Assignment 1 Design

20 October 2017

Fall 2017

Kevin Frazier

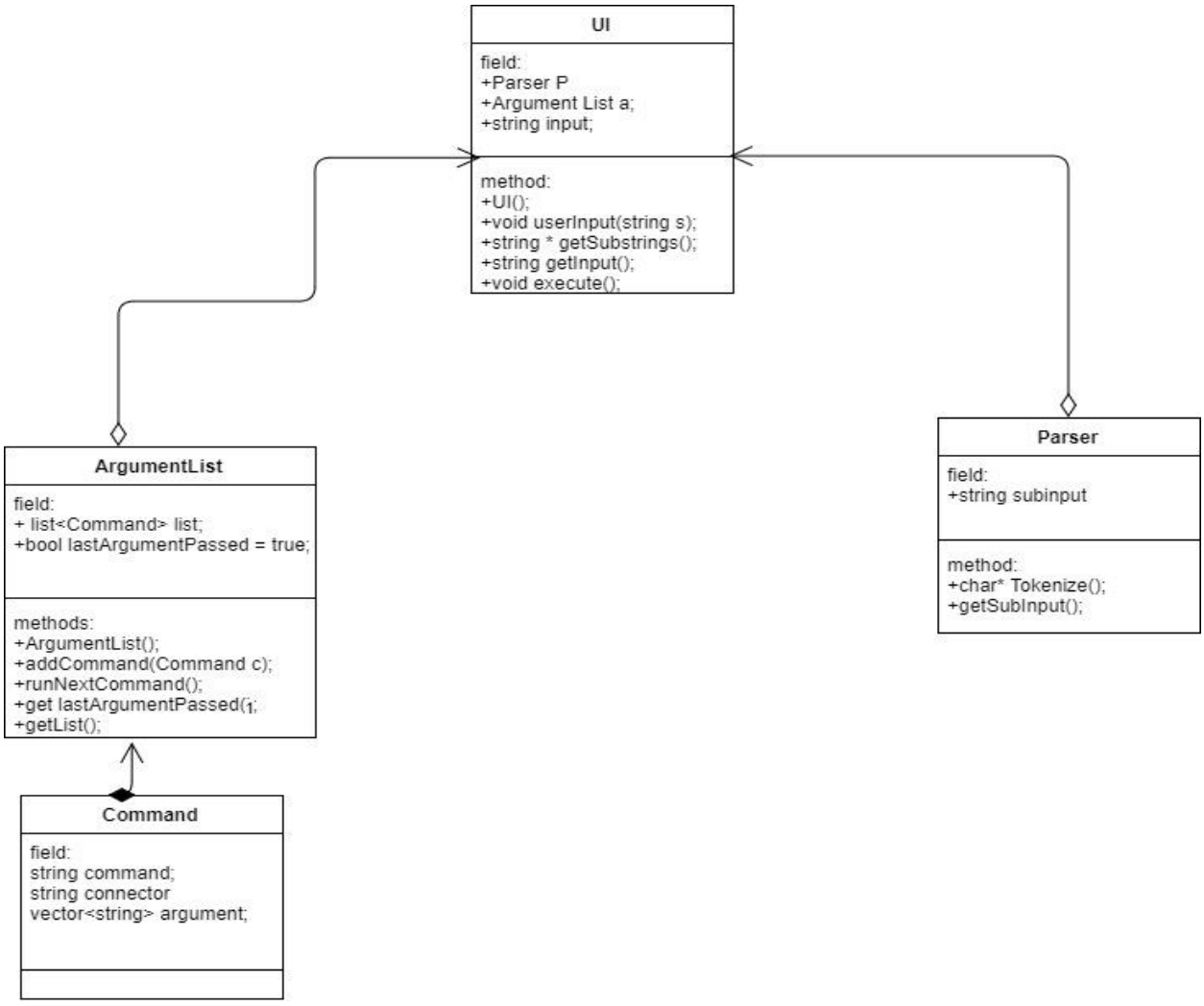
Dinh Bui

- **Introduction : Give a brief overview of your design**

  Epic: "The user should be able to execute a one-line Bash command by entering the command on a keyboard."

