Project Software Reengineering

Robbe Claessens

robbe.claessens@student.uantwerpen.be

Tim Leys

tim.leys@student.uantwerpen.be

April 23, 2019

1 Introduction

We are going to refactor the codebase of the popular tool JFreechart written in Java so that we can easily implement a new feature. The new feature is, giving the user the possibility to alter the shape of the points used in graphs. At this moment it is just a simple dot, but the user should be able to chose whether he would like a dot or a rectangle for example. Before diving in to the code, there are some thing to be done. We need to analyse the project to see if for example the test suite is good, where the hotspots are, ... After the analysis, we can (based on the analysis) add extra test cases or be able to tell where there will be lot of problems when we alter certain code. This report describes how we will refactor and what important decisions we are going to make. First, we will take a look at the tools we have chosen for visualization, refactoring, code analysis, test coverage and mutation testing. After that, we talk about the test coverage.

2 Tool Choice

2.1 Tools for visualization and metrics

In order to have a better understanding about the project at hand, there are some tools that help us visualize the code. Since staring at code for hours is not efficient, we searched for some tools which gave us some important metrics to estimate the quality of the good and to get some better and deeper understanding about the codebase. The tools that were reviewed are: Codescene, JDeodorant and Moose.

A first tool we tried out was Codescene (https://codescene.io), this tool gave us some metrics to play around with. Some important metrics we gathered from Codescene are: Lines of code, hotspots in the code, number of active contributors, 'age' of the code, which files need refactoring, which files are coupled to one another. These metrics are really valuable, but not for what we are doing. Maybe the coupling part can be useful, the others are mere information.

Another tool that was worth taking a look at, was JDeodorant (an eclipse plugin). This tool helps with detecting code smells like Feature Envy, Type Checking, Long Method, God Class and Duplicated Code. But another handy thing is the visualization of the tool. JDeodorant helps us visualize the code smells. This is handy, because we can quick and easily take a look at the code smells and refactor them accordingly.

Moose is a known tool, but we were not able to make it work.

So we will use Codescene to get a better understanding about the code base and to gather some important metrics and we will use JDeodorant which will tells us all about the code smells in the code.

2.2 Tools for refactoring

When refactoring, we can do everything manually. But there are tools available which assist us with this process, so it is worth taking a look at.

Eclipse has built in tools for these, but since we are going to develop in IntelliJ, we are not going to explore this functionality. But IntelliJ itself also has refactoring tools available, which we are goign to use.

Also in this section is JDeodorant an adequate tool. It helps us detecting the possible refactorings, but it does not aide us with the refactoring. Besides that, the tool is worth mentioning here.

2.3 Tools for Duplicate Code Analysis

For the duplicate code analysis, the tools that were reviewed for this project are Intellij, iclones, DuDe, Clone Doctor and PMD CPD.

The InteliJ integrated tool for duplicate code analysis analyzed code syntactically, but enables you to choose whether to anonymize local variable, fields, and method names. Which allows us to detect duplicates of type 3: structurally identical clones with gaps. The tool is also integrated in the IDE that was mainly used for this project. It allows for fast navigation through the source code, with text highlighting.

IClones is a research tool by the university of Bremen. In contrast to IntelliJ, the tool allows for more detailed configuration of finding near miss clones. In IntelliJ,

could choose whether variable names or method invocations are anonymized, in iClones, we can determine the minimum of identical tokens that can be merged as near miss clones. The creators of iClones also provide a tool to visualize the code clones in a clear and understandable way.

Instead of looking at token similarity, like iClones, DuDe uses line similarity to discover clones. Because of this, the tool is language independent, but is also able to determine near-miss clones. DuDe features it's own GUI that clearly represents each of the clones found and where they are located in the source code.

While iClones is a token based matcher, Clone Doctor inspects the abstract syntax tree of the analyzed code. The extra information we have by inspecting the abstract syntax tree might result in fewer false-positives, however, the results will be very similar to the token-based tools.

PMD offers a wide collection of static code analysis tools. The CPD or Copy Paste Detector can be used to discover code duplication. This tool is, just like iClones, a token based detector with similar workflow.

For the project, we will use the IntelliJ integrated tool for detecting the clones throughout the whole source code. Code clones that are an exact copy are the most important, since they are most vulnerable to the problems of duplicate code. For the classes that we determine to be feature hotspots (and thus are expected to change a lot), will be analyzed with a token based tool, to find more complex code clones. For this, we will use the iClones tool, because of our experience with the tool and the good visualization options.

2.4   Tools for Test Coverage

To select the tool for the test coverage, we selected four of the most widely used coverage tools for Java and compared their results. The tools that were reviewed for this project were: EMMA, JaCoCo, IntelliJ IDEA built in analyzer, and Clover. The results of each tool are presented in Table 1.

As seen in Table 1, most tools produce similar results, with the exception of Clover. Clover also does not feature line coverage, but features statement coverage. However, though we would expect similar results to the line coverage, it is much less.

The built-in tool of InteliJ does not feature any way of showing the branch coverage and produced a different output in a previous run.

This left us with two remaining tools, EMMA and JaCoCo. Both tools are able to compute the line coverage, the branch coverage, and the method coverage and produce similar results. Because the JaCoCo tool is integrated in the IDE that was used in this project and is compatible with the maven site plugin, this was the best choice for the project.

	EMMA	JaCoCo	IJ IDEA	Clover
Line Coverage	71.2%	71%	71%	57.2% (statement)
Branch Coverage	46.7%	46%	/	48.7%
Method Coverage	72.1%	71%	71%	62.8%

Table 1: Test coverage according to EMMA

	Coverage
Line	71.2%
Branch	46.7%
Method	72.1%

Table 2: Test coverage according to EMMA

2.5 Tools for Mutation Tests

As for mutation tests, the available tools were LittleDarwin, PITest and Major Mutation Framework.

The Major Mutation Framework only works file per file or in combination with the Apache Ant tool. Since the current project is managed by Maven and other tools were available that worked out of the box, we chose not to look into this tool.

PITest seemed very promising, it is a mutation testing tool that is available as plugin for Intellij that promises to be much faster and much more efficient than competing tools. However, we were not able to make it work with the JFreeChart project.

Though Little Darwin is a research tool, it was the only tool that was able to work out of the box. It also provides a good and precise report of the coverage of each class in each package. Therefor, it is our preferred tool for mutation testing.

3 Test Coverage

3.1 Initial Test Coverage

	Coverage
Line	58.3%
Class	82.8%
Method	63.3%

Table 3: Test coverage according to IntelliJ IDEA

	Coverage
Line	71%
Branch	46%
Method	71%
Class	89%

Table 4: Test coverage according to JaCoCo