Assignment 1: Design

January 29, 2018; Winter Quarter

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:

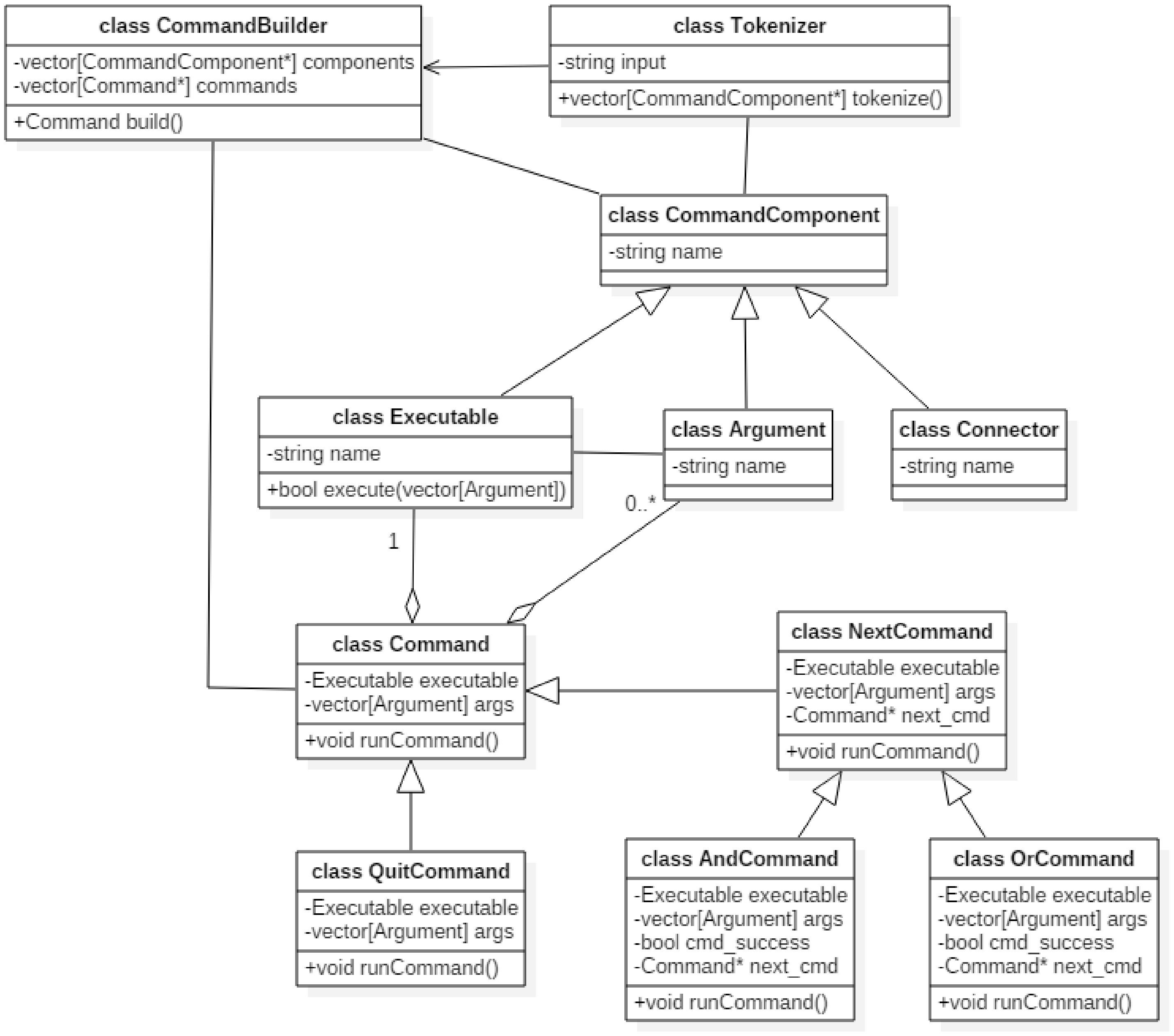
$ [command] [arguments] [connector]

