Introduction

I have been tasked to investigate and re-engineer Project “JFreeChart”. Firstly I investigate the system to gain an understanding and get familiar with the technical and business logic. Secondly we carry out static analysis and dynamic analysis to get some visualizations and graphs. We analyse our visualizations to see which parts of the system could be problematic or good candidates for refactoring or rewrites.

Project Investigation

There were multiple re-engineering patterns used in the investigation of “JFreeChart” project. The patterns used have been listed below along with how it was applied and what information was found using the pattern. I summarize the initial investigation with my findings using all the patterns listed.

Patterns used

This section is broken down into two sections

* Reverse Engineering Patterns used
* Reengineering Patterns used

The “Reverse Engineering Patterns used” describes all the patterns used to when we started to investigate the system and when we analysed the system. The “Reengineering Patterns used” is when we started to refactor the system to improve its functional and non-functional requirements.

Reverse Engineering Patterns used.

**Read all the Code in One Hour** (First Contact)(NEED TO COME BACK)

Pattern Application – The project source code was cloned from Github. JFreeChart uses Maven where I setup the packages and prepared it for Eclipse. I inspected the code using Eclipse. I had firstly went to see if there was functional / unit tests and ran all of those to make sure the system was running as expected. The project uses Javadocs and it is extensively commented within the sourcecode where I was able to find the developers intentions on some of the project functionality.

Pattern Information Finding - The project was developed using Java where it uses Object-Oriented principles. The project seems to be using the generic Java Programing Style Guidelines even though this isn’t explicitly declared on their website or forums. The project is considered to be classed as a medium sized project. There are varying degrees of code quality found within the project but the overall quality was considered to be above average. The project uses various programming idioms / design patterns used in most modern Object-oriented programming projects although I did manage to find a few code smells whilst looking at the org.jfree.chart.renderer.\* and org.jfree.chart packages. I had a look at some abstract classes and interfaces to see the design intention. I had found some very large classes with over 1900 lines of code and uses conditional statements extensively mainly from org.jfree.chart.renderer.\*

**Skim the Documentation** (First Contact)

Pattern Application – The majority of the documentation found in the project was found on

* Website - <http://www.jfree.org/jfreechart/>
  + Project overview
  + Project Samples (Screenshots and a Demo application that needs to be installed)
  + API Documentation (Up to date JavaDocs)
  + Frequently Asked Questions
* Project Source Code - <https://github.com/jfree/jfreechart>
  + ReadMe.md (Quick overview and versioned change logs)
  + ReadMe.txt (Project setup and technical documentation)
* Documentation - <https://sourceforge.net/projects/jfreechart/files/> (Includes versioning of documentation)
* Project History - <https://sourceforge.net/projects/jfreechart/files/1.%20JFreeChart/>
* Bugs - https://sourceforge.net/p/jfreechart/bugs/
* Patches - https://sourceforge.net/p/jfreechart/patches/
* Feature Requests - https://sourceforge.net/p/jfreechart/feature-requests/

We elicited as much information from the above sources of information. By reading through the various sources I was able to find out –

* the project’s overview and project intention for developers and users
* How up to date the documentation is compared to the source code
* Key points in the project and any common objects
* The technology stack and a little bit of the project design
* Requirements specifications came from the projects version control and desired features from the featured requests on source forge.

Information Finding – Our findings found out that the documentation is up to date with the most recent core project release even though this was in the year 2014. It still addresses the main functionality. We were not able to find any architectural information, design information and models describing the system. We did not find any database information either however we did find an up to date JavaDoc / Api Documentation. The Website provides a good overview of what the system does and what charts it supports. There is also instructions on how to install and build the system. We did find any email address of the chief designer on the system which we could contact later on if we need to. There is a Forum where we are able to find out other information if we need to.

Do a Mock Installation (First Contact)

Pattern Application – A JFreeChart Demo from <http://www.jfree.org/jfreechart/samples.html> was used to install and use JFreeChart Samples.

Information Finding – We found out the different types of charts that could be used with this application. I then tried to cross reference those charts in the source code to find out where they are. This would be an area we would like to focus on since it is the core functionality of the project. Org.jfree.chart seems to cover most of these charts. We also managed to do a source code build with no problems just by following the online instructions. We did manage to find two unit tests that did not run successfully out of the 2000 unit tests made. We did learn what the project intentions were which helped us focus more around Charts.

