Assignment 1: Design

January 29, 2018; Winter Quarter
Last Updated: February 11, 2018

Jonathan Oaks joaks001 SID:861303298

Introduction

This design is for a command shell program “rshell” written in C++. The rshell program will present a command prompt ($) which will read commands in the form:

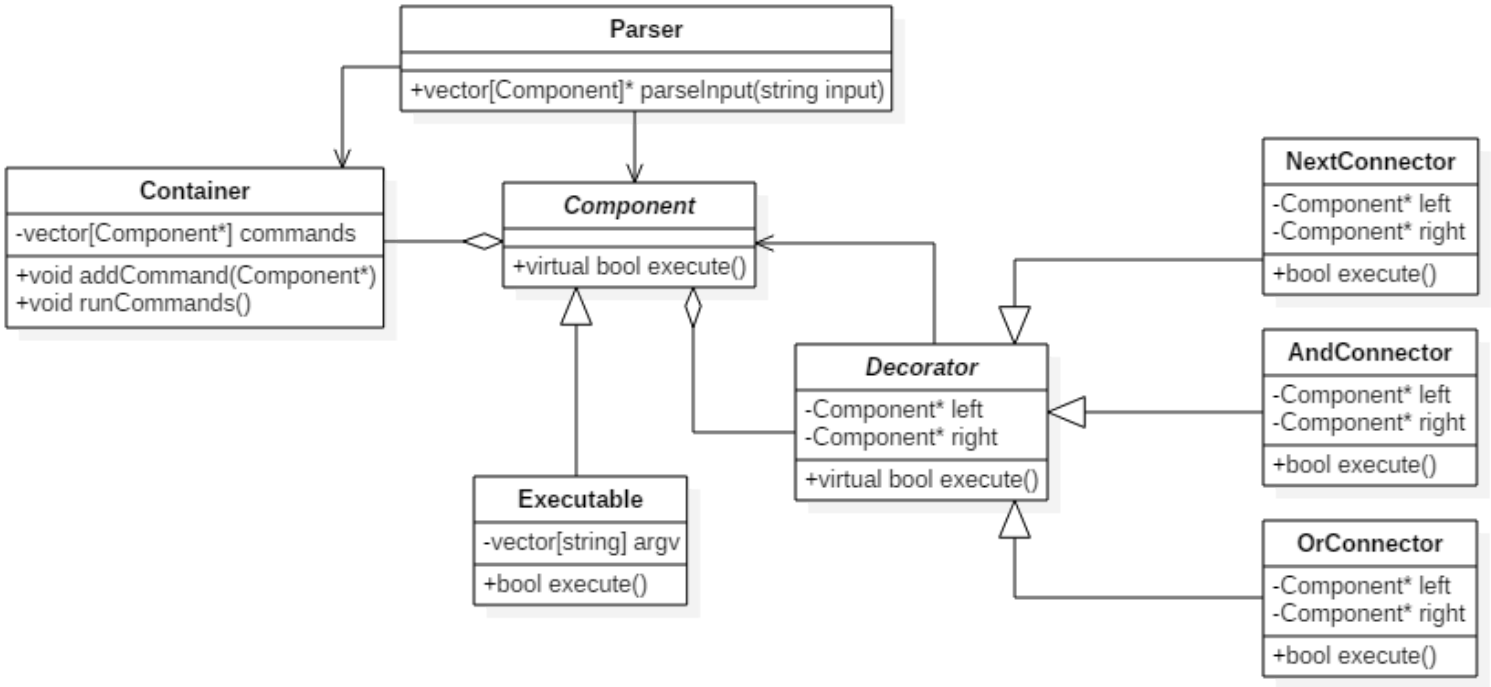
$ [command] [arguments] [connector]

where [command] is any bash command located in /bin, [arguments] are any arguments those bash commands may take, and [connector] is one of {;, &&, ||}.

Connections allow for multiple commands to be run from one line based on the condition of the connection provided: ; will execute the next command, && will execute the next command only if the first command was successful, and || will execute the next command only if the previous command was unsuccessful.

There is no limit to the amount of commands that can be chained together with connections, and there can be any combination of of connections in a single line. The program will run until the user exits rshell using a custom command.

