KS489

Assignment 3 – JFreeChart

Kyle Spindler

CO7506

Contents

[Introduction 2](#_Toc471802767)

[Project Investigation 2](#_Toc471802768)

[Patterns Used 2](#_Toc471802769)

[Reverse Engineering Patterns used. 2](#_Toc471802770)

[Findings 4](#_Toc471802771)

[Reengineering Patterns used 4](#_Toc471802772)

[Find a Problematic Class / Group of Classes 5](#_Toc471802773)

[Static Analysis 5](#_Toc471802774)

[Code Duplication Analysis 5](#_Toc471802775)

[Weighted Methods Per Class Analysis 5](#_Toc471802776)

[Lines of Code Analysis 6](#_Toc471802777)

[Combining the Lines of Code Analysis with Weighted Methods Per Class 6](#_Toc471802778)

[Dynamic Analysis 7](#_Toc471802779)

[Total Count on Class Executions 7](#_Toc471802780)

[Total Execution Time Per Class 7](#_Toc471802781)

[Fix the Problematic Class / Group of Classes 7](#_Toc471802782)

[Refactoring Changes to ChartFactory 8](#_Toc471802783)

[ChartFactory before refactor - 8](#_Toc471802784)

[ChartFactory after refactor 8](#_Toc471802785)

[Refactoring Changes Summary 11](#_Toc471802786)

[Appendix 1 0](#_Toc471802787)

[Appendix 1.1 0](#_Toc471802788)

[Appendix 1.2 1](#_Toc471802789)

[Appendix 1.3 2](#_Toc471802790)

[Appendix 1.4 3](#_Toc471802791)

[Appendix 2 4](#_Toc471802792)

[Appendix 2.1 4](#_Toc471802793)

[Appendix 2.2 5](#_Toc471802794)

[Appendix 3 6](#_Toc471802795)

[Appendix 3.1 6](#_Toc471802796)

[Appendix 3.2 7](#_Toc471802797)

[Appendix 3.3 8](#_Toc471802798)

[Appendix 4 9](#_Toc471802799)

[Appendix 4.1 9](#_Toc471802800)

[Appendix 4.2 10](#_Toc471802801)

[Appendix 4.3 11](#_Toc471802802)

# 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. We then apply the appropriate refactoring to a selected class and re-run our static and dynamic tests. I then compare the initial and refactored graphs and analysis to see if any improvements have been done.

# 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 which introduced high cohesion and lowing coupling which is good for further project development. 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 Execution Time Per Class

## Static Analysis

### Code Duplication Analysis

The classes which we felt were duplicated the most were the Dataset, Resources and Entities classes. The Resource files were excluded as they don’t relate to the reengineering project. The average duplication for these classes ranged from 28% - 38% which is consider above average for a project. The code duplication did remove curly bracers and empty whitespaces. We did see a need to potentially refactor most Dataset and Entity classes as this is where we saw the most code duplication. We also saw Chart related classes with an above average code duplication.

(INCLUDE THE CHART FOR REFERENCE)

### Weighted Methods Per Class Analysis

We used two types of Weighted Methods Per Class –

* Weighted Methods Per Class (Shows total Cyclomatic Complexity for all methods on that particular class)
* Weighted Methods Per Class (Shows average Cyclomatic Complexity for all methods on that particular class) WMPCAverage = WMPC / MethodCount

We found the classes that had the highest WMPC scores were -

* Plot.\* classes (e.g. XYPlot.java, CategoryPlot.java)
* AbstractRenderer which ranked 6th with a score of 281
* ChartFactory which ranked 76th with a score of 57
* JFreeChart ranked 24th with a score of 123

We found the classes that had the highest Average WMPC scores were –

* Utilities.\* classes (e.g. PaintUtils.java, DatasetUtilities.java, LineUtils.java etc...)
* Renderer.\* classes (e.g. CyclicXyItemRenderer.java, LayeredBarRenderer.java, StackedBarRenderer.java etc…)

This is where a lot of the most complex logic sits. The renderer classes render graphs in different ways and how they renderer is complex. This also applies to the utilities classes.

### Lines of Code Analysis

(INCLUDE THE CHART FOR REFERENCE)

As you can see in figure (CHART REFERENCE) the top 5% of the project reaches above a 1000 lines of code which is way above average for this project. This could indicate potential God Classes that could be good candidates for refactoring. The following classes were recorded as potential reengineering classes for this project.

* Plot.\* classes (e.g. XYPlot.java, CategoryPlot.java etc…) a lot of these plot type classes appeared in the most lines of code in the project
* AbstractRenderer this class ranked 6th most lines of code with 3323 lines. This class falls in our initial investigation which belongs to org.jfree.charts.renderer which we tried to focus our efforts into.
* JFreeChart & ChartFactory these two classes also fell in our initial investigation which belongs to org.jfree.charts. JFreeChart ranked 15th with 1579 lines and the ChartFactory ranked 17th with 1526 lines of code.

### Combining the Lines of Code Analysis with Weighted Methods Per Class

Lines of code could indicate a potential problem with a class but it could also be ambiguous as the duplicated lines of code could be braces such as {} or () but could also include comments such as /\* or \*. So I paired the Total Lines of Code with Weighted Methods Per Class and found the following

|  |  |  |
| --- | --- | --- |
| **Class Name** | **Code Duplication** | **Weighted Methods Per Class** |
| **JFreeChart** | 1579 (rank 15th) | 123 (rank 24th) |
| **AbstractRenderer** | 3323 (rank 6th) | 281 (rank 6th) |
| **ChartFactory** | 1526 (rank 17th) | 57 (rank 76th) |

When these two metrics are combined we get to see possible candidates for reengineering. All these classes have above average code duplication and Weighted Methods Per Class.

## Dynamic Analysis

The Dynamic Analysis was set around the unit tests that targeted the following packages org.jfree.chart and org.jfree.chart.renderer. We felt that this was the two areas that we were going to focus on based on our initial project analysis and static analysis. Even though these unit tests did not have good coverage so these figures are not completely accurate. But they do give us an indication of the importance of these figures along with the total lines of code and weighted methods per class analysis.

