Assignment 1: Design

October 20, 2017

Fall 2017

Moker(Ke) Bellomo

&

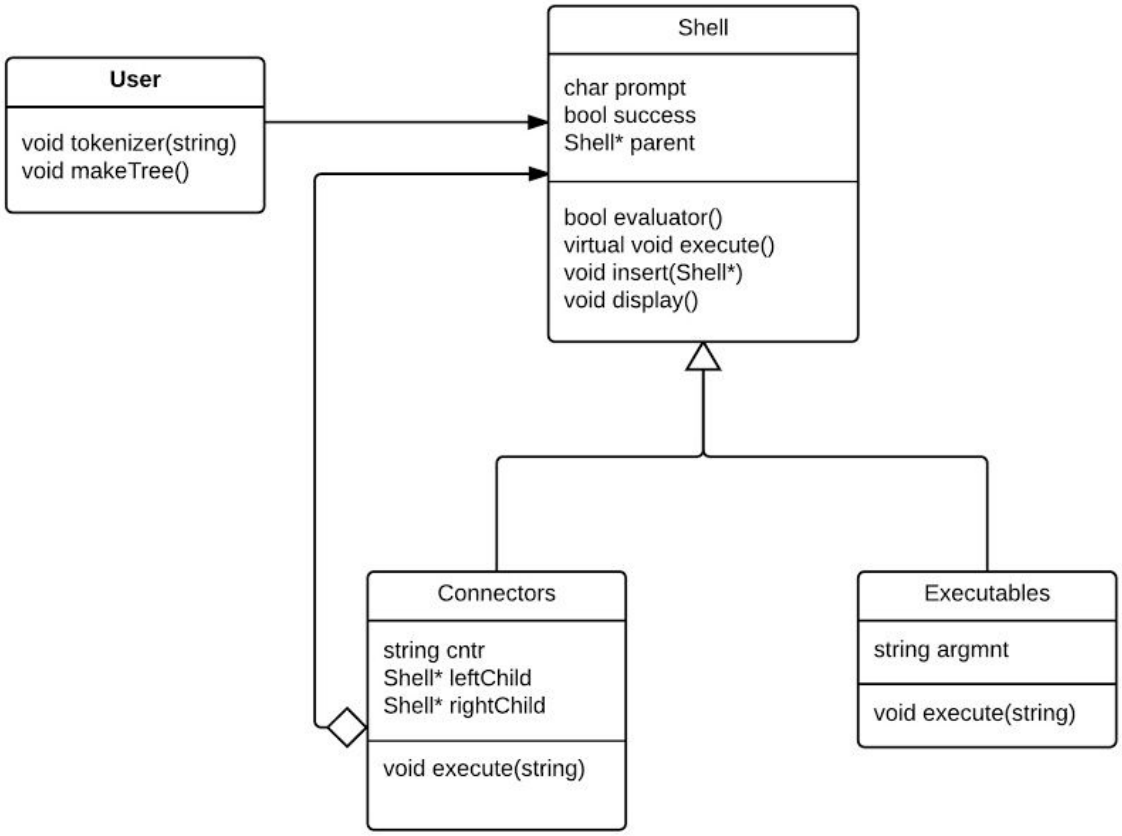
Jason Terrazas

Introduction

We are creating a basic command shell in C++. Our shell will first print a command prompt and then read in a command on one line. The commands will take the form of ‘executable [argumentList] [connector cmd]’. The connectors, or delimiters, such as ‘ || or && or ;’ will be able to run multiple commands at once. If the command is followed by ‘;’, the next command will always execute. If the command is followed by ‘&&’ and succeeds, then the next command will also execute. However, if the command is followed by ‘||’ and fails to execute, then the next command will run. There should be no limit to the number of commands that can be chained or combined together. There will be a built in exit command that will force an exit from the shell. A hash, ‘#’, character can be used in front of a line to write comments and will not execute instead.

This assignment will be implemented by breaking down the user entered command line into smaller tokens and turning them into nodes. Using these nodes, we will create a binary tree consisting of a root, the last connector, internal nodes, consisting of Connector class nodes and leaf nodes made up of Executables class nodes. We will have a function that will traverse the binary tree and execute each leaf node/Executable based on the success of the argument recorded by a bool variable and its parent.

UML Design Diagram



Classes/Class Groups

The project for creating a shell terminal will consist of one class group made up of the Shell base class and its two derived classes: connectors and executables. The User client will have access to the tokenizer function which parses the command line into smaller parts to be made into nodes. The User will also have access to the makeTree function that creates the binary tree. The Shell base class will handle most of the work of the program by displaying the interface and traversing through the binary tree. The binary tree will consist of Shell nodes that will be either a Connectors/composite node or a Executables/primitive node.

After the binary tree is completed, the leftmost node, which is the first argument, is executed by the execute function. If successful, the evaluate function will update the bool variable success within the Shell node to true. Based on the outcome and its parent node, the connector, it will either end the process and await the command line from the user or move on to executing the next argument. This process will continue to the end of the command line unless terminated by not meeting the connector's conditions.

The two children of the Shell base class are the Connectors and Executables. The Connectors are internal nodes as they are composites in the binary tree unlike the Executables which will only be leaf nodes as they are primitives. As such, Executables will not have any child nodes and Connector will have either an Executable right child and a left child is a Shell node because it can be an Executable or a Connector.

Coding Strategy

We will follow the Kanban strategy and will plan, develop, and test one feature before moving onto the next feature. Since the Shell class is the most important part, we will both implement it together using the driver and navigator strategy as there is no cost and also help us both understand how the base shell will work when writing the child classes. We will then separate the work between the Connector and the Executables with each member doing one class respectively. We will then reconvene and discuss and how we made each subclass before integrated them to the base shell. We will more than likely end up doing this multiple times due to mistakes and bug fixes.

All the work will be split into multiple branches in Github. We will first branch off the master branch, containing the final version of the program, into a development branch where most of our code will be done and tested. This branch can be branched off into multiple feature branches that will be our main workspaces for developing functions. Both master and development will have a bugfix branch to fix any mistakes that are made.

Roadblocks

Our biggest roadblock could possible be is trying to parse the command line into individual commands successfully. We will either use the strtok function or the tokenizer class from the boost library to overcome this potential issue. There could also be compatibility issues when we merge our feature branches into the development branch. Finding the mistakes and from the merges and any logical mistakes which could take up most of our time compared to actually implementing our code. Outside of programming, another roadblock could scheduling issues.