Diagram



Classes/Class Groups

class Component

The base class for what will become the executable commands and the decorators to those commands. Has one virtual function execute() that returns a bool indicating the success of the commands its children will run.

class Executable : public Component

On construction, assigns a vector of strings to an argument vector. When execute() is called, the process is split with fork(). The child process runs the command with arguments contained in argv. The parent process waits for the child process to exit and returns the success status of the child process. If the exit command is entered, a global variable is updated to stop running the remaining commands and to exit the command prompt loop.

class Decorator : public Component

The base class for the three types of connectors. Contains two pointers to Components for a left node and a right node. The decorator structure indicates that the Components will be organized as an expression tree, with the Decorators acting as the

branches and the Executables as the leaves. The execute() function will, in general, return the success of the child nodes.

class NextConnector : public Decorator

Representing the semicolon. When its execute() function is called, it calls its left node's execute(), then returns the value of its right node's execute().

class AndConnector: public Decorator

Representing the logical && operator. When its execute() function is called, it calls its left node's execute(), then if it was successful, it will run its right node's execute(). If both nodes return a success, then AndConnector will return a true, otherwise it will return a false.

class OrConnector : public Decorator

Representing the logical || operator. When its execute() function is called, it calls its left node's execute(), then if it was not successful, it calls its right node's execute(). If either node returned true, then OrConnector will return true, otherwise, it will return false.

class Parser

Converts the user's input string to a full fledged expression tree. It first splits the string into tokens by " and ' characters. Then it reconstructs compounded strings into a single token so that strings can have quotes within them (e.g. " a 'bc' d" or 'a "bc" d'). It then splits the tokens into more tokens separated by a space, a semicolon, or parenthesis, keeping the semicolon and parentheses tokens as individual tokens. During this step, if the comment character (#) is found at the start of a token, a comment has started and the remaining tokens are discarded as comments. The remaining tokens are converted from an infix notation (ls && ls) to a postfix notation (ls ls &&). The postfix is converted to a Component expression tree, declaring the necessary classes for each token. The expression tree is then added to the Container.

class Container

Contains a vector of Components, which are expression trees. When its runCommands() function is called, it iterates over its vector, running each command.

Coding Strategy

The main steps of this design are:

1) Tokenizing user input to commands to be run
2) Chaining commands together
3) Executing /bin programs in parallel to the rshell

Implementing from the inside out seems to be the most prudent path.

1) Verify the syntax used by the system calls to run /bin programs, updating the specifics in the design document as necessary
2) Implement chaining using premade Commands to test functionality and edge cases.
3) Implement string tokenization and Command construction.

By implementing the Executable first, we ensure that if a Command is formatted correctly, testing it will produce the correct outcome. This reduces the possible errors encountered when implementing Command chaining; rather than wonder if the programs are executing properly or if the chains are running properly, we can be sure the error lies with the chains. Once Command chaining is implemented, the use of premade Command chains can highlight errors with certain combinations of Commands or Connectors. After both Executions and Commands are verified to work correctly, tokenizing the input string properly will have the minimal room for external errors.

If these tasks were to be completed independent of each other, the outer layers should be tested with placeholder functions that print the expected state of the program following their completion, e.g: Completing the Parser independently can be rudimentarily tested by printing out the final lists of tokens and Components. Completing the Command task independently should print the expected programs to run and arguments to pass in. Test cases could be made to demonstrate the validity of the Next, And, and Or Connectors by assigning test values to the boolean variables. Executing the /bin programs is the most inward step of this process, so it can be completed as normal. Integrating these tasks together should be seamless, but each case should still be tested for possible errors introduced during integration.

Roadblocks

The Parser class currently holds a lot of functionality inside its only function. It's possible each step in the parsing process could be its own private method or class that interfaces with the Parser, and it would probably be cleaner on memory that way as well. However, as it already functions as expected, I wouldn't want to split the functionality at this point. If in the future, more functionality is required of the parser, then I would focus on decoupling the functionality at that point.

The Container class currently is not doing much. In the original design, I had it set up so that many expression trees would be pushed into the Container, delimited by the semicolon, until I realized that the semicolon could just be its own sort of connector within the expression tree. The Parser could just as easily run the final tree when it's done creating it, but to avoid giving even more functionality to the Parser and as it's still possible that future assignments could have a use for a Container type class, I'll leave it in.