Programming Assignment: Writing a Basic Command Shell

11/29/18

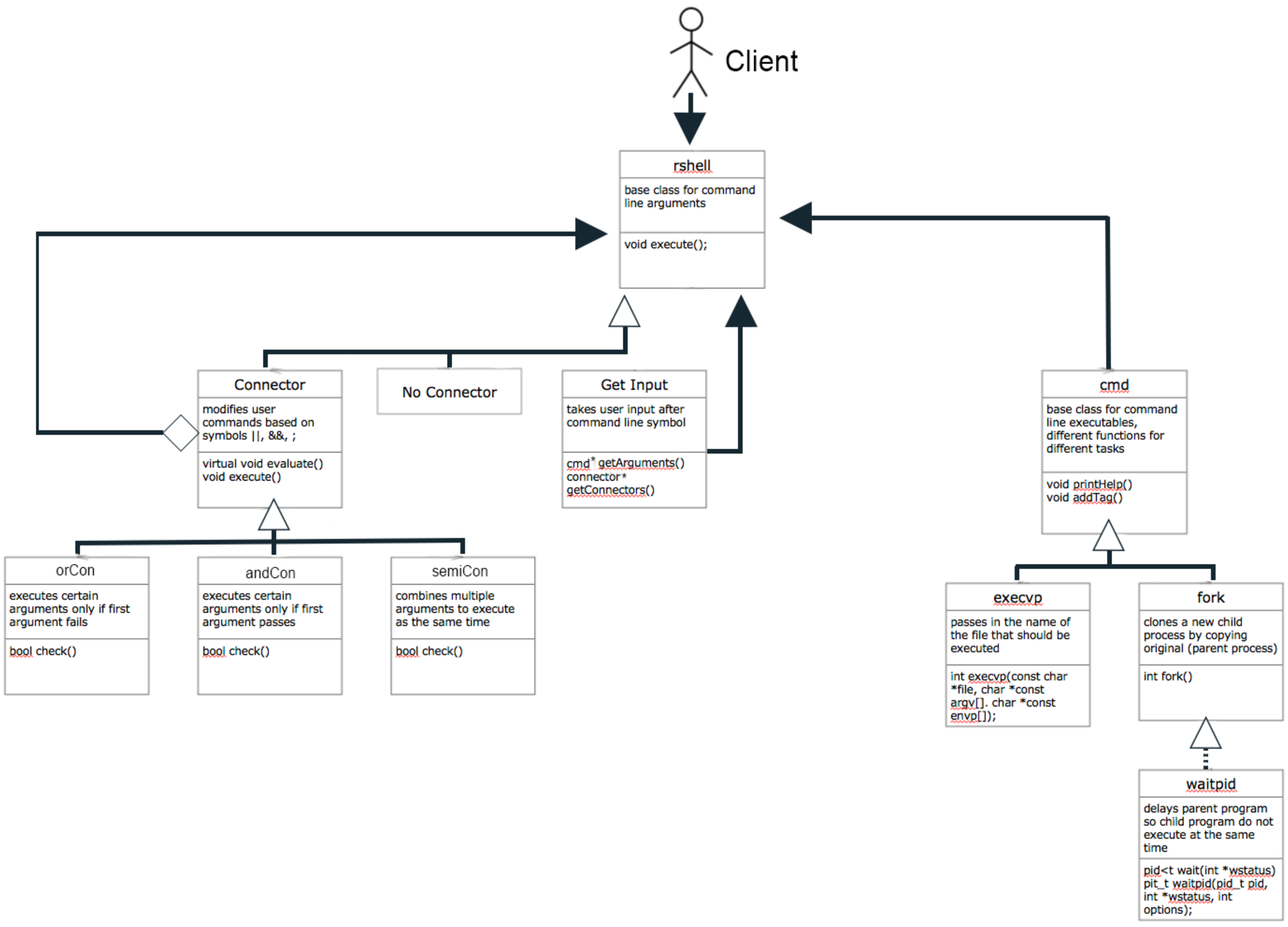
Fall 2018

Raymond Kim, Lucas Song

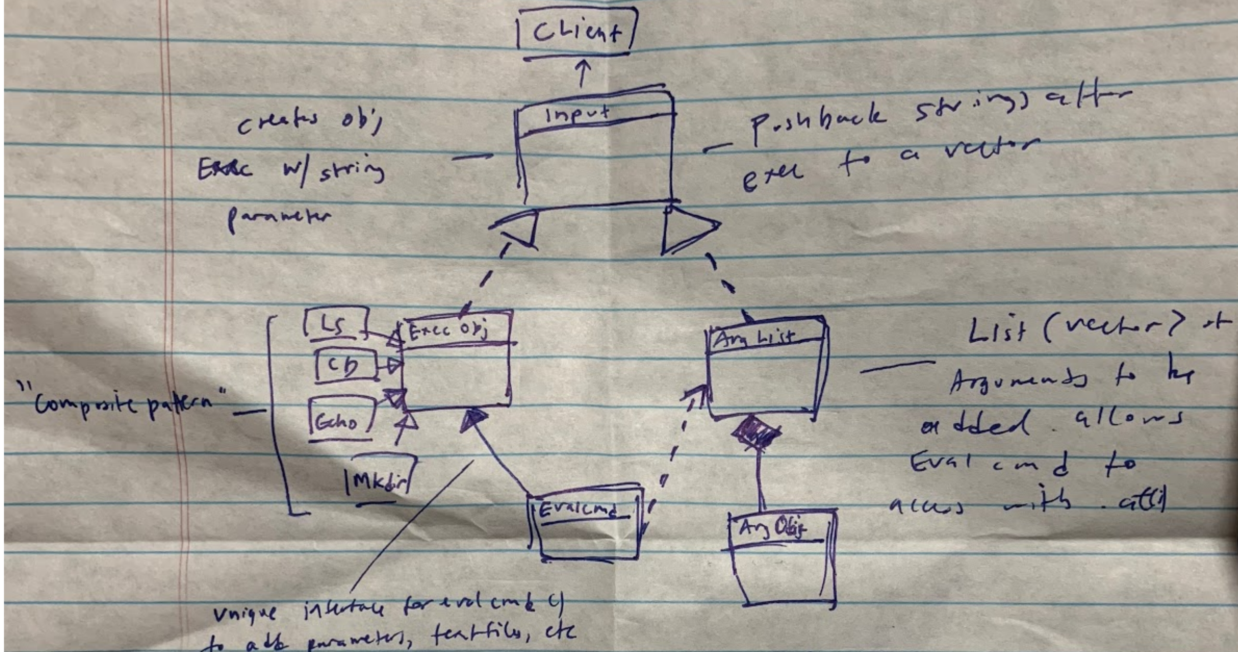
Introduction:

For this project, we will be designing a basic command shell. This command shell will be able to print a command prompt, read in a line of user input, segment this input into an executable, argument list, connector, and more, and use said input to execute the proper commands using fork, execvp , and waitpid. The shell should be able to run multiple commands at once, utilizing symbols to determine how multiple commands should be run in unison. These include "| |, &&, or ;."

Diagram:


Client
rshell
base class for command line arguments
void execute();
Connector
modifies user commands based on symbols ||, &&, ;
virtual void evaluate()
void execute()
No Connector
Get Input
takes user input after command line symbol
cmd* getArguments()
connector* getConnectors()
cmd
base class for command line executables, different functions for different tasks
void printHelp()
void addTag()
orCon
executes certain arguments only if first argument fails
bool check()
andCon
executes certain arguments only if first argument passes
bool check()
semiCon
combines multiple arguments to execute as the same time
bool check()
execvp
passes in the name of the file that should be executed
int execvp(const char *file, char *const argv[]. char *const envp[]);
fork
clones a new child process by copying original (parent process)
int fork()
waitpid
delays parent program so child program do not execute at the same time
pid<t wait(int *wstatus)
pit_t waitpid(pid_t pid, int *wstatus, int options);