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



class CommandBuilder
-vector[CommandComponent*] components
-vector[Command*] commands
+Command build()
class Tokenizer
-string input
+vector[CommandComponent*] tokenize()
class CommandComponent
-string name
class Executable
-string name
+bool execute(vector[Argument])
class Argument
-string name
class Connector
-string name
0..*
1
class Command
-Executable executable
-vector[Argument] args
+void runCommand()
class NextCommand
-Executable executable
-vector[Argument] args
-Command* next_cmd
+void runCommand()
class QuitCommand
-Executable executable
-vector[Argument] args
+void runCommand()
class AndCommand
-Executable executable
-vector[Argument] args
-bool cmd_success
-Command* next_cmd
+void runCommand()
class OrCommand
-Executable executable
-vector[Argument] args
-bool cmd_success
-Command* next_cmd
+void runCommand()

Classes/Class Groups

class CommandComponent

The base class for the family of components that make up a full command. All components store a string for their name which will be used to identify specifically what kind of component they should be and, if it's an Executable, which program to run.

class Executable

Inherits from CommandComponent. Its name field corresponds to an executable in /bin, which it will try to run through its execute() method. The execute() method will run the C++ system calls for parallel processing and execution and attempt to run its corresponding program. It optionally accept a vector of Arguments that it can pass to the /bin program as well. The execute() method returns a boolean indicating whether the /bin program was successfully run or not.

class Argument

Inherits from CommandComponent. Stores the string of an argument to be used by a /bin program, if any.

class Connector

Inherits from CommandComponent. Stores the string indicating what kind of connection the Command will have, if any.

class Command

A basic command type and the one that will be formed if no Connector is found when building the Command. It is composed of an Executable and a vector of Arguments. The runCommand() method will tell its Executable to run its program, passing in any Arguments associated with running that program.

class QuitCommand

Inherits from Command. A custom quit command that will exit the rshell program upon running.

class NextCommand

Inherits from Command. Corresponds to the presence of the semicolon (;) Connector in the input string. Contains a pointer to the next Command to be run after the current Command has finished, regardless of its success.

class AndCommand

Inherits from NextCommand. Corresponds to the presence of the && in the input string. Will run the next Command only if the Executable from the current Command returns a successful boolean.

class OrCommand

Inherits from NextCommand. Corresponds to the presence of the || in the input string. Will run the next Command only if the Executable from the current Command returns a successful boolean. Otherwise, the next Command will be skipped.

class Tokenizer

Takes the user-input string and converts it to a vector of CommandComponents. The input string is initially delimited by spaces into a vector of strings. That vector is then processed further to split the semicolon connector into its own string and remove any comment blocks. The remaining strings are converted into CommandComponents and returned by the tokenize() function.

class CommandBuilder

Build commands out of the vector of CommandComponents received by the Tokenizer. Since the user input string should be in the form executable-argumentList-connector, the front component in the components vector will likely be an Executable, and the following items in the vector will either be Arguments, Connectors, or nothing. The arguments are stored in a vector until either a Connection or the end of the components vector is found. Depending on the name of the Connection, or if there is no Connection, the appropriate Command type will be created and the Executable and Argument vector will be passed into the new Command. This Command will then be stored in the Builder’s command vector.

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 Tokenizer independently can be rudimentarily tested by printing out the final vector of Commands or CommandComponents depending on the scope of the Tokenizing and Command assignments. 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 Commands 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 C++ system calls may require a different specification of variables than the ones currently listed. If so, methods should be included to handle the conversion to the proper format.

Tokenizing the input string may present other possible outputs than presumed. The described method for tokenizing strings to components may be doing more work than necessary. Depending on the results, the Tokenize class may be able to be simplified.

The CommandComponent family of classes may prove to be unnecessary depending on the efficiency of the Tokenizer and Builder classes. If so, then Argument and Connector could be removed and Executable moved to its own class.

The assignment instructions aren’t clear on how comments should be handled. The current plan is to ignore comments until a connector is found and proceed from there, however the expectation may be to ignore every character after a comment is announced. Either way should be an easy fix, but the requirement isn’t clear yet.

There may be an issue with blank commands, or multiple connectors strung together with no commands in-between. This can be handled by treating an empty executable field as either a failed or successful execution and continuing from there.