X-Y Plotter

by

Connor McGarty, cmcgarty

Report Submitted on 31 May 2018

EE 333 – Engineering Programming Using Objects

Department of Electrical and Computer Engineering
The University of Alabama at Birmingham

Abstract

A software design problem was provided to the developer to demonstrate the complete process of an object-oriented software solution design. An X-Y Plotter system was to be designed and implemented that reads numerical data from Comma Separated Value (CSV) files and transforms the data using modular filter components into a graphical representation in the form of a Portable Network Graphics (PNG) file. The project was defined in full, examining constraints, goals, standards, and potential features. The chosen design was explained, including the use of Unified Modeling Language (UML) diagrams, and design alternatives were discussed. The process of finding supplemental information to assist implementation as well as debugging procedure was examined. The results of the design and implementation of the system were presented and discussed.

Contents

INTRODUCTION

The objective of this report is to demonstrate a complete object-oriented design and demonstration of a Java program that simulates an X-Y plotter. Software design is a complex process that often involves large teams coordinating efforts across multiple disciplines in order to meet requirements for clients, management, or other end users. As a result, great care is taken in the early stages of project development in order to have a clear, well-defined process developers can use as a guide along the way, as well as documentation of system information, implementation details, and mistakes made to learn from in the future. This project should provide a small scale example of the process taken by software developers in taking a product from problem requirements to a functioning prototype.

PROJECT DEFINITION

The project consisted of producing an object-oriented design and implementation of a program in the Java programming language that simulates an X-Y plotter. The requested program takes input in the form of a comma-separated table (.csv file format) where each column in the file is a data stream that can be modifed by filters the user chooses. Filters can be layered to take input from other filters, as well as output to the input of other filters. After filtering, two data streams are left, one mapping to the X-axis of the plot and the other mapping to the Y-axis of the plot. A Portable Network Graphics (.png) file is produced by the program displaying the plot resulting from the filtered input data.

Constraints

The following project constraints were identified.

- CO-01: The program shall be implemented in the Java programming language.
- CO-02: The input data to the program shall be supplied in the form of a comma-separated values (.csv) table, in which each column is a separate data stream.
- CO-03: The modeling classes shall not implement program input or output.
- CO-04: At least **two filters shall be implemented.**
- CO-05: At the end of the filtering stage of data processing, two data streams shall be left for passing to the plot.
- CO-06: Data streams shall be indexed by column number, beginning from one (1).
- CO-07: The program shall output a plot in the Portable Network Graphics (.png) file format.

- CO-08: The filters shall have the capability of layering on top of one another, in such a way that a filter can accept input from the output of other filters.
- CO-09: The X and Y ranges of the plot's map shall be definable by the user.
- CO-10: The size of the images output by the plotter shall have a consistent resolution, and the datapoints drawn on the image shall be scaled to this resolution regardless of the horizontal and vertical ranges of the input data.

Goals

The following implementation goals were identifed by the developer during the design process. These goals were identifed to help guide the design and identify possible tradeoffs that were needed to be made to complete the project.

- GO-01: The program should not modify the original input data.
- GO-02: The program should be simple to use, with no configuration on the part of the user in order to install the program, nor to use the program.
- GO-03: The user should be able to select filter chains through prompts via the command line, as opposed to writing their own main class (see Could-07).
- GO-04: The program should print verbose progress messages during data stream proccessing.
- GO-05: The developer should attempt to implement most, if not all of the filters suggested by the program requirements (see CO-04).
- GO-06: The program should be tested by writing JUNit unit testing for all modeling classes.
- GO-07: The system should be modular, supporting an arbitrary construction of filter types in any order and any size.

Features

The desired program features were identified as follows.

- Must-01: Accept a .csv file as data input.
- Must-02: Ask the user for the range of the X and Y axes.
- Must-03: Output a .png plot.
- Must-04: Implement two (2) data stream filters.
- Should-01: Implement greater than two (2) data stream filters.
- Should-02: Provide verbose program progress messages to the user as data is being processed.

- Could-01: Implement all of the filters supplied in the project requirements list.
- Could-02: Write JUnit unit tests for all modeling classes.
- Could-03: Provide Java exceptions for expected possible error handling (however, would be better to use proper condition checking for flow control).
- Could-04: Allow the user to configure the look (color, line density, etc.) of the outputted plot.
- Could-05: Output image formats other than .png files.
- Could-06: Allow the user to save custom filter chains for reuse with different data sets.
- Could-07: Signal chains be completely configurable as the program runs by prompting the user: no coding required by the user.
- Won't-01: Implement a GUI interface.

Applicable Standards

The standards listed below will be followed during implementation of the product.

