Assignment 1: Design

Friday, November 2, 2018

Joseph DiCarlantonio
Sameh Fazli

CS 100 Fall 2018

Introduction:

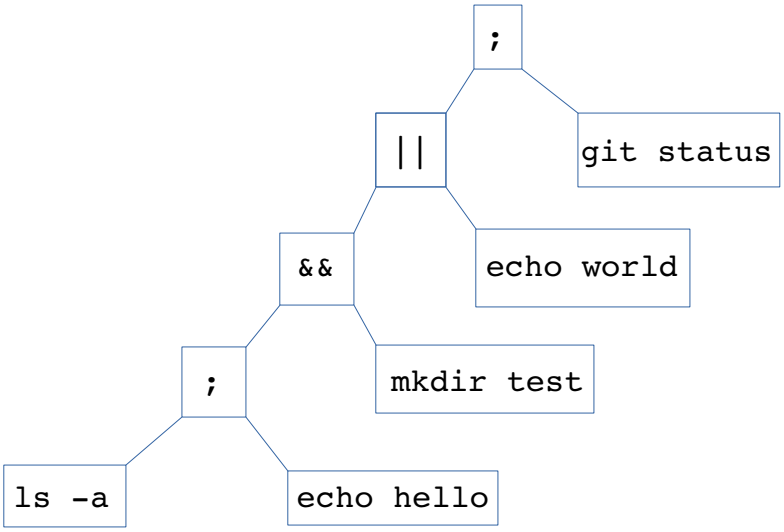
First we take a high level view of the lifetime of our shell from initialization to destruction: (>*.*)=B

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`

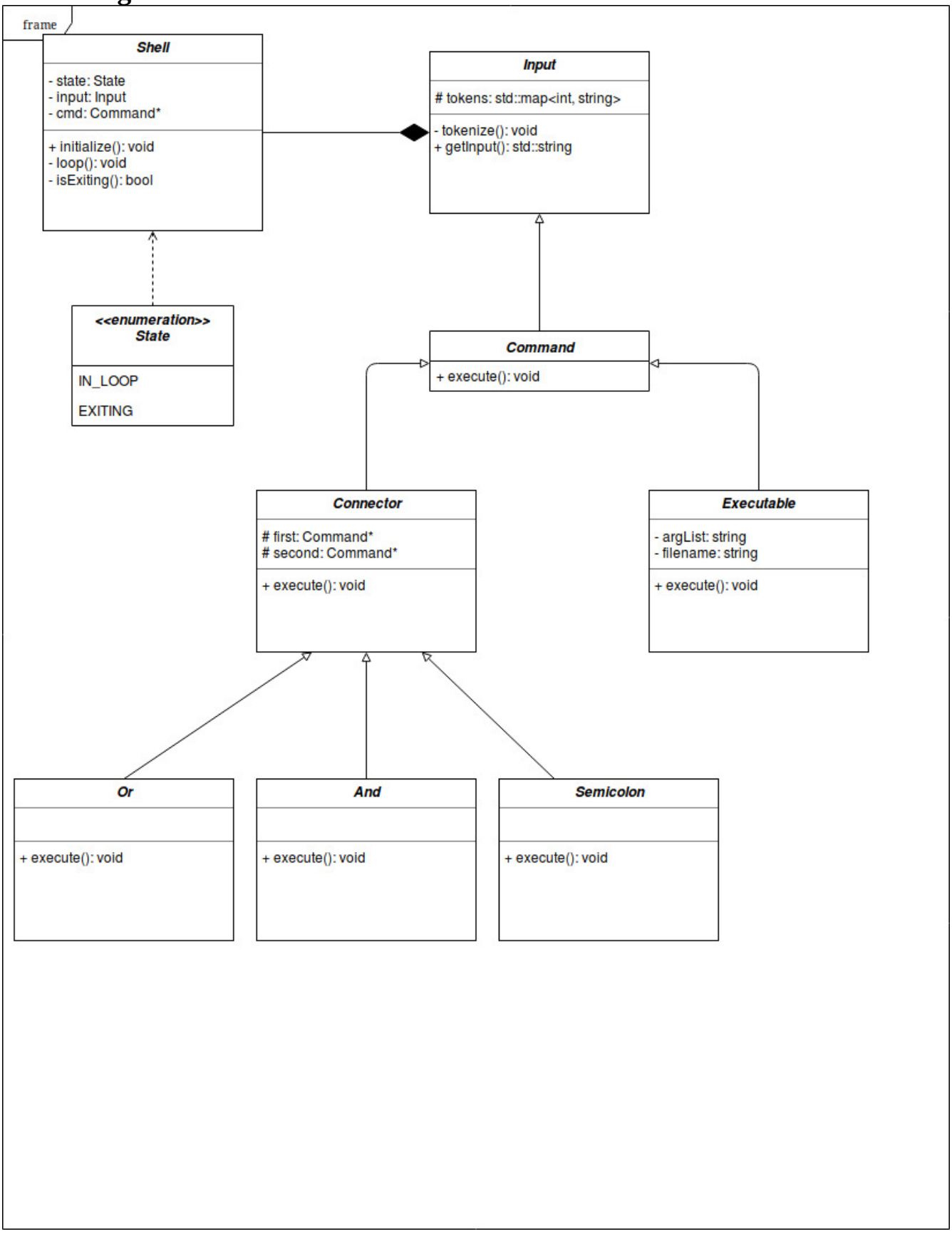
and we can break this up into an expression tree and execute this command recursively:



