|  |
| --- |
| MSC SOFTWARE DEVELOPMMENT |
| Advanced Object Technology |
| Assignment 1 |
|  |
| **Jimmy Collins – R00145569 – Jimmy.Collins@mycit.ie** |
| **3/20/2017** |

|  |
| --- |
|  |

Contents

[Section 1 – Introduction 3](#_Toc477704513)

[Section 2 – Demonstrated Use of Object Orientated Concepts 5](#_Toc477704514)

[Reuse 5](#_Toc477704515)

[Encapsulation 5](#_Toc477704516)

[Abstraction 6](#_Toc477704517)

[Inheritance 6](#_Toc477704518)

[Overloading 7](#_Toc477704519)

[Dynamic Binding 8](#_Toc477704520)

[Nested Class 8](#_Toc477704521)

[Inner Class 9](#_Toc477704522)

[Abstract Class 10](#_Toc477704523)

[Generic Method 10](#_Toc477704524)

[Generic Class 11](#_Toc477704525)

[Bounding 11](#_Toc477704526)

[Custom Exception Handler Type 11](#_Toc477704527)

[Section 3 – Usage of Design Patterns 12](#_Toc477704528)

[Strategy Pattern 12](#_Toc477704529)

[Observer Pattern 13](#_Toc477704530)

[Factory Pattern 14](#_Toc477704531)

[Section 4 – System UML Diagram 15](#_Toc477704532)

[Section 5 – Screen Captures of Most Impressive Code Snippets 16](#_Toc477704533)

[Generic Method (Utility.java) 16](#_Toc477704534)

[File Parsing Logic (BinetFile.java) 17](#_Toc477704535)

[Clean & Simple way to Support Different Display Strategies (PieChartStrategy.java shown) 18](#_Toc477704536)

[Section 6 – Evaluation of Work 19](#_Toc477704537)

[Appendix 1 – Code Commit Analysis 21](#_Toc477704538)

[Contributions 21](#_Toc477704539)

[Code Frequency 22](#_Toc477704540)

## Section 1 – Introduction

The application that I have developed can be used to parse net flow files and present some basic statistics.

Currently supported statistics are:

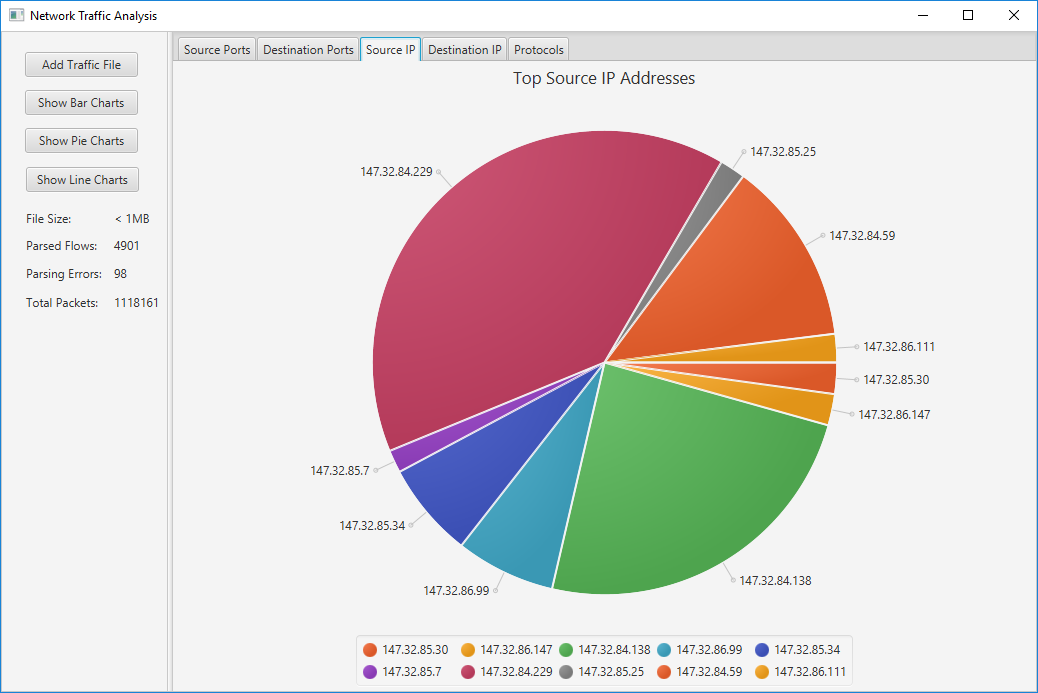
* Top Source Ports
* Top Destination Ports
* Top Source IP Addresses
* Top Destination IP Addresses
* Top Protocols.