### Total Count on Class Executions

Classes and how many times they were executed during the Dynamic Analysis.

|  |  |  |
| --- | --- | --- |
| **Class Name** | **Rank** | **Times Executed** |
| **AbstractRenderer** | 13 | 2234 |
| **JFreeChart** | 114 | 32 |
| **ChartFactory** | 84 | 83 |

### Total Execution Time Per Class

Class and how long it took to execute the class on average during the Dynamic Analysis. The total time all methods took divided by the number of methods executed.

|  |  |  |
| --- | --- | --- |
| **Class Name** | **Rank** | **Times Executed** |
| **AbstractRender** | 37 | 10753605.68 |
| **JFreeChart** | 111 | 2306247.49 |
| **ChartFactory** | 7 | 33254980.44 |

# Fix the Problematic Class / Group of Classes

The class that we have opted for as our initial project reeingeering is the ChartFactory class in the org.jfree.charts package. We felt this would be a good candidate for reeingeering since it has displayed problematic metrices during our static analysis. It has a big influence on the system and gets called frequently. It is also one of the core aspects to the system where Charts get generated. This was not the biggest problem or concern when analysing our data but it was amoungts the top. We felt choosing this class doesn’t pose a big risk for us or the client when reeingeering which will help us gain confidence with our customer and development team as a first iteration project, but it has enough influence to show that the system can benefit from the reengineering. The class isn’t extremely complex and does display signs that this is a God class that has High coupling, which is a problem for further software development on the project which could increase the project’s bug count. We decided to use a design pattern called “Template method” pattern. Using this pattern we hope to decrease the following –

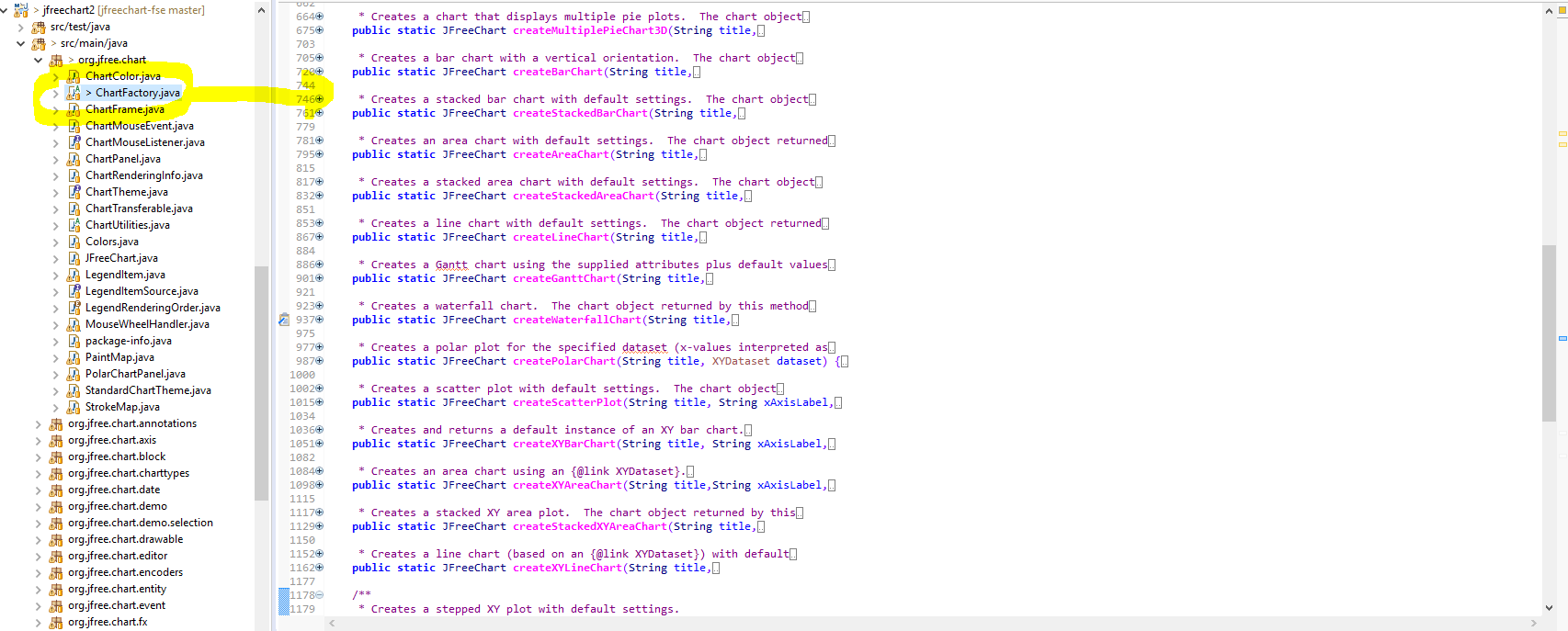
* Reduce Total Lines of Code.
* Reduce Total Weighted Methods Per Class and Cyclomatic Complexity.
* Reduce Total amount of times the class gets called.
* Reduce code duplication

Using the “Template method” pattern will help us with the following –

* Increase cohesion and lower coupling
* Abide by the SOLID Principles
  + Single Responsibility Principle
  + Open / Close Principle
  + Liskov Substitution Principle
  + Interface Segregation Principle
  + Dependency Inversion Principle
* Make it more maintainable for future development.
* Reduce complexity

## Refactoring Changes to ChartFactory

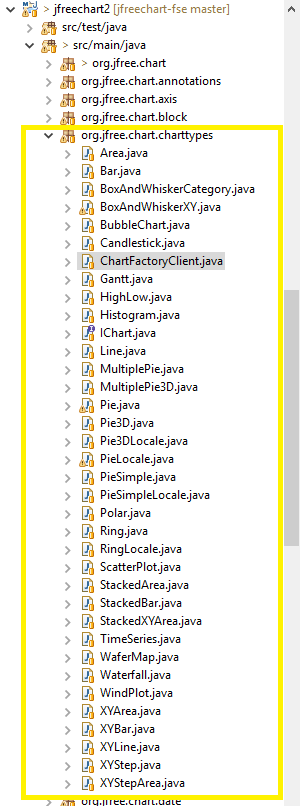
### ChartFactory before refactor -



The ChartFactory had multiple methods that would create a specified chart type. This lead to over 1500 lines of code which would place this class as a “God Class”. This class belonged to org.jfree.chart

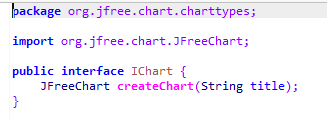
### ChartFactory after refactor

ChartFactory was renamed to ChartFactoryClient and was placed in its own package under org.jfreechart.charttypes. All methods were placed in its own class to represent a chart type.



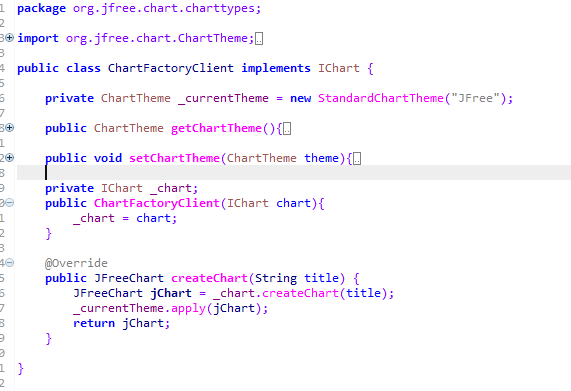
The graph shows the interface, ChartFactoryClient and the multiple Chart types that were created when refactoring the original ChartFactory.

