Assignment 3: Design Document

1 November 2018

Fall Quarter

Nathaniel Moore, Reuben D'cunha

Introduction

The current state of this project is a command line housed inside of a single class that takes in a string input and passes that string into a struct which parses a given user input. The parser heavily utilizes a tokenizer class from the boost library to break the input string into several parts based off of the three connectors (Semicolon, And, and Or) and the comment symbol (#). The structure of the program begins with an abstract Base class which has two composite subclasses Command and Connector.

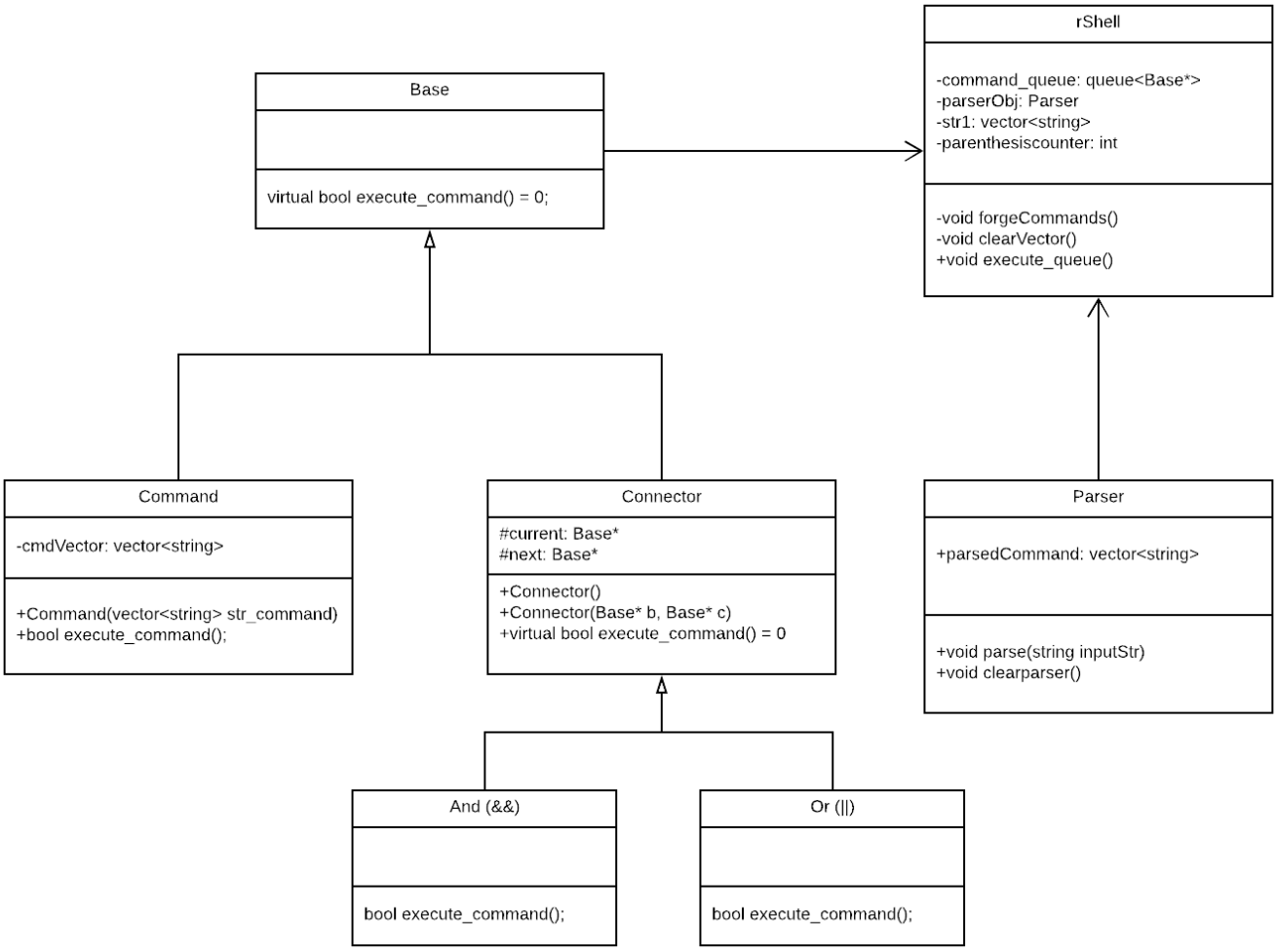
The Parser struct takes the user input and separates it into tokens and pushes it back into a vector. rShell utilizes this by creating a Parser object and grabbing the vector of parsed string tokens to use in rShell's own functions to instantiate Command and Connector objects that serve as a way to prime command inputs for execution.

The Connector object will be associated with its Command by sharing that Command's particular container member. Additionally, the Connector has two Base type pointers. The pointer named "current" will be used to detect whether or not the previous command ran successfully and the pointer named "current" will be used to execute the next command. Depending on what condition Connector receives the indication from the "current" pointer, the program will run differently through the bool execute_function.