The data can be presented to the user as pie charts, bar charts or line graphs. Data can be transformed to these formats dynamically at runtime. The Strategy pattern is used to accomplish this.

While a file is parsing, there is a panel on the left of the user interface that allows the user to monitor how many flows have been parsed, any parsing errors that have occurred, and the total number of packets being parsed. The dynamic updating of this panel while parsing is in progress is implemented using the Observer pattern as described later.

The rest of this document describes the specific requirements that have been implemented as per the specification.

The full source code has been submitted as per the specification. Note that this is a NetBeans project due to my personal preference for that over Eclipse.



*Figure 1: The main application user interface.*

## Section 2 – Demonstrated Use of Object Orientated Concepts

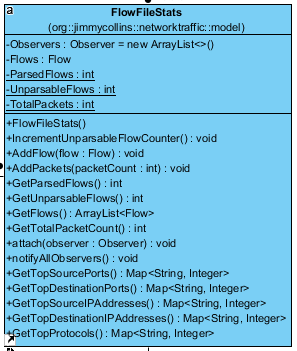
### Reuse

This is demonstrated by the LogUtil class. This is an object responsible for providing logging/alerting functions. It’s used throughout the application to log things to the application log file, and to show user interface alerts to the user.

In implementing the Strategy, Observer and Factory design patterns, the application is also demonstrating Pattern reuse.

### Encapsulation

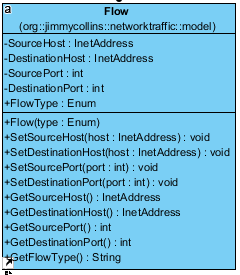
Encapsulation is demonstrated by the FlowFileStats class.



*Figure 2: The FlowFileStats class.*

### Abstraction

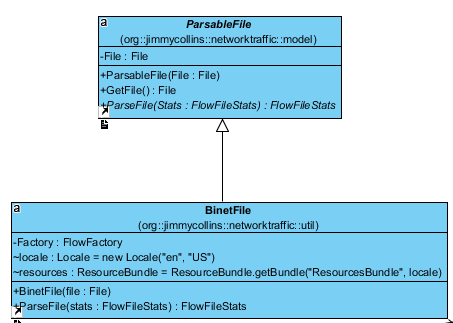
Abstraction is demonstrated by the Flow class.



*Figure 3: The Flow class.*

### Inheritance

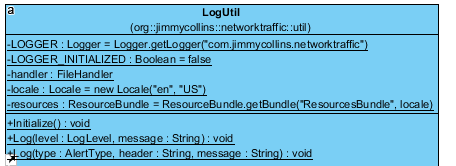
The concept of Inheritance is demonstrated by BinetFileParser extending ParsableFile. The parsing of other types of traffic files could be supported here in the future by introducing subsequent classes that extend ParsableFile and override the necessary functionality.



*Figure 4: Use of Inheritance*

### Overloading

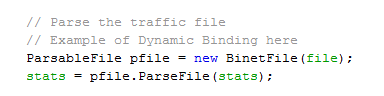
Function overloading is demonstrated by the logging class I use within the application. There are different logging functions that can be used depending on, for example, if a log is being written to the log file or to the user interface for display to the user.