- STD-01: Input data complies with RFC4180, a CSV, Comma Separated Values format definiton. [1]
- STD-02: Output data complies with the Portable Network Graphics image file format as specified in RFC2083. [2]
- STD-03: Code complied with Java SE8.

DESIGN

The object-oriented design process for the X-Y plotter began, after receipt of project, with formalizing the project requirements and formulating them into constraints that the developer used to begin to conceive of possible implementations. The main components of the project were split into phases: an input phase, a processing phase, and an output phase. The input phase would handle parsing the .csv format input file and constructing data streams based on this data. The processing phase would apply the implemented filters to the datastreams and handle passing the mutated data onwards to the next component of the chain. Finally, the output phase would take the final two data streams and produce the .png image. Once a rough conception of the project was understood, identification of key goals and features followed. Through the formalization of these guiding points of the project's design, more concrete ideas of implementation details began to materialize. Multiple classes and interfaces were brainstormed, and UML (Unified Modeling Language) class diagrams were instrumental in laying out the structure of class relationships and visualizing for refinement. The results of this process can be found in the Object Oriented Design section of this report.

Appropriate for Object Oriented Approach

An object-oriented approach proved to be an apt means of tackling the X-Y plotter problem. The system required a clear delineation between input, processing, and output processes, and the major one of those three, the processing stage, was particularly well-suited for OOD. Due to the fact that a filtering "toolbox" of multiple difierent components with some shared qualities, and the necessity of the components to be modular, as well as have the capability of coupling with other components, this particularly lent itself to modeling with objects, especially using interfaces in Java. Java's object modeling allows for easily creating complex data structures, which this design takes advantage of.

Design Decisions

During the design process, a few possible design issues were encountered. The following section details those issues, design alternatives that were considered, which alternative was selected, and why.

- AL-01: Implementing GO-03/Could-07 proves to be too difficult, tedious, time-consuming, etc., or the implementation result is too unwieldy to use via the command line.
- **AL-01A: Completely scrap the command line user-interface and instead require user to provide their own main classes to configure a filter chain. This could not affect modeling classes: keep in mind this alternative when writing implementation in case this issue comes up.**
- AL-01B: Allow the user the *choice* whether or not to code their own main class, or to configure a filter chain from the command line via a user interface.

Alternative **AL-01A** was selected in the case of being unable to deliver a user interface in time to provide a more layperson-friendly method of using the plotter system. This proved to come up; other projects competed for the attention of this one and as a result, not as much work was done as was initially expected. Since the miminmum requirement was to have the system be configured using main classes, this alternative was selected.

- AL-02: I/O must not be a part of modeling classes.
- AL-02A: Design separate classes specifically for handling reading from.
- **AL-02B: Handle I/O in main class: choosing a file to pass to a parsing class and image writing should be implemented in the main class.**

Initially, the writing code for writing the BufferedImage of the plotter's output was contained inside the *plot()* method of the Plotter class. This was revised along the lines of **AL-02B**, such that the *plot()* method now returns a BufferedImage object, and the main classes can handle writing to a file—alternatively, the BufferedImages could be further manipulated in other projects, providing more modularity to the use of the modeling classes. This complies with constraint CO-03.

- AL-03: Implementing filter chain process, which filter is hooked up to which and in what order, etc., becomes too messy.
- AL-03A: Rework data stream flow.
- **AL-03B: Create a Chainable interface, add methods that implement a tree structure that can hook up filters, data streams, and a plotter.**
- AL-03B: Implement a stack-like structre for handling layered filters.

The chosen design for managing the connection of filters, data streams, and a plotting class ultimately was a tree-like methodology, where the plotting object forms the root of the tree, filter objects are connected in series to the Plotter, and arbitary length of the chain is possible, so long as the leaves of the tree structure are DataStream objects. This design, which supports the entire modularity of the system (see GO-07)of the system, proved to be extremely capable at handling the constraints and desired features of the system.

Object Oriented Design

The X-Y plotter system consists of a StreamSet class that handles parsing of the input data; the fields that make up each column of the file are stored in a DataStream object that contain an ArrayList of floating-point numbers (Double objects). The DataStream objects stored in the StreamSet ArrayList class variable "streams," where the array index corresponds to the column in the .csv file that data came from (the 0-ith index of the list is assigned "null" to meet constraint CO-06). The contents of these datastreams can then be manipulated by objects that implement the Filter interface. Filter objects implement an "apply" procedure that operates on the input and passes along the modified data as output. The implementation contains two Filter objects, a ScalerFilter class and an AdderFilter. The ScalerFilter takes two floating-point numbers upon construction, a base value and a gain value, and applies the formula $x_{out} = x_{in} * gain + base$ to the input data points. The AdderFilter calculates the arithmetic sum of two input data streams and produces a DataStream object that contains the list of these sums: $x_{out} = x_i + x_j$, where $i = j$.

