Assignment 4: rshell

Thursday, December 13, 2018

Joseph DiCarlantonio
Sameh Fazli

CS 100 Fall 2018

Introduction:

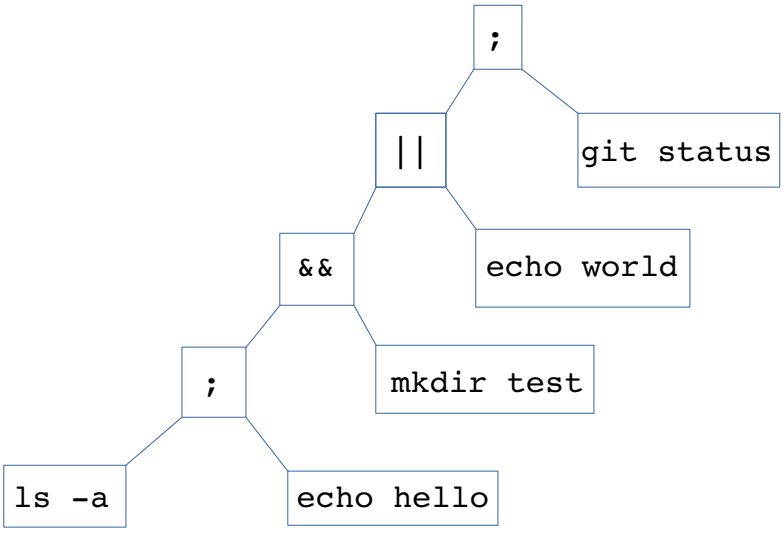
First we take a high level view of the lifetime of our shell from initialization to destruction:

1. First the shell needs to start up and idle upon request from the client.
2. While the shell is running, it needs to get a command input from the user, parse that input and then execute the command.
3. Finally, when the user enters exit from within our shell, it should exit.

For our design, we decided to have a class called Shell that will handle initialization and contain the main loop of the program where it will handle input. For the input, we are using the composite design pattern, since the user can enter commands such as

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

and we can break this up into an expression tree and execute this command recursively:



To do this we have a helper class called Input, which will prompt the user for input, parse that input, and initialize our component, called Input which will get the input from the user, parse it, and initialize the appropriate command. The Input class is an aggregate of the Shell class. The Shell class is the main class which initializes and maintains the loop state.