ASSN 2: R'shell


Client
Input
creates obj Exec w/ string parameter
Pushback strings after exec to a vector
Exec obj
Ls
Cd
Echo
Mkdir
"Composite pattern"
Arg List
List (vector) of Arguments to be added. allows Eval cmd to access with .at()
Evalcmd
Arg Obj
unique interface for eval cmd () to add parameters, textfiles, etc

mkfile (vim) → string (getline (cin, string))

mkdir → vector<string>

cmd: Execobj → evaluate(ArgList<string> a)

Classes/Class Groups:

Input

- **Input();**
    - Constructor
- **~Input();**
    - Destructor
- **Input(string userString);**
    - Stores input from user into userInput
- **void clearInput();**
    - Clears input and vectors for next input
- **removeComment();**
    - Removes comment from userInput to prepare for parsing
- **vector<baseNode* > returnParsedNode();**
    - Returns vector of parsed nodes (helper function)
- **vector<vector <string> > returnParsedNoSpace();**
    - Returns parsedNoSpace vector (helper function)
- **vector<string> returnStrings();**
    - Returns parsedStrings vector (helper function)
- **vector<Connector* > returnConnectors();**
    - Returns connectors vector (helper function)
- **baseNode* returnHead();**
    - Returns top of executable tree
- **void callExit();**
    - Sets bool exit = true
- **bool checkExit();**
    - If bool exit == true, quits command prompt
- **void runInput();**
    - Prints command prompt and takes input from user
- **vector<string> parsePar(string &userString);**
    - Parses userInput based on parentheses found
- **void parseTest(string &userString);**
    - Parses userInput based on “test” / “[]”
- **void parseInput();**

