CSCE 470 CS Capstone Project
Dynamic Data Viewer

for
Dr. Kenrick Mock
Professor of Computer Science
University of Alaska Anchorage
Department of Computer Science & Engineering
NSB 221

by
Matthew Wareham
wareham.mc@hotmail.com

May 1, 2014

Table of Contents

CONTENTS.....ii

I. Overview.....1

1. Abstract 1
2. Problem & Purpose 1
3. Parties Involved 1
4. Planning Process 1

II. Project Requirements.....2

5. Functional Specifications 2

III. System Design.....4

6. Interface Mock-up 4
7. Computer System Specifications 6
8. Data Information 6
    - 8.1 Supported Abstract Data Type(s) 6
    - 8.2 Attribute Value System Format 6
9. System Architecture 7
    - 9.1 MVC Overview 7
    - 9.2 Module Summaries 8

IV. Software Development Process.....10

10. Development Methodology 10
11. Testing & Debugging 10
12. Challenges 10
13. Schedule---Planned and Realized 11
14. Peer Code Review 13

V. Final Notes.....14

15. Results 14
16. Conclusion 14
17. References 15

I. Overview

1. Abstract

The goal of the project was to create a plot viewing application that allows dynamic exploration of a data set or parameter space. Mathematica was the language chosen for implementation initially, and this decision turned out to have pluses and minuses. Decent progress was made over the course of the semester, but eventually development was severely hindered, mainly due to the lack of a thorough understanding of the language's inner workings and best practices. The project will not cease here and is a continued work in progress, with the next step being to become more familiar with Mathematica to decide if it really is the best choice for the fulfilling the objectives, and then to reassess current work and move on from there.

2. Problem & Purpose

In general, plotting a data set is usually a very important first step toward gaining insight about the data. For the purposes of rigorous data exploration, most plotting applications are either too rigid or specific, like the charts of Google/Yahoo Finance, or too static, like the charts of Microsoft Excel. It appears that most general data mining tools focus on the statistical algorithms and less on the visualization of the data. With visualization-oriented tools, like curios.IT, many seem to provide a "do it for me" scheme, where the program finds an appropriate plot type that matches the data on its own. The results of these programs are often unfortunately not very dynamic---they are fine for certain purposes, especially if you just need something quick. This project's goal is to improve upon the data viewing solutions that are currently available, which have a number of issues when it comes to trying to "swiftly" and efficiently explore both the big picture and the intricate detail of a data set. Since 2D and 3D Cartesian coordinate plots are the most widely applicable visualization technique, they will be the first focus, with other types of plots possibly implemented later on.

3. Parties Involved

For now this is intended as a personal tool, and thus I am both the client and sole developer.

4. Planning Process

The primary, high-level use scenario is depicted by the flow chart in Figure 1. The user interacts with the Main Window and one-to-many Plot View Windows via special on-screen controls as well key and mouse input. This scenario was the guide for the design of the system.

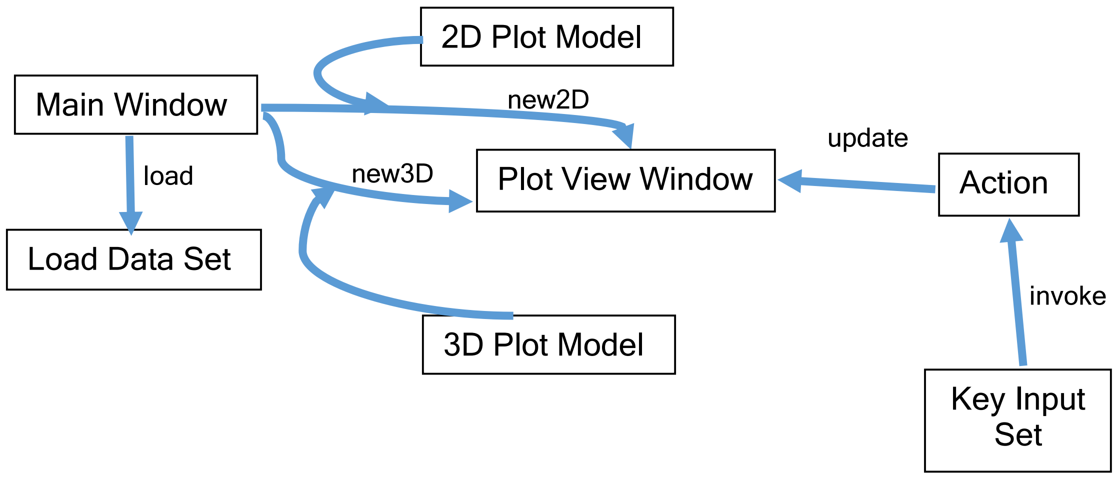


Figure 1

II. Project Requirements

5. Functional Specifications

The primary features that I wish to implement are for the most part understood. They haven’t changed much from the original project proposal, so the listing below is mostly the same as in that document.

