University of Sheffield

Re-Engineering Assignment1 Report

COM3523 Software Reengineering

Zhicong Jinag

acc20zj

Department of Computer Science

March 24, 2023

Initial analysis

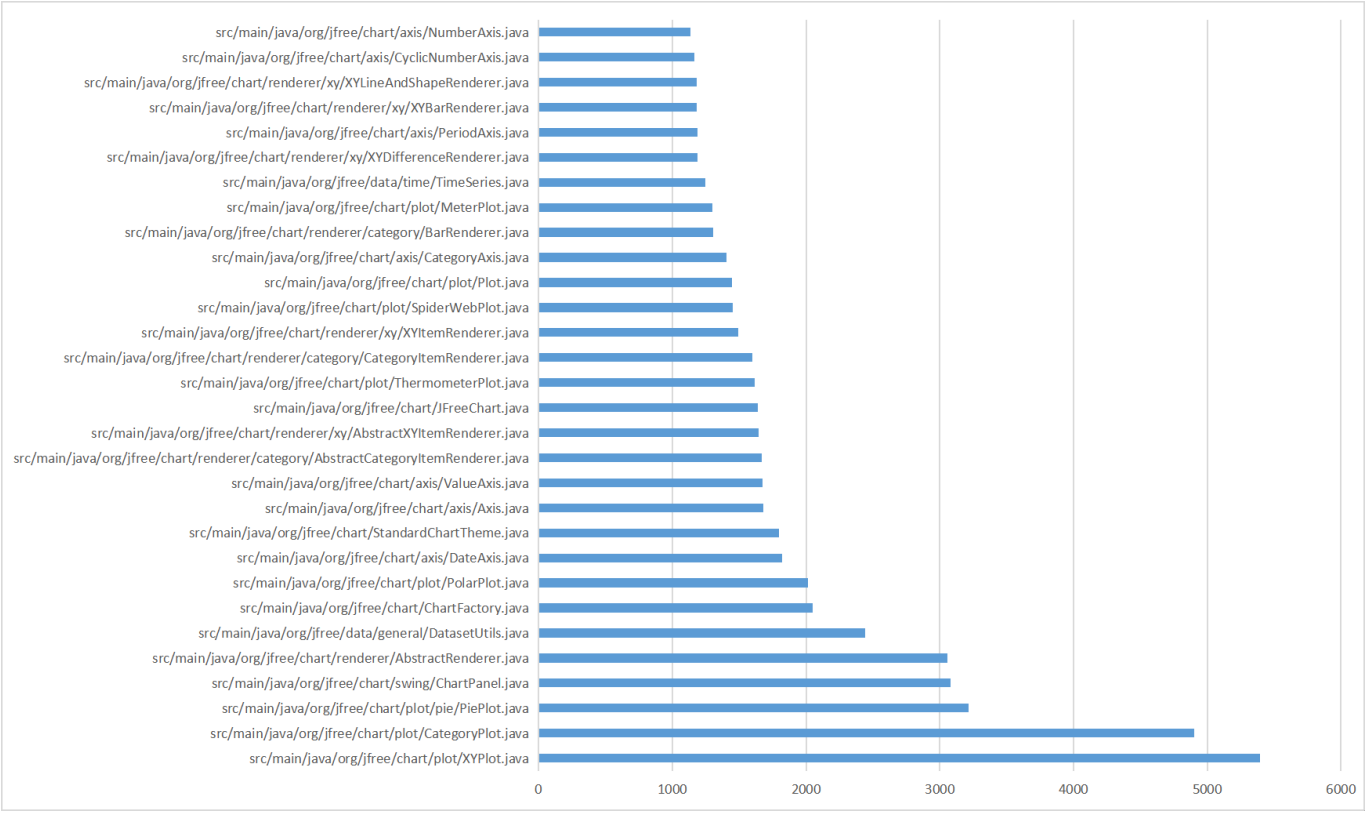

Figure 1: Top30 classes with Line of code value

Re-engineering is the process of reconstructing and enhancing already-existing software in order to enhance software quality and maintainability. Analyzing the number of lines of code can clarify the program scale, maintainability, and reconfiguration requirements. The number of lines of code is a sign of the quality and maintainability of the software. It is clear that several classes in the JFreeChart project predominate in terms of lines of code from the diagram fragment above, which displays the top 30 classes in the project with the most lines of code. Indicating their more significant importance in the overall project, the "XYPlot.java", "CategoryPlot.java", and "PiePlot.java" classes, for instance, have more than 3,000 and even 5,000 lines of code.