    - Takes in user input and tokenizes string into substrings based on connectors, and then tokenizes yet agian to remove spaces in the substrings. Returns a vector of strings to be then used to generate argument objects.
- **vector<string > parseSpaces(string withSpaces);**
    - This is a helper function that is virtually the same as the first step of parseInput(), using spaces as delimiters instead of the connectors to remove spaces from string withSpaces
- **void parseConnectors();**
    - Takes connectors from userInput and pushes into vector of connector objects
- **baseNode* makeNode(vector<string> exec);**
    - Helper function that returns a baseNode object based on the zero index of the vector passed in and takes the rest of the vector as the argument list
- **void makeExecutableTree();**
    - Constructs executable tree based on 2D vector and connector vector, sets baseExec* head to the head node of tree.
    - head->eval should execute the tree with respect to the connectors
- **void callExecute();**
    - Calls execute on the head node. Recursively calls execute for elements of executable tree with respect to connector conditions
- **string returnUser();**
    - Returns name of user (Extra Credit)
- **string returnHost();**
    - Returns name of host (Extra Credit)

Arg
- **arg();**
- **Virtual void execute() = 0;**

baseNode
- Base class for connectors and executables
- **baseNode();**
    - Sets leftChild and rightChild pointers to NULL
- **~baseNode();**
    - Deconstructor
- **Virtual void execute();**

- **Virtual void setLeft(baseNode* leftChild);**
- **Virtual void setRight(baseNode* rightChild);**
- **Virtual baseNode* getRight();**
- **Virtual string returnType();**
    - Helper function
- **Virtual string returnCheck();**
    - Helper function for test cases

Connector : baseNode

- **Connector();**
- **Void setLeft(baseNode* leftChild);**
    - Sets left child
- **Void setRight(baseNode* rightChild);**
    - Sets right child
- **baseNode* getRight();**
    - Returns rightChild
- **Virtual string returnType();**
- **Virtual string returnCheck();**

And : Connector

- **And();**
    - Constructor, points leftChild and rightChild to NULL
- **Void execute();**
    - Calls execute for right child if left child execution succeeds

Or : Connector

- **Or();**
    - Constructor, points leftChild and rightChild to NULL
- **Void execute();**
    - Calls execute for right child if left child execution failed

SemiColon : Connector

- **SemiColon();**
    - Constructor, points leftChild and rightChild to NULL
- **Void execute();**
    - Calls execute for left and right children

baseExec : baseNode

- **baseExec();**

- **Void addArg(vector<string> arg);**
    - Adds arguments to vector
- **Virtual bool execute();**
    - Executes non-specified command using execvp()
- **Virtual string returnCheck();**
    - Mimics error class's returnCheck();
    - Used to print error statement

echo : baseExec

- **echo();**
- **Void addUserInput(string userInput);**
- **Void execute();**
    - Prints arguments on newline

error : baseExec

- **error();**
    - Constructor, points leftChild and rightChild to NULL
- **Void execute();**
    - Outputs error statement

exitCall : baseExec

- **exitCall();**
- **Bool execute();**
    - Sets exitBool = true
    - Returns true
    - Used to exit command prompt

test : baseExec

- **test();**
    - Constructor, sets dashE to true, sets dashF and dashD to false
- **execute();**
    - Checks for flags in arglist
    - If not, checks if file path exists

Par : baseExec

- **Par();**
    - Constructor: sets left and right children to NULL
- **setSubString(string s);**
- **Bool execute();**
    - Returns true
    - Simply acts as a step in recursively calling parseInput on subString data member

Coding Strategy:

We intend to split the work in such a way that we are both able to familiarize ourselves with the program. We will both implement the classes and their respective functions, however, we will test each others' code so that we are exposed to all aspects of the program and its design.

Roadblocks:

An issue that we anticipate is in parsing the user input. Since there are many ways for spaces and stray symbols to introduce problems in the program, we must ensure that the data is properly compartmentalized.

In addition, coding out the logic of connectors seems a little confusing. We are not completely sure as to how to approach the case of multiple connectors with multiple arguments that must execute in different cases.

Not only this, we infer that there will be many changes to the structure of our design. As such, keeping an accurate account of changes that have been made and will be made should allow for smoother transitions between adding new features.