Assignment 2: Writing a Basic Command Shell

October 28

Fall 2018

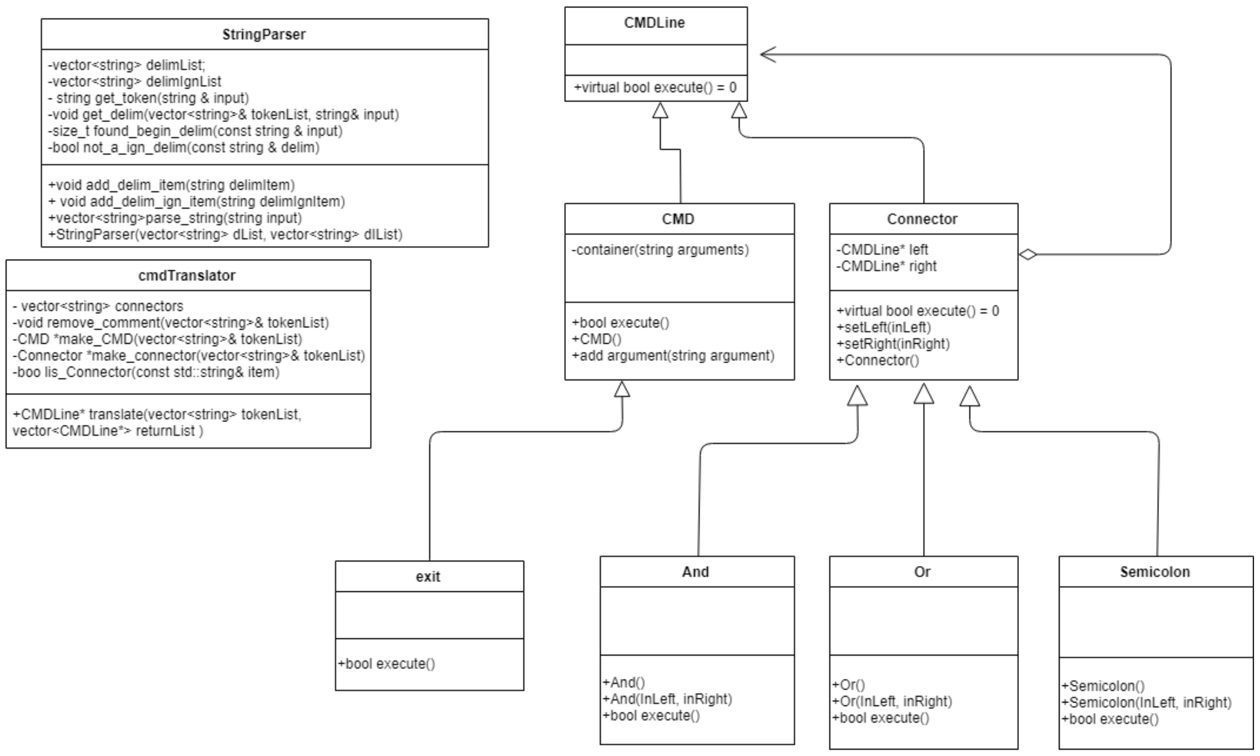
Christian Campos

Brandon Tran

- Introduction:

  The main task of the assignment is to create a shell for the user to enter commands , arguments, and connectors in one line. Further, we are mainly using a composite pattern to implement the user interface. The commands and connectors will be implemented as hierarchy tree structure. The top of the hierarchy is CMDLine abstract class. The CMDLine base class will have virtual bool executed function for all the subclasses to inherit. The next layers of the trees are CMD and Connectors and then children of the CMD and Connectors are Exit,And, Or, and Semicolon. The CMD class will execute the appropriate commands commands using fork, execvp, and waitpid. Lastly, a Parser class was created to parse the user inputs from invalid inputs.

- Diagram:


StringParser
-vector<string> delimList;
-vector<string> delimIgnList
- string get_token(string & input)
-void get_delim(vector<string>& tokenList, string& input)
-size_t found_begin_delim(const string & input)
-bool not_a_ign_delim(const string & delim)
+void add_delim_item(string delimItem)
+ void add_delim_ign_item(string delimIgnItem)
+vector<string>parse_string(string input)
+StringParser(vector<string> dList, vector<string> dlList)
cmdTranslator
- vector<string> connectors
-void remove_comment(vector<string>& tokenList)
-CMD *make_CMD(vector<string>& tokenList)
-Connector *make_connector(vector<string>& tokenList)
-boo lis_Connector(const std::string& item)
+CMDLine* translate(vector<string> tokenList, vector<CMDLine*> returnList)
CMDLine
+virtual bool execute() = 0
CMD
-container(string arguments)
+bool execute()
+CMD()
+add argument(string argument)
Connector
-CMDLine* left
-CMDLine* right
+virtual bool execute() = 0
+setLeft(inLeft)
+setRight(inRight)
+Connector()
exit
+bool execute()
And
+And()
+And(InLeft, inRight)
+bool execute()
Or
+Or()
+Or(InLeft, inRight)
+bool execute()
Semicolon
+Semicolon()
+Semicolon(InLeft, inRight)
+bool execute()

- Classes/Class Groups:

  - ➢ CMDLine (base)

    The command line is the abstract base class. It will be the interface of the sub classes. A pure virtual function execute() will require all subclasses that inherit from the interface to execute a task, whether is is a composite or a leaf.