In order to produce a graphical output of the results of the filter processing, a Plotter class requires two DataStream objects to feed into its X input and a Y input. These DataStream objects can come directly from the input file, using the *StreamSet.getStreams().get()* method, or as a result of the DataStream objects that are produced as a result of the *Filter.apply()* interface methods. A Plotter object on construction requires a minimum and maximum X and Y value; this is the range of points that will be visible on the output image. It is not necessary for these values to match the actual minimum and maximum values of the data set; only the points that fall within the supplied range will be visible. The Plotter object scales this supplied range to the 500x500 pixel resolution PNG output image, such that regardless whether the input data ranges from negative one to one or from negative one million to one million, the full plot will be visible on the image.

In order to chain together any number of filters and data streams in any sort of configuration, DataStream, Filter, and Plotter objects implement the *Chainable* interface, which

provides a *connectInput()* procedure and a *requestData()* procedure. Chainable objects are able to be linked together using their *connectInput()* methods into tree-like structures, where a Plotter forms the root of the tree and DataStream objects form the leaves. In between can be an arbitrary number of Filter objects that can have any Chainable object connected to their inputs, DataStream or other Filter objects. After the "tree" is constructed using the *connectInput()* methods, the Plotter object has its own connect method to wire Chainable objects to its X or Y inputs. When the Plotter's *plot()* method is called, the Plotter looks to its two inputs. If both of the inputs are DataStream objects, the Plotter iterates through the two streams, constructing a BufferedImage object and utilizing the 2DGraphics class to draw lines between successive pairs of points. The image is then written to a Portable Network Graphics file and the program terminates.

If either of the Plotter's inputs are Filter objects, then the Plotter must request the Filter to apply its operation on the Filter's own inputs before that Filter can pass along the output to the Plotter. The *requestData()* method handles this behavior. When a Filter object's *requestData()* method is called, it looks to its own inputs; if they are DataStream objects, the Filter's *apply()* method is called on the stream or streams and returns the result. If the input is itself another Filter, then the **input's** *requestData()* method is called, and then the Filter's *apply()* method is called, applying the operation to the received data stream. In this way, the Chainable tree makes recursive *requestData()* calls down, originating from the Plotter, through the Filter objects until only DataStream objects remain; the bottom-most filters apply their operation to the data and return that data to the object that its *requestData()* method was called from. The data works its way up the tree until the Plotter has two input DataStreams, and finally the image can be constructed.

This design provides for a modular system capable of any arbitrary filter chain. The interface-based design allows expansion to incorporate many different types of filters, and the Chainable interface allows both data and procedures to be treated interchangably; from the perspective of other Chainable objects, it does not matter if its connected input is data or a procedure that will produce data.

