Assignment 1: Design

October 20th, 2017

Fall Quarter

Authors

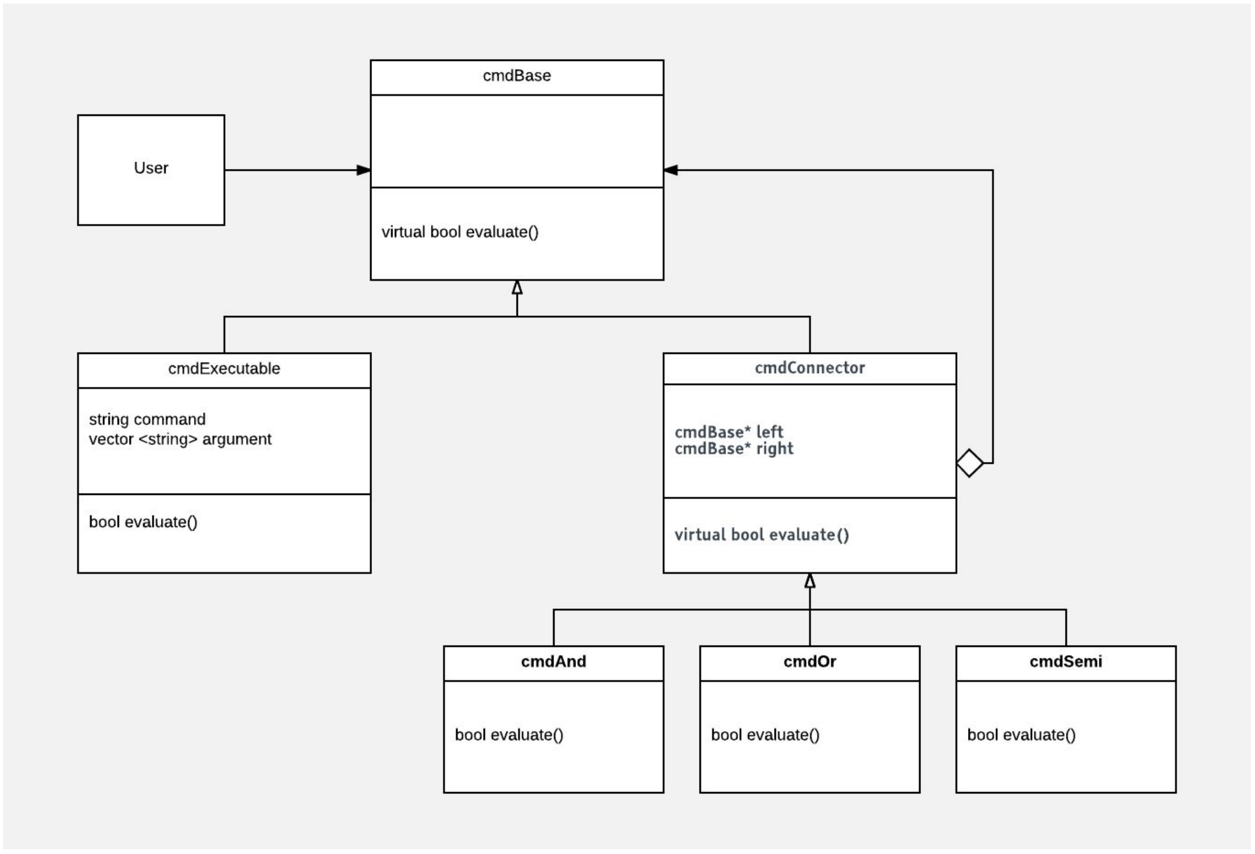
Ji Hwan Kim

Nikhil Gowda

Introduction

We will be creating an RShell. To handle the complexity of commands inputted into our RShell as well as proper execution, a Composite Design Pattern can break these problems into primitive and composite classes. The composite pattern allows the user to use an object uniformly without consideration that there may be very different objects existent. The nature of this program, taking in a shell command, our program must be able to evaluate the argument, even nested ones. We also realized that regardless of number of arguments, the evaluation function is existent through the use of connectors (composites) or executables. Having the interface contain a virtual evaluate function, we can have connectors have a children of connectors or executables and use polymorphism to adhere to each class's evaluation specifics. This will create a tree like structure for the RShell.

