemies, the code will become unmanageable and with low scalability, so to improve the organization of the pacman.ui package we recommend to use the Decorator design pattern, in order to:



- It is flexible than inheritance because inheritance adds responsibility at compile time but decorator pattern adds at run time.

- We can have any number of decorators and also in any order.
- It extends functionality of object without affecting any other object.

3. Vulnerabilities affecting the system

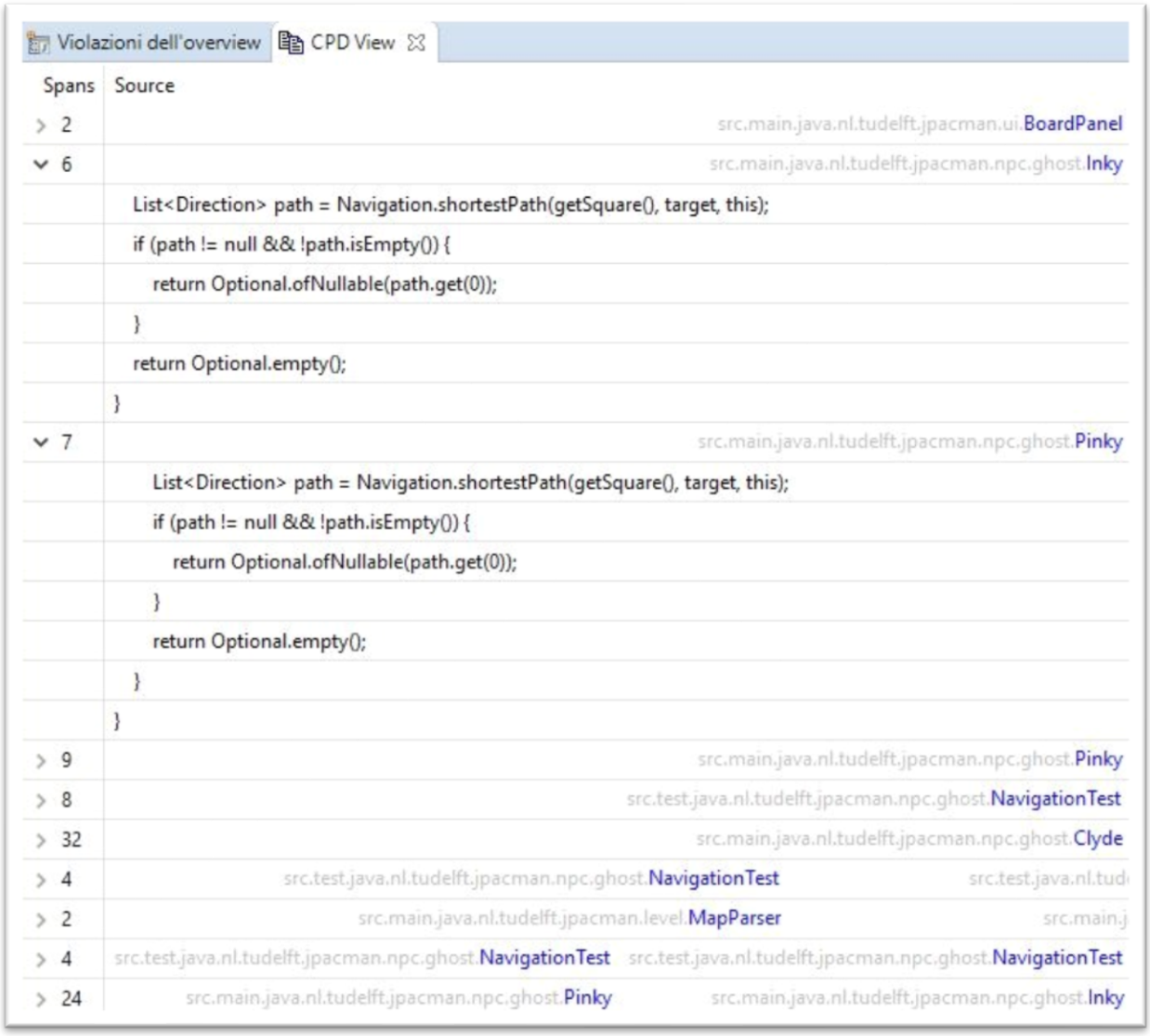
A vulnerability is an error, made by the developer that can be used to provoke a failure of the system.