Both the Charts and ChartFactoryClient need to abide by the following contract



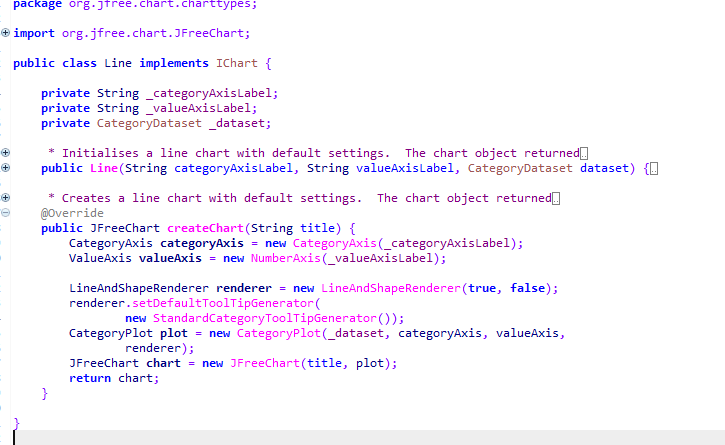
Each Chart Class should be responsible for how it wants to implement its chart type and not the god class which was ChartFactory.

The following as the ChartFactoryClient that generates the different kinds of Charts available.



Here we are able to pass implementation into the ChartFactoryClient using interfaces which makes our code less coupled and helps us with code maintenance or adding new Chart Types in the future without changing core classes and there is no hard dependencies which will makes out project less likely to break due to changes. The ChartFactoryClient does have two common methods, setChartTheme and GetChartTheme and is responsible to apply the chart theme. This does not differ between chart types and therefor the Factory will be responsible for all chart themes.

Below is an example of a Chart Type that is used by the ChartFactoryClient



The Chart Type in this case, the Line Chart, has its own implementation on how this type of chart should be generated and this will be different other Chart Types. So the implementation stays on the class and not integrated into one class that has multiple implementations. So we removed the responsibility from the original ChartFactory into its own classes.

The refactored project can be found at <https://github.com/ks489/jfreechart-fse> where the reengineering took place.

## Refactoring Changes Summary

We used a number of Reeingineering patterns and design patterns when refactoring the ChartFactory Class. The Reeingineering patterns used when refactoring our code is described in the Project Investigation along with the other Reeingineering patterns used in the initial investigation of the project.

|  |  |  |  |
| --- | --- | --- | --- |
| **Metric Type** | **Before Refactor** | **After Refactor** | **Graph** |
| **Weight Method Per Class** | 57 | 6 |  |
| **Weight Method Per Class Average** | 1 | 1 |  |
| **Total Lines of Code** | 1526 | 51 |  |
| **Class Count** | 84 | 252 |  |
| **Class Time Execution** | 33254980.44 | 36256560.93 |  |

**Weighted Method Per Class**

We have reduced the total class complexity from 57 to 6. This was the charts implementations which was transferred into their own classes.

**Weight Method Per Class Average**

The Cyclomatic Complexity for a given method in this class is 1 which is good. This value did not change during the refactoring.

**Total Lines of Code**

The total lines of code changed from 1526 to 51 lines of code of which half of the 51 lines of code is not java related such as comments. The source code got split up into its own classes along with their own implementation. Each class relates to a chart type and is sent to the ChartFactoryClient to generate its own implementation since each chart will have a different implementation than another chart. There is common functionality between all these charts which is the theme of the chart and hence why we opted for the Template Method Pattern. This pattern helps us resolve this problem of reducing ChartFactory God Class. As you can see the 1526 lines of code has been split up into multiple classes as shown in FIGURE The average lines of code is 61 instead of the one god class with 1526.

**Class Count**

Class count did rise from 84 to 252 which isn’t always a bad indicator. We have made our new refactored ChartFactory more cohesive which is indicated in the increase of class executions. These “references” use interfaces which means there isn’t any hard dependencies in the project which is good for project maintenance and is less likely to break during changes.

**Class Time Execution**

Class Execution has roughly stayed the same but this was not our focus for this iteration on the project. We wanted to reduce class complexity first before addressing performance issues. This could be addressed in the actual Chart Type implementation for each chart which is indirectly related to the actual ChartFactoryClient. Each chart is in charge of how the chart is implemented which is a good candidate for another project refactor in the future.

Other areas to look into for further reengineering and refactoring projects

* Plots.\*
* Datasets.\*
* AbstractRenderer.java
* Renderer.\*
* JFreeChart.java

# Appendix 1

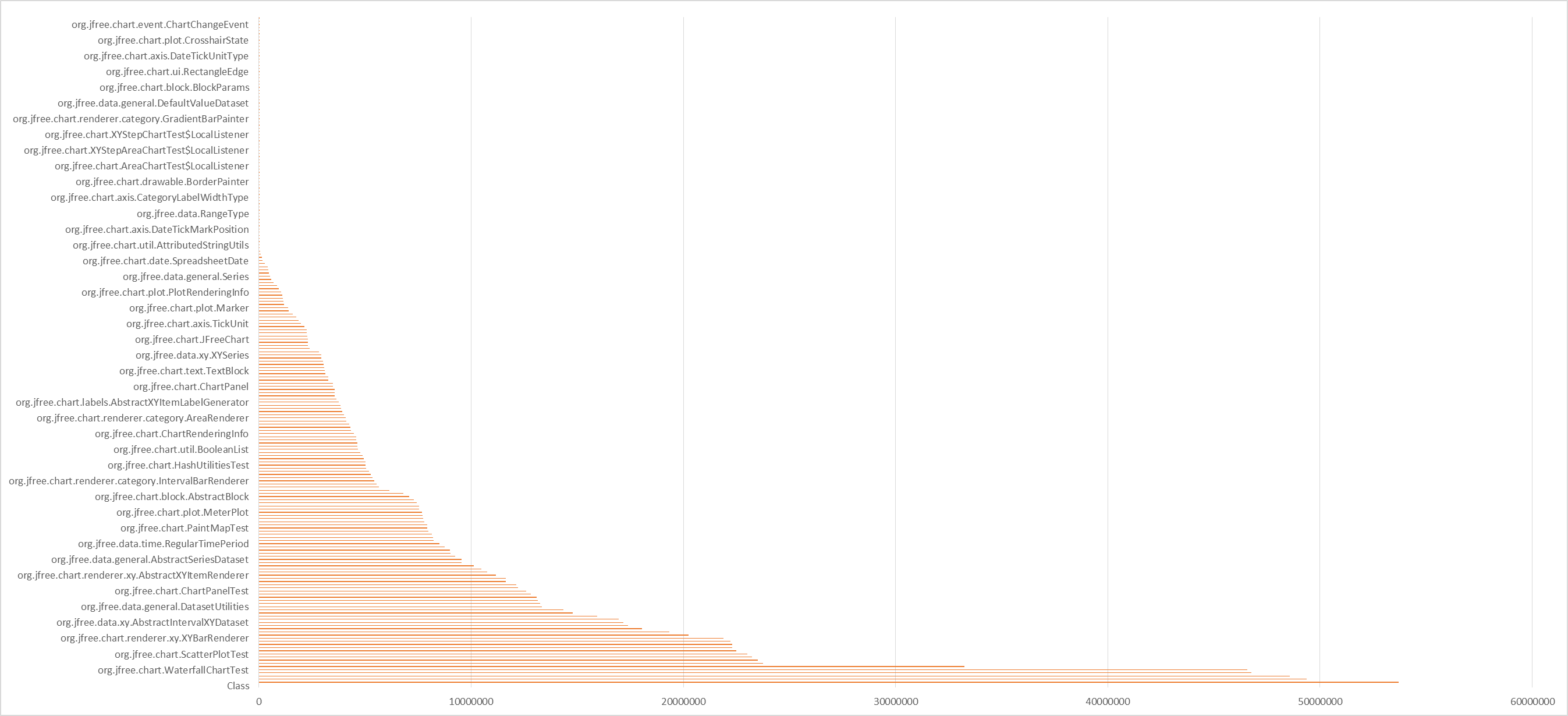
These graphs display the initial dynamic analysis on the JFreeChart project.

