# Assignment 1 : Design

Date: 10/20/17

Fall 2017

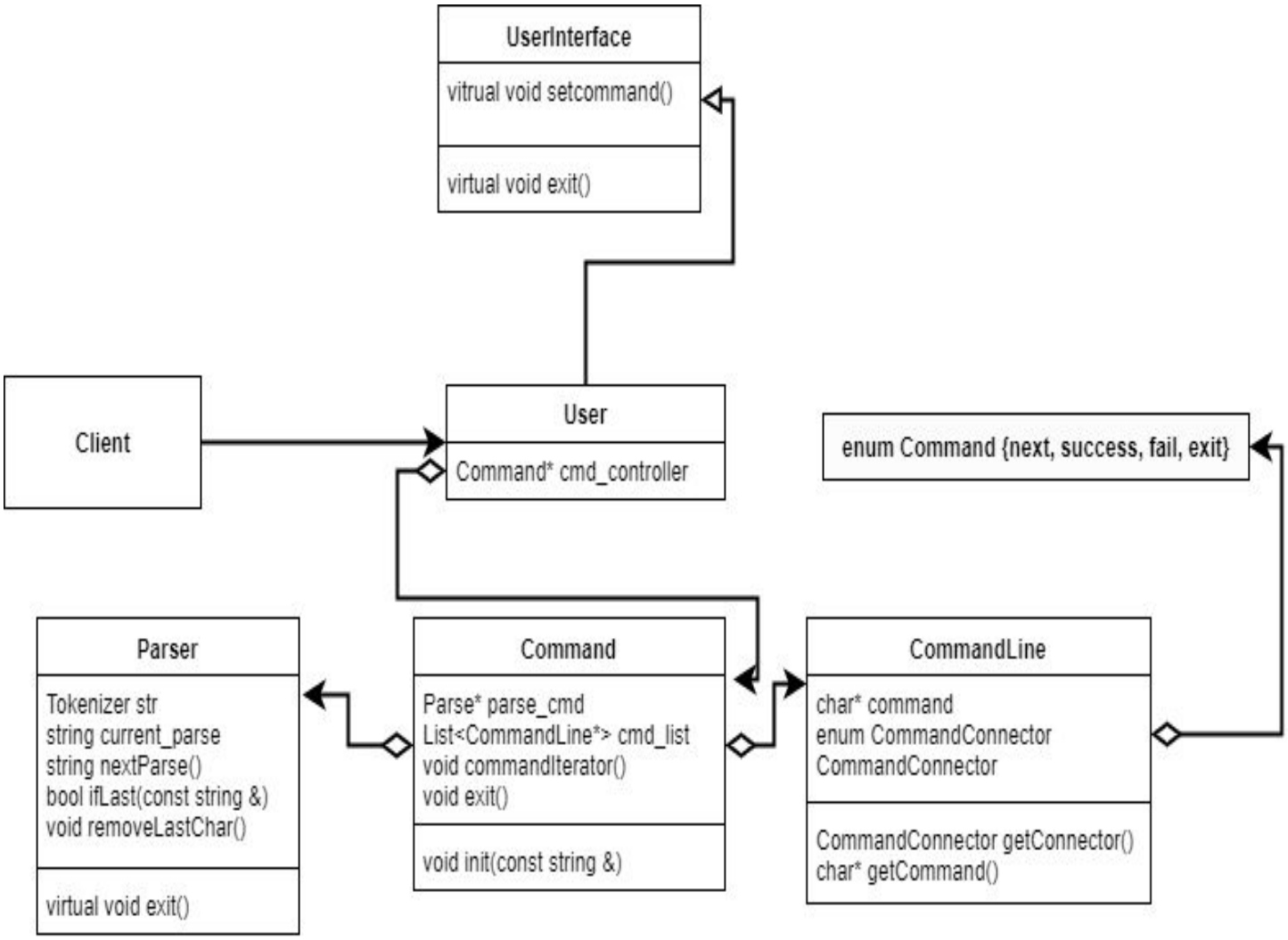
Salud Lemus

Jose Garcia

# Introduction

We are creating a command shell called rshell. The rshell will be capable of reading and carrying out executable programs found in the PATH. The rshell is also capable of three optional “connectors,” which enables the user to chain multiple commands within one line. There is the standard “;” which simply executes commands one after the other. There is also the “&&” and “||,” which only executes the next command if the previous either succeeded or failed. Lastly, rshell contains a custom exit command which simply exits out of the rshell.