We used some static analysis tool to analyze the code of the project, like **CheckStyle** that execute a lexical analysis of the code and check if the code respects standards and conventions of the community. For JPacman, CheckStyle shows only some minor format problems like the usage of the tab character.

Details of Checkstyle violation "Line contains a tab character." - 45 occurrences

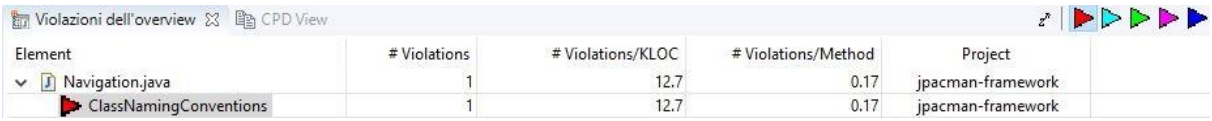
Resource	In Folder	Line	Message
apple.png	/jpacman-framework/src/main/resources/sprite	4	File Tab Character: Line contains a tab character.
bell.png	/jpacman-framework/src/main/resources/sprite	4	File Tab Character: Line contains a tab character.
cherry.png	/jpacman-framework/src/main/resources/sprite	4	File Tab Character: Line contains a tab character.

Another tool, **PMD**, shows all the duplicated lines of code. Generating a Copy-Paste report, we can see that in JPacman there are a lot of duplicate lines. Why should we care about duplicates? Assuming duplicated blocks of code are supposed to do the same thing, any refactoring, even simple, must be duplicated too, and this means more work for the developer. Now, who want to work more of the necessary? Not us! So,if the code may never change in the future, then this will be not a problem. I'll simply have 2-3 or more equal blocks that do the same thing. Otherwise, if the code could change, I should refactor the same block, in the same way, many times. For jPacman, there are several copy-paste detection:

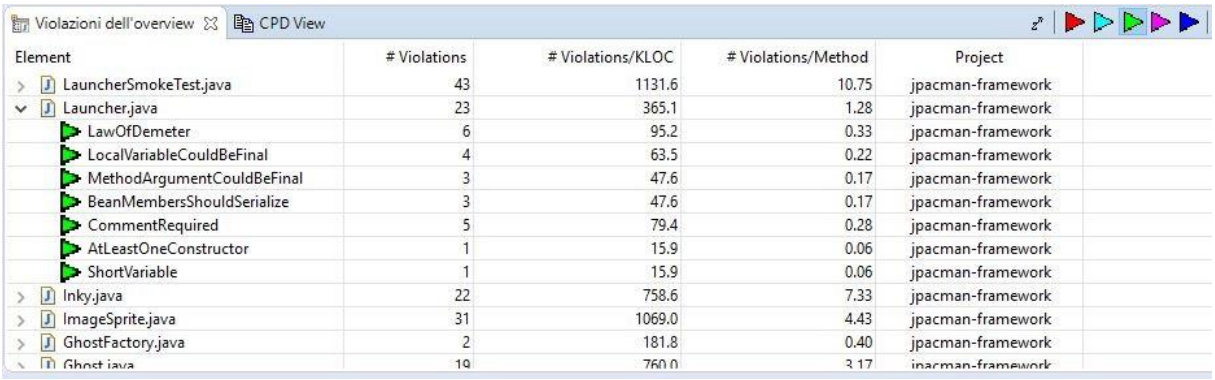


Once we have located some duplicates, we can apply several refactoring strategies. If the duplication is local to a method or single class, we can extract a local variable if the duplicated logic is not prohibitively long, into a private method. If the duplication occurs in many classes, we can use the template method design pattern.

PMD, show also some other violation, and while no critical violations are reported, there is a Blocker violation in the class Navigation because the class naming convention is not respected.