## Appendix 1.1



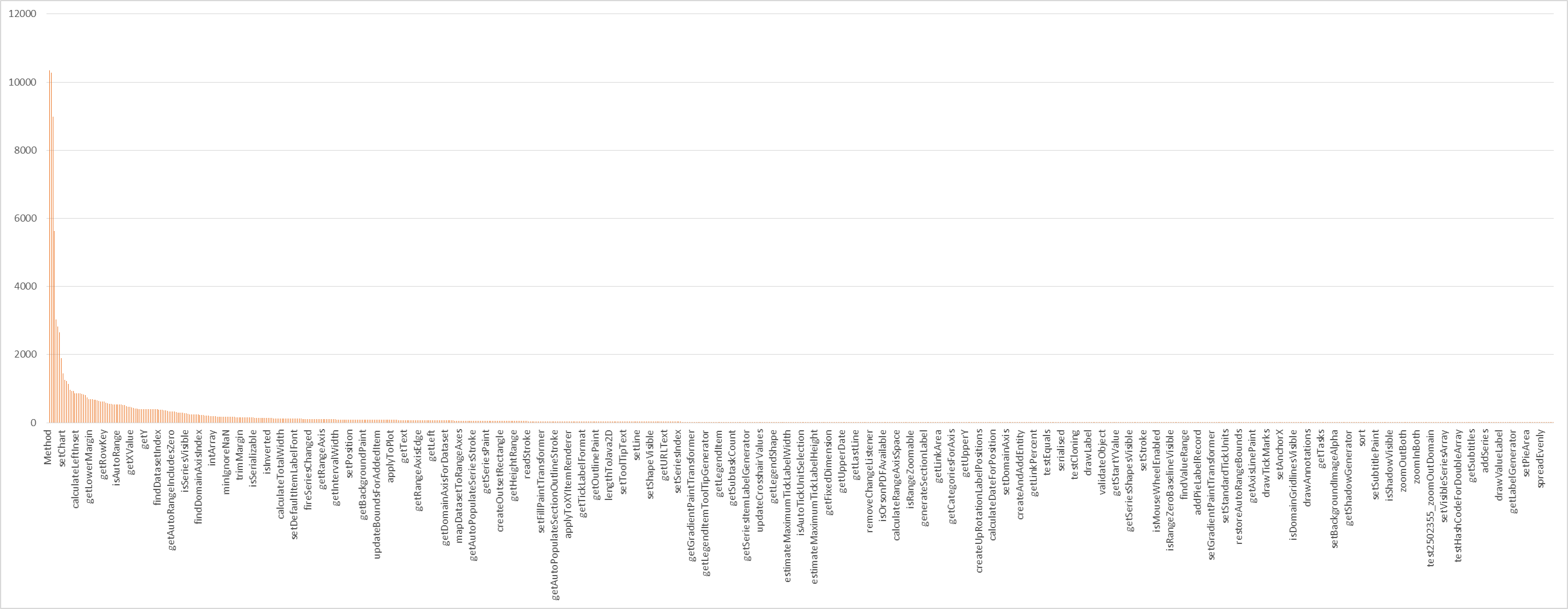
**Graph - Class Count**

## Appendix 1.2



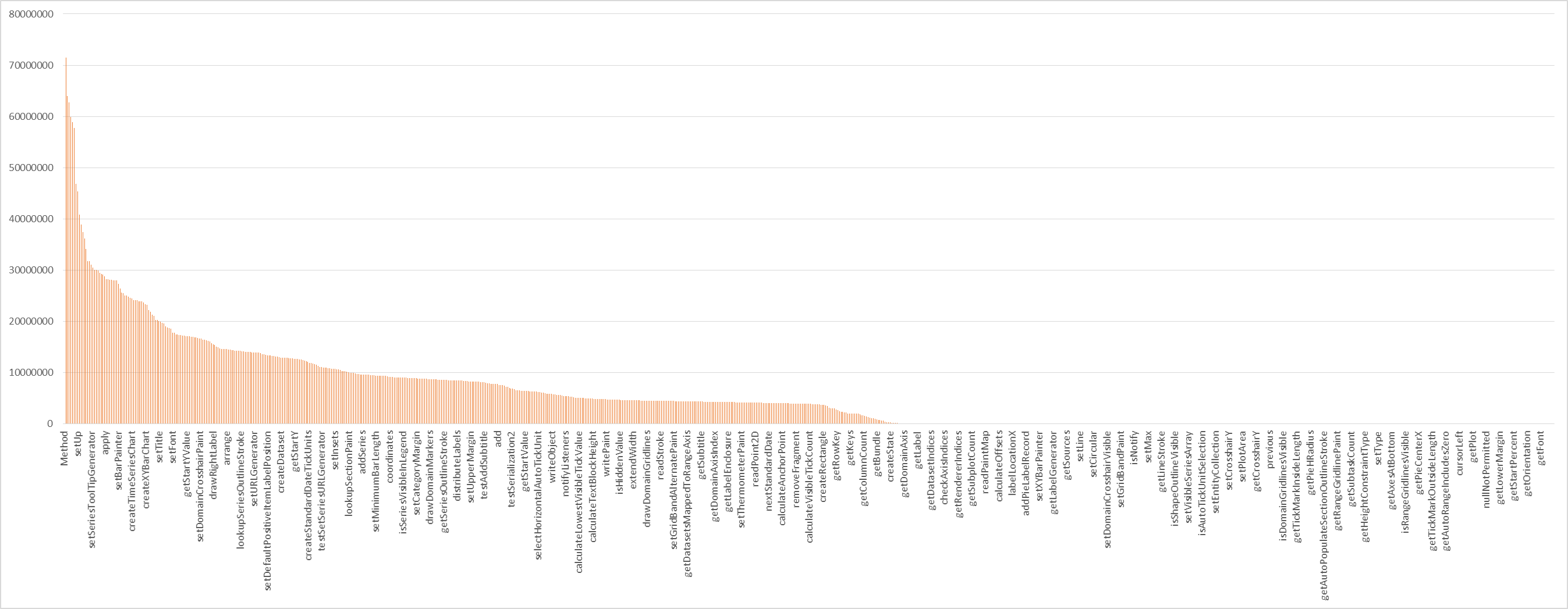
**Graph - Class Times**

### Appendix 1.3



**Graph - Method Count**

## Appendix 1.4