Study the Exceptional Entities

Pattern Application – We decided to run our analysis tools on the two packages we thought were core to the project and could potentially find issues with. We focussed around org.jfree.charts.\* and org.jfree.chart.renderer. We ran various static and dynamic analysis to inspect the code and find matrices to help us understand more about the project and also to find potential problems. We used Aspect Oriented Programing to find Aspects on the system when we ran through the Charts and Renderers unit tests. We also recorded Total Lines of Code, Cyclomatic Complexity, Weighted Methods Per Class, Average WMCA, Average Class / Method execution Time, Class / method execution Count and Method / Class dependency count.

Information Finding – We managed to find out that the two packages we focused on did indeed have issues. We found the classes belong to those two packages had performance issues, duplicated code, large lines of code count referring to a “God Class”. On further inspection into those classes we did find various code smells. This analysis is further explained later on in the document.

Step though the execution

Pattern Application – We stepped though the source code to confirm our understanding of the system but to also confirm our initial speculations on the system design and design patterns used to find out whether the system was using code idioms / design patterns as an anti-pattern. We did find the ChartFactory class was trying to use a Factory Pattern but it did not conform to this pattern correctly which could be one of the causes to the poor performance metrics.

Findings –

Through our re-engineering patterns that we used to analyse the project, we found that the ChartFactory, JFreeChart classes both found in the org.jfree.chart package and AbstractRenderer class found in the org.jfree.chart.renderer package would be our focus areas for futher re-engineer. We did not find the need to use other re-engineering patterns since we felt comfortable with our knowledge of the system and problematic classes we had found. Also time was an issue which meant we were not able to use all the re-engineering patterns we would of liked to have used.

Reengineering Patterns used

**Write Tests to Enable Evolution**

Pattern Application – we updated and wrote new unit tests to keep up with the latest refactored changes to the system.

Pattern Information Finding – We found that not all unit tests were written for all system functionality. We did update a few but found that this would be another project in its own right. We have kept this as a record for further development.

**Test the Interface, Not the Implementation**

Pattern Application – We refactored the code to use the latest SOLID principles and design patterns which means we used interfaces and therefor were able to test out interfaces and not the class. This is good since class implementation changes could break our unit tests in the future.

Pattern Information Finding – We found that there were a lot of unit tests that did not adhere to this principle and have kept a note of this in case of further project development.

**Regression Test After Every Change**

Pattern Application – We ran tests after every change we made to the system and any refactorings to make sure we did not break any core functionality and to make sure we adhered to the original function requirements.

**Compare Code Mechanically & Visualize Code as Dotplots**

Pattern Application – we wrote a tool that would inspect all files for code duplication and then inspected those files with the highest percentage code duplication. We did exclude all tests and demo files which don’t influence the projects core functionality.

Information Finding – We did find a lot of classes that had at least 30 % - 43% code duplication. We did not have enough time to inspect all those classes to find if they are suitable classes for reengineering but we did keep the graphs and metrics in case of further project development. We did find a lot of code duplication In the JFreeCharts project file which lies within our analysis.

**Move Behavior Close to Data & Split Up God Class**

Pattern Application – We used these two patterns include a template method pattern used in most object oriented programming languages. We refactored the ChartFactory Class. The refactoring used did adhere to the SOLID Principles as well in case there was further project development on this area of the project.

Information Finding – We did find other classes besides our ChartFactory that we refactored that could use these design patterns but were out of our scope for our reengineering project. These were recorded.

Find a Problematic Class / Group of Classes

We employed various analysis tools and metrics to help locate our problematic class. The following are the analysis techniques that were used for the reengineering analysis used on the “JFreeChart” project.

* Static Analysis
  + Cyclomatic Complexity
  + Weighted Methods Per Class (Total complexity)
  + Weighted Methods Per Class (Average complexity)
  + Total Lines of Code Per Class
  + Code Duplication
* Dynamic Analysis
  + Total Count on Class Executions
  + Total Count of Method Executions
  + Total Execution Time Per Class
  + Total Execution Time Per Method

Fix the Problematic Class / Group of Classes

Before ChartFactory 57 wmpc and average 1 cc

After ChartFactory 6 wmpc and average 1 cc

Lines of code

Before 1526

After 51 (Show the graph to show how even the amount of lines of code are) the average is 62.77

Class Count = 84 and 84 (Exact the same)

Class times

Before

33254980.44 7th place

36256560.93 7th place (This boils down to the actual chart implementation which needs to be refactored)

After

From your analysis you should identify one class, or a group of classes, upon which to focus your re-engineering efforts.

You should write up your findings over 4-7 pages (including any charts or other visualisations)

Document changes 2-3 pages (include some evidence that your change has made a significant difference)