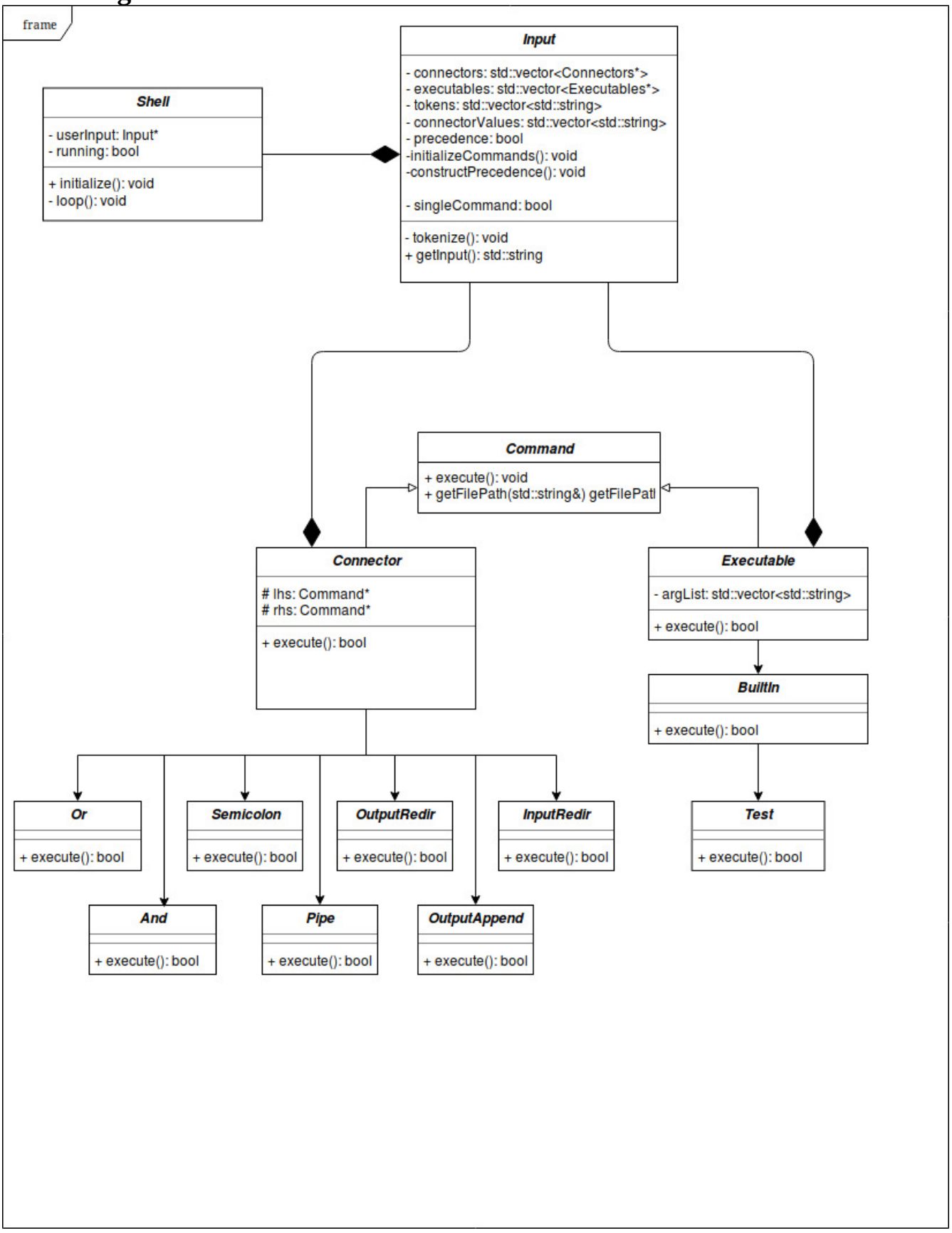
We also have to account for precedence, I.E. as in the command:

```
(echo A && echo B) || (echo C; echo D)
```

To do this, we consider a different tree which can be more balanced depending on the user input. We looked at the problem more as a grouping problem, i.e. grouping different sub-commands, and connecting them all with a connector. To do this, we used a play on postfix and pushed and popped onto two stacks. One stack represented connectors and parenthesis (symbols), while the other was an actual command (which could also be a connector or parenthesis object). We pushed and popped onto these stacks accordingly to user input, placing precedence on parenthesis.

From here, we used the composite pattern to execute commands recursively. We used the Command class as our base component. The Command class defines the interface for our executables and connecors. For the executables we use a leaf class Executable. For the connectors, we use a composite class Connector which contains children leaf nodes, And, Or, and SemiColon. The Executable class is where we handle all the syscalls to execute the command.

UML Diagram:


frame
Input
- connectors: std::vector<Connectors*>
- executables: std::vector<Executables*>
- tokens: std::vector<std::string>
- connectorValues: std::vector<std::string>
- precedence: bool
-initializeCommands(): void
-constructPrecedence(): void
- singleCommand: bool
- tokenize(): void
+ getInput(): std::string
Shell
- userInput: Input*
- running: bool
+ initialize(): void
- loop(): void
Command
+ execute(): void
+ getFilePath(std::string&) getFilePatl
Connector
lhs: Command*
rhs: Command*
+ execute(): bool
Executable
- argList: std::vector<std::string>
+ execute(): bool
BuiltIn
+ execute(): bool
Or
+ execute(): bool
Semicolon
+ execute(): bool
OutputRedir
+ execute(): bool
InputRedir
+ execute(): bool
Test
+ execute(): bool
And
+ execute(): bool
Pipe
+ execute(): bool
OutputAppend
+ execute(): bool

Class Descriptions:

class *Shell*:

The shell class handles the initialization and oversee the life of the entire program. It will contains a bool flag to keep track of when it is running.

