ASSIGNMENT 1: DESIGN DOCUMENT

Kinjal Mugatwala (Discussion Section: 026)
Tanya Malik (Discussion Section: 026)
Fall 2018

INTRODUCTION

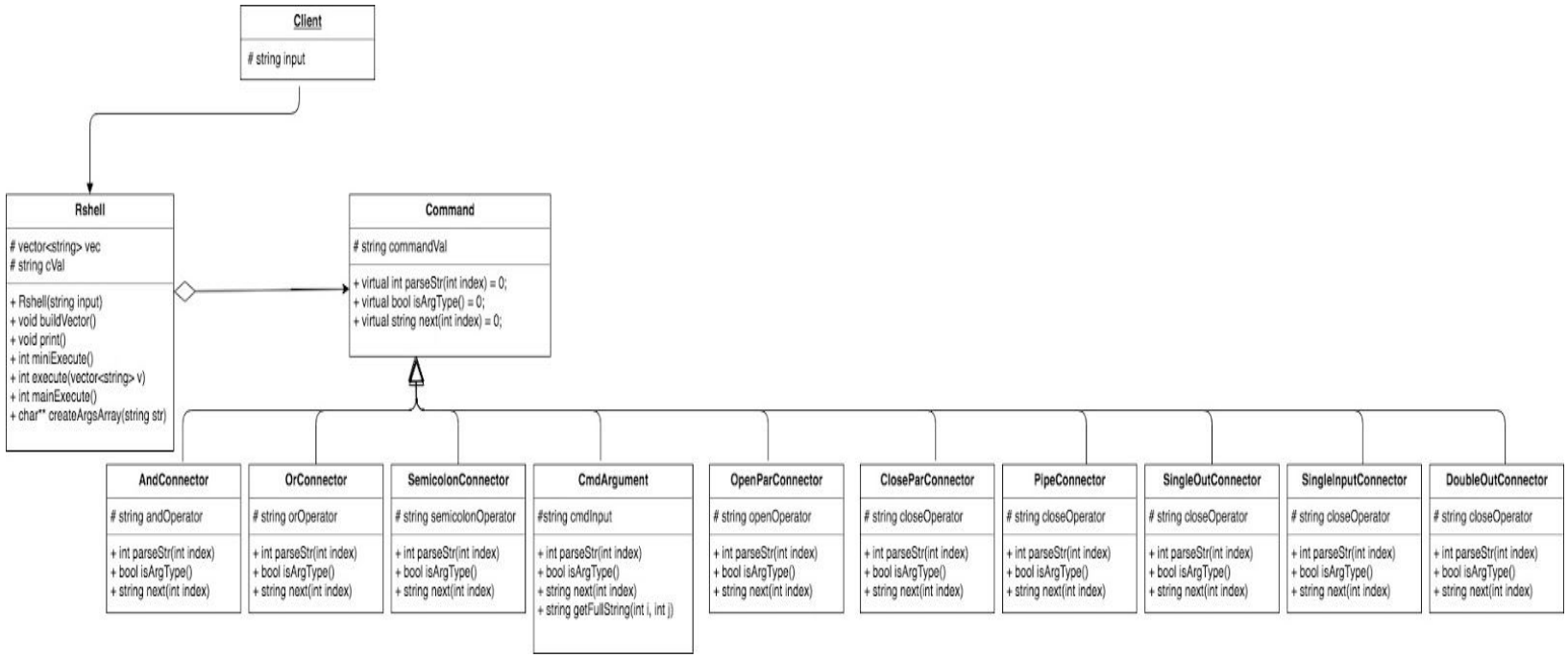
The main objective of our program is to use our command shell to read a command line, to organize the arguments into its special class, and then ultimately use these command keys to execute three main commands, which are fork, execvp, and waitpid. In order to make this happen, we need to have the client call on the Rshell class and input a command string. Using this string input value, we can divide the string into individual arguments by using Command's subclasses functions.

This is where the Command class comes in. The Command class is broken down further into Connector subclasses and a CmdArgument subclass. The Connector subclasses contains all the information about the operators related to its AndConnector, OrConnector, SemicolonConnector, OpenParConnector, CloseParConnector subclasses as these operators connect the individual arguments together. The CmdArgument subclass contains the string values of the special bash commands like echo, ls, and mkdir.

In order to execute the three main commands (fork, execvp, waitpid), we need an Rshell class. This class uses a vector data structure to help organize the individual Command elements and using this vector and strtok(), we can print a command prompt and execute those three commands as mentioned before.

Also, we have included the code for redirection for >>, >, <, and |. We used dup(), dup2(), pipe(), open(), and close() to help make these commands work properly.