Diagram


cmdBase
virtual bool evaluate()
User
cmdExecutable
string command
vector <string> argument
bool evaluate()
cmdConnector
cmdBase* left
cmdBase* right
virtual bool evaluate()
cmdAnd
bool evaluate()
cmdOr
bool evaluate()
cmdSemi
bool evaluate()

Class Description

cmd Class Group:
The design of the RShell follows a composite pattern made up of one base class and the child classes that inherit from the base class.

cmdBase class:
The base class defines the interface of the Rshell. It is an abstract class because it would make more sense to not implement the class and allow its child classes to edit the bool evaluate function to fit its own functionality while retaining the overall structure.
Thus, the bool evaluate() is a pure virtual function and the base class is abstract.

cmdExecutable:
This is a primitive class that inherits from the base class. It has string command and vector<string> argument as private data to store the name of the executable and the argument(s) passed in respectively.
The evaluate() function will take in the string and vector private data as parameters, find the command and run the executable. If the executable is run, the function will return true, else false. Recursion is not involved in this class.

cmdConnector:
This is a composite class that inherits from the base class but also has three children classes. The left and right base class pointers allow the connector class to have children. Because they are base class pointers, the connector class can have both executable class and another connector class as its children.
The virtual evaluate() is a pure function inherited from the base class. Similar to the abstract base class, this allows the connector class to let its children classes to edit the evaluate function to fit its individual functionality.

cmdAnd:
This is a primitive class that inherits from the cmdConnector and represents && connector. The evaluate function will recursively call evaluate on the left pointer first. If this returns true, evaluate is called on the right pointer, else the original function returns false without considering the right pointer. Then if the evaluate call on the right pointer returns true, the function returns true, else the function returns false.

cmdOr:
This is a primitive class that inherits from the cmdConnector and represents || connector. Evaluate function will recursively call evaluate on the left pointer first. If this

returns false, evaluate is called on the right pointer, else the original function returns false without considering the right pointer. Then if the evaluate call on the right pointer returns true, the function returns true, else the function returns false.

cmdSemi:
This is a primitive class that inherits from the cmdConnector. Evaluate function will recursively call evaluate on the left pointer first. Regardless of truth value returned, evaluate function is then recursively called on the right pointer.

Coding Strategy

We plan to start the program off by the agile method of pair programming. Starting off the program like this will allow the driver to code while the observer can catch obstacles in the turbulent beginning of constructing every class/function. Switching frequently, we hope we can come to quick conclusions of why certain aspects of our code are not working. After this initial strategy to get over the beginning hurdles, we plan to break up the code into manageable user stories that we can create together.

We hope the pair programming phase will allow a manageable structure where we can then branch off. While one of us works on cmdExecutable and managing the general user interface, the other will work on the composite cmdConnector class and the children. If there is any time differences in coding time, we plan to work on the same implementations when those time differences appear. After this coding phase and constantly updating our Kanban board, we plan to harness pair programming again to get final glitches to be cleared. This beginning pair programming and end pair programming will keep our focus on a similar development strategy.

While coding individually, we plan to branch off into two separate development branches within git to reflect our personal development. While coding branches off our development branches may be obsolete, we will create possible feature branches if necessary. Coming across merge conflicts, if they do appear, will be a process of team observation. Our strategy, relying on a higher level of agile methods will also take modified parts of Scrum, specifically using “concrete deliverables” that we can hopefully use to better track our progress. And finally, after the days we do code, even independently, we hope scrum-like meetings or simple informing will keep the efficiency or give discussion about strategies to fix our efficiency issues.

Roadblocks

A central issue designing this program is the specific operating structure of each connector. We knew we had to harness the capacity of using an evaluate boolean but we had discussions of whether it was superfluous to create additional boolean values. This question was also amplified by the need to handle comments. And also, how do you handle whether the next argument should be executed? We fully realize we may have to make adjustments to our UML diagram, but keeping a simple structure and putting emphasis on inheritance specific evaluate functions will allow us to proceed with our current construction.

This led to our belief our ability to parse properly and have nested arguments needed to be fixed. We implemented the connectors/ actual logic operators to be composites and the latter being primitive. Our obstacle here is to see if our current interface can function properly with our current class structures. We decided that having our connectors derived from a composite connector class, we can modify specific functions at this level without altering the entire functionality of the entire structure similar to Lab 3. This is how we will most likely handle future implementations and features.