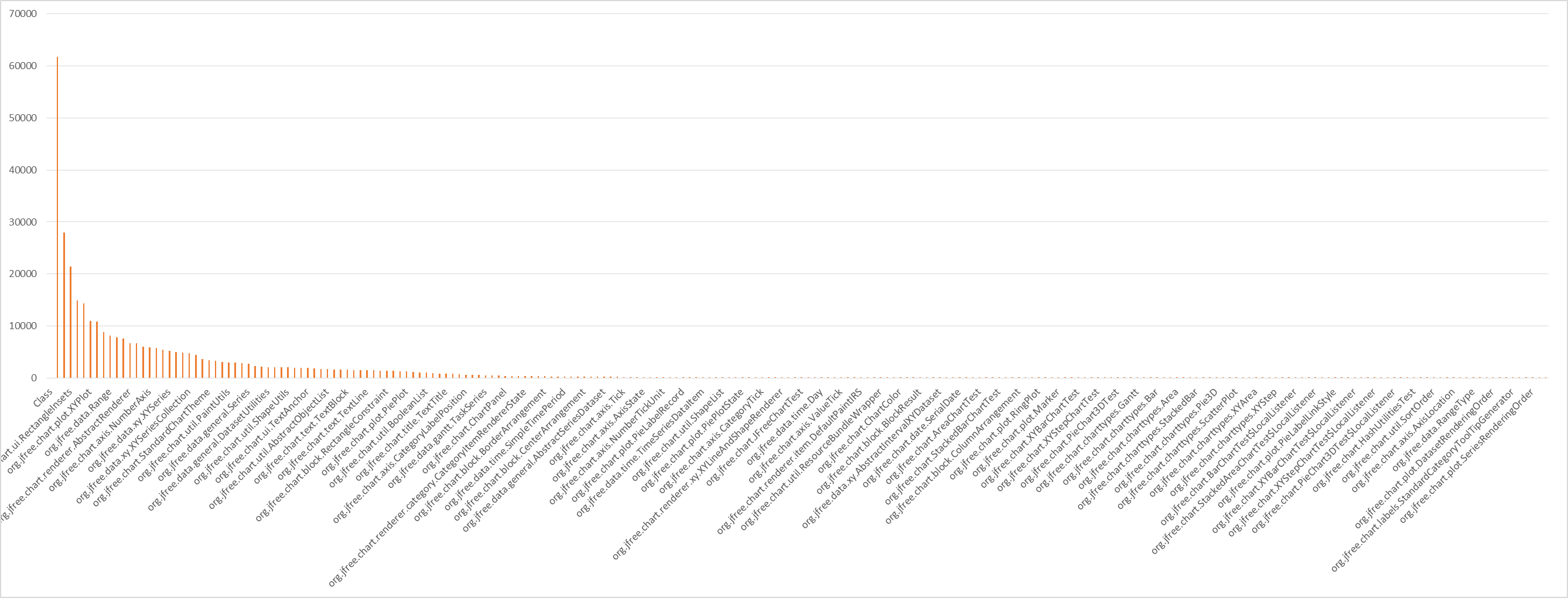


**Graph - Method Times**

# Appendix 2

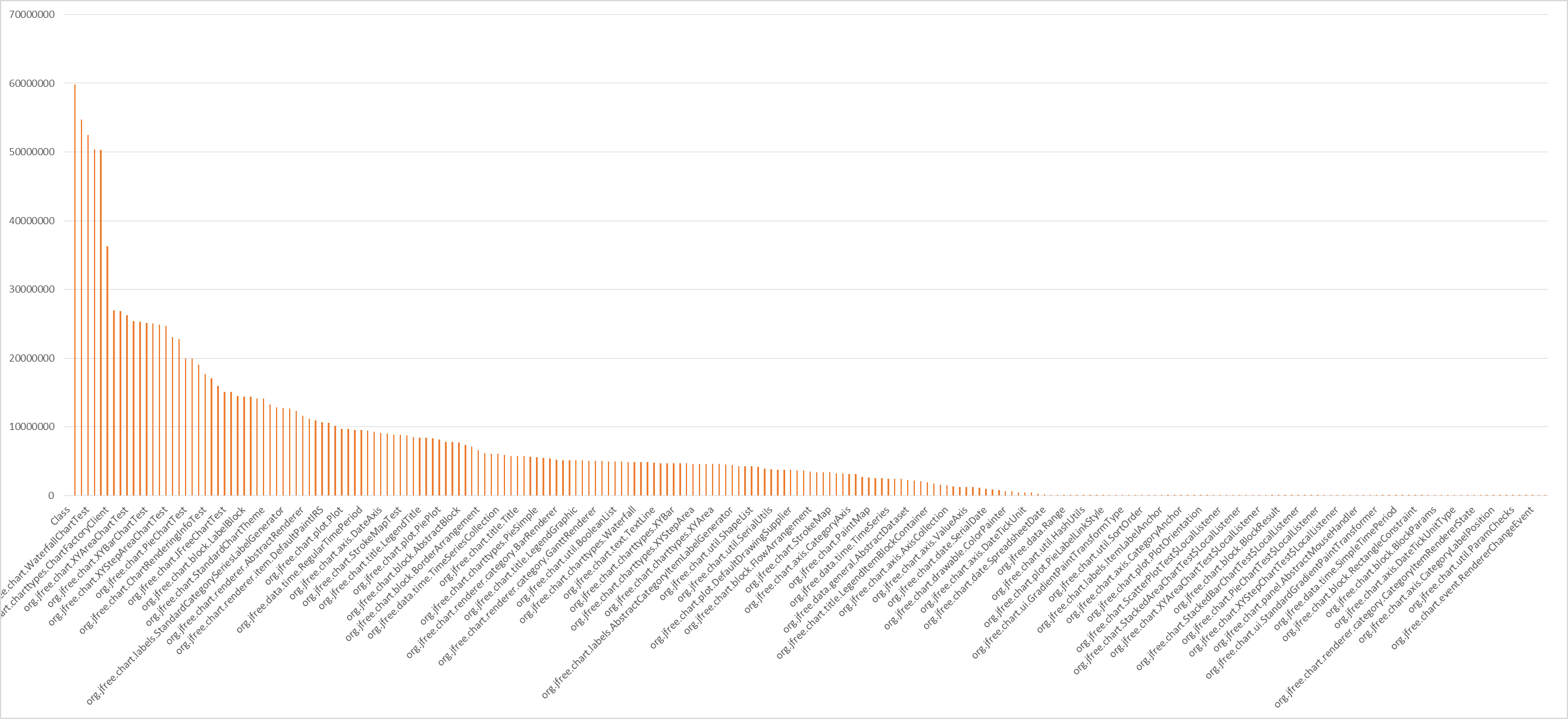
These graphs were generated after the project was refactored to show any differences on the dynamic analysis.