1. Data Type Support:

   1a) For now support the ability to load only a certain CSV format, with the "attribute vector" of a data set as the first line (row), and any subsequent lines being instance vectors corresponding to the attribute vector to create a full "Attribute-Value Matrix/System." Support numeric and categorical data types.

   1b) Data structure storing/representing the relevant metadata for each dataset and each attribute in a dataset in regard to plotting (to be determined / discussed later).

2. Global/Main Navigation Window:

   2a) Ability/menu(s) to load data sets from CSV files, and display "loaded" data sets (just basic metadata, like the name), along with one to many "PlotView" objects (again, just basic metadata about each PlotView object). This is meant to be a succinct, well-ordered, list-like hierarchy visually, summarizing what data and PlotViews are loaded/open.

   2b) *desired but not crucial*: ability to save aspects of a PlotView object (the parameters of the plot essentially)---adding this to #4 would be useful as well.

3. A general model (special class/module) for representing the state of a Cartesian plot's parameters (labels, plot range of each axis, etc.). Will need one for 2D and one for

3D. The goal will be to get everything created for 2D first, and then if there is time do 3D too.

4. The class/module I was referring to as "PlotView":

   4a) Visual display, labeling, buttons, etc. of a plot within a panel/pane, which itself is within its own independent window frame; can ideally be more than one PlotView object (multiple windows) at once.

   4b) Necessary controls like buttons, sliders, checkboxes, textfields, etc. to control the state of a plot's parameters.

   4c) Ability to quickly and easily change which variables are "assigned" to the x, y, or z plot axes. Any remaining attributes (dimensions) should be held constant (treated as parameters) at any given time, but should be manually alterable. Eventually it is more preferable to allow a more general "subset" to be specified for each of these remaining attributes, but this is not as important as other features, so will not be dealt with *yet*.

   4d) *desired but not crucial*: for categorical types, ability to change the order of the attribute values plotted...for instance, if the values of an attribute were: {dog, cat, fish}, you'd be able to align those on the corresponding plot axis in whatever order you want.

5. Key Bindings to Invoke Commands:

   5a) Key bindings which invoke Actions (see #6) to alter plot parameters. Example: hit right arrow key to move/increase upper bound of the x-axis by a parametric amount [or for categorical, just show/add "the next" category on the "right"] of a 2D plot. Define a default set of key bindings.

   5b) *desired but not crucial*: integrate a key binding tool (such as the one found here: https://www.atagar.com/keybindingUtil/) and build the necessary structure to allow a user to pick their own key binds to bind with the static hard coded set of Actions (see #6), and easily save them as sets and load saved sets. Have the default set always something to fall back upon, and thus hardcoded in the program.

6. List/Map of actions/commands; when invoked, each action causes changes to parameters in the plot model mentioned in #3.

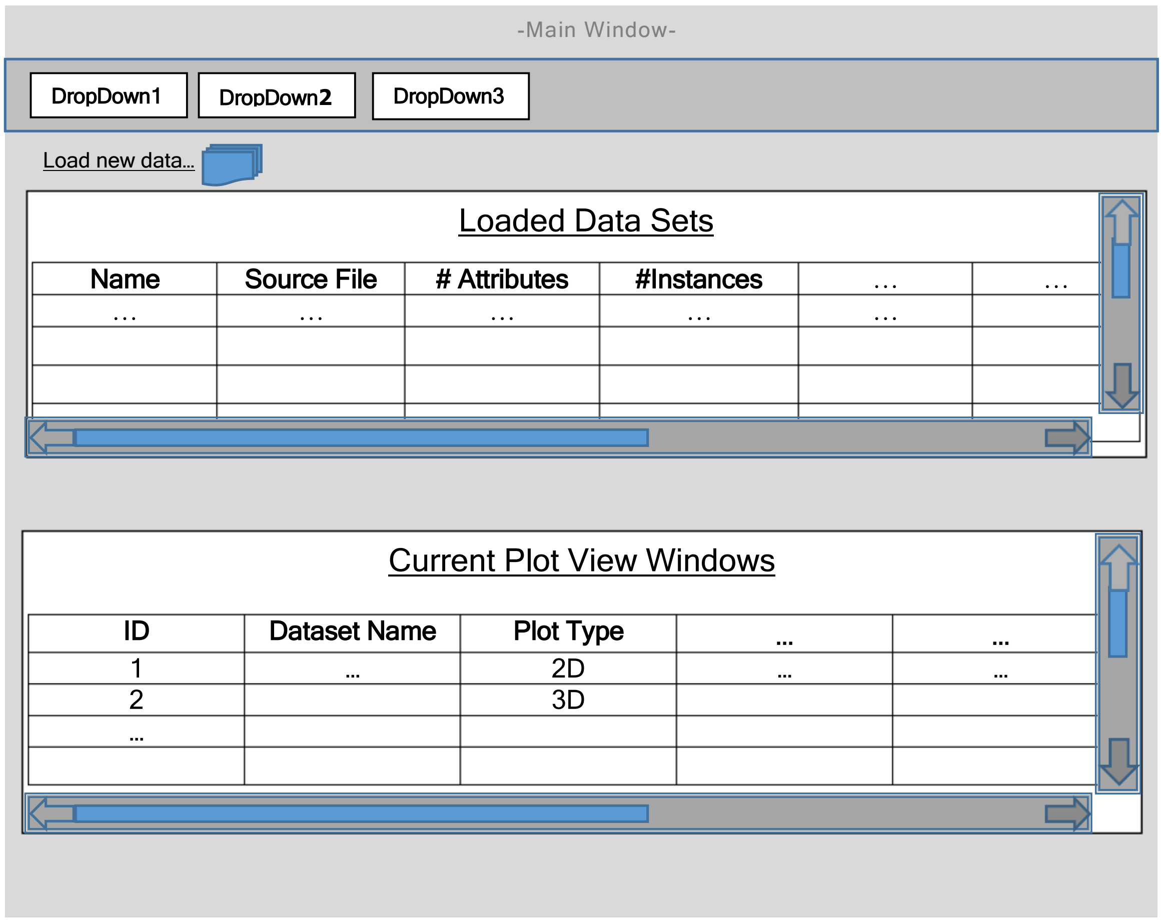
7. Relevant logic to continuously update the plot visual based upon changes in the underlying parameters. Using Mathematica, this should mostly be done using the function Dynamic[].