To do this we decided to use a base class, our component, called Input which will get the input from the user and tokenize it. We will be using a map to store and distinguish our tokens between an executable, argument, and connector. For now Input will have one child composite class, Command.

The Command class will contain a leaf node and another composite called Executable and Connector, respectively. Executable will handle executing the executables in /bin. Since we are not concerned with built in commands right now, Executable will be a leaf node. Connector is a composite that has leaf node children for | |, &&, and ;. This design should be scalable, so we could easily add on new features such as built-in commands by adding leaf nodes to Executable.

UML Diagram:


frame
Shell
- state: State
- input: Input
- cmd: Command*
+ initialize(): void
- loop(): void
- isExiting(): bool
Input
tokens: std::map<int, string>
- tokenize(): void
+ getInput(): std::string
<<enumeration>>
State
IN_LOOP
EXITING
Command
+ execute(): void
Connector
first: Command*
second: Command*
+ execute(): void
Executable
- argList: string
- filename: string
+ execute(): void
Or
+ execute(): void
And
+ execute(): void
Semicolon
+ execute(): void

Class Descriptions:

class *Shell*:

The shell class will handle the initialization and oversee the life of the entire program. It will contain an enum to handle the state, for now we are only concerned with two states:

- IN_LOOP – when the program should be reading and executing commands entered by the user
- EXITING – when the user enters the exit command

Contains an instance of Input and a pointer to the abstract class Command.

Functions include:

initialize() → starts the shell and enters the user into the prompt
loop() → main loop of program
isExiting() → returns true if state is changed to EXITING

class group *User Input*:

This heirarchy of classes is used to handle user input, parsing, and execution of commands.

Base class *Input*:

contains a map that holds tokens from the user input. A map was chosen here so we can split the different tokens into groups ie) Executable, Argument List, File, and Connector.

Functions:

tokenize() → parses user input into tokens (which are stored in the map). Parsing will probably be handled using Tokenizer in the Boost Library
getInput() → returns the user's entire input as a string

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().

Leaf node 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

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.

Coding Strategy:

First, we can split the work between the Shell and the input class such that we can get it done concurrently. This will get the backbone of our shell going and we can work on the Command class and all of its children.

So far, the Command class should be fairly straightforward, however we can split up the work between its children, Executable and Connector. The connector class should be straightforward as it is a composite, so we can split up the work between its leaf nodes, And, Or, and Semicolon. We are anticipating the Executable class to not be as straightforward, so we plan splitting up the work and working collaboratively on the Executable class.

Roadblocks:

One major problem we are anticipating is running into issues with parsing, but to counter this, we plan on using either the strtok() function from the C standard library or another third party parsing library. This will ensure we use a well documented and well tested method for parsing.

More issues we can expect during development are errors introduced from working on the same code at the same time. To prevent this we plan on working on our own branches and doing pull requests to review each other’s code as we progress.

Another issue we expect is not fully handling a complex user input such as the one used in the example above. We are planning for this by using the composite strategy.