Contains an instance of Input.

Functions include:

initialize() → starts the shell and enters the user into the prompt
loop() → main loop of program

class *Input*:

contains vectors for the Connectors and Executables. Creates and runs each command as per user's input.

Members:

connectorValues → vector of strings holding all the connects contained in the users input
tokens → used for storing tokens for parsing
connectors → vector contains all Connector objects
executables → vector contains all Executable objects
precedence → keeps track if user wants to use precedence

Functions:

getInput() → Prompts the user for input and makes a call to tokenize for parsing. The parsing differentiates between executables and connectors and constructs an Executable object.
tokenize() → Parses user input into tokens. Uses the strtok function from the standard library to split the input up into tokens
run() → Runs the command entered by the user. First checks if the user entered a single command, or one using connectors. If connectors are in the command, it calls initializeCommands() to construct Connector objects And, Or, or SemiColon appropriately. If a single command, it call execute() on the only executable, and if connectors are in the input, it calls execute() on the last connector in the vector.
InitializeCommands() → constructs Connector objects, based on whats stored in the connectorValues string vector.
ConstructPrecedence() → Basically like initializeCommands() but accounts for precedence. This function uses two stacks (as vectors) and converts the input into postfix, constructing a tree to execute the composite pattern.

Class group (composite pattern) *Commands:*

Abstract base class *Command*:

Our component for the composite design pattern. Contains the interface for Executable, Connector, and the children of Connector. For now, it only contains execute().

Composite class *Executable*:

Derives from Command. This is the class that will handle the system calls. It has a string member which holds the argument list for execvp().

Functions:

execute() → will call fork(), execvp() accordingly to execute the command with its arguments

Leaf node class *BuiltIn* and its derivative *Test*:

BuiltIn defines the interface for built in commands. In this case we only have the test command. Both classes only have the function execute().

Composite class *Connector*:

Also derives from Command. Handles the case when a connector, ie) ;, &&, and | | is in the user input. It contains two members first and second of type Command*. These members will represent whats on either side of the connector for example: `echo Hello && ls -la` will have `echo Hello` in first and `ls -la` in second. The execute() function in this class will not be implemented here, but in its children classes And, Or, and Semicolon.

Leaf node classes *And*, *Or*, and *Semicolon*:

Derives from Connector. Right now, it only has the function execute() which will call the execute() function of the Command* members first, and second depending on the specification. For example && will call the second execute() only if the first command executes. | | will call the second execute() only if the first command does not execute. And the ; will call the first and second execute() all the time.

Leaf node classes *Pipe*, *InputRedir*, *OutputRedir*, and *OutputAppend*:

Derives from Connector. These classes handle the Pipe and redirection. The execute() function defined in *Command* now takes two parameters for file descriptors. We have one parameter for the input file descriptor and one for the output file descripter, namely *fdin* and *fdout*.

Pipe:

In the execute() function, we create the array fds[2] for the pipe system call. We then call pipe() and pass in the returning file descriptors into the lhs and rhs execute() functions appropriately

Redirection:

For the redirection and append, we get the file path name from the rhs executable and call the open() system call with the appropriate flags and modes.

Coding Strategy:

Our coding strategy was to break down the entire problem into small sub-problems which either of us could work on and easily integrate. For instance, one of us was able to work on parsing, while the other handled the processing of the commands, I.E. the system calls and execution logic. Then we were able to work as a pair getting everything working together and fixing errors.

Roadblocks:

We had two major roadblocks during the development of this shell. One of the major problems we ran into was differentiating between connectors and executables during parsing, for this we were able to use strtok to split the user's input up into tokens which made the problem much simpler. From there, we used two different vectors, one to store all tokens and one specially for connector values.

The other roadblack we had was system calls not working as expected. One problem we were having was with zombie processes, which was solved by a bit of research and changing the way we handled errors.