Assignment 1: Design

November 1st, 2018

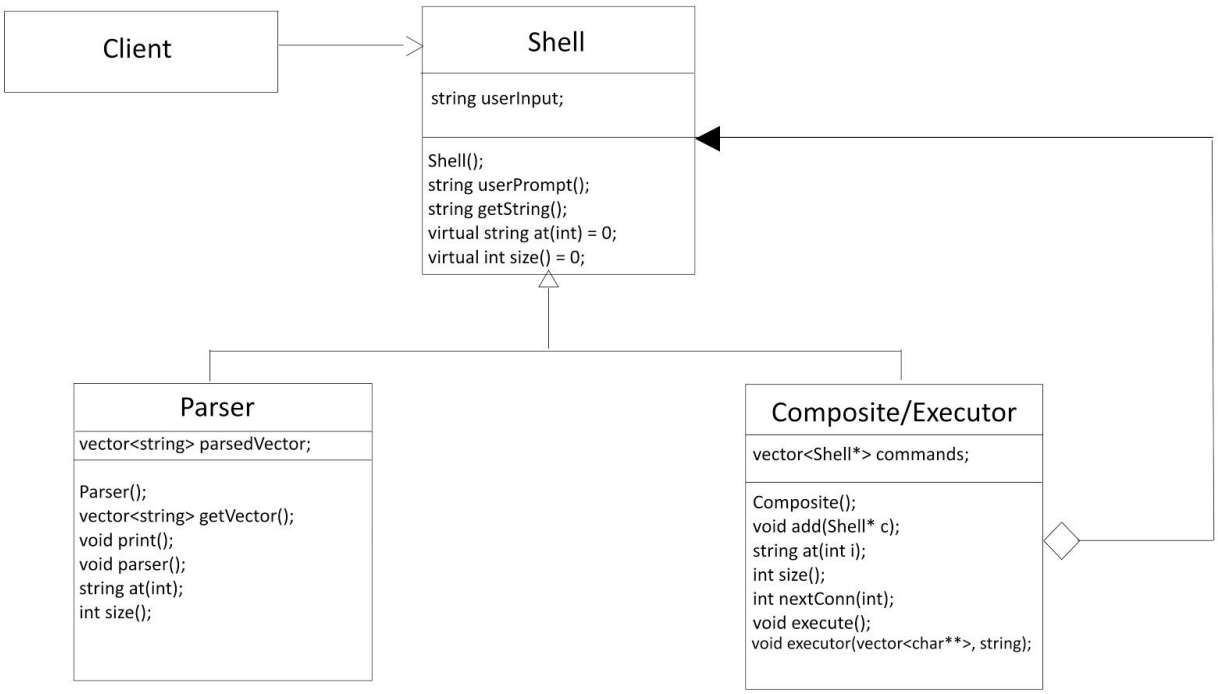
Fall 2018

Bryan Nguyen & Brandon Cai

Introduction:

Our design breaks the program down into a few simple steps. We have an interface “Shell”, we take user input, we parse the input in order to find how the user would like to have their commands executed, and then we will execute those commands in the order that it was parsed. We created a parser class which parses and stores the parsed strings in a vector of strings. We separated them by connectors and words. After that, we created our composite class, which will take the parsed object into a vector of our base class pointer which will then unpackage the string into char** and finally execute to make it appear like a real, basic command shell.

Diagram:


Client
Shell
string userInput;
Shell();
string userPrompt();
string getString();
virtual string at(int) = 0;
virtual int size() = 0;
Parser
vector<string> parsedVector;
Parser();
vector<string> getVector();
void print();
void parser();
string at(int);
int size();
Composite/Executor
vector<Shell*> commands;
Composite();
void add(Shell* c);
string at(int i);
int size();
int nextConn(int);
void execute();
void executor(vector<char**>, string);

Classes/Class Groups

CLASS GROUP SHELL

string userPrompt(); - This method is handles getting the input from the user via a getline. It also displays “username@hostname $ “ before that to let the user know the shell is active.

string getString(); - This method just allows us to get the userInput from the shell in order to utilize it in other classes.

virtual string at(int) = 0; - Gets the position at i of the vector in the class.

virtual int size() = 0; - gets the size of the vector in the class.

CLASS GROUP PARSER

Parser: Within the parser we have 5 methods and 1 attribute.

vector<string> getVector(); - This method allows us to get the vector from the parser in order to be used by the composite/executor class.

void print(); - This method prints the parsed vector when called. Mostly for testing purposes.

void parser(); - This is our parsing method. We chose to use the tokenizer from the boost library as it was the most efficient way of using delimiters to break up the userInput into indexes of a string vector in order to traverse through it to execute later on.

string at(int); - Returns the string at a given position.

int size(); - Returns the size of the parsed vector when called.

CLASS GROUP COMPOSITE/EXECUTOR

void add(Shell* c); - This method is to add the leaf objects into the composite class. In our program, the leaf is a vector of strings. We then add that our vector object leaf into our vector composite class.

string at(int i); - This method will return the element of the leaf class from the composite. It is used to take the strings outside of the vector of a vector to later convert them to char** to get ready for execution

int size(); - Returns size of the vector

int nextConn(int); - Finds the next connector in the string. And returns the position.

void execute(); - Our ultimate execute method. The strings inside the vector of a vector object is taken out and placed into a char**. It checks for connectors and depending on the connector, executes once it finds a connector, or finds the next command (for && and ||). This method unpackages data and packages the data to be ready for execution.

bool executor(char, string); -** The use of fork, execute, and waitpid are found here. It is what we see when we run rshell. The strategy of && and || are also found here. It takes in the vector of char** and a string that represents the connector. Once that is found, it executes the vector at element 0 and then at element 1 depending on the connector. Perror is also found here to check if the execute failed somehow.

Coding strategy:

We will break up the work as 50/50 to the best of our ability. Brandon will take care of the Composition part of the code and the Bryan will be in charge of the strategy implementation. We will integrate the segments together by pushing our code to the github repository and notifying each other of changes so we can make sure that the code is integrated along the way instead of waiting till the end when the code is done to implement in which the code may not be compatible. This way we won't run into catastrophic errors because of the incompatibility in our parts.

UPDATE:

The coding ended up being about half and half. Brandon took care of the parsing, while Bryan took care of the composition and execution. We also took care of everything related to our responsibilities such as gtests.

Roadblocks:

Even though we have planned this so that adding new nodes and classes should not be much of a problem, we still expect to have difficulties implementing new strategies and compositions into our code. Also, both of us are new to using the excvp(), wait(), and fork() commands and have little to no understanding on how to use them. We plan on just using the trial and error technique and researching how and when to use these methods. Furthermore, other than file i.o, we have never worked on executing commands that affect things outside of the compiler. One of the main roadblocks will definitely be in the limited knowledge of implementation and lack of experience in building shells and the bash environment. We plan on overcoming this roadblock by working on the project in a timely manner and being able to seek help regularly from external sources.

Another roadblock would be straying from our design as the project calls for it. We think that that would be a challenge to realign our thought process for the project as it comes together. We can overcome this roadblock by consulting the professor and teacher assistants for guidance in this regard.

One roadblock that can also occur is conflicting schedules on when one person can work on the project. We plan on solving this roadblock by setting up dedicated meeting times where we can group code and collaborate.

Another roadblock is that we feel as if our code is not what it should be. Even though it works, our code is most likely inflexible in its design. We strongly believe that we will have to redo this project in hopes to make it more flexible.