III. System Design

6. Interface Mock-up

There are two primary user-level interfaces: the MainWindow object and the window for each independent PlotView object. For the custom key bind solution, there will probably be a pop-up menu type interface as well. Figure 2 shows the mock-up for the MainWindow and Figure 3 shows the mock-up for a preliminary idea for a PlotView window object.

Figure 2



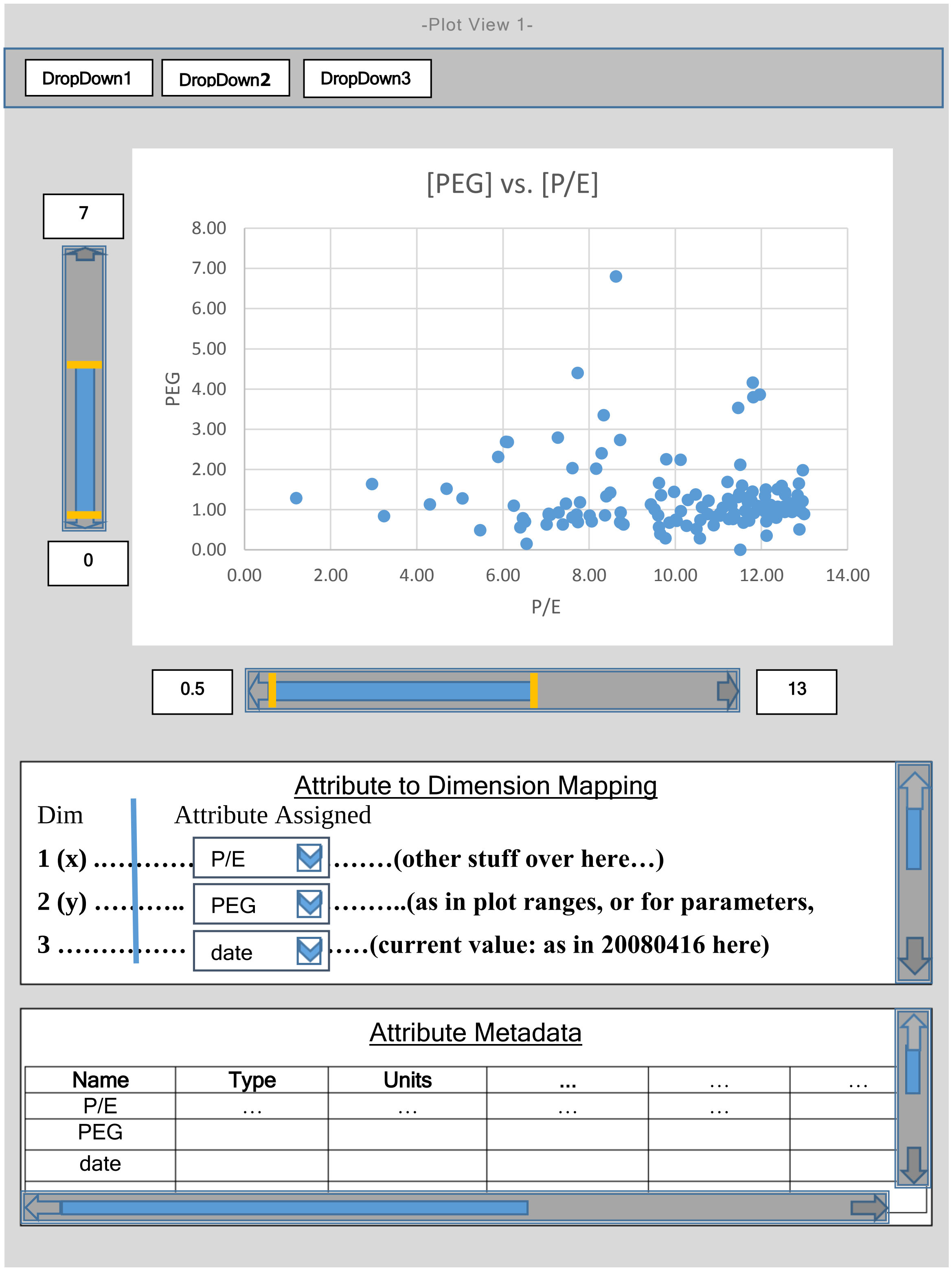
Loaded Data Sets

Name	Source File	# Attributes	#Instances	…	…
…	…	…	…	…	

Current Plot View Windows

ID	Dataset Name	Plot Type	...	...
1	...	2D	...	...
2		3D		
...				

Figure 3



Attribute to Dimension Mapping

Dim	Attribute Assigned	
1 (x) …………	P/E	…….(other stuff over here…)
2 (y) ………..	PEG	……...(as in plot ranges, or for parameters,
3 ……………	date	….(current value: as in 20080416 here)

Attribute Metadata

Name	Type	Units	...	…	…
P/E	…	…	…	…	
PEG					
date					

7. Computer System Specifications

Again, this is intended as a personal tool for now. As such, I am mainly a Windows user, but even despite that I would like to make a tool that is usable on other operating systems in case I prefer other systems in the future or in the event that I wanted to provide this to other people. Luckily, Mathematica and Java are suitable for use on many systems.

Originally I planned to write this almost completely in Java since it is the language I know best, with relevant plot graphics created/retrieved from interacting with the Mathematica kernel through an interface called J/Link. Despite the risk of learning a new language, since the Mathematica language is quite high level and does a lot of the dirty work "automatically" under the hood, choosing to use solely Mathematica was a reasonable decision. In part x on page y, I discuss the good and bad of this decision.

8. Data Information

8.1 Supported Abstract Data Type(s)