  - ➢ CMD

    The CMD class is a leaf to our composite structure. The CMD's purpose is to encapsulates one command and its arguments into an object. Example: If we have the command ls -a; pwd, then ls -a will be made into one CMD object and pwd will be different. We are treating a command input and its arguments as a single object.

  - ➢ Connector

    The connector class is composite class that contains two CMDLine base objects. It is a parent class which contains only valid connectors that separating two command line objects. The children are And , Or , and Semicolon by the following assignment specifications. The commands should obey to the connector rules to decide what it should be executed.

  - ➢ Exit
    The Exit class sole purpose is to exit the program when the user enters the exit command. It is a subclass that inherits from CMD class.

  - ➢ And(&&)

    The And class is a child of connector. The And operation holds two base pointers which are data pointers. The left data pointer will executed first and it was able to execute properly, then the other data pointer will executed. The right data pointer will executed if and only if the left data pointer was able to complete its instruction(s).

➢ Or (||)

The Or class is a child of connector. The Or operation inherits two base pointers from the connector. The Or operation will executed only if one of the base pointers has a valid command input. The right hand data will executed if and only if the left hand data failed.

➢ Semicolon (;)

The Semicolon class is child of connector. The Semicolon separates two CMD objects from the input. The semicolon operation will always executed the next command.

➢ Parser

The Parser class is to parse the user's input and convert to CMDLine objects. It's a class that doesn't inherits or composite the base class. Further, the Parser class will contain c string library strtok function to parse the spaces , connectors, commands from the terminal.

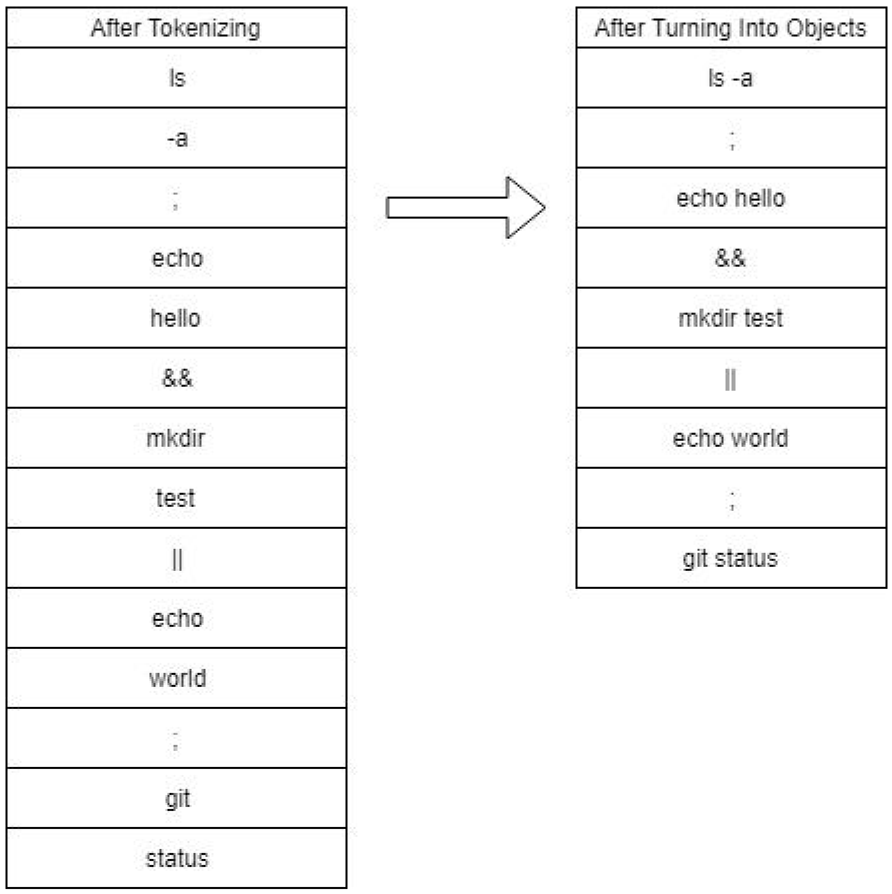
- Coding Strategy:

Brandon Tran - *Parsing the user's input*

Brandon's task will parse the command string the user enters. We are going to used the strtok function from the C standard libraries to parse the input. We are going to use strtok function to split the input from the client/user into strings tokens which will be turned into CMDLine objects. This will be implemented by the Parser class. For example: the client enters input,

```
$ ls -a; echo hello && mkdir test || echo world; git status
```

The input will then be parsed using spaces and connectors as the separators. After the input is tokenized, the tokens will then be created into CMDLine objects as the diagram shows below.