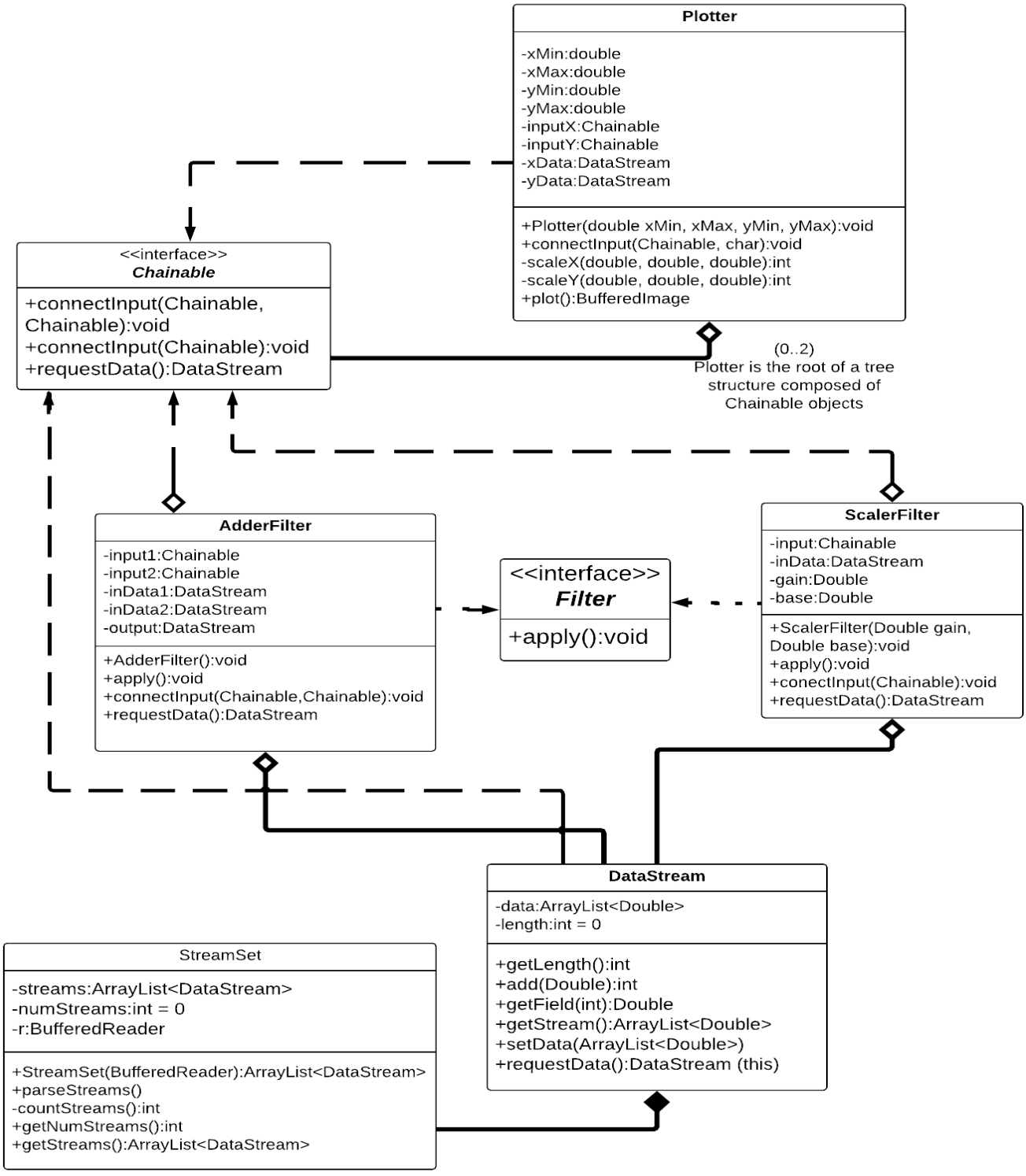

Figure 1: UML Class Diagram

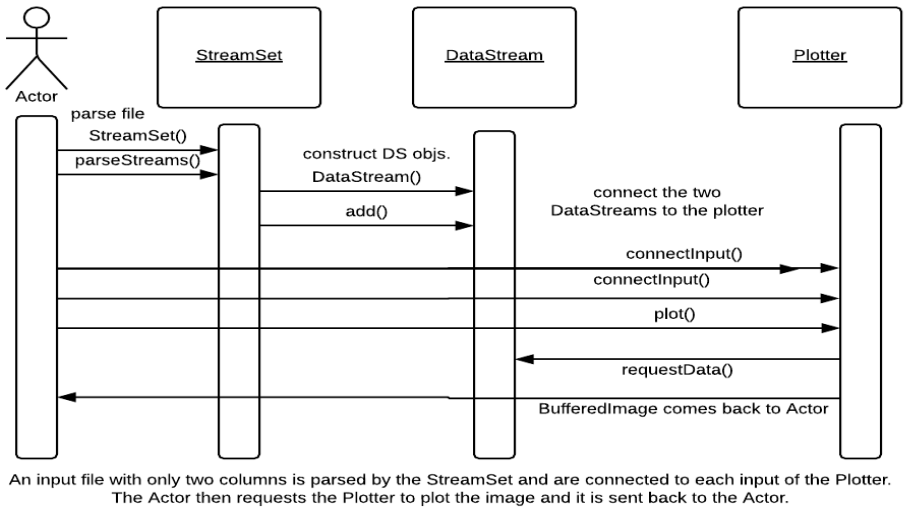

Figure 2: Interaction Diagram showing plotting with no filters.