Generally, most data can be lumped into one of two groups: categorical or numeric. In short, categorical attributes usually take on a finite discrete list of values (like {red, blue, yellow}), while numeric attributes can take on any number as a value usually, or otherwise a subset of the real numbers (like the natural numbers, or sometimes a finite set). Mathematica deals with numbers just fine, and to deal with categorical values, my first approach was to map the possible values for a given categorical attribute to the natural numbers, and keep a copy of this "legend." This was a good solution for now.

8.2 Attribute-Value System Format

As mentioned in #1 of the functional specifications section (page 2), the first line in an input CSV file containing a data set should be the attribute vector. Every subsequent line should be an instance vector. However, within the attribute vector, the following convention was used, which was a simple and elegant solution to include relevant metadata about each attribute:

Within each attribute between commas in the attribute vector, instead of placing just the attribute name, use underscores as a secondary delimiter to specify two more pieces of information: the data type (categorical 'c' or numeric 'n') and a text string specifying the units of measurement (like centimeters, cm, seconds, etc.) if applicable. In the event that units should be effectively "null," just using a dash character "-" is a suitable marker to convey this, and is handled by the program fine (as it is just "another string"). Data type should always be specified though. Here's a concrete example:

```
length_n_cm,color_c_-,id_n_-
```

As opposed to just doing the following:

```
length,color,id
```

9. System Architecture

9.1 MVC Overview

The model-view-controller (MVC) design pattern was utilized as it fits well with the project's requirements. Figure 4 shows the primary high level components of each of the 3 parts of the pattern, while Figure 5 shows the actual modules implemented for each thus far.

