Assignment 1: Design

October 20, 2017

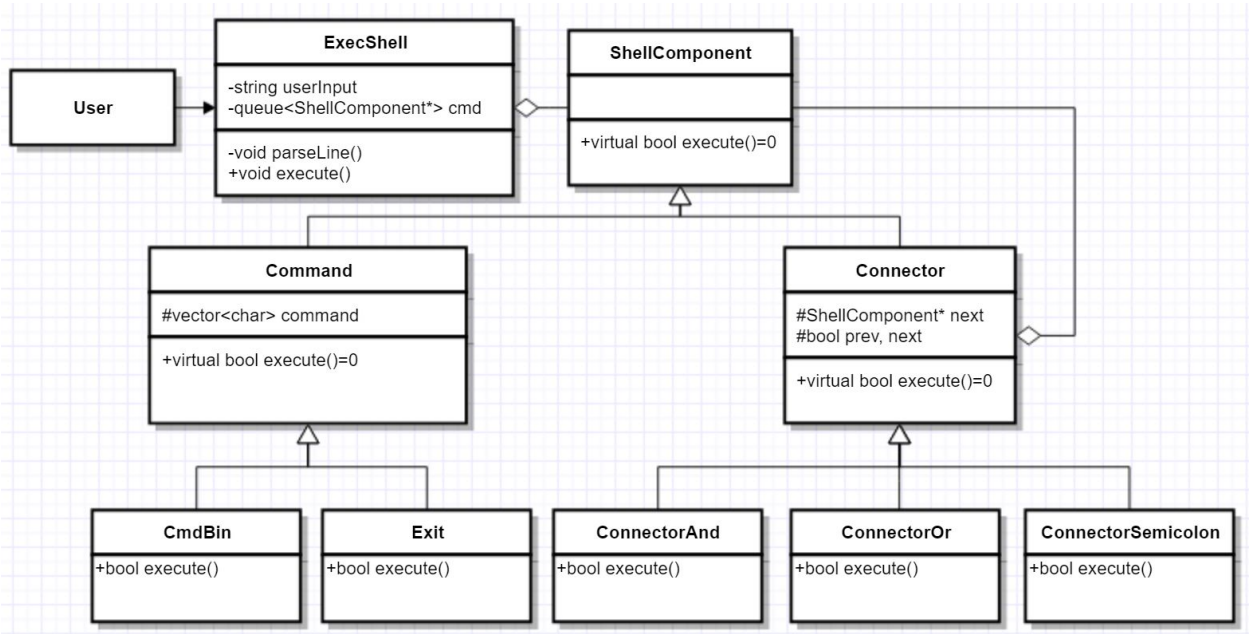
Fall 2017

Liang Xu & Patrick Porter

Introduction

This is the RShell program of Liang Xu and Patrick Porter written in C++ for CS100 Fall 2017. Our design treat each command and connector as a component. During execution, the program executes each component with its order determined by a queue. The program will continuously acquire bash commands including connectors from user and execute these commands, and the loop will stop whenever our own command, the exit command, being inputted. It also support reading and executing multiple commands separated by ";" in one line. When a line is being read, it will be parsed into program-readable segments and sent to corresponding shell components waiting for executions.

UML Diagram



Classes/Class Groups

ExecShell class is designed that the users only needs to call the public function, execute(), to execute their commands; all the specific execution processes and orders will be treated internally. After calling execute(), the program will first call the private function, parseLine(), to parse through the user input and format it into program-readable segments, and will push each command segments (after converting

to corresponding component) into a queue, cmd. When it finishes parsing, execute() will pop each element in the queue and call execute() for each component. Later, PID will be considered to determine the priority of execution.

ShellComponent class is an abstract class which is an interface for its derived classes, command and connector. It has a pure virtual class called execute() which executes commands or connectors and returns whether it executes successfully.

Command class is a primitive class of ShellComponent for treating and executing shell commands. It has a protacted data member which is an vector of characters of command. The program will read each character of command at one time and separate them into shell command and target command which command operates on. Command class has two derived classes, CmdBin and Exit. CmdBin subclass deals with all bin commands whereas Exit subclass only deals with the exit command defined by ourselves.

Connector class is a composite class of ShellComponent for treating connectors, such as “&&”, “||”, “;”, etc. It has a data member, next, is a pointer of component. It can be the pointers of command objects which execute commands. The class also has a boolean data member which indicates whether the previous command being successfully executed. The state of the next command will be returned by execute() function. It has three derived classes corresponding to different connectors for now.

Coding Strategy

We decided to break up the project into two major tasks: parsing user input into readable commands and creating a system to execute these commands properly with the connectors. Patrick will be in charge of parsing user input and Liang with execution.

Our original idea to parse the user’s input was to use the getline() and cin functions, but we realized that this can lead to complications in both performance and potential tests. Instead, we are going to use the cstdlib’s strtok function. This function will allow us to break up the user’s input using a simple delimiter, the “ “ character. Since all the commands, their parameters, and the connectors are separated by spaces, we thought this would be a good implementation. After calling the strtok function, strtok will return a series of tokens that we can read in and combine and execute.

In terms of the execution of each command, we plan to push each token onto a queue. After that, we will pop off the elements of the queue one at a time and execute based on the value.

Roadblocks

One potential roadblock we can see is, during the parsing of the input, figuring out how many parameters one function might have. For example, if we have “echo ‘Greetings user’”, the tokens would be “echo”, “Greetings”, and “user”. We will need to figure out, per command, how they end and execute them accordingly.

Another potential roadblock can be seen with our planned implementation of the execution. Using the queue, like with our other roadblock, we will need to figure out where a command ends and how to execute properly.