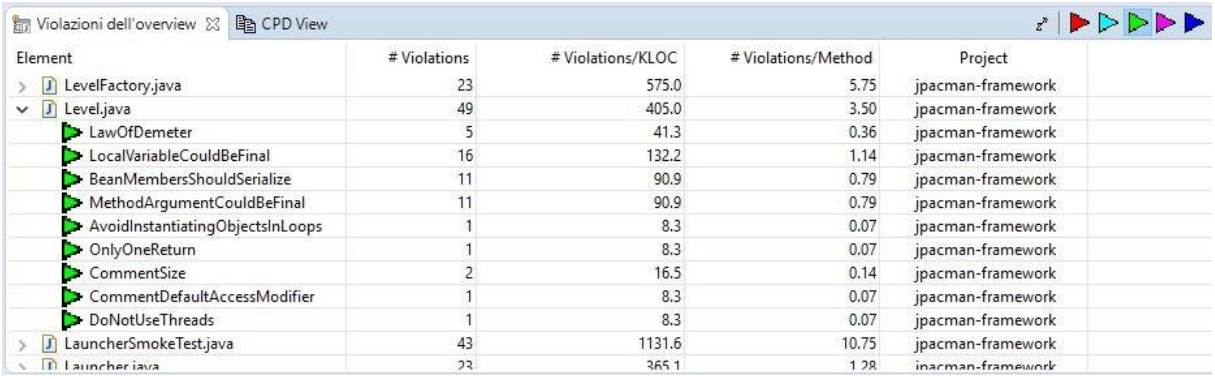


There are many Urgent violation in the most of classes, let we se the violation reported by PMD, for the blob class Launcher and Level that we saw in the previous chapter.



Violazioni dell'overview | CPD View

Element	# Violations	# Violations/KLOC	# Violations/Method	Project
LauncherSmokeTest.java	43	1131.6	10.75	jpacman-framework
Launcher.java	23	365.1	1.28	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
BeanMembersShouldSerialize	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
Inky.java	22	758.6	7.33	jpacman-framework
ImageSprite.java	31	1069.0	4.43	jpacman-framework
GhostFactory.java	2	181.8	0.40	jpacman-framework
Ghost.java	19	760.0	3.17	jpacman-framework



Violazioni dell'overview | CPD View

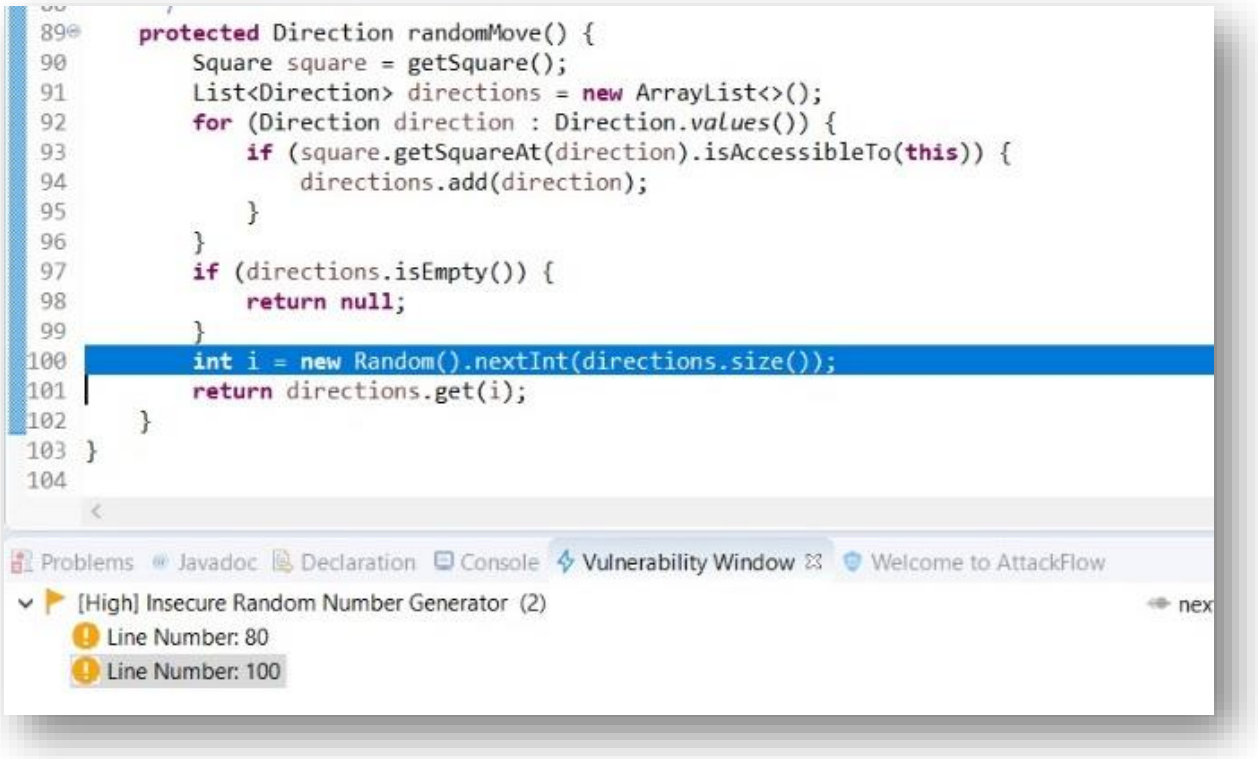
Element	# Violations	# Violations/KLOC	# Violations/Method	Project
LevelFactory.java	23	575.0	5.75	jpacman-framework
Level.java	49	405.0	3.50	jpacman-framework
LawOfDemeter	5	41.3	0.36	jpacman-framework
LocalVariableCouldBeFinal	16	132.2	1.14	jpacman-framework
BeanMembersShouldSerialize	11	90.9	0.79	jpacman-framework
MethodArgumentCouldBeFinal	11	90.9	0.79	jpacman-framework
AvoidInstantiatingObjectsInLoops	1	8.3	0.07	jpacman-framework
OnlyOneReturn	1	8.3	0.07	jpacman-framework
CommentSize	2	16.5	0.14	jpacman-framework
CommentDefaultAccessModifier	1	8.3	0.07	jpacman-framework
DoNotUseThreads	1	8.3	0.07	jpacman-framework
LauncherSmokeTest.java	43	1131.6	10.75	jpacman-framework
Launcher.java	23	365.1	1.28	jpacman-framework