Figure 4

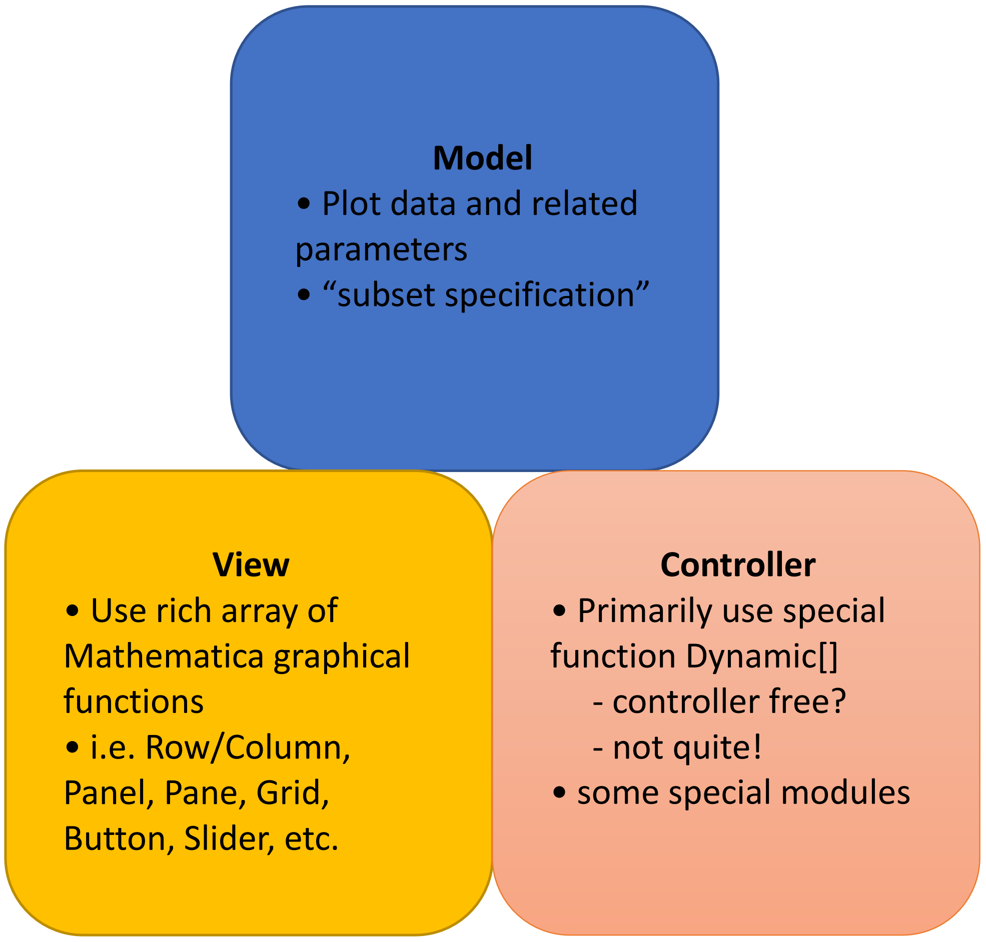
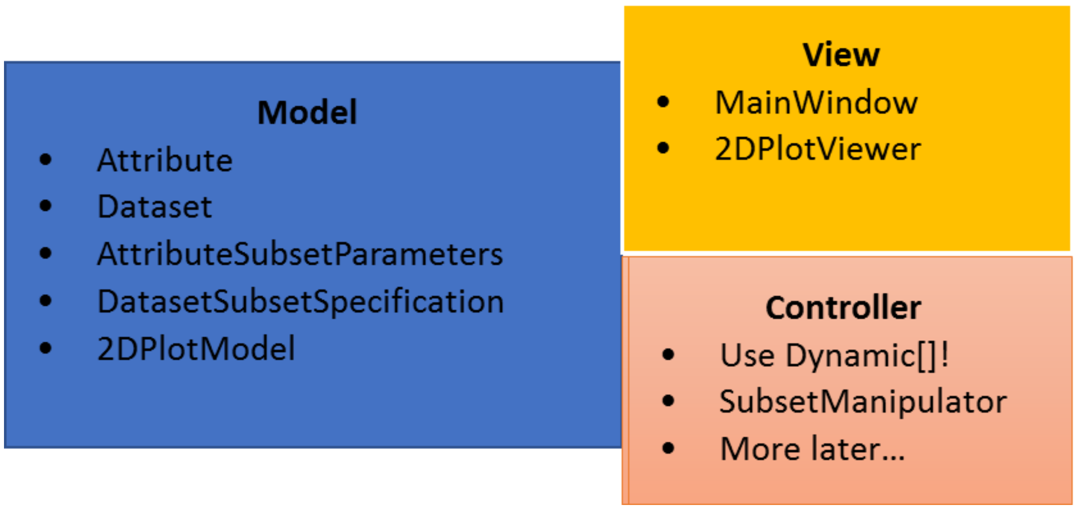

Model
• Plot data and related parameters
• "subset specification"
View
• Use rich array of Mathematica graphical functions
• i.e. Row/Column, Panel, Pane, Grid, Button, Slider, etc.
Controller
• Primarily use special function Dynamic[]
- controller free?
- not quite!
• some special modules

Figure 5



9.2 Module Summaries

Here I will briefly describe each of the modules implemented (those shown in Figure 5).

<u>Model:</u>

1) **Attribute**

   The first and simplest module, which represents the metadata of an attribute: name, type, and units. It can practically be seen as a struct. Name can be any string, such as “id”, “color”, “count”, “length”, etc. Type must be either “n” for numeric or “c” for categorical, so it is a simple enumeration effectively. Units can be any string, such as “m”, “meters”, “lbs.”, etc.

