GEDIVA Team 1 Design Document

Introduction

Data abounds in today's information age, and sometimes the vast collections of data can be overwhelming for people attempting to use that data. The GEDIVA project addresses this issue; it is a data visualization project that allows a user to see a simple representation of otherwise difficult to understand data. Our team's implementation focused on working with stock data. We aimed to make a reusable framework for data visualization, to make widely reusable sections of code, and to make a specific implementation that can turn tables of stock data into visual representations of that data.

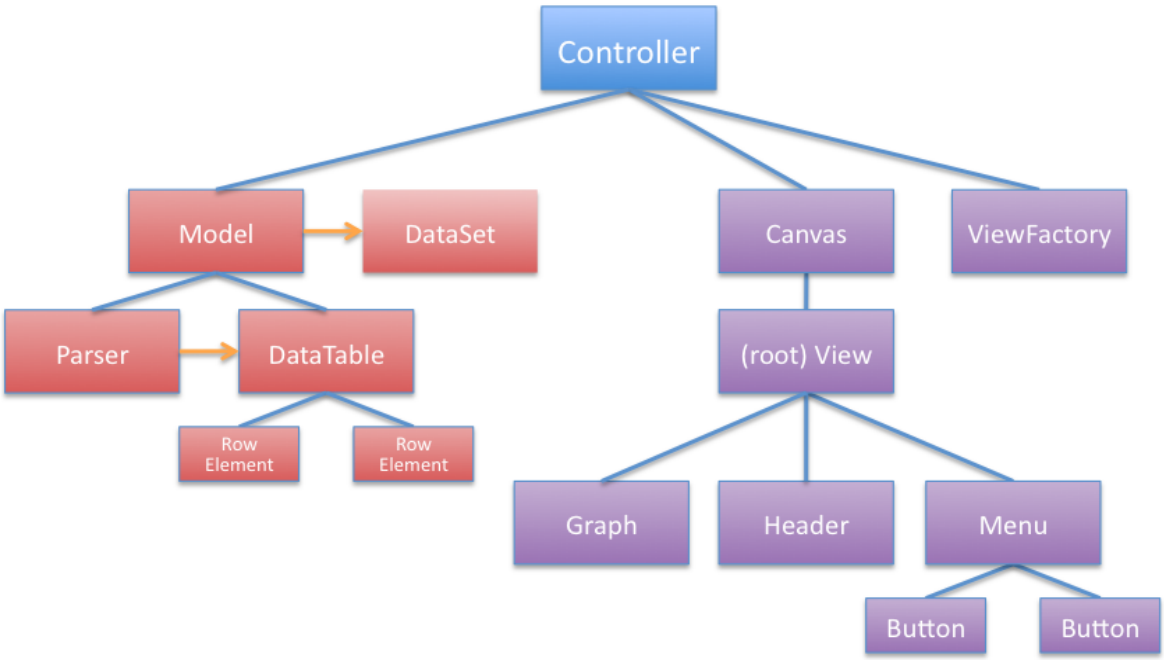
We use the Model View Controller (MVC) design pattern as the basis for our framework. The model is in charge of parsing input files, storing the data, and processing the data. The view is in charge of displaying data, providing a user interface, and accepting user input. The controller is in charge of instantiating the model and the view and mediating interactions between the model and the view. Together, these components maximize code modularity and reusability, while also providing a simple conceptual model for how the program actually works.

Overview

The program begins as a blank window with multiple options to select how to import data. The user can choose between loading from a file, from a URL that they input, or they can enter a stock's symbol and the program will load data from the internet using the stock symbol. After the user selects the import data option they want, the information for a certain stock will be imported in the form of a line graph. The user then has the option to toggle graph views between line and bar types. The user also has the option to select a new stock from which a new set of data will be imported onto the screen. If an error occurs, such as bad data format or unreadable source, the user should see an error message appear on the screen.

On initialization, the controller tells the view factory to put a menu in the canvas, allowing the user to see the buttons which can load from a file, URL, or stock symbol. When the user inputs their choice, controller tells the model, and the model finds a parser that can accept the chosen input and populates the database. When this completes, the model gives the controller a dataset representing the newly parsed data, which it gives to the view factory which makes a title (header) and a plot that represents the data.

Throughout this chain of events, the model was never exposed to part of the view, and the view saw only a returned dataset from the model. These two parts of the project were designed independently of one another. The model does not care at all how the data is displayed or how user input is taken. The view does not care how files are read or how data is stored. MVC encapsulation allows these components to be independent and reusable beyond this project. Even if they are not reused, the encapsulation make testing and debugging easier. If there weren't this conceptual separation, one change with the user input could break the way data is stored.



Design Details

The model is broken down into 4 main parts: parsers, the database, responses, and the central model. The parsers are a family of input reader factories that generate buffered readers for some input (like a file or a web connection). The model uses these parsers to turn the name of an input into a readable version of the input. These "parsers" do not actually parse the input themselves; because they are factories, they create other objects that can read the input. Each parser specializes in a different input type and generates the corresponding reader. There is an abstract generic parser which defines the primary use of these factories: the generateReader function. Given

some desired file or url (or other data source), the generateReader function tries to access the source and generate a buffered reader. If it fails, it throws an appropriate exception.

The model's database stores the model's data while providing a bit of useful functionality through management and simple access. It is conceptually a table with each row representing one data point and each column representing one attribute. The database is comprised of two levels of objects: the upper level DataTable, and the lower level RowElement. The upper level DataTable's job is to deal with requests and managing data manipulation. The DataTable can create, add to, and sort its rows, parse and populate its column names, and pick out individual columns of data. The lower level RowElement (instantiated many time per DataTable) is in charge of storing data and providing simple functionality that makes the DataTable's job easier. Each RowElement stores one datapoint and can compare itself to other RowElements for sorting based on a requested attribute.