*Figure 5: LogUtil Class*

### Dynamic Binding

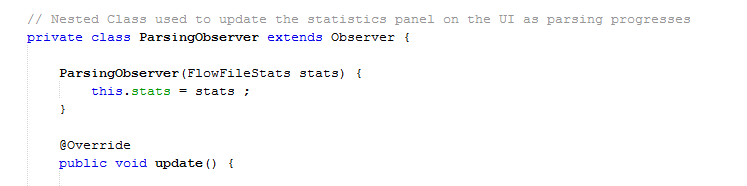
Dynamic Binding is illustrated in the logic I use to parse the traffic file. In the example in *Figure 6* below, the compiler cannot determine the type, because the instance of BinetFile is also an instance of ParsableFile, so the type is determined at run-time.



*Figure 6: Example of Dynamic Binding*

### Nested Class

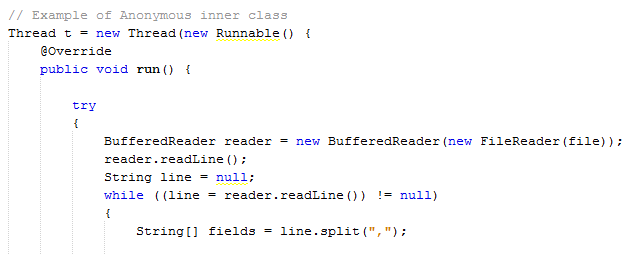
The ParsingObserver class (shown in the code snippet in *Figure 7* below) defined in MainUIController is an example of a Nested Class. This class is used as part of my implementation of the Observer pattern (see Section 2), and contains functionality for updating the user interface as parsing of a file progresses.



*Figure 7: Nested Class Example*

### Inner Class

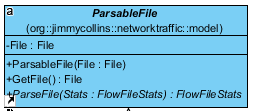
There are a couple of examples of inner classes. Shown in *Figure 8* below is a snippet that illustrates the usage of an anonymous inner class in the function I use to parse the traffic file.



*Figure 8: Example of Anonymous Inner Class*

### Abstract Class

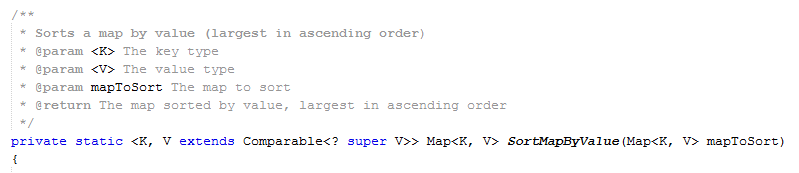
A demonstrated used of an Abstract class is ParsableFile. This defines an abstract class with the fields and methods relevant to parsing a file.



*Figure 9: ParsableFile Class*

### Generic Method

Usage of a generic method is demonstrated by SortMapByValue contained with the static Utility class. See Figure 10 below for a snippet of the method signature.



*Figure 10: Snippet from Generic method SortMapByValue*

### Generic Class

I do not create a new generic class in the application currently, but I do make extensive use of the ArrayList class, which is demonstrating usage of a generic class.

### Bounding

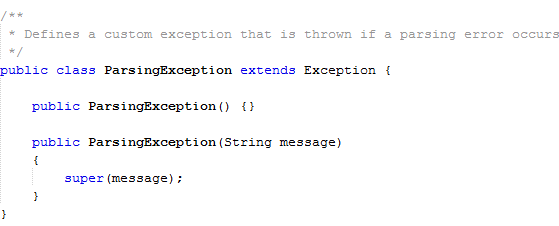
In my generic method, I’m demonstrating the use of bounded type parameters to restrict the type of objects that can be used in the parameterized type, to ensure that these objects are Comparables.



*Figure 11: Example of a Bounded Type Parameter*

### Custom Exception Handler Type

I have defined a custom exception used when parsing the net flow file. See Figure 12 below for a code snippet illustrating this.