# Diagram


UserInterface
vitrual void setcommand()
virtual void exit()
Client
User
Command* cmd_controller
enum Command {next, success, fail, exit}
Parser
Tokenizer str
string current_parse
string nextParse()
bool ifLast(const string &)
void removeLastChar()
virtual void exit()
Command
Parse* parse_cmd
List<CommandLine*> cmd_list
void commandIterator()
void exit()
void init(const string &)
CommandLine
char* command
enum CommandConnector
CommandConnector
CommandConnector getConnector()
char* getCommand()

# Classes/Class Groups

Parser: This class will be in charge of parsing through the user's input and handling one string at a time to ensure correctness.

- Tokenizer str: This data member will hold the entire user's input
- string current_parse: It will hold the current string that we are parsing
- public void init(const string&, List<CommandLine*> &): It will initialize the list and tokenizer via the string passed in.
- private string nextParse(): It will retrieve the next string in the str data member via Tokenizer's next() method.
- private bool ifLast(const string&): returns true if the last char of a string is ";"
- Private void removeLastChar(): If the current string contains a ";," it will remove it via string's substr() method.

Command: This will commence the program by asking the user for input, and it will handle the each command as we parse through. Also it will exit whenever exit is encountered through the parsing.

- Parser* parser_cmd:
- List<CommandLine*> cmd_list: List of CommandLines
- public void init(const string &): It will clear and fill the cmd_list with the passed in string.
- private void commandIterator(): Executes the commands in the cmd_list one by one while also accounting for the connectors as well.
- private void exit(): The program will exit whenever "exit" is encountered.

CommandLine: This class stores the char* and command connector required for the Command class to perform its various functions.

- enum CommandConnector { next {;}, success {&&}, fails {||}, exit}
- private char* command: stores the command (e.g. cd NewDirectory), which will be of use whenever we will call execvp()
- private CommandConnector connector: stores what the Command class will do after the execvp()
- public getCommand(): returns command
- public getConnector(): returns connector

UserInterface: It will enable the User class to interact via the command line.

- virtual void setCommand(): This will prompt the user to enter their desired command (e.g. cd thisdirectory).
- virtual void exit(): It will exit the rshell whenever an "exit" is encountered or we are done with the entire command itself.

User: It enables the user to execute the rshell.

- Command* cmd_controller: It will traverse the entire string and will execute each command that is encountered via its methods.

# Coding Strategy

Before anything else, the header for all classes must be written, just the variables and functions. The source files for all classes must be written as well, but they only require a return line.

From there, Jose will implement the code for the Parser and CommandLine class while Salud will do the same for the User and UserInterface class. Salud will then proceed to implement the Command class in order for the User class to utilize its data member.

For the UserInterface class, both pure virtual functions will be implemented in the User class. SetCommand() will be implemented first which will be tested that the user's input succeeds. Lastly, exit() will be implemented which will exit the rshell.

For the User class, it will begin by the user calling its method of SetCommand(). This in turn will enable the use of its data member cmd_controller, which will handle the input via parsing and executing that/those command(s).

For the CommandLine class, implementation is very simple and straightforward. This must be done before the Parser class.

For the Parser class, the nextParse(), removeLastChar(), and ifLast() functions should be finished first before the init(). The init() will use a while loop to add CommandLines into the List.

For the Command class, the init() function should be implemented first because it will initialize the buffer for the tokenizer class, this is where the data member parse_cmd will come in. This function will also do a push_back on the list data member. If there any instance of "exit" or the buffer is completed, exit() will be called, which should be implemented next. Finally, the commandIterator() will be implemented lastly, which will call upon execvp().

Once Jose finishes the Parser class, he will begin the main file and prepare a test case.

# Roadblocks

Jose and Salud have very little experience with execvp() and Tokenizer. A lot of time will be spent reading and understanding the functions and reading their respective manpages that are readily available. Thus, some conflicts may arise regarding the system call functions' usage.

The Command class depends a lot on the Parser class. Until the Parser class is completed, testing the Command class will be inconvenient.

Some edge cases may not be prominent during testing, which may cause the program to fail.

If future expansions of the project require changes to the CommandLine, then most of the entire project has to be rewritten. On the bright side, there isn't must code to change.

Perhaps the user would want to use their previously entered command again, like the up arrow to retrieve it, so we would most likely consider on implementing a stack specifically for that scenario.