Project Software Reengineering

Robbe Claessens
robbe.claessens@student.uantwerpen.be

Tim Leys
tim.leys@student.uantwerpen.be

April 23, 2019

1 Introduction

In this report, we are going to talk about which tools we are going to use in order to guarantee a smooth refactoring of the JFreechart code. In what follows, we are going to describe for each major refactoring topic (Code Analysis, Test Coverage) which tools are available, which tools we will be using and why we choose those specific tools. The third chapter is about testing. When refactoring a system, we need certainty that the newly implemented changes or refactoring will not break the existing functionality. With testing, we are able to give us a certain degree of confidence to refactor the code. We will explain some test strategies we will use and how good the current code base performs on different testing metrics.

2 Project Scope

In this section, we will discuss the scope of the project as well as the risks that are involved with it.

2.1 Purpose and Scope

The purpose of this project is to inspect and improve the current code base of JFreeChart, such that implementing new functionality becomes easier. The new functionality that is desired is possibility to create time sequences, where each data point of the graph has a different shape. The data points are also given by a tuple *(x, y, shape)* and are stored in a database.

To identify the parts of the code that are related to this features (the feature hot-spots), we will first use visualization techniques as well as code metrics. Not only will this give us a better understanding of the code base, we can also use the metrics to quantify the improvements that we will make by refactoring.

We will determine the test coverage of code base. If the coverage is not adequate for the refactoring process on the 'feature hot-spots' that where identified in the previous step, we will update the test suite with more tests. We will not, however, add tests to improve the test coverage of whole code base.

In the refactoring step, we will improve the code structure, such that it is more suited to implement the new functionality. We will not implement the new functionality ourselves, or change the functionality of the code.

2.2 Non-Functional Requirements

During the re-engineering process, we will also aim to improve the code to better meet the non-functional requirement.

The first requirement that we will consider is maintainability. By choosing clear and understandable solutions, we aim to aid future maintenance projects in the code base. Also adding missing documentation will improve much.

For this project we only need to look ad improvements for extending the functionality for one specific use case. However, by considering the extensibility of the project, we can facilitate possible extensions in the future.

When refactoring the code, we will also consider testability. If the testability is improved, it would be easier to cover the code by the test suite. Not only will this improve the credibility of the library, but also facilitate future refactoring processes.

Re-usability encourages programmers to reuse pieces of codes as much as they can. In turn this will improve the maintainability. A well known problem of code clones is that if a bug exists in one clone, it exists in every clone and needs to be fixed in every clone. By reusing code as much as possible, bugs are not copied.

We will also aim to improve robustness. Because the current library can interact with a database as well as a file, we will take into account to make improvements such that it would be easier to incorporate more input formats.

2.3 Objective Solution Metrics

What makes a solution *good* is often very subjective. However, we will aim to quantify the improvements we will make, to objectively justify our solutions.

As for the main requirement of implementing the new functionality, we can use the amount of lines of code that are needed to implement the new feature. However, since implementing the new functionality is not in the scope of the project, we will make an estimation based on the the amount of classes and methods that are needed.
This also holds for the extendibility. We can only make estimations of how much changes are needed to implement any new functionality.

A metric that we will use for our solution is the decrease of god classes that are identified by the analysis tools that we will use. God classes are classes that have too much responsibilities. Since instances of god classes are tightly coupled to a large amount of other classes, they impede the maintainability as well as the extendibility of the code.

We would also like to see a decrease in code clones. Duplicate code is bad for maintenance, since duplicated code also means duplicated bugs. Bugs that could be fixed by updating a single class or method, have to be updated in each clone. It is also bad for reusability, since programmers would more likely clone code and adjust it to their needs instead of reusing generic functions.

Inheritance chains should not be too long. We will therefor aim to limit each chain to approximately 5 to 10 nodes. Longer chains indicate a bad design and limit the readibility of the code, thus decreasing the maintainability. We will, however, not avoid inheritance, since it is a key feature of object oriented designs.

3 Tool Choice

In this section, we will discuss all tools that were reviewed for this project. For each tool, we will briefly describe their features, their advantages, and their disadvantages, as well as explaining which tools we will use throughout the project.

3.1 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.

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 plug-in). 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.

3.2 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 going 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.

3.3 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

	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

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 hot-spots (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.

3.4 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 plug-in, this was the best choice for the project.

3.5 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 plug-in 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.