Assignment 1: Design

October 20th, 2017

Fall Quarter

CS100

By:

Huzaifah Simjee & Hamza Awad

Introduction:

Assignment 2 will be written in C++. We will be designing the project with two functionalities. First off, we will add functions to allow a user to print a live command prompt line. Second of all, functions will be included that will read in and execute a/multiple command(s) within the same line.

Epic:

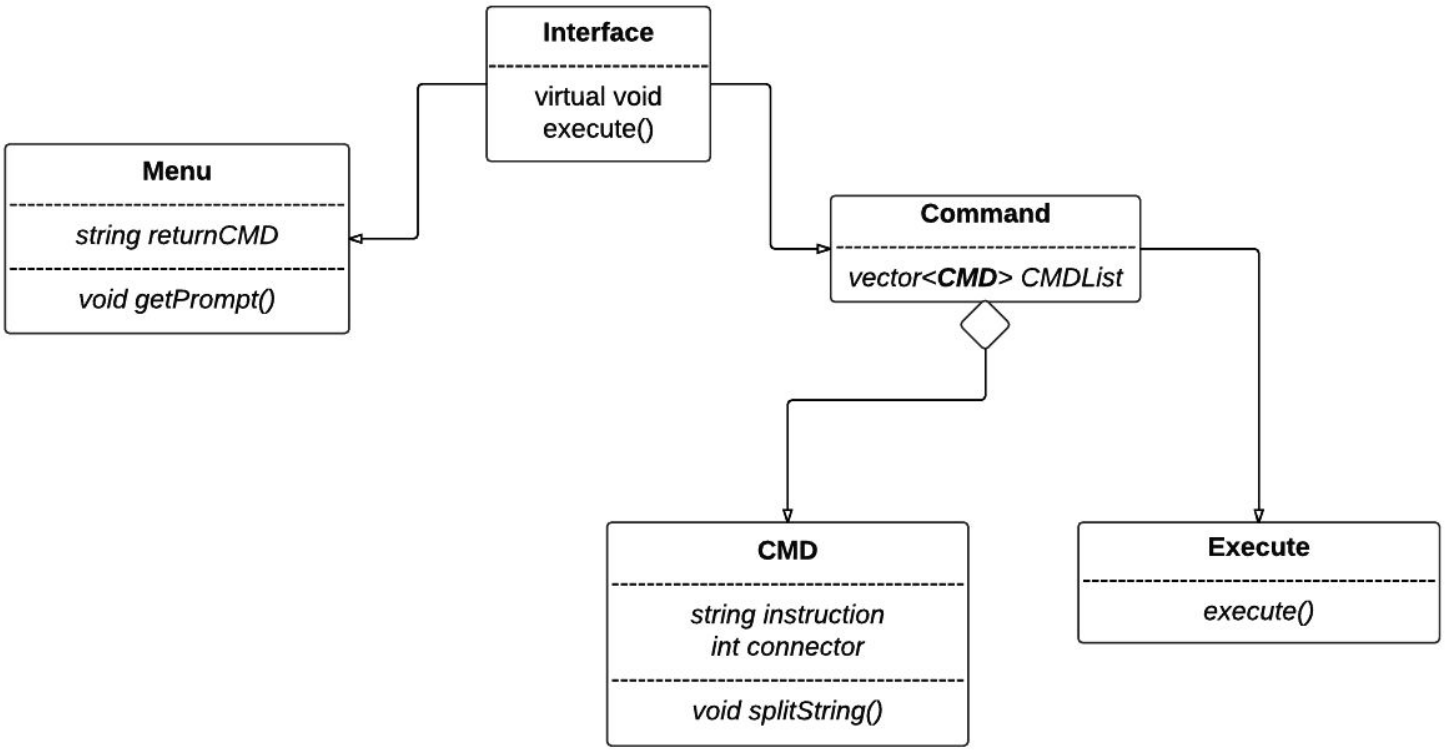
The chosen design style for accomplishing this is composite. This is because it will allow modularity which will make adding additional components simple and will allow smaller objects to be morphed together to make more complex objects.