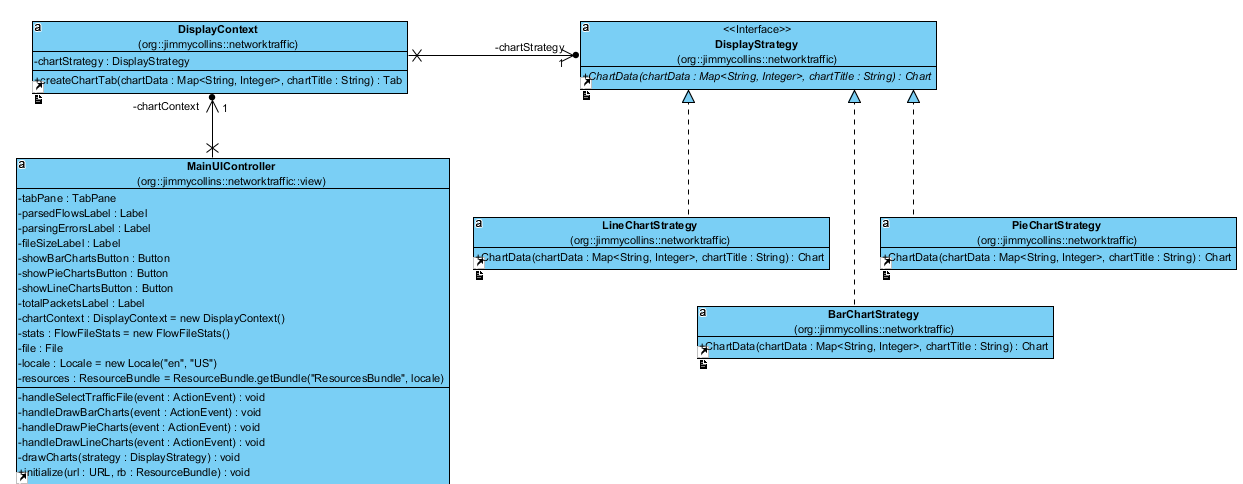
*Figure 12: Custom Exception*

## Section 3 – Usage of Design Patterns

### Strategy Pattern

This is used to allow choosing of different chart types to show the net flow data – pie charts, bar charts and line charts are supported out-of-the-box.

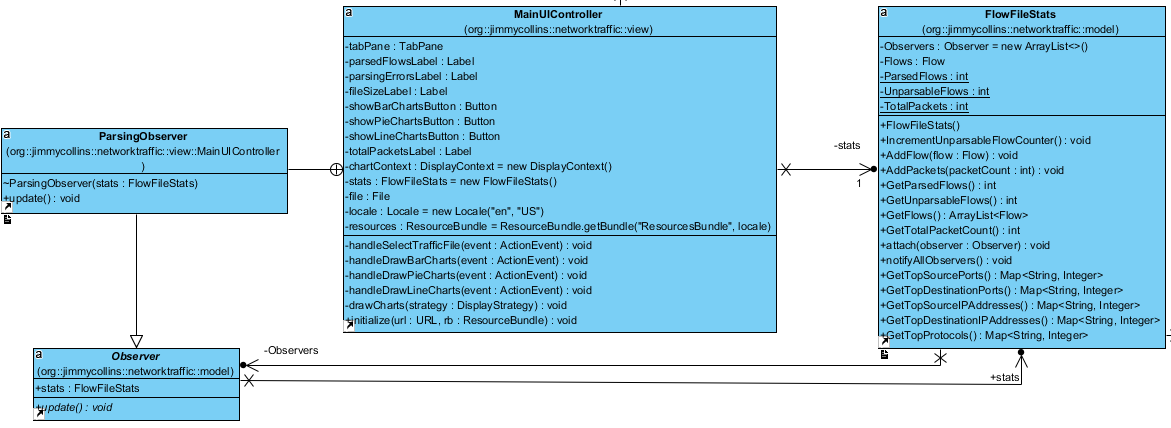
Support for other charts could be added easily by adding another class and implementing DisplayStrategy.



