Assignment 4: Updated Design

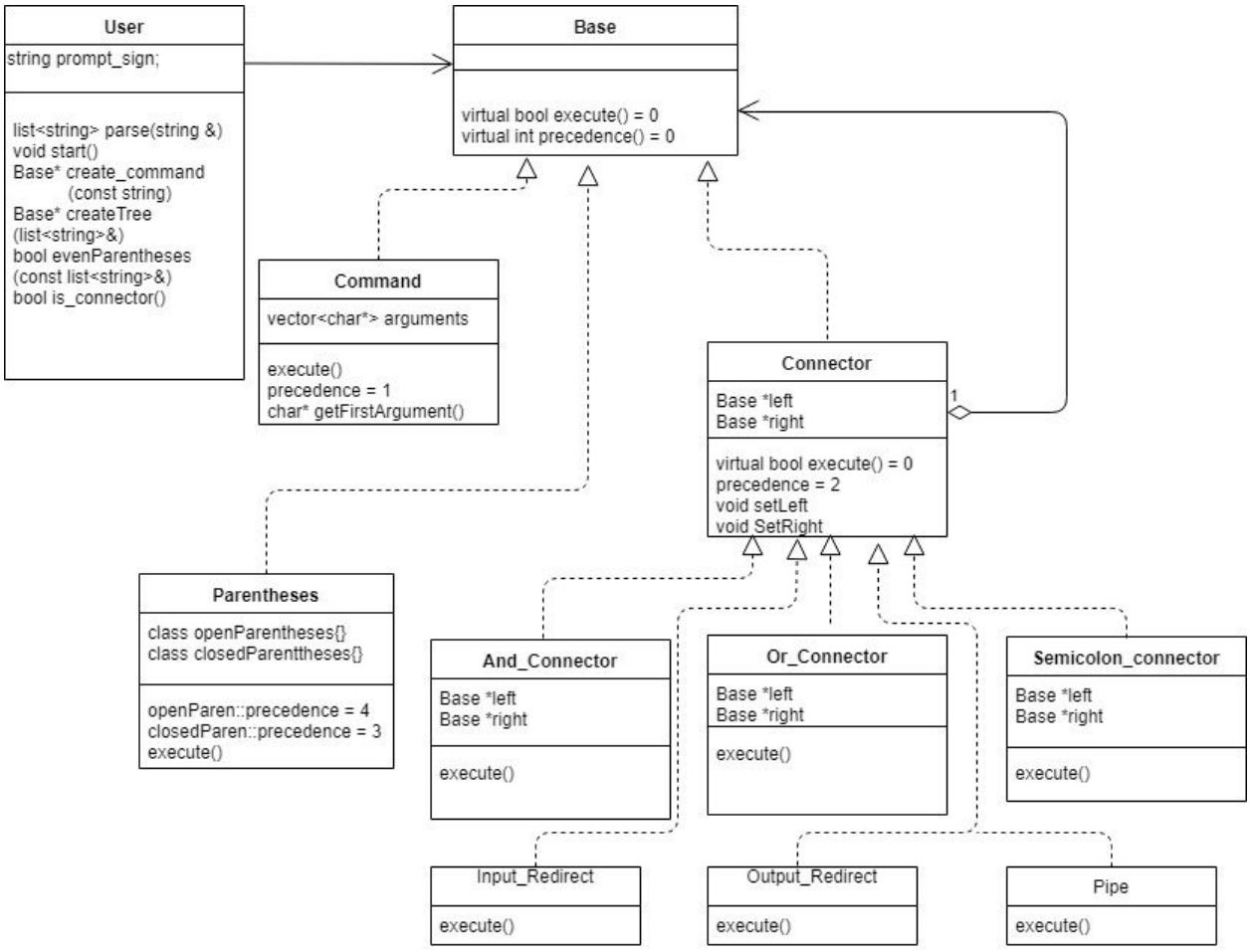
December 11, Fall 2018

Fangsheng Chao
Phillip Nguyen

Introduction:

The project is a C++ program named Rshell, that accepts Bash commands and executes those commands like with the Bash shell. The design should have no restrictions to the number of commands that can be chained together and it should handle any combination of operators. These commands and connectors do not have precedence except the inclusion of parentheses around the commands, which will group the commands together and change the precedence. In addition, Rshell now includes the test command, alongside its symbolic equivalent([]), which can test files and directories. Now Rshell supports input and output redirection, <, >, >> and it also supports piping, '|'.

UML Diagram:


User
string prompt_sign;
list<string> parse(string &)
void start()
Base* create_command
(const string)
Base* createTree
(list<string>&)
bool evenParentheses
(const list<string>&)
bool is_connector()
Base
virtual bool execute() = 0
virtual int precedence() = 0
Command
vector<char*> arguments
execute()
precedence = 1
char* getFirstArgument()
Connector
Base *left
Base *right
virtual bool execute() = 0
precedence = 2
void setLeft
void SetRight
1
Parentheses
class openParentheses{}
class closedParenttheses{}
openParen::precedence = 4
closedParen::precedence = 3
execute()
And_Connector
Base *left
Base *right
execute()
Or_Connector
Base *left
Base *right
execute()
Semicolon_connector
Base *left
Base *right
execute()
Input_Redirect
execute()
Output_Redirect
execute()
Pipe
execute()