As we can see there are a lot of possible violations, like methods that have more than one return points and some other minor violation like missing comments (or too large comment size). Both classes are potentially violating the LoD, Law of Demter that is a design guideline to loose coupling that can be summarized with:

- Each unit should have only limited knowledge about other units:
  only units "closely" related to the current unit.
- Each unit should only talk to its friends; don't talk to strangers.
- Only talk to your immediate friends.

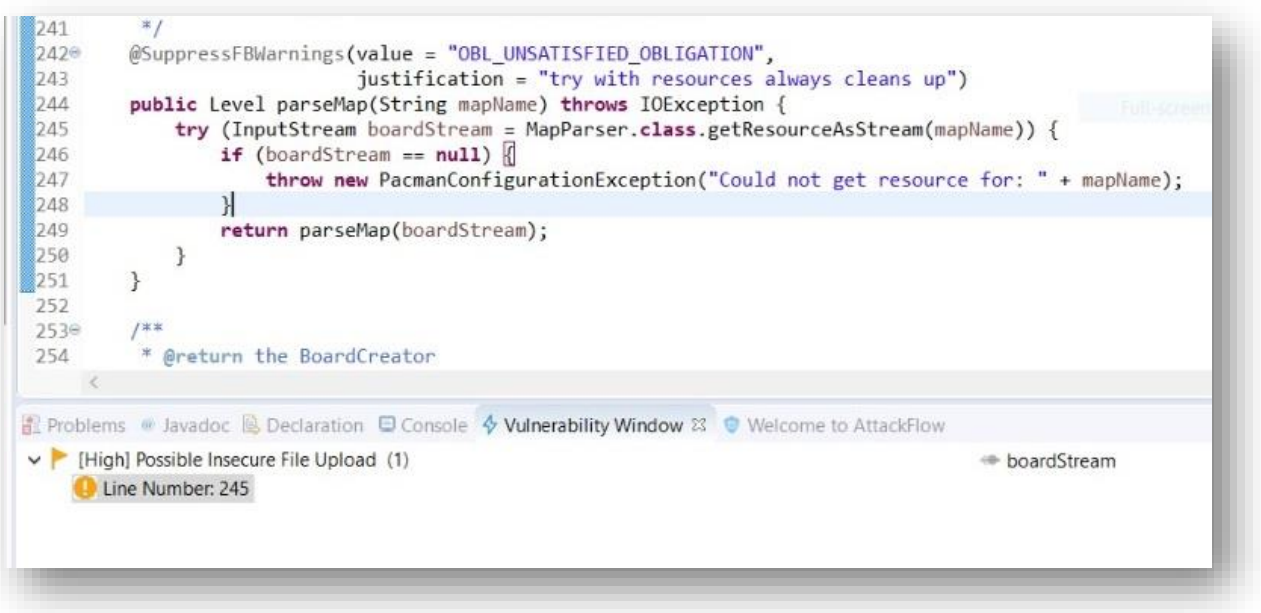
The last tool we used, is **AttackFlow**, another eclipse plugin-In that try to find every possible security weakness of the system.