It is worth noting that there are a lot of lines of code for large classes in the "src/main/Java/org/jfree/chart/" directory, and there are more files there, indicating that the main role of the project is to create graphics. All in all, the project has several classes that have more than 1000 lines of code, which indicates that the classes are more complex or have a lot of important logic to implement and may be less readable and maintainable in terms of readability and maintainability. When maintaining and refactoring the JFreeChart project, you can pay attention to the files and classes in the "org.jfree.chart" package.

Reverse-engineer class diagrams

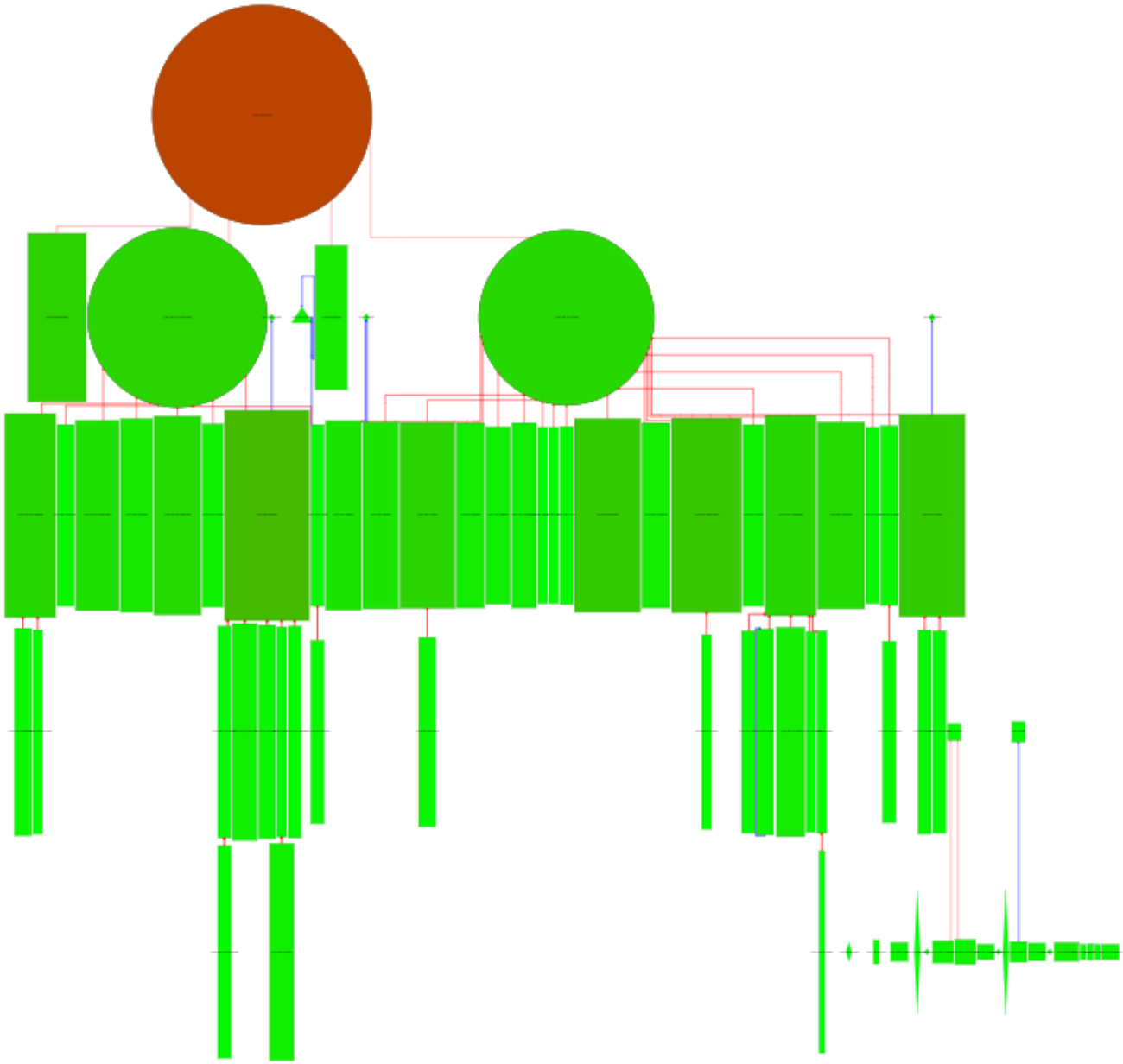

Figure 2: Enhance Class Diagram

Reverse engineering class diagrams are the process of creating a visual representation of classes, their properties, methods, and the relationships between them, based on existing code or software. This process can reveal aspects of a software system, such as its structure, design patterns, dependencies, and behavior. The above diagram is a reverse-engineered class diagram of all the classes in the “org.jfree.chart.renderer” package of the JFreeChart project. According to the type of Class, the class graph is divided into four categories: ”Interface”,”Enum”,”Abstract”, and ”Class”, which are represented by different shapes. For example, the diamond represents Interface, the triangle represents Enum, the circle represents Abstract, and the box represents the general class. In addition, by calculating the number of members and methods of each class, the width and height of the graph are corresponding respectively, and the color of the graph also reflects the number of members of the class. The darker the color, the more members it contains. It is worth mentioning that the reverse engineering class diagram also reveals the corresponding relationship between each class.