## Appendix 2.1



**Graph - Refactored Class Count**

# Appendix 2.2

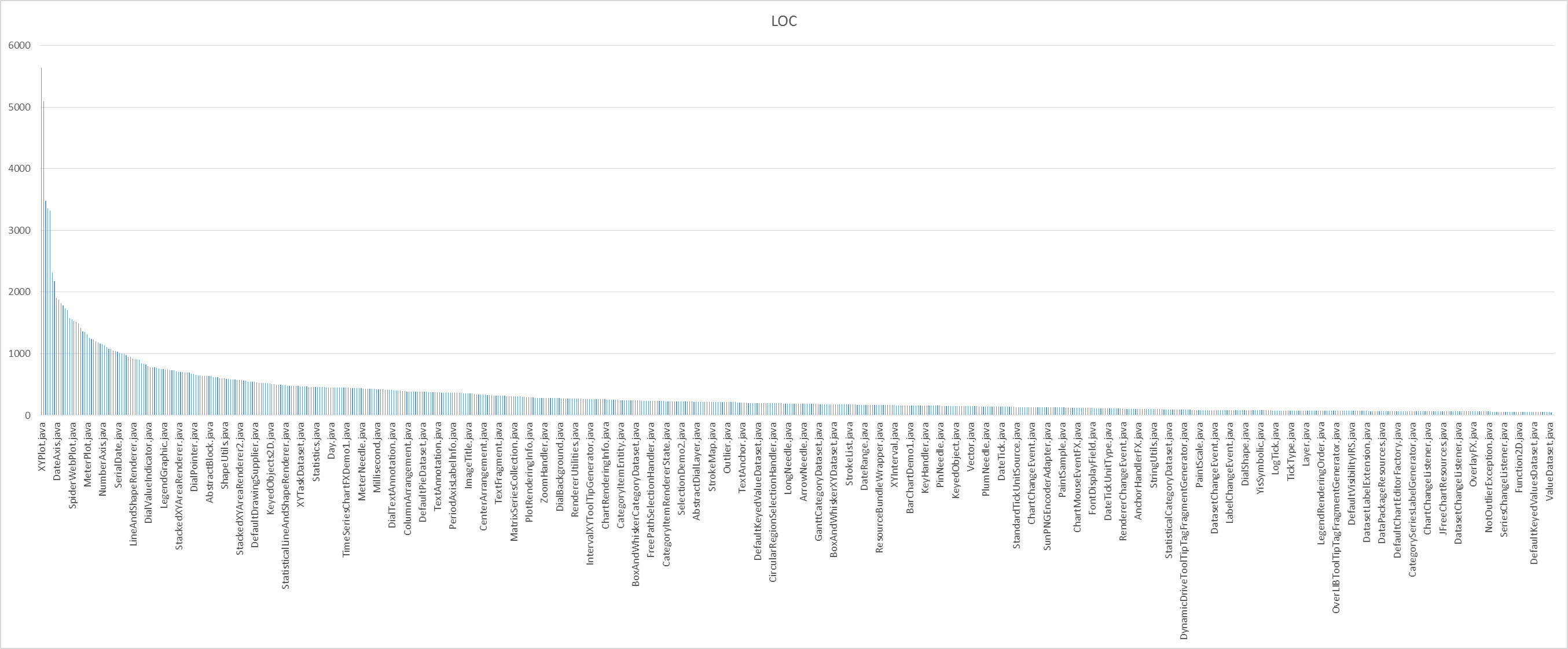


**Graph - Refactored Class Times**

# Appendix 3

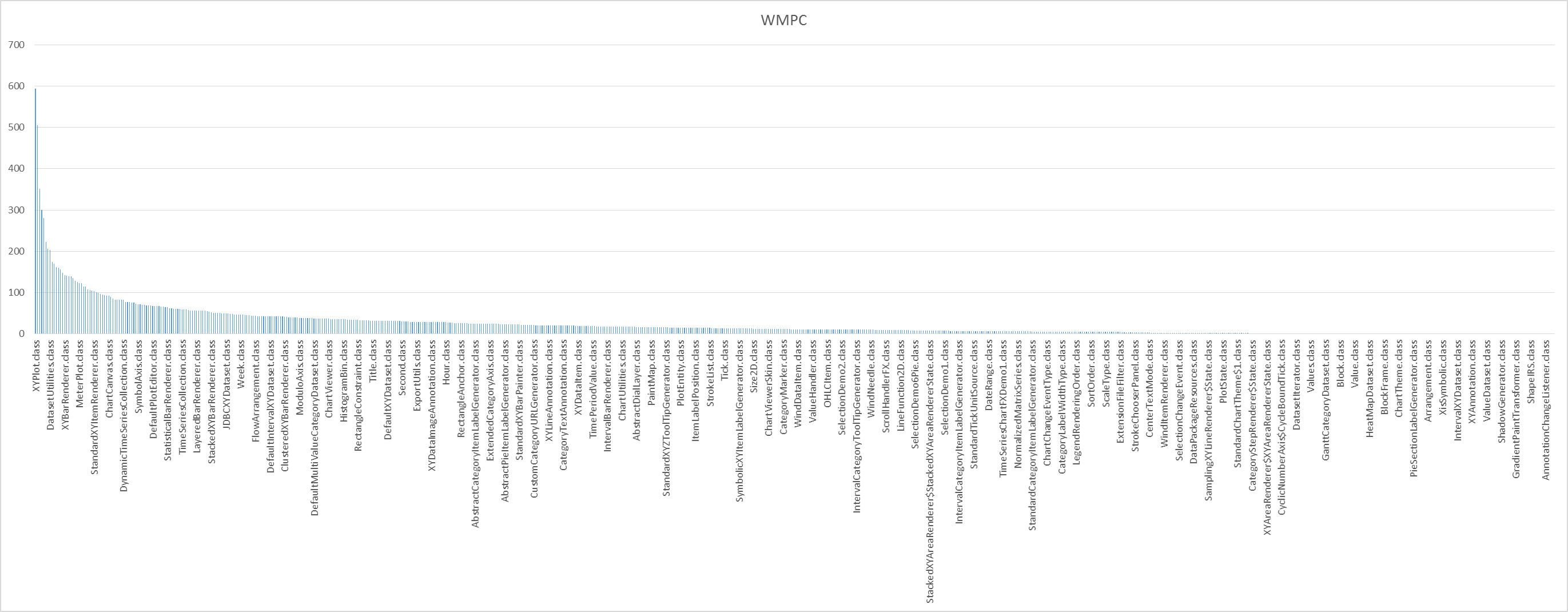
These graphs show the static analysis that was done on the initial project analysis.