2) **Dataset**

   This module contains information for a given dataset, including small things such as id, name, source file, etc., and three main “big” things: attribute list, instance list, and a simple algorithm that constructs these lists from the imported .CSV based file format.

   attribute list example: {id, color, count}
   instance list example: { {1,blue,6}, {2,green,5}, …}

3) **AttributeSubsetParameters**

   Each attribute in a dataset is associated with one of these objects, which holds state information relevant to plotting. They are the following six parameters:
   - attribute name
   - lowerBound (for x or y axis plot range)
   - upperBound (for x or y axis plot range)
   - dL (amount to change lower bound when panning)
   - dU (amount to change upper bound when panning)

- constParamVal (for dimension 3+)

In the 2D case, each attribute in a dataset will either be assigned to the x-axis (dimension 1), the y-axis (dimension 2), or the dimension 3+ set. For the former two, the lower and upper bounds are applicable for specifying the plot range on the given plot axis for the attribute assigned to that axis. For the latter, at this time the simple case of just specifying a constant parameter value for all attributes assigned to dimension 3+ is the purpose of constParamVal. This has the effect of only choosing x-y pairs for display from instances that have the appropriate constant parameter value for each of the dimension 3+ attributes. Eventually, generalizing this to specifying any subset would be very useful.

4) DatasetSubsetSpecification

The big picture of this module is that it contains three things:
- reference to Attribute object assigned to x-axis
- reference to Attribute object assigned to y-axis
- list of AttributeSubsetParameters for each Attribute

The overall idea is this module represents the state of a subset definition (or as I call it specification), which can be dynamically updated.

5) 2DPlotModel

It would make more sense to make this module "have-a" DataSubsetSpecifcation object, but for now, there are 5 main pieces to the 2DPlotModel:
- reference to x-axis Attribute (again)
- reference to y-axis Attribute (again)
- passed in x-y pair data (calculated elsewhere)
- special logic to transform (map) categorical attributes to natural numbers
- special logic to calculate certain plot parameters

<u>View:</u>

6) MainWindow

This turned out just like the mock-up, and there are 3 sections to the interface:
- typical drop-down menu bar with typical "File" options list as well as the option to load a data set, and Plot list, which contains options to create a new 2D or 3D plot
- loaded data sets table, including a special button above the table to load a data set
- current plot viewer windows table, which lists metadata about the current plot viewer windows

7) 2DPlotViewer

This also turned out very close to the mock-up, with 4 sections to the interface, all within a scroll pane:
- typical drop-down menu bar (this may be removed until it has a specific use)
- plot section, which contains the plot as well as special on-screen controls for altering the x-axis and y-axis parameters dynamically

<u>Controller:</u>

As mentioned previously, a special function called Dynamic[] gets me much of the controller for free. Using this function in the correct way implements what is effectively an elegant observer type pattern under the covers, leaving almost no work for the programmer. It is handy, but tricky to use correctly.

8) **SubsetManipulator**
   This is the one separate module with control code that I used so far, which does exactly what the name suggests---manipulates a subset (DatasetSubsetSpecification object). Right now it is just one function though, and based on what it does it might make more sense to be a static part of the PlotModel module.

More controller modules may be necessary eventually. Once issues with the project's stalled progress have been ironed out (to be discussed in a subsequent section), the typical "Main" module with an "init" call will be implemented ultimately as well.

IV. Software Development Process

10. Development Methodology

The software development methodology I planned to use at first was prototyping. I soon realized some combination of Agile and prototyping would be best because I wanted to keep track of things and approach the project from an Agile standpoint with prototyping as an integral part of the process. This was a good choice as it gave me flexibility, mainly in the ability to not have to design all at once up front, which was helpful. The idea of "rapid prototyping" is very much a possibility and really even could be called the norm for developing Mathematica programs, since code can be "executed" easily at will without requiring compilation, similar to Scheme programming with Racket and other declarative languages in general. This was in fact extremely helpful since I was learning as I went along.

11. Testing & Debugging

The ability to run programs at any point on the fly was extremely useful for testing, debugging, and learning. I found that I tended to incrementally write smaller chunks of code so that it should theoretically work, then run it, then if it didn't work, go back and figure out why, and repeat. Mathematica has debugging tools built in that could probably help, and it would probably behoove me to get familiar with them, but I have not used them for so far.

For testing I wanted to try to do test first, but I realized it was difficult without being familiar with the language and with the requirements not being crystal clearly spelled out and absolute. Therefore I ended up writing test cases after I finished a module successfully, and my intention was for them to be "preliminary" testing modules, with more rigorous testing modules created later. This worked fine I think, except that some of the later more complicated modules were trickier to write test cases for since they inevitably relied on other modules. This might have been faulty design at work though, and refactoring might help make this easier.