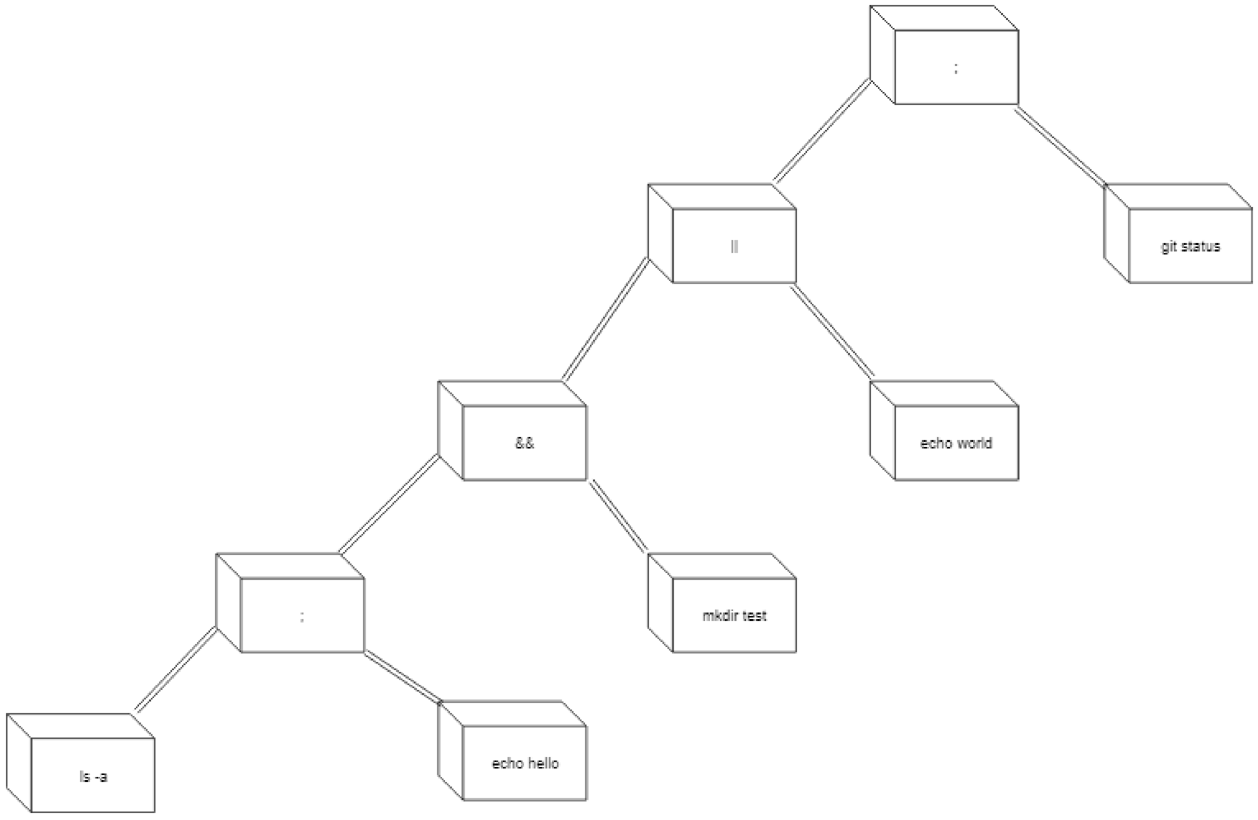


The CMDLine objects will be added from left to right ordering for its execution:

```
$ ((((ls -a; echo hello) && mkdir test) || echo world); git status)
```

Christian Campos - *Commands and Connector Tree*

Since the basic command shell is based on a hierarchy system, the first command and its arguments from the input will be the lowest level of the tree structure. For example: let's use the command input used in the parsing. The top node of the tree will be the last command or connectors that will be executed from the user's input. Once the execute() function is called, the root node of the class will execute() and traveled down to the lowest level of the tree which is "ls -a". The command "ls -a" will be executed and returns a boolean value to transerve one level up towards the connectors , ";".Then a boolean value will returned based on what connectors command interacts. If the bool value returns true from "ls -a" , then execute() will then travel to the right side of the tree to perform execute on the command " echo hello". As a result, it will recursively transverse to the top node of the tree until all the nodes in the tree are executed with the true boolean value. The following recursive and tree structure can be shown in the diagam below:



- Roadblocks:

  A roadblock that can hinder our project structures is the system calls. We don't fully understand how fork, execvp, and waitpid because we never used system calls before. However, we will thoroughly read multiple articles and watch youtube videos to help us comprehend the unix functions.

  An another potential roadblock that can occur is parsing the user input with strtok function. Since our previous projects we were mostly creating our own parse for the user input, strtok is new to us. Nevertheless, we will read c++ documentation and articles, and watch youtube videos on strtok function.