## Appendix 3.1



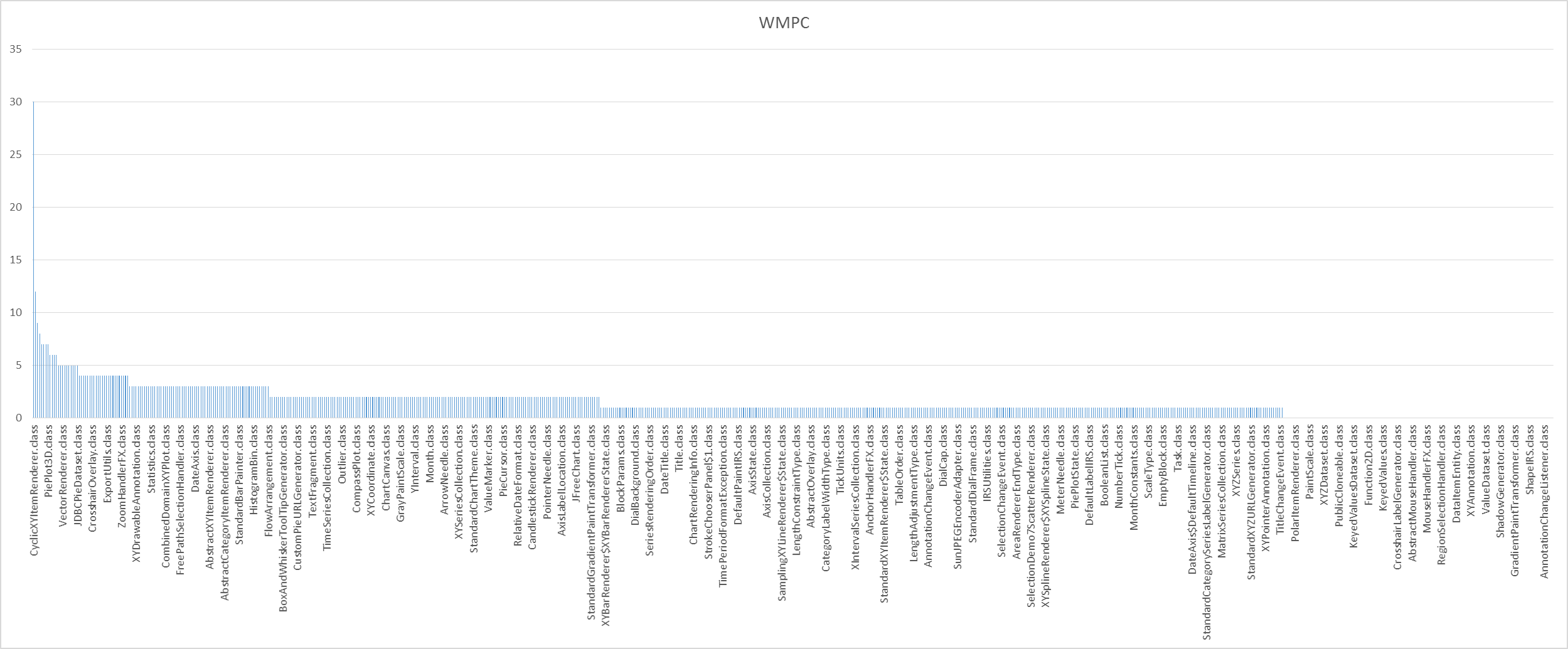
**Graph - Lines of Code**

## Appendix 3.2



**Graph - Weighted Methods Per Class**

## Appendix 3.3

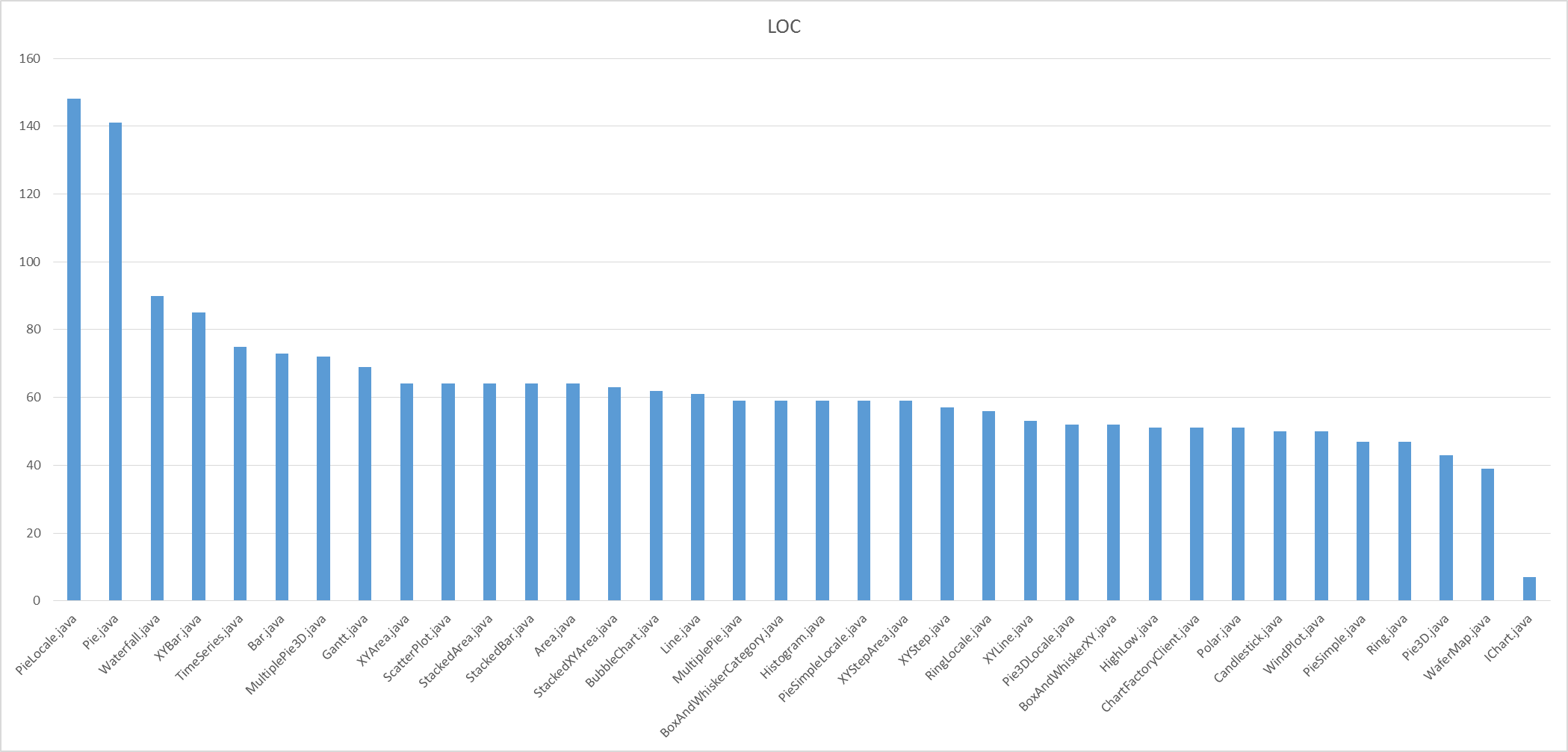


**Graph - Weighted Methods Per Class Average**

# Appendix 4

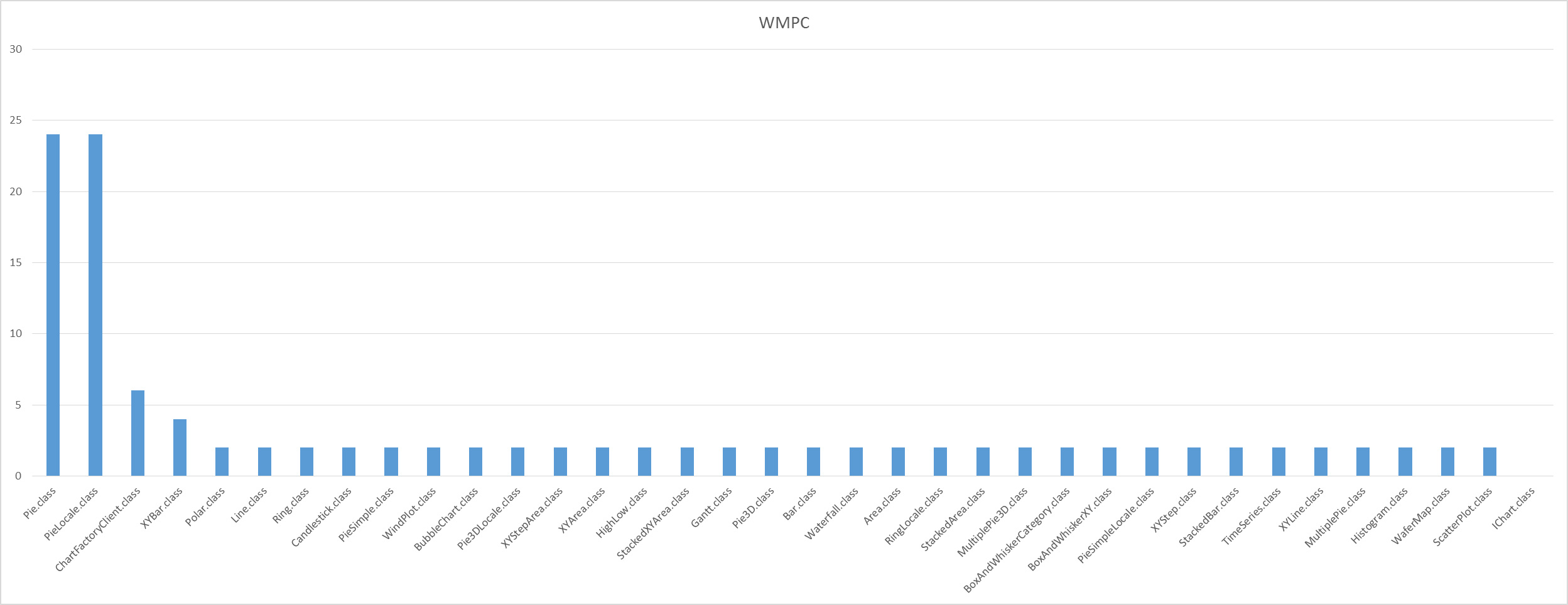