UML DIAGRAM


Client
string input
Rshell
vector<string> vec
string cVal
+ Rshell(string input)
+ void buildVector()
+ void print()
+ int miniExecute()
+ int execute(vector<string> v)
+ int mainExecute()
+ char** createArgsArray(string str)
Command
string commandVal
+ virtual int parseStr(int index) = 0;
+ virtual bool isArgType() = 0;
+ virtual string next(int index) = 0;
AndConnector
string andOperator
+ int parseStr(int index)
+ bool isArgType()
+ string next(int index)
OrConnector
string orOperator
+ int parseStr(int index)
+ bool isArgType()
+ string next(int index)
SemicolonConnector
string semicolonOperator
+ int parseStr(int index)
+ bool isArgType()
+ string next(int index)
CmdArgument
#string cmdInput
+ int parseStr(int index)
+ bool isArgType()
+ string next(int index)
+ string getFullString(int i, int j)
OpenParConnector
string openOperator
+ int parseStr(int index)
+ bool isArgType()
+ string next(int index)
CloseParConnector
string closeOperator
+ int parseStr(int index)
+ bool isArgType()
+ string next(int index)
PipeConnector
string closeOperator
+ int parseStr(int index)
+ bool isArgType()
+ string next(int index)
SingleOutConnector
string closeOperator
+ int parseStr(int index)
+ bool isArgType()
+ string next(int index)
SingleInputConnector
string closeOperator
+ int parseStr(int index)
+ bool isArgType()
+ string next(int index)
DoubleOutConnector
string closeOperator
+ int parseStr(int index)
+ bool isArgType()
+ string next(int index)

CLASSES/CLASS GROUPS DESCRIPTIONS

Client: The client class simply takes in an input argument. The client calls on the Rshell class to access other classes. The input argument is the attribute of the base class and this input argument contains an entire command line. This command line input string must be parsed using the parseStr() function as it needs to be broken down into its individual pieces like connectors and main argument commands.

Rshell: RShell contains a vector of string elements. The vector would be our aggregation structure. The vector is filled with string elements because we need to acknowledge individual string arguments and each string connector in order to help execute three commands (fork(), execvp(), and waitpid()) in the correct order. We also need to implement the exit() function as we need to be able to exit the program. The print() function will help us keep track of what elements we have in our vector just for our own checking basis. The reason why we have addElement(), at(), and size() methods just in case if we need to connect the vector to another class where such a class needs to get to the vector's elements. In order to run the execution of commands, the mainExecute() will go through the parsed vector and run execute(vector<string> v) and this function will run individual commands with the miniExecute() function.

Command: Command reads in the input and separates each argument to determine the function of the argument. These arguments are then added to Rshell. Command has its own virtual functions and they are getArgType() and next(). The getType() function lets us know whether the type is a CmdArgument or a Connector. This is a virtual function as we need to know the type of each individual connector and possibly CmdArgument and so they should be implemented in the Command's child classes as well.

CmdArgument: CmdArgument is a subclass of Command. Any argument that is not a connector is determined to be a CmdArgument. Examples of CmdArgument are ls and mkdir. These arguments should individual words with no space in between. Based on what the argument is, next() must be implemented to take care of the user input and to allow us to get to the next input individual element.

AndConnector, OrConnector, SemicolonConnector, CloseParConnector, OpenParConnector, DoubleOutConnector, PipeConnector, SingleOutConnector, SingleInputConnector: These are all subclasses of the Connector class. The next() function is implemented with their own algorithms in each of these classes. And means that the command after the connector is only executed if the first command executes. Or means that the command after the connector is only executed if the first command fails. Semicolon means that the command after the connector is always executed. The parseStr() and getType() functions are inherited from Connector but remain empty in these classes. Open and Close Par connectors were created in order to help deal with the stacks formed in mainExecute() function to run commands in the proper order.

CODING STRATEGY

We decided to figure out how we want to parse the given input string and in order to do that, we decided to make parseStr() a virtual function in the Command class. We decided to change the return type of parseStr() from void to int just so that we can keep on moving along the input string and constantly storing these broken elements from the parseStr() into our vector in rshell.h. Then using our vector, we can execute the proper commands in the correct order using execvp(), fork(), and waitpid(). After coding, we will try testing extremely small commands and slowly build up on the way we create the input string and make additions to our code as we go along.

For assignment 3, we realized that there are two main objectives that need to be accomplished. One objective is to deal with the test command and the other objective is to handle parentheses and the order of the commands ran. Because we are a team of two, we decided that Tanya will handle the test command and Kinjal will handle the parentheses situation. After, we are done with our corresponding parts, we will merge our code.

For assignment 4, we decided to break it down by the redirection connectors (>, >>, <, |). We decided to create a miniExecute() function for each of these connectors because they all work so differently. So, by doing that, working on the coding became much more efficient. However, at first, we thought that pipe() would only be used for |, however, we later understood that it is also used for the other redirection connectors.

ROADBLOCKS

We are beginning to face another roadblock that can affect our future assignments and that is deciding whether we want to use an STL stack or STL queue to handle the parentheses placed within the commands during runtime. The problem with stack is that the commands will probably be handled in the opposite direction.

One main roadblock that we may face is how we are exactly going to "parse" the strings into its individual elements. The main problem we had was that we may have made our parse string extremely specific and so when we have to consider other possible connectors in the future, it could cause us to make so many more classes. Later, we will have to find out a way to generalize the parseStr() function.

Another roadblock we will probably face in the future is that we will have to make our execute() process much more detailed in order to look over every possible error that can occur. Because of our execute() function, we have to see what kind of tests we need to look over. We faced a huge problem with our test.cpp in our tests/ directory because we didn't know how to connect our src/ directory to the tests/ directory.

One more roadblock that we are going to face is how we are going to connect the tests director to the src directory and make sure the tests run. Testing can also become a huge challenge for us as commands become more complex and it will become much harder to see if the specific and complex commands work properly.

We have been facing several roadblocks in our 4th assignment. We had a huge struggle converting the filename from our vector to fit into the open() function. Our open function worked with string literals, but it didn't work with our fileName variable. To fix that, we found out that there was a space involved that was preventing us from running the open() function correctly.