### Observer Pattern

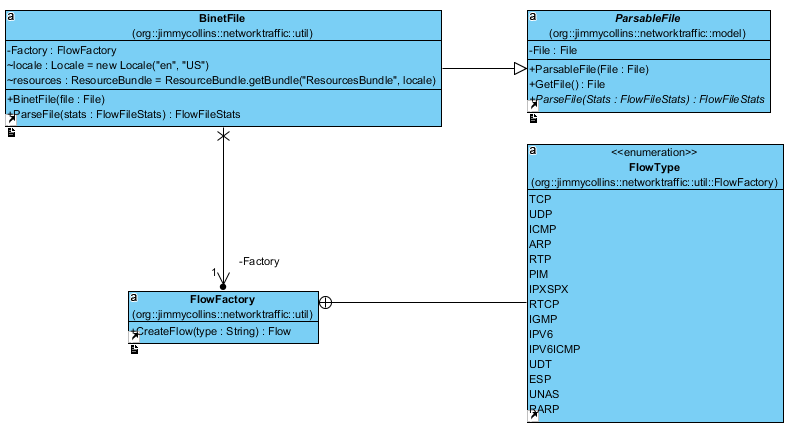
The Observer pattern is used to monitor file parsing information as a file is being parsed, and display a summary on the user interface.

FlowFileStats is Observable, and certain fields are observed by MainUIController and updated on the user interface as they change.