As a user, I would like to input multiple commands through an instruction I will write into a program with different connectors that will make each command execute depending on whether or not a previous command has been executed successfully.

Design:

RSHELL ASSIGNMENT 2 DESIGN DIAGRAM

Huzaifah Simjee and Hamza Awad | October 20, 2017



Classes/Class Groups:

First and foremost, our overarching class group is called **Interface**. This abstract base class will inherit certain data members and methods to its two children: **Menu** and **Command** class. This **Interface** will be, not only responsible for sharing data members across classes, but to orchestrate its children in a way that the rshell prompt can function in sync with regards to displaying, reading in, and splitting commands. This interface class will eventually loop to a point where its subclass, **Menu**, will receive the "exit" command; thus, our interface operations will terminate and all functionality of our software will cease.

The first subclass of **Interface** to discuss is **Menu**. **Menu** will have a string data member *returnCMD* and a function which requests command input from the client *getPrompt()*. *getPrompt()* will have no return value and if the command(s) can be identified and isn't the exit command, the function will set *returnCMD* to the inputted command line. Essentially, *getPrompt()* is a function that prints '$' and waits for the user to type in the desired command(s). **Menu** is a subclass of **Interface**.

The second subclass of **Interface** is **Command**. This class is responsible for organizing its own subclasses to split commands received through **Menu** and store them in **Command's** vector data member: *vector<**CMD**> CMDList*. While **Command** does inherit from **Interface**, it will have two children of its own: **CMD** and **Execute**.

CMD's purpose is defined through one of **CMD's** functions: *splitString()*. On a larger scale, **CMD** objects will hold information regarding the single split command and its command type (if numerous). These **CMD** objects will be the vector type of which *CMDList* is made of. Therefore, **CMD** has to contain two data members, aside from its *splitString()* method: a string *instruction* data member and an int *connector* data member. *instruction* is fairly straightforward in that it will be set through *splitString()*, along with *connector* being set, and then dynamically allocated and pushed onto **Command's** *CMDList. instruction* is simply going to store a single command, while *connector* is going to figure out what connector-type follows this command. If the connector type is || , then the identifier we will use in *connector* is 2. If && , we will use 3. If

the connector-type is just a semicolon, then *connector* will be 1. If the connector-type doesn't exist (i.e. there is nothing following the current command) then *connector* will be set to 0.

The second subclass of **Command**, **Execute**, inherits its parent's vector of **CMDs**. **Execute** will be responsible for going through the entire *CMDList*, tossing out any commands that don't exist or that our command prompt doesn't know about yet. And for each valid command(s) that it goes through, they will be executed then be popped out of the vector, so by the end of **Execute** going through all **CMDs** in *CMDList*, the resultant vector should be empty of all stored **CMDs.**

Once this cycle of command execution is complete, it is up to the interface class to signal to **Menu** that it is time to get the next prompt from the client.

Coding Strategy:

My partner and I plan on breaking up the assignment by working on seperate classes not including the Interface Class. Hamza will be working on the Command Class and Huzaifah will be working on the Menu Class. After each of those classes are finished, we will each tackle either the CMD Class or the Execute class.

Everyday we make progress in our respective classes, we will call or meet each other and talk about what design changes, details, and implementations we have made to them. We will also try to talk about how those design choices will affect the overall project and what changes we might have to make to adapt the project to our advantage.

After we finish each of those classes we will meet together and go over all of our classes and ideas to ensure coherency. Once we are satisfied with our execution and progress we have made towards those classes, we will make the Interface Class together to make sure that we include all the design details we previously implemented in our other classes.

We will work to debug any bugs that pop up together until the program works as we planned. We will then celebrate with cake, donuts, oreos, and milk!

Roadblocks:

First and foremost, bugs. Every program will have a bug at the first compile. Secondly, since we are coding our classes separately, we might not be able to weave in our classes together flawlessly. In addition to that, it may be difficult to explain our code to one another so that our partner knows how a function exactly operates.