Some of the possibile vulnerabilities reported by this software are abut the insercure generation of random numbers in the Ghost class.



The random numbers are used to define the direction of ghosts during the game. Assuming that random generated numbers are not really random, but defined with some algorithm, if someone knows which is the used algorithm, he can predict the next ghost's move. So, a possibile solution is to define an own algorithm to generated random number safely.

Another possibile vulnerability reported with this tool, is about the creation of the game map. In the ParseMap class, there is a possibile insecure file upload.

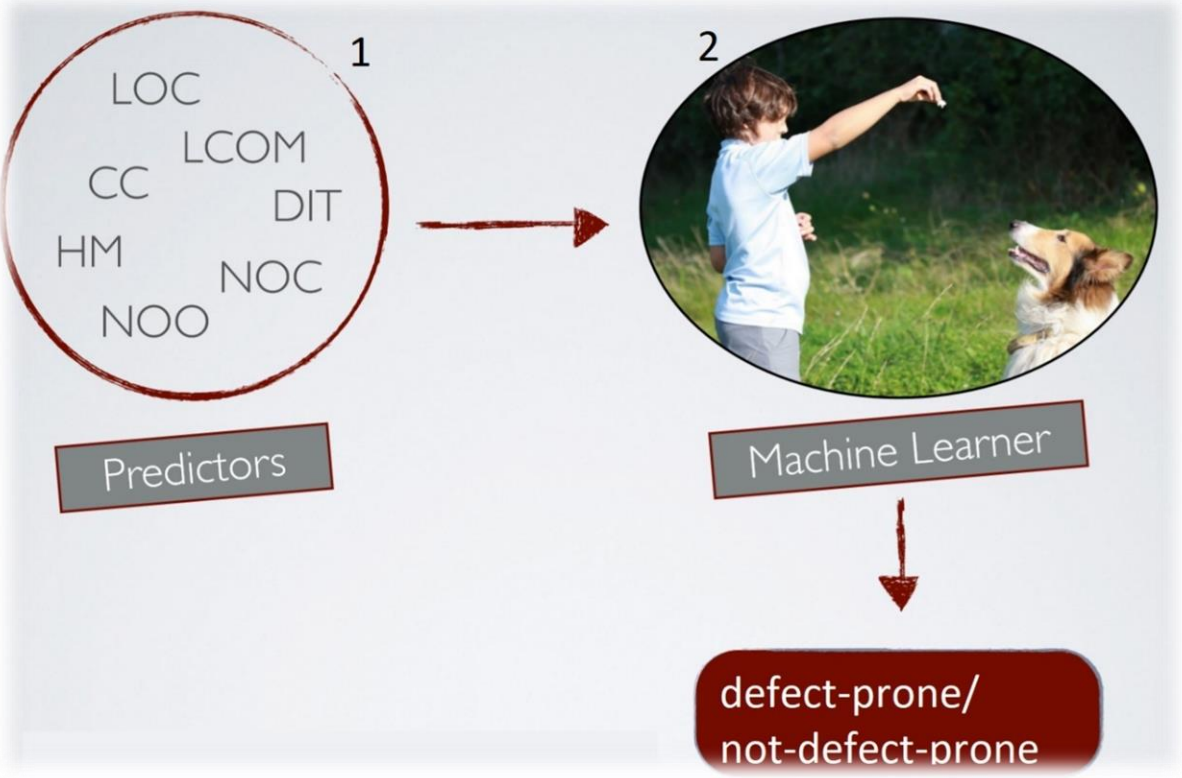


This is a possible vulnerability because if the user Is able to upload a different file from the expected one, the result can be unexpected. Malformed data or unexpected data could be used to abuse application logic. Anyway, in this case, this is not possible because the input file's path is specified using a final string.

```
public static final String DEFAULT_MAP = "/board.txt";
```

4. Defect prediction model

4.1 Dataset building



To build our defect prediction model, we need to extract data from the jPacman repository. So, we used PyDriller, a Python framework that allow us to extract information from the repository of the project.