12. Challenges

The primary challenge was being new to Mathematica. Ultimately, the coding precedents I went with from the beginning probably aren't best practices for Mathematica, and they resulted in odd warnings and eventually not just warnings but complete errors that kept me from making any progress. For one, the precedents I used tried to emulate aspects of object-oriented programming (especially information hiding), but doing this was perilous because it was only object-oriented "in spirit," not in reality. Therefore, continuing down that path was eventually going to cause a problem once things got complicated enough, and sure enough that is what happened. Therefore, a crucial lesson was that it is not the best idea to go full steam ahead with a project before a sufficient understanding of the language/technology involved is reached.

Despite spending 20-30 hours before even beginning the project studying Mathematica documentation, the problem is that it is rather unstructured, and it is mainly a giant source of ad hoc examples. This is definitely good from a certain perspective, but from the perspective of trying to get up to speed and understand the language (especially its nuances) in a timely, structured, and all-encompassing manner, it is unfortunately not sufficient. Therefore, I ended up crashing and burning so-to-speak (with respect to hitting a wall progress-wise). My plans for what to do next to overcome this issue will be discussed in the conclusion section.

13. Schedule---Planned and Realized

The data in the figures below include time spent on the project specifically only, so not on other aspects of the class like other assignments/activities. But it does include all time spent designing, implementing, testing, etc. In total I spent 134 hours in 8 weeks' time on the progress I made so far. My goal was to get 20 hours a week in average given other responsibilities, and I ended up falling short of that goal. Along the way I recorded hours that could be classed as "learning," where I was mostly having to fiddle with unfamiliar things to figure out how they worked, and this constituted a very large chunk of the total time, roughly 30-35% overall, which is almost certainly not ideal.

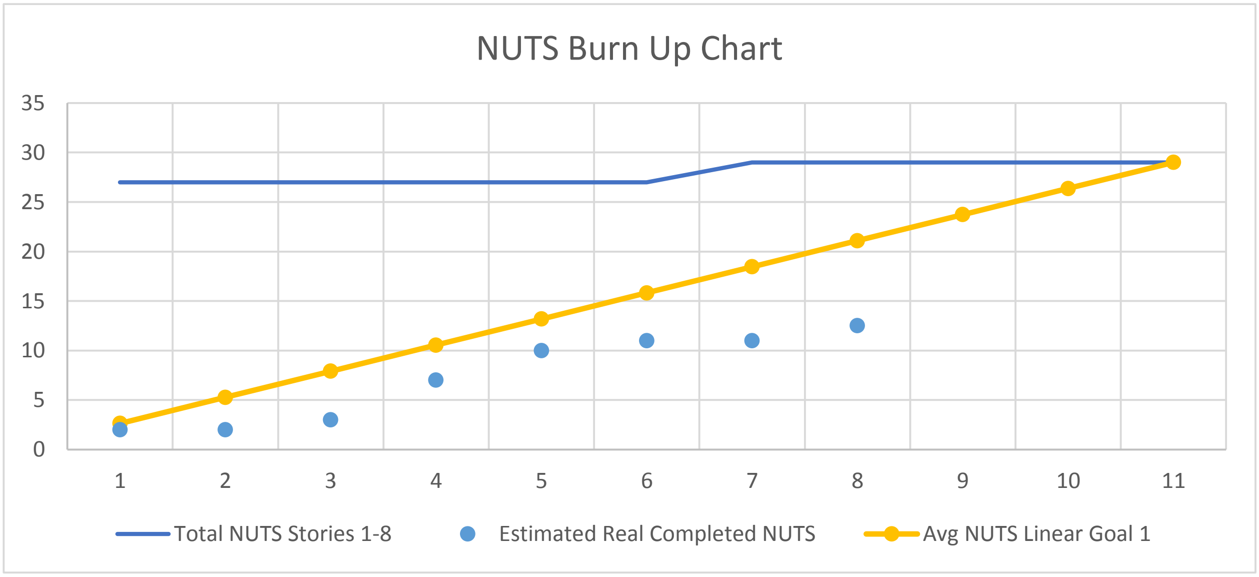
Table 1 below shows the primary eight feasible user stories that I hoped to complete, with the time I spent on each over the course of the semester. In the end, I ended up only finishing half of the NUTS I wanted to, but this is mainly because of hitting a wall with story 4, the 2DPlotViewer module. Ultimately, my NUT estimates were not completely wrong, but they weren't the best either, and this will likely get better with more practice/experience. Note, I ended up counting a lot of hours that should fundamentally be a part of story 3 as part of story 4. This will become more balanced when I refactor the plot model modules in the future.

Table 1

Story	Description	Nuts	Hours
1	Data set state modules	1	6
2	MainWindow module	4	38
3	2D plot model modules	3	9
4	2D plot viewer modules	7	81
5	Action module	4	0
6	Define list of 2D plot actions	3	0
7	Key binding functionality module	4	0
8	Default key combo to action binding module	3	0

Figure 7 shows my burn up chart for the 8 weeks (1 week per iteration) until I hit a wall. I started out behind, began to accelerate, and then fell behind again with respect to the linear goal.