In the figure above, the red line represents the inheritance relationship, and the darker the line color, the deeper the inheritance depth of the class. And the blue line is the correlation.

According to the analysis of a class diagram, which can see clearly in the inheritance hierarchy, inherit the shallow depth of class is the "org.jfree.chart.renderer.AbstractRenderer", it is the most basic class and origin of the entire hierarchy. The base class is extended by several other classes such as "AbstractXYItemRenderer" and "AbstractCategoryItemRenderer". These abstract classes outline a number of typical rendering techniques. An example of a general framework for creating XY (x-y) charts is "AbstractXYItemRenderer". This framework involves sketching axes, data points, lines, and other geometric shapes. Each concrete renderer class implements its own distinct rendering logic while deriving from these abstract classes. For instance, "XYLineAndShapeRenderer" creates an XY chart that includes both lines and data points, but "XYDifferenceRenderer" just creates an XY chart that shows the difference between the two lines. In JFreeChart, this class diagram illustrates the interdependence and inheritance between renderer classes. The foundational elements of the JFreeChart package, which let programmers choose how their data is shown, are these renderer classes.

Call Graph

Call graphs show the calling relationships between components in a software system, helping to better understand and maintain code, quickly locate problems, and make targeted optimizations. Based on the call graphs constructed for the jFreeChart project, the following two tables were generated using Fan-in and Fan-out as metrics.