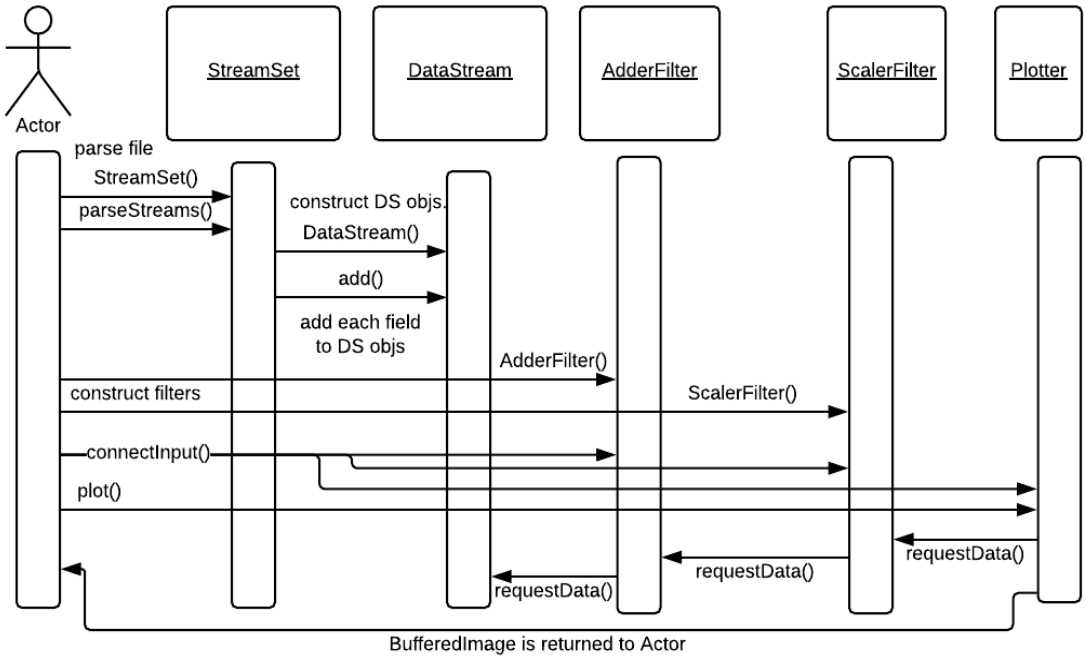

Figure 3: Interaction Diagram showing plotting with a filter chain.

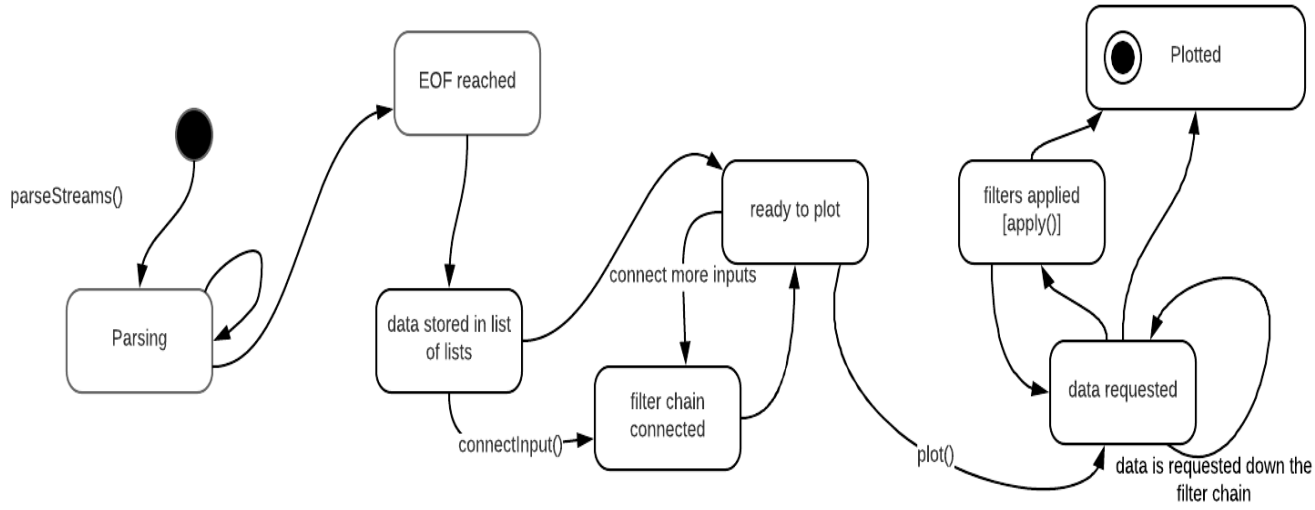

Figure 4: State diagram.

DISCOVERY AND USE OF ONLINE INFORMATION

One instance in which online research was required to aid the design of the prototype was in handling the problem of scaling an arbitrary range of data points to a constant image resolution (in this case 500 by 500 pixels). The output of the plot should be a constant size, but getting the entire range of data to fit in the image required scaling of some kind. The first resource consulted was the official Java SE8 documentation from Oracle [3] for the BufferedImage object. The only relevant result discovered was the *getScaledInstance()* method; however, this method returns an Image object which cannot be written to a file, instead a new BufferedImage must be constructed, and the Image object drawn onto the BufferedImage [4]. This implementation was attempted, but the result was not as expected. For data sets with few points, the initial, unscaled image would be very small (a data set with 5 rows of information 5x5 pixels for example). When this was scaled up to a 500x500 size image, the former 5 pixels would be scaled up to 100x100 size pixels drawn on a 500x500 image. This was not the desired behavior.