These graphs show the static analysis that was done on the project after refactoring

## Appendix 4.1



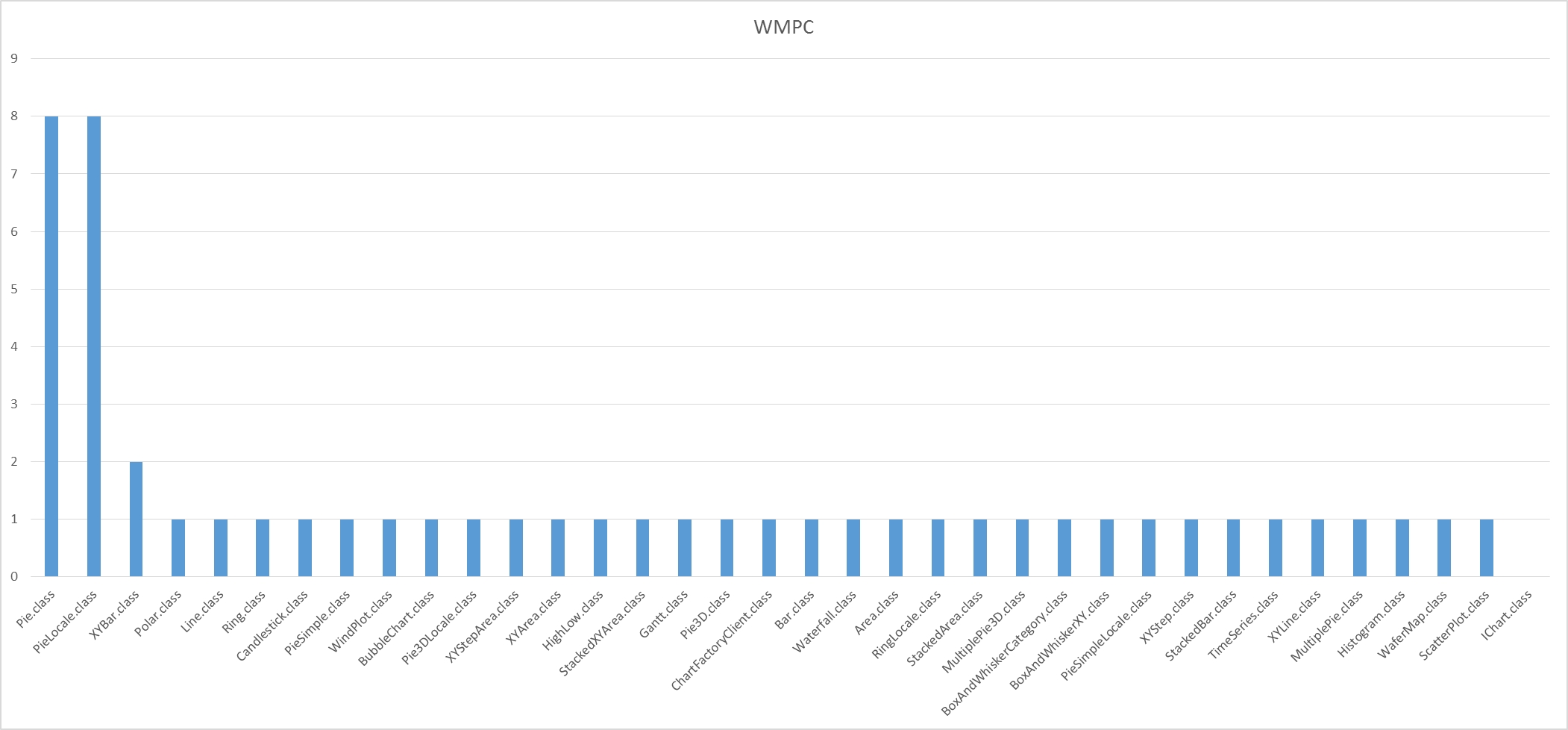
**Graph - Refactored Lines of Code**

## Appendix 4.2



**Graph - Refactored Weighted Methods Per Class**

## Appendix 4.3



**Graph - Refactored Weighted Methods Per Class Average**