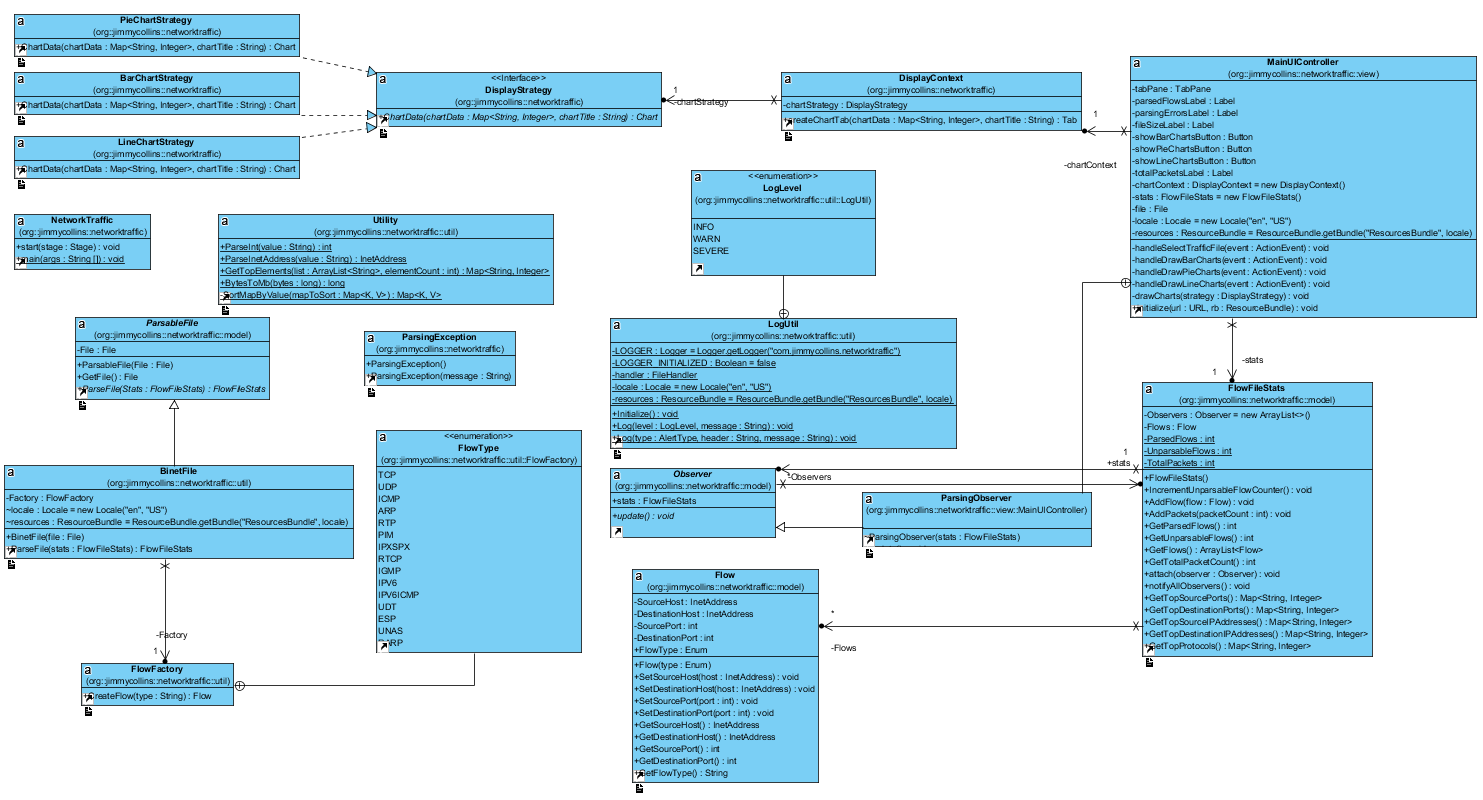


### Factory Pattern

The Factory pattern is used in the application when parsing the file. A class called FlowFactory is used to set the type field on the Flow object being created and return a new Flow object.

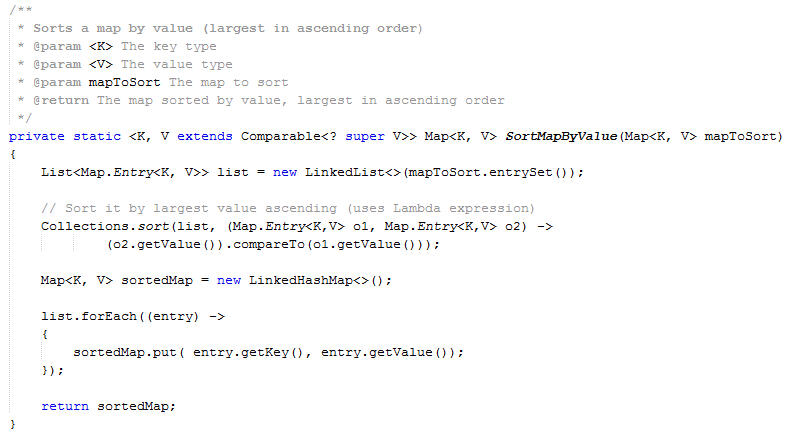


## Section 4 – System UML Diagram



## Section 5 – Screen Captures of Most Impressive Code Snippets

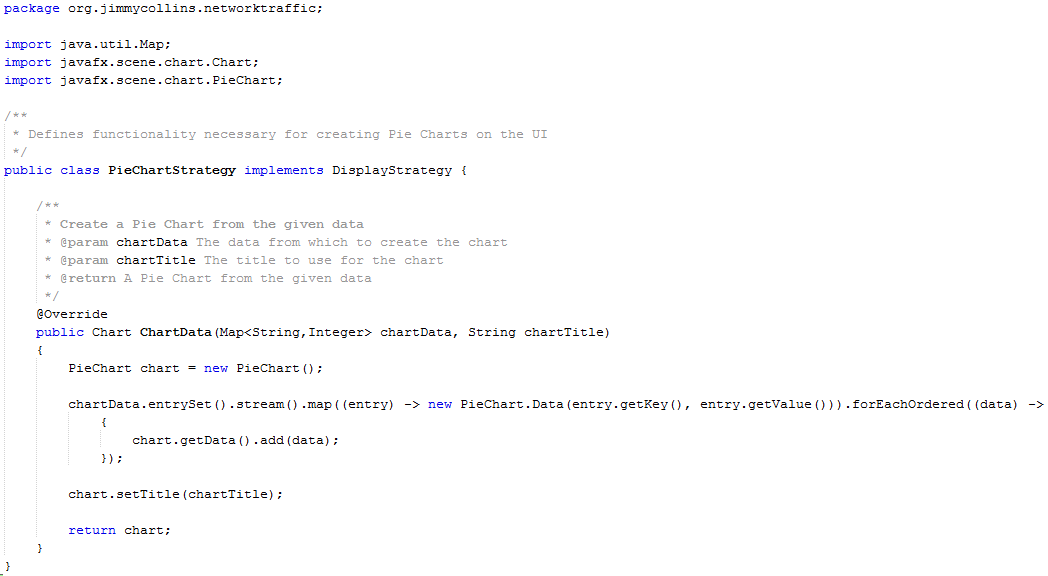
### Generic Method (Utility.java)