As a result, this method of scaling the using image IO operations was replaced with a simple algorithm that employs a formula that translates the original, unscaled data point into its equivalent point on a 500x500 grid. This way, the drawn lines are still one pixel in width, but the full plotted information is scaled to the size of the canvas; for example, a list of points of the line $y = x$ on the ranges $0 \leq x \leq 5$ and $0 \leq y \leq 5$ will be shown as a straight, thin line from the bottom left (graphical point (0, 500)) to the top right (graphical point (500, 0)) of the image.

Although the researched technique did not end up in the final implementation, the research was a learning experience, and also helped the developer come to the conclusion that an

original algorithm was necessary.

DEBUG

The process of developing an algorithm to scale an arbitrary number range of data points to a fixed range was inhibited by a few issues. The Netbeans debugger was a necessary tool in aiding this design.

In graphical applications, a pixel grid's origin is located at the upper left corner of the "canvas". The most upper left pixel of an image is referenced by the point (0, 0), while the most bottom right pixel of an image is referenced by the point (500, 500). However, the input data to the plotter system uses the tradition Cartesian coordinate system. Thus, imagining that the plot's output image is meant to display the first quadrant of a graph on a Cartesian system, the Cartesian origin (0, 0) must be translated to the graphical coordinate (0, 500). This task was simple enough for translating positive numbers, but became much more difficult if the user wishes to graph data ranging from -1000 to 1000 for example. Multiple attempts at an algorithm were brainstormed and implemented, but this case of negative coordinates translated to a positive-only coordinate system, that also utilizes a flipped vertical axis was too complex an issue to work out without systematic examination of each step of the algorithm. Trying to plot a data set and attempting to figure out what was wrong with it by solely inspecting the output image visually did not suffice for refining the algorithm—all that could be determined was that the full dataset was not being displayed as expected, but no specific information could be gleaned from just looking at the image, besides the fact that the scaling procedure was probably bugged, since a basic example with no scaling plotted as expected. Thus, the debugger was a necessary tool for the situation.

The debugging process was carried out as follows. First, a very basic data set was created for testing purposes. The graph of $y = x$ on $-10 \leq x \leq 10$ and $-10 \leq y \leq 10$ was chosen—this dataset consists of negative and positive points on both axes, and is also simple enough to visually inspect for correctness: the expected result was a straight diagonal line bisecting the image from the bottom left to the top right of the image. A breakpoint was set on the call to the Plotter's *plot()* method in the main class, and the program was executed by the Netbeans debugger. The program halted on the call to the plot method, and the 'Step In' command was selected to enter the body of the *plot()* method. Although the developer was nearly certain the issue was with the *scaleX()* and *scaleY()* private methods were the culprit of the issue (many different implementations of these methods were tried before this particular investigation, thus it was reasonably safe to assume that the code for constructing the image and graphic objects as well as getting points was sound), each line of the entire plotting method was stepped over. When the call to the Graphics2D object's *drawLine()* was made, the debugger's **Variables** window was used to check the values of the lastXCoord, lastYCoord, nextXCoord, and nextYCoord variables. These variables make up the points that the Graphics2D object draws a line between into the BufferedImage. These points were recorded by hand on paper and the "image" was hand drawn to see the points as the program would draw them, one at a time. The process of "playing computer", carrying out

the program instructions by hand, helps to get a more clear understand of what is happening to the state of the program over time. This was done for each datapoint until the program was finished executing.

At this point it became immediately clear that the the X component of the data points were being scaled properly—they were equally spaced across the image—however location of the points in the vertical direction were not; the spacing was correct, but the points were reflected across the horizontal axis. The bug was a simple typing error introduced from a quick revision to the scaling formula as it was being refined, as it was copied and pasted from the *scaleX()* method—the difference between the point to be scaled was being calculated from the minimum Y point provided on the Plotter's construction, as opposed to the maximum Y point. This was a simple mistake, but it proved to be somewhat difficult to discover without the use of the debugger, partly due to the fact that many different interations of a scaling algorithm had been tried, and it was easy to lose track of all the changes made and what effect they had on the output. Using the debugger to carefully inspect values as they were passed to and returned from the *scaleY()* method was critical in fixing this feature quickly.

RESULTS

The functionality of the system was tested against varied input data sets and filter configurations, from simple to somewhat more complex. Input data contained all positive data, and combined positive and negative data to test for handling proper scaling in each scenario. The first test main class utilized no filters, and a small data set containing only two data streams (see packaged file test1.csv). The expected result was a digonal line drawn from the bottom left to the top right of the image, which matches the output seen in Fig. 5.