Lastly, rShell is the primary interface for creating, storing, and executing the commands that have been inputted by the user and parsed by the Parser object. rShell's execute_queue function is used to execute and print the command line (if user's NetID is rdcu001, then execute_commands will print [rdcu001] $) and request user input to be passed in.

UML Diagrams

Composite Diagram



Classes/Class Groups

Base Class: An abstract base class which contains a constructor and a virtual void evaluate() function.

Command Class: A class derived from Base whose primary function is to obtain a vector of strings and convert them into a proper char array for the rShell to execute commands. It then stores converted commands into its member vector of char pointers. Command contains special conditions for detecting whether the command "exit" is parsed or whether the two versions of the test function ("test -e test.cpp" / "test test.cpp" or "[-e test.cpp]" for example) are parsed as a command. It then executes the appropriate steps for these commands.

rshell Class: rshell is where the execution of the commands occurs. It contains the main functionality of the program which includes a function forgeCommands() which creates the Command and Connector objects and executes the queue of parsed commands. The rShell's forgeCommand() function utilizes the helper function clearVector(). Additionally, the ability to handle precedence operators has been input into forgeCommands().

Connector Class: A class derived from Base which serves as the composite for the three connector types. The connector subclasses will be called based on the input provided by the user. The two base pointers housed by the connector class contain the current and the next command to be executed by the rshell. It contains two base pointers which can be assigned the Connector subclasses "And(&&)" and "Or(||)" which depend on the success or failure of the previously parsed command. It can also contain a simple Command object which will always execute the next command contained in the Base pointer.

And(&&): A connector which will execute(call upon rShell) the "next" command only if the "current" command was successfully executed. After execution, it will call on the container to refresh its commands.

Or(||): A connector which will execute(call upon rshell) the "next" command only if the "current" command was not executed successfully. After execution, it will refresh its commands by calling on the container.

Parser: A special struct which contains the means for tokenizing strings. The user input is filtered through here using the boost library tokenizer and stored in a vector of strings.

Coding Strategy

The work needs to be split up into implementing the classes, parsing the string correctly, and ensuring that no errors arise from using syscalls. Parsing and classes need to be created at the same time in order to integrate them together properly.

Classes need to be implemented in a way that grants us flexibility in adding new functions while still accomplishing the goal of each assignment. For example, the last and next pointers in the Connector class will both be Base type pointers just in case a new kind of conjunction appears later on in our assignments. **Reuben** will be responsible for implementing the functionality of most of the classes.

Parsing will need to involve taking in a string from the command line and passing it into a function which will tokenize the string based off of the key connectors Semicolon, And, and Or. We need to design the parsing function so that it is straightforward and simple. All it needs to do is associate a Connector with a Command and then locate the Connector's neighbors so that it can check if the condition for its previous neighbor was met or verify if they even had a condition in the first place. Parsing the string relies heavily on classes being implemented correctly, or else unintended behavior will ensue. **Nathaniel** will be in charge of creating a proper parsing system both outside and inside the classes.

Commands need to be executed without causing hidden errors. We will need to take extra care in using the fork(), exec(), and waitpid() functions so that the program does not behave in an unintended way. It would be wise to design our program such that memory is properly deallocated through a destructor in our classes, however this will be addressed when we become more competent with using syscalls. Learning to use and actually implementing syscalls will be **both** our responsibilities.

Roadblocks

First of all, we may encounter difficulties in leaving the program as open-ended as possible. Depending how we choose to develop the program later on, our design choices might restrict to us to specific solutions that could be unnecessarily difficult to implement. We need to ensure that we consider other options as well and explore how well they solve any problem we encounter. Some solutions may seem easier, but if we choose hastily then problems may show up later on.

Additionally, memory leaks may eventually be an issue as well because of our inexperience with bash and syscalls. Our greatest fear is that, depending on how we approach the program, memory leaks will become apparent. Therefore, we will have to check for memory leaks by creating unit tests specifically for the deallocation of processes and pointers.

As of the second iteration of the program, we had run into a variety of roadblocks which challenged our ability to successfully create the program. When creating the Parser's parse() function, it was difficult to find an exact approach to correctly read the commands and their tokens into the storage vector. There was not a standardized way of presenting the commands to the rShell, and because of that we had run into huge problems with using system calls because the non-standard way of parsing strings. The char pointers were not passed into the execvp() syscall correctly, and resulted in unrecognized names being seen by execvp() for an otherwise correct command.

Additionally, we have discovered that we tend to overcomplicate our solutions. Previously, we created several different functions to handle the execution of commands, as well as using recursion to assign Commands and Connectors with one another. However, this only proved to delay our progress and it made bugs significantly more prominent. Therefore, we are going to take an extra step and lay out simple, concise plans on how the next portion of the assignment is going to be implemented. We believe a strategy such as Scrum or Kanban would useful for this.

Another roadblock has been the difficulty of creating nested precedence operators. Because of the way our forgeCommand() function is structured, we may have to utilize recursion in order to achieve nested precedence assignment.

-------X-------