We need to extract the defect-fixing commits of the project, the defect-introducing commits, and the defects-prone classes. If the commit has the "fix" word probably mean that the commit has fixed a problem, so, will be marked as defect-fixing commit. The `get_commits_last_modified_lines(fixed commit)` returns the set of commits that last "touched" the lines that are modified in the files included in the commit. So, we will have, for each commit, the last touched lines, that PROBABLY introduced the problem.

```
def get_buggy_commits():
    fix_commits = []
    for commit in RepositoryMining(repo).traverse_commits():
        if 'fix' in commit.msg:
            fix_commits.append(commit)

    gr = GitRepository(repo)

    buggy_commit_hashs = set()
    for fix_commit in fix_commits:
        bug_commits = gr.get_commits_last_modified_lines(fix_commit)
        buggy_commit_hashs.update(bug_commits)  # Add a set to a set
    return buggy_commit_hashs
```

With the results, we have build a csv file with the product metrics and process metrics defined in the calculate_metrics function.

```
def print_metrics_per_file(files):
    output = open('C:\\Users\\ferra\\Desktop\\Salerno\\jpacman-
framework\\output.csv', 'w')
    output.write('file,n-changes,added,removed,loc,complexity,defect-
prone\n'.format())

    for key, value in files.items():
        n_changes = len(value)
        added = value[n_changes - 1]['added']
        removed = value[n_changes - 1]['removed']
        loc = value[n_changes - 1]['loc']
        comp = value[n_changes - 1]['comp']
        buggy=False
        for i in range(0, (n_changes-1)):
          if(value[i]['buggy']==True):
            buggy=True
        # Append process metrics to CSV file
        output.write('{},{},{},{},{},{},{}\n'.format(key, n_changes, added,
removed, loc, comp, buggy))
```

```
def calculate_metrics(bugs):
    files = {}
    for commit in RepositoryMining(repo).traverse_commits():
        for mod in commit.modifications:
            if mod.filename.endswith('.java') and mod.change_type is not
ModificationType.DELETE:
                buggy = True if commit.hash in bugs else False
                process_metrics = {'change': mod.change_type, 'added':
mod.added, 'removed': mod.removed, '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)
    return files
```

And this is the output file.

File output.csv	n-changes	added	removed	loc	complexity	defect-prone
src\main\java\nl\tudelft\jpacman\Launcher.java	53	2	8	104	20	True
src\main\java\nl\tudelft\jpacman\board\Board.java	32	1	5	35	12	True
src\main\java\nl\tudelft\jpacman\board\Square.java	20	1	4	45	10	True
src\main\java\nl\tudelft\jpacman\board\Unit.java	26	3	3	51	13	True
src\main\java\nl\tudelft\jpacman\game\Game.java	15	80	80	52	12	True
src\main\java\nl\tudelft\jpacman\game\Level.java	1	9	0	5	0	False
src\main\java\nl\tudelft\jpacman\game\Player.java	4	50	5	41	8	True
src\main\java\nl\tudelft\jpacman\sprite\Sprite.java	8	43	43	8	0	True
src\main\java\nl\tudelft\jpacman\ui\Action.java	5	6	6	4	0	True
src\main\java\nl\tudelft\jpacman\ui\BoardPanel.java	9	101	101	51	7	True
src\main\java\nl\tudelft\jpacman\ui\ButtonPanel.java	7	23	23	21	2	True
src\main\java\nl\tudelft\jpacman\ui\PacKeyListener.java	7	33	33	25	5	True
src\main\java\nl\tudelft\jpacman\ui\PacManUI.java	19	1	3	50	4	True
src\main\java\nl\tudelft\jpacman\ui\PacManUiBuilder.java	20	1	2	58	9	True

src\main\java\nl\tudelft\jpacman\ui\ScorePanel.java	18	76	76	47	7	True
src\main\java\nl\tudelft\jpacman\board\Direction.java	10	56	56	19	3	True
src\main\java\nl\tudelft\jpacman\game\GhostColor.java	1	24	0	7	0	False
src\main\java\nl\tudelft\jpacman\sprite\AnimatedSprite.java	11	150	150	75	15	True
src\main\java\nl\tudelft\jpacman\sprite\EmptySprite.java	7	17	17	19	4	True
And so on…						

4.2 Builiding the defect prediction model

Now, we have all the data that we need to build the defect prediction model. To build a model, we used R language with caret library.

Defining the train-control:

```
train_control<- trainControl(method="repeatedcv", number=10, repeats=10, savePredictions = TRUE)
```

```
> summary(dataset$defect.prone)
False  True
    9    59
```

The prone class, has more true (59) than false (9), so data is unbalanced. using a machine learning algorithm out of the box is problematic when one class in the training set dominates the other. So, we decided to use Synthetic Minority Over-Sampling Tecnique (SMOTE) to balance data and solve the problem.