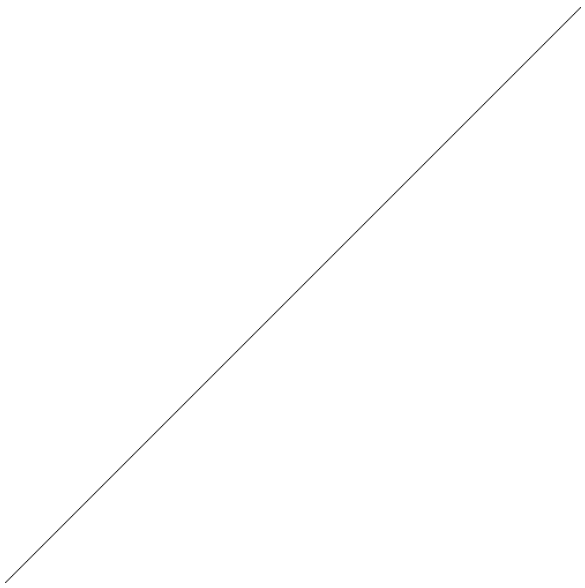

Figure 5: Output of TestMain1

The second test class tested the functionality of the AdderFilter. Two AdderFilters were used; the first summed the first two data streams in the input file (test2.csv), which expected to result in the integers from 1 to 20. The second summed the last two data streams in the input file, whose result should be the even integers from 0 to 20, and then the number 20 until the end of the file. This should result in a steeper diagonal line to the middle of the top edge of the image, that then continues along the top edge to the right hand side of the image (Fig. 6).

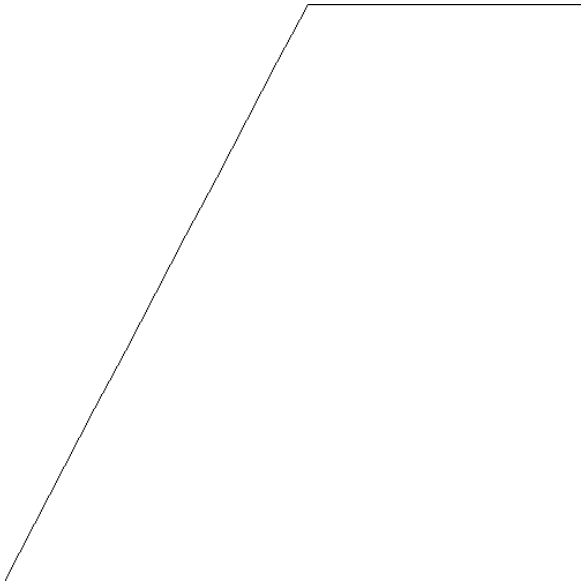

Figure 6: Output of TestMain2

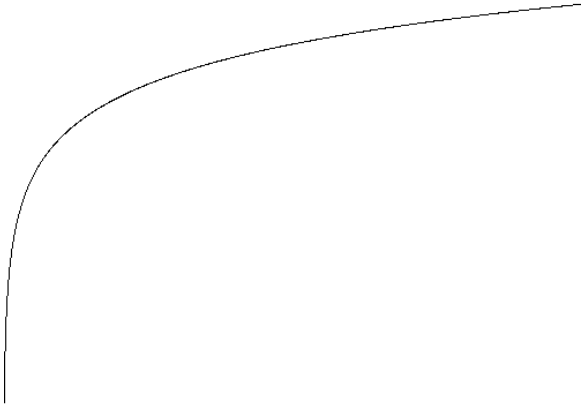

Figure 7: Output of TestMain3

The third test class tested the functionality of the ScalerFilter. The input file (test3.csv) contained a data stream of positive integers from 1 up to 1000, and another of the natural logarithm of these integers. The ScalerFilter scaled the results of the natural logarithm function by a factor of 3, and then added 5 to each point. The expected result was a more exaggerated natural logarithm function. This can be seen in Fig. 7.

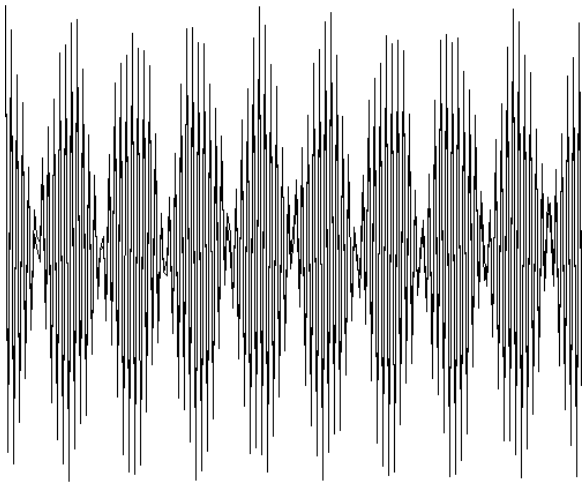

Figure 8: Output of TestMain4

The fourth test class tested a more complex filter chain, utilizing both types of filters that connect to the input of other filters, as well as a larger and more interesting data set. A test file (test4.csv) contained the following four columns:

1. All integers from -1000 to 1000. This data stream is the X input to the plotter.
2. $15 \sin 3x + 4$
3. $\cos x$
4. $\cos \frac{-x}{2} - 2$

The second and third streams are added together using an AdderFilter. This filter's output is connected to another AdderFilter, whose other input is data stream 4. The result of *this* filter's operation is then fed into a ScalerFilter, which is configured to scale the Y data points by a factor of 2.5. The output, shown in Fig. 8, is the graphical representation of the sinusoid resulting from the function $((15 \sin 3x + 4 + \cos x) + (\cos \frac{-x}{2} - 2))(2.5)$.

DISCUSSION

All constraints mentioned at the beginning of this report were satisfied by the final prototype. The complete project definition was covered, resulting in a fully-function filter-based X-Y

(CO-05) plotting system that reads input from .csv files (CO-02), processes data and plots it in the form of a .png image (CO-07). Modeling classes were separate from program I/O (CO-03). The minimum requisite number of filters (2) were implemented (CO-04). The design was modular allowing for the layering and chaining of different kinds of filters (CO-08). Plotting ranges are defined by the user (CO-09) and these ranges are scaled to the image's constant output size (CO-10), chosen to be 500x500 pixels.

Many of the goals chosen for the project were not met. Beyond the requirements for an operational plotting system, many of them either proved to either take too much time, be too complicated, turned out to not be necessary or particularly desired for sufficient operation of the system, or were reliant on the completion of another different goal that for any of the other listed reasons was not met. GO-01 and GO-07 were met, but the rest were not. GO-02, which aimed for no configuration on the part of the user, was not realistic due to the fact that it was dependent on a user interface which itself was a stretch goal for the project. Given more time it would have been a nice addition, but the current prototype requires programming knowledge to be operational. GO-03 and GO-04 were not met for the same reason as GO-02. GO-05, implementing most of the filter options, would have been trivial to implement, especially thanks to the use of the Filter and Chainable interfaces, but was not reached due to the deadline. GO-06, JUnit unit testing for modeling classes would have been relatively simple to complete, but was decided against for the same reason as GO-05. In hindsight, goals should have been more focused on smaller implementation details based on the requirements of the project, so that they could have been used more as a guiding force behind design decisions as opposed to stretch features.

The StreamSet class relies on the input data complying with STD-01 for Comma Separated Values file format, and thanks to built-in Java libraries, STD-02 for Portable Network Graphics files was followed. The source code is written in Java SE8.

Overall, the system complies with the constraints laid out as part of the problem defintion, and functions as a completely working prototype. Its construction is based in a solid object-oriented design that provides the groundwork for expansion into a much more complex system that could be developed and refined into a tool that could be useful to a wider audience. In the future, the lessons learned from this design process were the importance of design flexibility, such that design issues don't instantly derail a project's completion, time management, and taking more care in setting of realistic goals that consider all constraints.

CONCLUSIONS

This report described a holistic overview of an object-oriented software design project, from receipt of problem all the way to retrospective on the final result. Contraints were determined based on all information provided on problem receipt, which then guided the determination of features that were absolute must-haves, those that were desired but not required, as well as what would be impossible given the constraints. Different design options were taken into consideration based on identified goals for the project, and trade-offs were made along the

way. Visual aids such as Unified Modeling Language diagrams were made and revised to formalize design and view the system as a whole. Outside resources were considered to assist implementation, and industry software tools like the Netbeans debugger were used to resolve implemementation issues. The result of the process was a fully-functioning X-Y Plotter system capable of transforming a table of numerical input data into a graphical representation by the use of modular filter components that perform specific functions. Lessons learned from the process as well as a "post-mortem" on missed goals were described in detail.

REFERENCES

1. Common Format and MIME Type for Comma-Separated Values (CSV) Files. Y. Shafranovich. October 2005. `https://tools.ietf.org/html/rfc4180`
2. PNG (Portable Network Graphics) Specification Version 1.0. T. Boutell. March 1997. (Format: TXT=242528 bytes) (Status: INFORMATIONAL) (DOI: 10.17487/RFC2083) `https://tools.ietf.org/html/rfc2083`
3. "BufferedImage JavaDoc." Java SE8. Oracle, 2018. Web. 31 May 2018. `https://docs.oracle.com/javase/8/docs/api/java/awt/image/BufferedImage.html`
4. `http://www.jguru.com/faq/view.jsp?EID=134008`