The responses are very simple intermediates that wrap the database so that the view can interact with it without modifying it in any way. The IDataSet interface defines only 3 methods: attributes (returns a list of attributes of the data), sort (sorts the data by an attribute), and getData (returns a list of data corresponding to the desired attribute).

The model class wraps these components together and provides the external model API, which is quite simple. It is in charge of initializing its input reader, filling and managing its database, and provide methods for data processing. The AbstractModel that is the groundwork for the model provides a way to identify itself, methods for initialization through an input name or a pre-made buffered reader, and a method for processing of any type. The StockModel adds basic info about the stock it represents and implements the database loading and processing methods.

The controller is in charge of mediating communication between the model and the view, instigating the initialization of the model and view (not all of the specifics), and handling user input (without interfacing with the user). The controller can be simple because it provides an interface between the model and view without actually implementing very much functionality. This interface makes the view and model less interdependent, causing both sets of code to be more reusable.

User Interface Design

The view is less segmented than the model and instead uses a hierarchy to work with all of the parts. The view consists of 3 main parts: the view elements (which have a good deal of variation), the canvas that is the central point of access to all view elements, and a factory to populate the canvas with the desired views.

Each instantiation of the view class represents one element in the displayed window. A view may or may not have children (in which case that element would have sub-units), and if it does, it is in charge of passing commands down the hierarchy to all of its children. This structuring combined with one root view ensures all views can receive a command (e.g. paint) by giving it to the root view. Whether or not it has children, each view has a location, a size and a way to paint itself.

The view class is subclassed multiple times to allow many different visualizations and element types that are all accessible through similar commands. There are label-like subclasses which display text for the user. The Header class is a view that adds a title to the display with details about the stock of interest. The ErrorView class is a view that displays to the user when an error occurs, and it displays text relevant to the error. This usually occurs when a file is not found when attempting to load. The Button class both displays preset text and triggers a function on a user click. The Menu class is designed to hold buttons and make their generation easier for the developer. The view class also has graph subclasses which are specifically designed to plot data. The graphs can draw their axis, add tickmarks and labels to them, calculate where to put given data, and display that data. The abstract Graph class has 2 implementations: BarGraph and LineGraph. These two simply have different ways of actually plotting their data (a bar graph and a line graph).

The canvas is the primary container for the view. It is the holder for all of the views and the interface to classes outside of the view. It holds the root view, listens for mouse clicks, manages the root view's children, and tells the view hierarchy when to repaint. It is very extensible because it delegates responsibilities in ways that promote modularity.

The StockViewFactory is factory that is specific to the stock visualization project. It provides a way for the controller to populate the views without actually knowing how the views work. It has two public methods (other than the constructor): one that loads the menu to accept user input and one that adds a header and graph once the model is loaded. When creating the menu, it both creates the views in the view hierarchy and connects the button triggers to methods within the controller.

Design Considerations

The biggest challenge for us in coming up with the API was finding a good way to communicate information between the Model and the View. We considered a few different

possibilities for communicating the data 1) A more direct approach, where the Model simply returns data in the form of an immutable data structure. 2) A more abstract approach, where the Model returns an object which encapsulates different types of data. We opted for option 2 because it makes our design more flexible. We also had two different choices concerning how often to communicate data: 1) The Model sends all the data in one request, and the Controller decides which information to glean from the data and show in the view. 2) The Model sends the data in pieces, as needed. I.e. the controller makes multiple requests for each type of data it needs. The main advantage of option 2 is better performance, because complicated data analysis might take some time, and with this option is possible to display data to the user faster. However, after a little experimentation, we discovered that the computation time for the types of analysis we were doing were pretty trivial. We opted to use option 1, but leave our code flexible to option 2 if needed. If someone else went in and modified our code to add more complicated forms of data analysis, they could make individual requests to the model to improve performance.

On the View side, one challenge was figuring out how to respond to button presses. One choice would be assigning each button some sort of id. Then, when a button is pressed, a method is called in the controller which is essentially a big switch/case statement. Depending on the id of the button that was pressed, it performs some action. The drawback of this approach is that the code inside of the switch/case statement can get kind of messy, and it is harder to extend the programming by adding more buttons. Instead, we decided to make clever use of reflection. Each button has as instance variables the method which will be called when it is pressed, and the controller which contains the method. This makes it really easy to create new buttons or change what the existing button does. This approach was influenced by other frameworks and languages we have worked with. For example, in javascript you can assign an onClick attribute to a button to define what method will be called when the button is pressed.

Remaining Issues

The user never sees the implemented processing method. More user interface could be added to allow for choice of processing type.

The user will only see one stock displayed at a time. This is for two main reasons. The controller only holds one model at a time, when a list of models could held, each representing different stocks. The plots are not additive, and must be changed to handle more than one set of data at a time.

Based on current methods for toggling the graph type, there is only support for two types of graph. This could be fixed by changing the toggle functionality to a set functionality.

There is a functionality bug that causes a ConcurrentModificationException when a stock is loaded, the graph type is toggled, another stock is loaded, and the graph type is toggled again. This again has to to with the way that graphs are toggled.

Team Responsibilities

High level project design was lead by Lance, with help from Alex and minor help from Mark and Jesse. The parsers were written by Lance. The model databases were written by Lance with minor help from Mark. The model responses were written by Mark with help from Lance. The model was written by Lance and minor help from Mark. The view hierarchy was outlined and implemented by Alex with help from Jesse. The graphs were written by Jesse with help from Alex. The label views were written by Alex with help from Jesse. The controller and facilitators were written by Alex and Lance. The non-code documents were written by Mark with help from Alex.