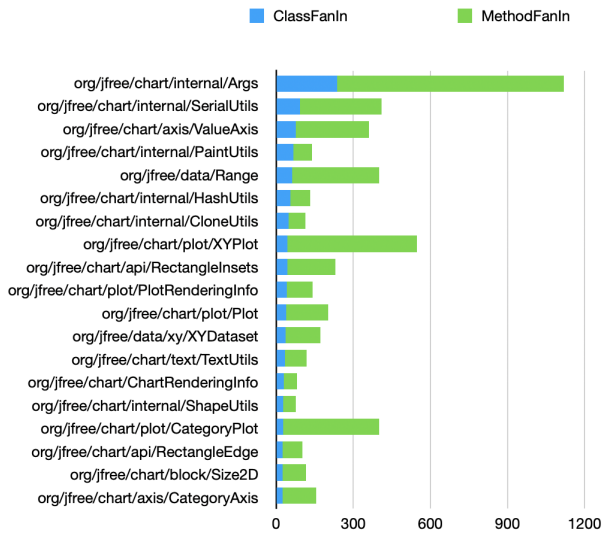

Figure 3: Class Fan In Graph

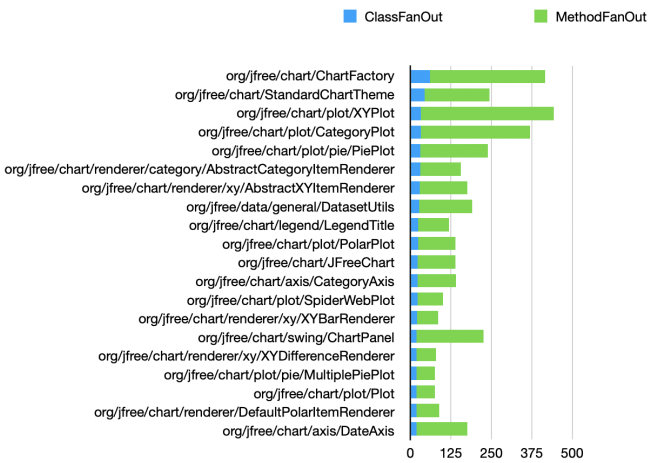

Figure 4: Class Fan Out Graph

Based on the analysis of the call graph, the classes are sorted by Fan-In and Fan-Out in order to more easily identify which classes are the most important, where the blue bar represents the Fan-In and Fan-Out of the class, and the green bar represents the sum of the Fan-In and Fan-Out of all methods of the class.

The Fan-in call graph shows that the "org/jfree/chart/internal/Args" class is one of the most influential classes in the library, as it has a very high ClassFanIn and MethodFanIn of 238 and 880, respectively. "org/jfree/chart/internal/SerialUtils" and "org/jfree/chart/axis/ValueAxis" follow with 93 and 77 references by other classes, and the Fan-In numbers of these classes may indicate that they are among the most

important components in the library and have a significant impact on other components. As for Fan-out, the "org/jfree/chart/ChartFactory" class has a relatively high number of Fan-Outs both for classes and methods, which indicates that this class needs to interact and collaborate more with other classes and they are more likely to be affected by changes in other classes. This makes these classes more fragile and difficult to maintain, and therefore requires consideration of how to reduce dependencies or optimize the design. It is worth noting that classes with high Fan-Out are related to drawing and presenting charts, and that these classes, such as "org/jfree/chart/plot/XYPlot" and "org/jfree/chart/plot/CategoryPlot", also have relatively high lines of code compared to Initial analysis. This is a good indication that plotting is the most complex part of the library,which deserves more attention from developers

Dynamic Analysis

Software Reconnaissance is the systematic understanding and exploration of the main features and behaviors of the target software, separating the code related to different features and behaviors. Here I chose to isolate the source code related to setting and rendering BarChart, and selected three other test cases that do not perform this function, collecting and recording relevant data and analyzing their trace information while observing the behavior of the target software.

Package	Class	Method
org.jfree.chart	BarChartTest	testReplaceDataset
org.jfree.chart	ChartFactory	createBarChart
org.jfree.chart.api	RectangleEdge	isLeftOrRight
org.jfree.chart.axis	AxisState	getMax
org.jfree.chart.axis	CategoryAnchor	values
org.jfree.chart.axis	CategoryAxis	<init>
org.jfree.chart.axis	CategoryLabelPosition	getAngle
org.jfree.chart.axis	CategoryLabelPositions	getLabelPosition
org.jfree.chart.axis	CategoryLabelWidthType	$values
org.jfree.chart.axis	CategoryTick	<init>
org.jfree.chart.event	RendererChangeEvent	<init>
org.jfree.chart.internal	ShapeUtils	rotateShape
org.jfree.chart.labels	AbstractCategoryItemLabelGenerator	<init>
org.jfree.chart.labels	StandardCategorySeriesLabelGenerator	<init>
org.jfree.chart.labels	StandardCategoryToolTipGenerator	<init>
org.jfree.chart.plot	CategoryPlot	getRangeAxis
org.jfree.chart.plot	PlotRenderingInfo	setPlotArea
org.jfree.chart.renderer.category	AbstractCategoryItemRenderer	findRangeBounds
org.jfree.chart.renderer.category	BarRenderer	findRangeBounds
org.jfree.chart.text	TextBlock	calculateDimensions
org.jfree.chart.text	TextFragment	calculateDimensions
org.jfree.chart.text	TextLine	calculateDimensions
org.jfree.chart.urls	StandardCategoryURLGenerator	<init>
org.jfree.data	DefaultKeyedValues	getValue
org.jfree.data	DefaultKeyedValues2D	getValue
org.jfree.data.category	DefaultCategoryDataset	getValue