### File Parsing Logic (BinetFile.java)



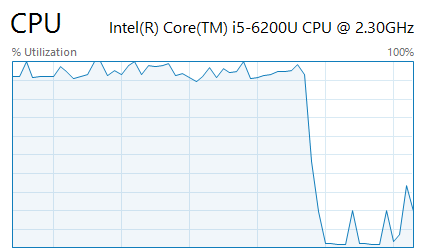
### Clean & Simple way to Support Different Display Strategies (PieChartStrategy.java shown)



## Section 6 – Evaluation of Work

I think overall the application meets the requirements set out; however, given some more time, there are a few things that I would like to improve in the design and code.

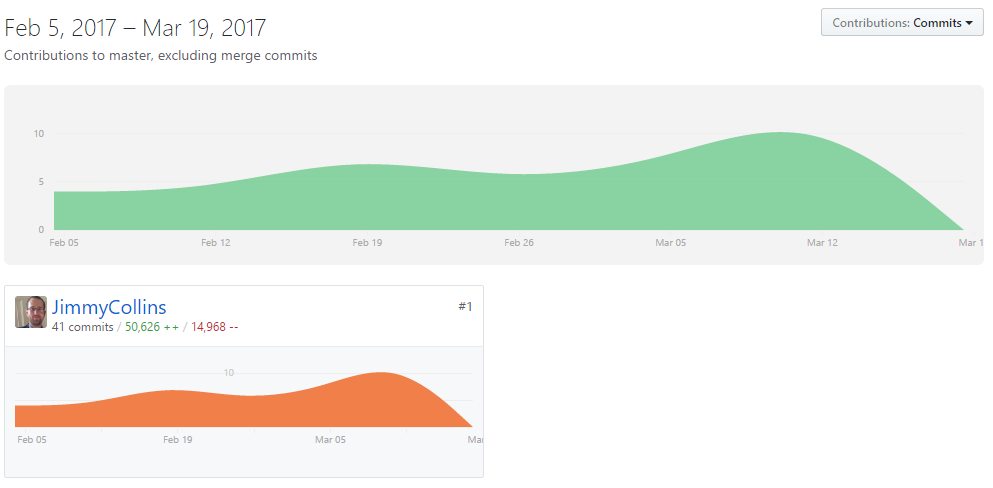
* Performance when parsing large files – when parsing large net flow files, the performance of the application could be improved. When tested with the 368MB file provided with the requirements, CPU utilization is high, and parsing can take upwards of two minutes to complete, depending on the machine specification (although worth noting there are nearly 3 million flows in this file). The below image illustrates this, the significant decrease in CPU utilization seen in the below graph happens immediately after parsing completes.



* Better data model – I think the design of the objects I use to parse the flow from the file could be better. Currently I use two, one that represent a single flow from the file, (I store a list of these), and one that represents high level statistics that are used to populate the charts on the user interface.
* Better implementation of Factory pattern – I am using a very simple implementation of the Factory pattern, this could be improved in a future version.
* For source and destination IP addresses, I experimented getting the FQDN in order to show it in the UI, or at least allow me to categorize the traffic flows better in terms of where they are originating etc. I found that using the built-in functions from the InetAddress class to do this significantly increased the parsing time (sometimes by 5x). In a future version I may just add this to the top 10 IP’s in the source and destination IP address charts, or I may design some way of persisting this information so that it only needs to be looked up once.
* In the next version, I would like to include more useful charts. One thing I think would be interesting to do is to see if there is a way to analyse the net flow files to detect Botnet activity.
* There are a few internationalization issues scattered in the code (e.g. the use of ‘*MB*’ is hardcoded in MainController.java). Having spent a few years fixing these types of issues in code due to developers not doing it in the first place, I’d like my code to be free of these types of issues.

## Appendix 1 – Code Commit Analysis

### Contributions



### Code Frequency