Classes:

- User: This will be the main class that interacts with the composite structure. We have a string that is the input the user will type in for their commands and we will pass that string into a parse function. To achieve this, we will use strtok() to parse our input to get the information we need to execute the commands. The start() function begins the program and allows the user to start entering commands. We have will create and pass in local vector of char pointers into the Command Constructors to create them. Then we have a function that creates the command to run that will handle edge cases and different combinations of commands and connectors.
    - Now we have a function that checks for even amount of parentheses and if it does not have a even amount of parentheses, it will output an error message.
    - The design has changed from createCommand to createTree which calls createCommand that creates single commands to build inside the tree. We are building the tree by taking the user expression and converting it into postfix notation and evaluating the postfix expression. We chose to build it this way because postfix notation handles parentheses and it simplifies the precedence part of the assignment.
        - We passed in a list of parsed commands changing it from a vector because the while loops were easier to work with and we would not have to worry about indexes. We tried to use a queue but the queue caused too many seg faults so we switched to a list for simplicity.
    - createTree: This function builds all the commands and handles the errors when the user does not enter enough arguments. The algorithm to convert infix to postfix was found online and the source is commented inside the code.
    - Parse: This function now adds spaces before and after each parentheses since our parser has stayed with using strtok() and it still parses by spaces.

Composite Class Group:

- Base: This is the interface for the composite class group. It contains a pure virtual function, execute, that returns bool so we know whether the command successfully ran or it failed. It also contains a pure virtual function called precedence(), that returns the precedence of each object.

- Command: This is the leaf class of the composite pattern. Its string data member will contain the command to be executed, and the function, execute, will be implemented by executing that command using syscalls(fork, execvp, waitpid). This now returns precedence of 1, it is an arbitrary number that we've chosen. Additionally, it contains our written test command that also replaces the symbolic test command with the word test; we did it inside the command class since it was easier to handle edge cases. This now

has a function that returns the first argument inside the vector of character pointers, since open() takes a char*, we made this function return a char* so it would be easier to open the files without changing how we built our commands in the user class.

- Connector: Interface for the connector class that contains two Base pointers because they can interact with the commands or operators. This returns precedence of 2 for all of the connectors since they all have same precedence.
    - And_Connector: This will implement execute by first calling execute on its left child. If the left execution returns true then it will continue calling execute on its right child and return that result. If the left execution does not return true, then it will return false and the process stops.
    - Or_Connector: This class will implement execute by calling execute on its left data member and if this result returns false, then it will call execute on its right data member and return that result. Otherwise, the connector will return true.
    - Semicolon_Connector: This class implements execute by calling execute on its left data member and then it's right data member regardless of the value that the data members return.
    - Input_Redirect: Input Redirection has the symbol '<' and its purpose is to pass the *contents* of a file into the command instead of passing the actual file to the command itself. The easiest way to see this is the command *wc* and use it normally and with input redirection and you see that it performs the same action but input redirection does not have it print out the file name since the command only received a stream.
    - Output_Redirect: This includes Single_Redirect('>') and Double_Redirect('>>') Classes inside the single file. What single output redirect does is redirect the standard output to a file, and it creates a file if it doesn't exist and overwrites the entire file if it does exist. Double output redirect does a very similar job to single output redirect except it does not overwrite the file its writing to but rather appends it to the end of the file.
    - Pipe: The symbol for pipe is "|" and it is a form of redirection the sends the output of a command to another command. It allows the user to create a pipeline of commands that is a temporary connection between multiple commands/programs. This connection allows commands/programs to operate simultaneously and allows data to be transferred continuously instead of having to store the data in a temporary location and then starting another command to read and use that data.

- Parentheses: This class is new for this assignment and it only inherits from Base. The only purpose for this class is to return its precedence for open and closed parentheses, which is 4 and 3 respectively. It defines execute but does not implement it. We needed this for our infix to postfix notation and the objects do not contain any data, they are not present for when we are evaluating the postfix notation since during the infix-postfix conversion, all the parentheses are removed.

Coding Strategy:

We plan on breaking up the work into small but testable packets and evenly split up the work so both of us share an even workload. We will utilize GitHub and create our own branches to develop on so the master branch will be stable once we develop and test on our branches. Fangsheng focuses on testing the code and Phillip is focused on developing the different requirements. Fangsheng also handles the edge cases and us together figure out how to implement error handling for different edge cases.

We continue to use different branches on Github to test our code and we develop and test on hammer to ensure the program compiles and runs correctly. We also have started developing on the hammer server for stability and so we do not crowd our commits with additionally commits just to transfer from Ubuntu on Windows to the Hammer server. We also freshly clone our repo every big addition to our system to make sure that it is consistent and we it runs smoothly.

Roadblocks:

Our current roadblock is the issue with forking on Phil's hammer server, it displays an error message of unavailable resources but continues to work fine on Ubuntu and Fangsheng's Hammer. This situation leads me to believe that in the future, additionally components in the system may create zombie processes that we are unaware of and that can create many issues down the line. And again our method of parsing by spaces is not the most elegant due to additionally I/O redirection in linux not being separated by spaces. So adding additional functionality for more redirectional commands will take more time than usual and our code would have to be refactored to keep it readable and organized.