Figure 5: Package Class Table

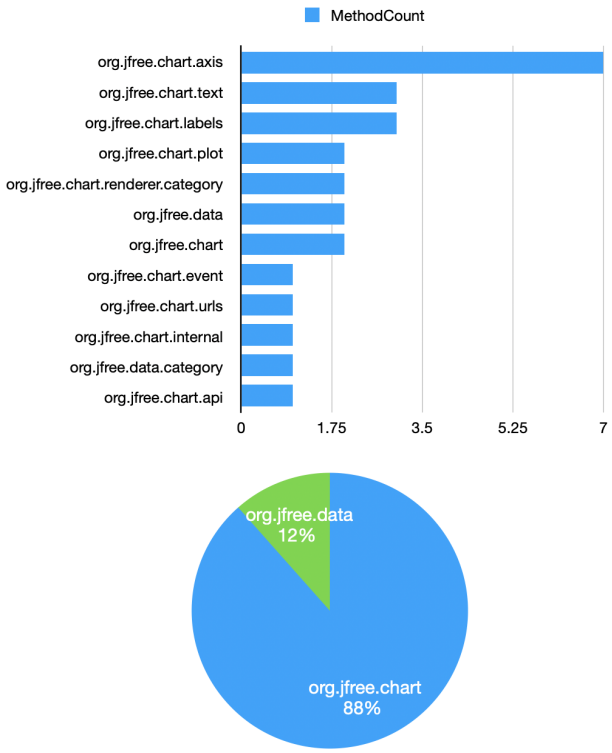

Figure 6: Package Class Graph

Figure 5 shows the source code of the classes and methods related to rendering BarChart ("BarChartTest") and the other three test cases ("ScatterPlotTest", "KeyedObjectsTest", " CategoryToPieDatasetTest") to isolate the resulting classes and methods, i.e. the source code related to a specific type of chart BarChart. However, this approach may be subject to error, because some code related to a specific function may also be related to other functions and thus be executed in test cases that do not contain the specific function, and thus may also be misclassified as irrelevant code. To avoid this, a combination of other methods of filtering is needed to ensure that the code filtered out is irrelevant to the particular function.

By organizing the data, we can find that the function of rendering BarChart calls many classes and methods in the "org.jree.chart.axis" package. It is worth noting that some of the classes in this package have high Fan-out values such as "org.jree.chart.axis.ValueAxis" as mentioned in the previous part of the call graph analysis, which shows the importance of this package in this function and even in the project. In addition, it is clear from the pie chart that this feature depends on the "org.jfree.chart" and "org.jfree.data" packages. Since this feature is mainly about BarChart rendering, the dependency of this feature on the "org.jfree.chart" package is 88%, which shows that the classes related to rendering are concentrated in this package, so they are highly localized and the code is highly readable and maintainable.

Conclusions

Overall, the JFreeChart project contains a large number of classes with a large amount of code, which may require more testing and debugging to ensure its correctness and reliability. In addition, too many lines of code may indicate that the class has too many responsibilities and does not conform to the single responsibility principle, which requires consideration of refactoring to reduce class complexity and improve maintainability and extensibility.

The reverse engineering class diagram shows that the project class naming is clear, class relationships are well designed, and developers can easily understand the code intent, reducing the difficulty of code maintenance and refactoring operations.

From the call diagram, we can see that the most influential class in this project is the class with the largest Fan-in value, which is referenced by many other classes, and once they change, it will affect the classes that call these classes. The Fan-out value identifies classes that are relatively fragile and difficult to maintain because they reference a large number of other classes and are vulnerable to changes to other classes. Therefore, we can consider how to reduce the dependency or optimize the design, fully consider the dependency and coupling between classes, reduce the dependency between classes, and improve the cohesiveness and maintainability of the code.

Finally, we analyzed that the project mainly relies on the packages "org.jfree.chart" and "org.jfree.data", the former focuses on chart building and rendering, the latter focuses on data presentation, and the code of related functions are stored in the corresponding packages and subdivided according to different functions, so they are highly localized and the code is readable and maintainable. However, since the trace information of only a few test cases are compared here, it is not possible to precisely isolate the specific source code for rendering barchart. Some code related to specific types of charts may overlap with code for other functions, and careful judgment is needed to avoid filtering out code related to specific types of charts.