Defining the new train-control:

```
train_control<- trainControl(method="repeatedcv", number=10, sampling="smote",  repeats=10, savePredictions = TRUE)
```

It's well to know that in most cases RF works reasonably well with the default values of the hyperparameters specified in software packages, so in our case, the usage of grid search for the research of the hyperparameters not improve the performance of RF.

To better validate our model, more than the 10-fold cross validation strategy, that depends on the initial random splitting and so can overestimate or under-estimate the real performance of the model, we repeated the validation process multiple times, to mitigate the random effect.

To train the model, we decided to use the Random Forest algorithm, we tried to use other algorithms, but empirically, the Random Forest gave better results.

```
model <- train(defect.prone~., data=dataset,  trControl=train_control, method="rf")
```

```
Random Forest

68 samples
 6 predictor
 2 classes: 'False', 'True'

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

Resampling results across tuning parameters:

  mtry  Accuracy   Kappa
   2    0.8752381  0.04444444
  37    0.8950000  0.59814071
  72    0.8848095  0.56385010

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was mtry = 37.
```

We had an accuracy 0.8950000, while the k-value is 0.59814071. So, we have a good model, optimized for our data.

5. CodeCity overview

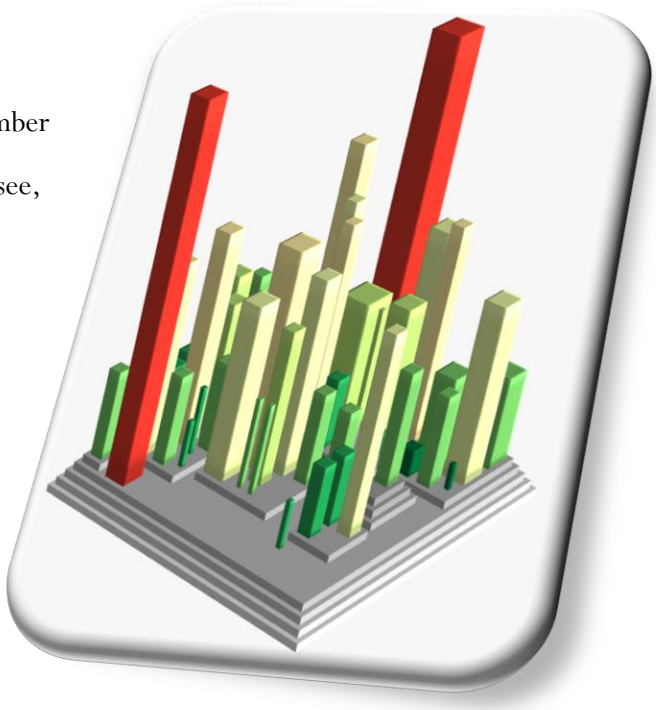
CodeCity is an Eclipse plugin that allow the user to see the project code in a different perspective. CodeCity allows managers and developers to analyze large software projects and to understand which parts of the software are troubling and need their attention. CodeCity describes the project as a city, in which each class is modeled as a building, this is one of the most popular and intuitive visualization techniques. CodeCity offers some interesting statistical data like the number of declared method for each class, the duplicated lines of code, the cyclomatic complexity and more. Let we see what is shown about JPacman:

This is the JPacman City!

This city has been modeled in order to the number of declared method for each class. As you can see, there are two buildings that are much higher than others. That means that there are a lot of methods in that classes.

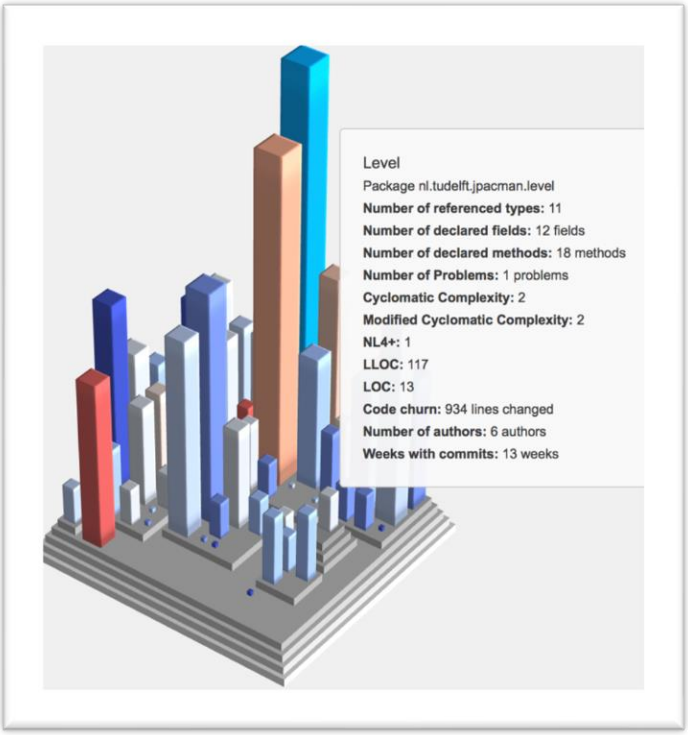
Guess what classes we're talking about.