Figure 7



14. Peer Code Review

Despite most of the class being unfamiliar with Mathematica, some good thoughts were mentioned, and I address them below from my Code Review file.

1) **String type pattern matching for passed in parameter**: This is something I haven't gotten to yet, but indeed I plan to figure out the best way to go about pattern matching.
2) **Comments on how code works**: I began to try to do this after the suggestion, and ultimately, I'll try to ingrain this so it becomes a natural habit. Normally I tend to get a module/class/method working correctly and fairly clear and neat, and then go back and add comments, but I realize starting out with well-formed comments can be a better technique, similar to using test first.
3) **Post evaluation warning (:= vs. =)**: Yep I realize this, and my intention is to use it this way in order to define what are effectively functions to be used when they are called. Even so, my incomplete understanding of how everything works underneath the hood means this is part of further research I must to with Mathematica.
4) **Isolate/declare all variables for persistent data all together**: This was something I realized was a possibility but that I fought the whole time in an effort to try to practice information hiding, even though since Mathematica variables/symbols are global by default, true information hiding is not natural. In further studying Mathematica, I'll come to the conclusion as to whether I should scrap information hiding totally, or how to do it in a more proper way.
5) **Function call or array access**? : Technically both actually. This is seen mostly when I create what are supposed to be getter functions for "member" variables within a module. Such "member" variables are really stored within the reference to a map/array object that represents an instance of the module/class. Indeed, this is somewhat fuzzy, and trying to do such things is what most likely contributed to the program going awry ultimately.
6) **Move closer to where used**: Yes, in some cases this would make sense, and it comes down to declaring everything in one place vs. closer to where they are used; we shall see.
7) **Temp vars declare in temp variable space**: Yes, I mostly did this okay but there are probably some places that are a little weird that need to be fixed.
8) **With to make constants**: I don't recall, but I'm guessing this was a "nuance of the language" type of thing that Keith must have mentioned. Yes, I can see how using the With function for constants would make sense for clarity, and indeed that is something I will probably start doing.
9) **Relative vs. Absolute scaling**: Indeed it is not ideal that I'm switching. I ended up trying different things, and once I got a certain part to work, leaving it. It would be preferable to figure out how to get or the other to work and stick with that, or worst case distinctly describe in some way why in some places the other must be used.
10) **Check if cancel selected for dialog**: Yes this is a good reminder, but in general I have "put off" validation, which is good for getting to more important functional requirements, but bad in that it is easy to break the program. So yes, similar to the comments, I should probably try to force myself to get in the habit of thinking of the appropriate validation required and implementing as I go along rather than later.

11) **Move to separate executable block**: Yep no worries was planning to do this eventually, though again like with the comments, maybe getting in the habit of doing this from the start would be best.
12) **Add clear button for data set**: Not a bad idea---unfortunately doing this has dependency issues to deal with, but related to this is the notion of exiting a PlotViewer window, at which point the list of PlotViewer windows would need to be properly updated to reflect this change, as this does not happen with the current implementation. In general indeed the issue of "deleting" and "clearing" needs to be addressed to.

V. Final Notes

15. Results

Despite the setbacks, I did get quite a bit done all things considered. Most of the infrastructure for the state of plot data and plot parameters is there, and the only issue is that some of it needs to be refactored/combined. The MainWindow works as hoped except for a weird input glitch still, but I suspect this is the result of other more fundamental problems with the program (as in not coding properly with Mathematica). The 2DPlotViewer works to an extent, and could actually be used to dynamically view data to some degree! There is much to be done still though.

16. Conclusion

Considering that I started out a novice with Mathematica (now I would say I have an intermediate level understanding at this point), what I was able to produce in 134 hours' time split between 8 weeks and other responsibilities was respectable progress I think. I learned how moving forward without a thorough understanding of the tools (language/technology) to be used is not a good idea. I also learned that one must be careful trying to apply principles that are applicable and preferable in one domain (object-oriented programming) to another domain (symbolic programming), as it can cause problems or simply be inappropriate.

Looking forward, the best course of action will be to step back and get a more thorough understanding of how Mathematica works, how to use it correctly, and which programming paradigm is best suited to building the tool I want. I plan to look deeply into functional vs. pattern/rule-based programming in Mathematica to see if adhering to those paradigms strictly makes the most sense. Ultimately, if Mathematica does not work out, I can return to my original plan of using Java. This project is therefore a work in progress, and I will continue it alongside other projects.

17. References

Yahoo! Finance
http://finance.yahoo.com/

Google Finance
http://www.google.com/finance

curios.IT
http://www.kanohi.ch/index.php?id=9

Statistical Variables/Attributes
http://changingminds.org/explanations/research/measurement/variables.htm

Mathematica Documentation:
http://reference.wolfram.com/mathematica/guide/Mathematica.html