  We are creating an rshell program using C++. This prompt asks for a string input. The string input can consist of commands, its arguments, and connectors. The UI class will split each command by its connectors into separate substrings. Each substring will go into a parser object and return tokens of each substring. These tokens in the substring will be created into a command structure, then the command will be stored into the Argument List object. After each of the commands are parsed, the Argument List will execute each command systematically.

- **UML Diagram :**



- **Classes/Class Groups:**

The

```
class UI
    private:
        string input
        Parser P                        // Represent parser object
        Argument List A                 // Represent an object containing the argument list
        string* getSubstring()          /* Obtain substring, then keep substring's location
                                           inside a pointer */
        void execute()                  // Execute command

struct Command                          /* Public class containing string variables connector
&                                               command, and list of vector argument*/

    public:
        string connector                // Stores ;  , || , && signs
        string command                  // Stores a single bash command
        vector<string> arguments        // Stores arguments for the command


class Parser
    public:
        string subinput                         //string input given by the user
        getSubInput() const;            //accessor function that returns the subinput
        char* Tokenize()                // tokenizes the string, subinput

class ArgumentList
    public:
        list<Command> listOfCommands    // Stores a list of commands
        bool lastArgumentPassed = true;  /*boolean to indicate if last command
                                                succeeded*/

    private:
        ArgumentList()                          // constructor for the ArgumentList class
        addCommand(Command c)   //inputs a command within the list of the class
        runNextCommand()                //executes next command
        getLastArgumentPassed()         //returns Boolean variable lastArgumentPassed
        getList()  const                        //*accessor function that returns list of
                                           stored in listOfCommands */
```

The class UI represents the user interface that uses the ArgumentList and the Parser classes to execute a set of command. The UI has a composite relationship with the Parser and the Argument List class.

The Parser class is a child of the UI class. The Parser class takes the user's input and tokenizes the input into smaller inputs. The Parser class then gives the subinputs to the UI for the ArgumentList class to store those subinputs.

The Command Class derives from the ArgumentList class. The Command class consists of two strings called command and connector. The command string stores a single command, and the connector string stores one of these four symbols: $; || &&. Besides the two string variables, the Command class also holds an array of strings called argument. The argument string stores all the subinputs that the UI receives from the Parser class.

The ArgumentList class is another child of the UI class. The ArgumentList class reads in the array of strings passed from the Command class. The ArgumentList class then iterates through the array of strings to store the Bash commands into a list of commands.

Relationships:

The UI has a composite relationship with the Parser and the Argument List class, because it will just be calling the Argument List functions.The Argument List Class will have a composite relationship with the Command Class as it contains and implement commands.

- **Coding Strategy**

The project can be broken up into 2 distinct sections: the parsing of the commands and the reading of the commands. Dinh will be focusing on the implementation of Parser and User Interface classes. Because the user interface will also create substrings, it is necessary that the Parsing should be done by the same person to smooth the process. Each process will create one command and giving it an easier process each command within the ArgumentList.

Kevin will be working on the implementation of the ArgumentList and Command classes. All of the commands inputted and parsed will have to be systematically inputted one by one into the list from the ArgumentList class. After all of the commands are inputted into the list, the ArgumentList class will execute the commands in chronological order. After the implementations are complete, the work will be divided into the technical details of executing the commands.

- **Roadblocks :**

Technical issues such as executing the command through bash script. Although, if the documentation to call these commands properly, then we will have no problem implementing them. When creating substrings, we might have white spaces that will cause issues with the main process of converting the string into commands. We could run into difficulties when checking whether a command succeeded or not. This could lead to problems in terms of making the || and # connector work specifically. Although this can be fixed if we need to address past arguments

by changing the data structure of the list into a vector. Therefore, we would have to keep the current index of the command that is being executed and each command on the list will have a bool variable whether or not the command succeeded.