YES, the blob classes that we saw in chapter 2!



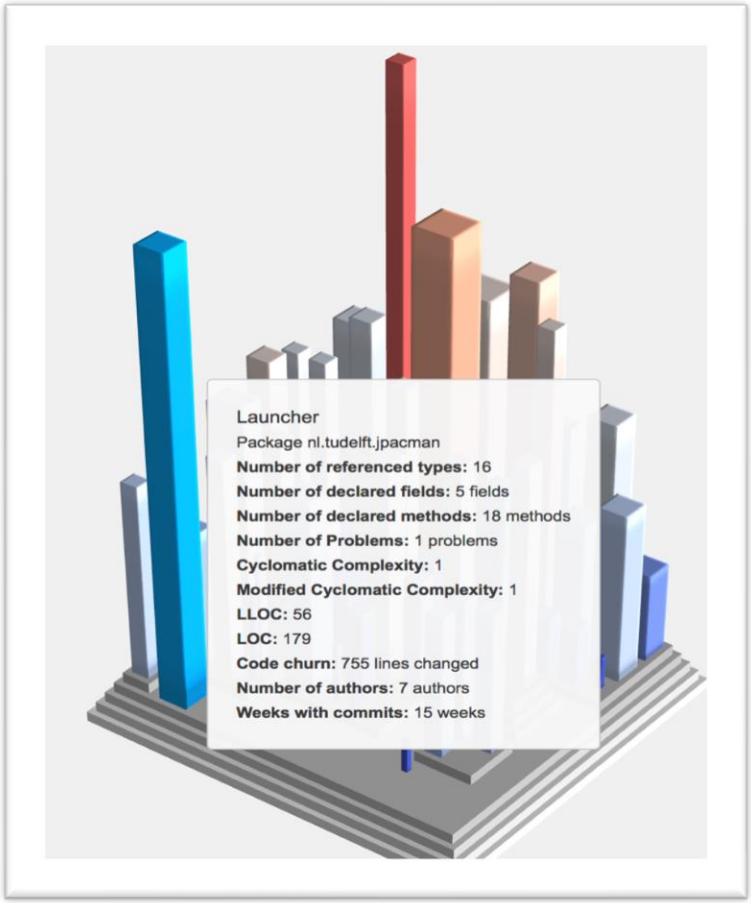
Level
Package nl.tudelft.jpacman.level
Number of referenced types: 11
Number of declared fields: 12 fields
Number of declared methods: 18 methods
Number of Problems: 1 problems
Cyclomatic Complexity: 2
Modified Cyclomatic Complexity: 2
NL4+: 1
LLOC: 117
LOC: 13
Code churn: 934 lines changed
Number of authors: 6 authors
Weeks with commits: 13 weeks

Launcher
Package nl.tudelft.jpacman
Number of referenced types: 16
Number of declared fields: 5 fields
Number of declared methods: 18 methods
Number of Problems: 1 problems
Cyclomatic Complexity: 1
Modified Cyclomatic Complexity: 1
LLOC: 56
LOC: 179
Code churn: 755 lines changed
Number of authors: 7 authors
Weeks with commits: 15 weeks



The Level class is also the class with the major number of declared fields (12).

While the Launcher class is one of the classes with the higher number of referenced types (16).



Some other interesting information are about the cyclomatic complexity of the classes. The classes with the higher complexity are the ones that model ghosts.

The most complex is Clyde class. In the game Clyde is the orange ghost (the worst!), and the Cyclomatic Complexity is higher than other ghosts because Clyde has 2 AIs, the first one is use when Clyde is far from Pac-man, while the second is used when Clyde is near him.