Assignment 1: Design

1/29/18

Winter 2018

NAMES

REMOVED

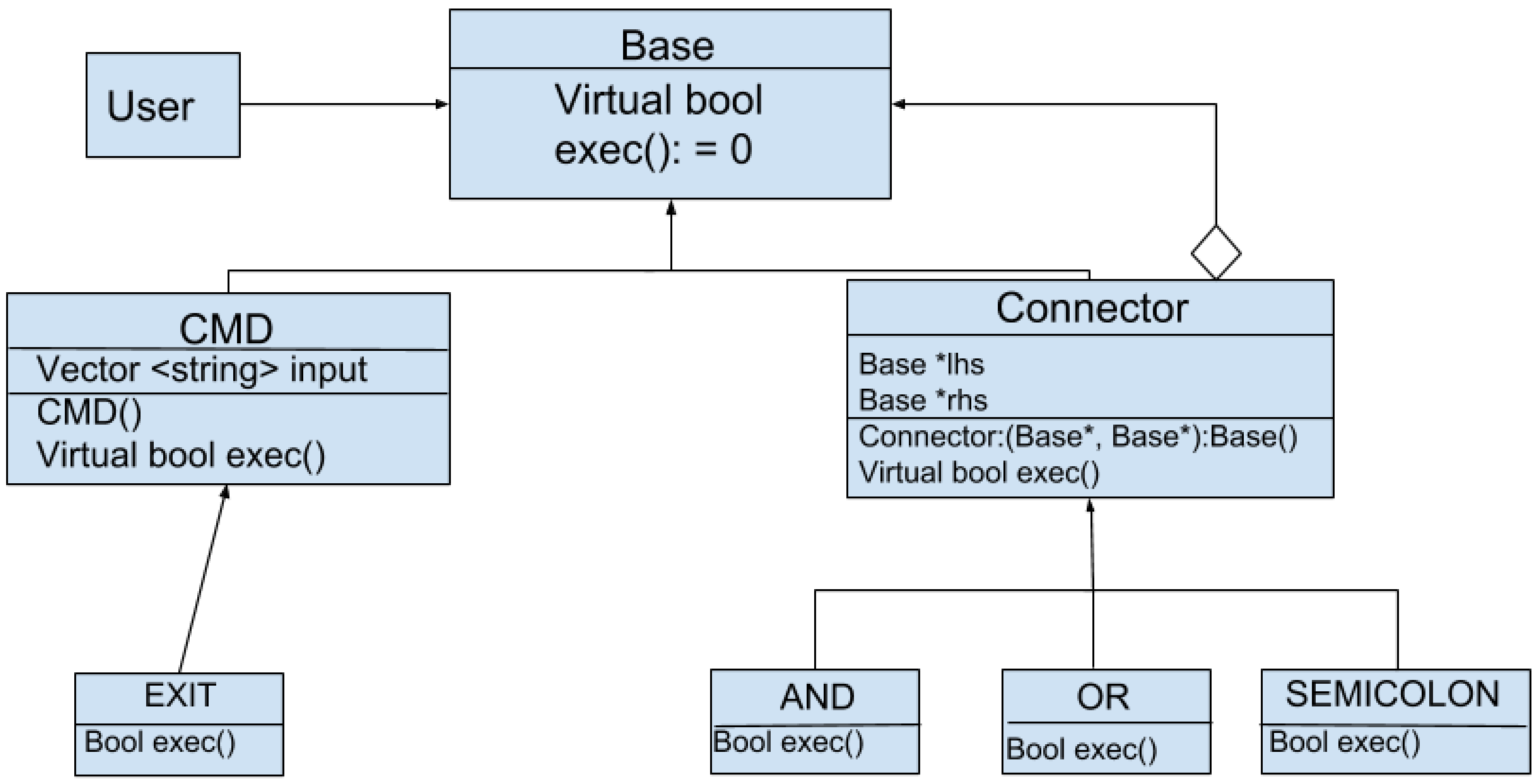
SHELL

Introduction brief overview of design:

For this program we will read in user input take the string they typed and check for text flags (i.e. || && ;) then we will separate the string into smaller strings that are made up of the information separated by each flag. Then we will execute each of the smaller sub-strings.

We will start by prompting the user for an input. Once the user input is received, the code will be read until a connector or null character is reached. This segment will be separated into its own process, and the code will continue. If a connector was reached, the code will continue reading the input for more commands. If a null character is reached, the input is over and the previous commands will be executed. At any time, if the program reaches a pound sign, the rest of the input is ignored as a comment. Each command of the input is checked for validity, then forked. The parent process is told to wait, while the code's logic is run on the child. The value of the child is returned and the child is killed, with the code continuing using the parent.

Diagram:


User
Base
Virtual bool exec(): = 0
CMD
Vector <string> input
CMD()
Virtual bool exec()
Connector
Base *lhs
Base *rhs
Connector:(Base*, Base*):Base()
Virtual bool exec()
EXIT
Bool exec()
AND
Bool exec()
OR
Bool exec()
SEMICOLON
Bool exec()

Classes/Class Groups:

Base: Our Base class is our abstract base class that all other classes inherit from. Its purpose is to execute the functions by requiring a virtual exec function..

Command: The command class is the leaf of our composite pattern. It contains the user's input and tracks the connectors used within in order to properly execute the code using the fork, waitpid, and execvp system calls.

Connector: The connector class is the composite class that holds a left and right operand. This class determines how the two operands will behave.

Exit: Special command that causes the RShell to exit

Class Group - Connectors:: Each of the following inherit from the connector class, and use it;s left and right pointers. They differ only in their logic.

Pound: Causes the code to ignore any input following the pound sign during execution

AND: Runs both operations on either side of the AND connector. The right hand side will only execute if the left hand side succeeds.

OR: Runs only one of the operations on either side of the OR connector. The right hand side will only execute if the left hand side fails.

Semicolon: Runs all operations on either side of the semicolon connector regardless of success and failure.

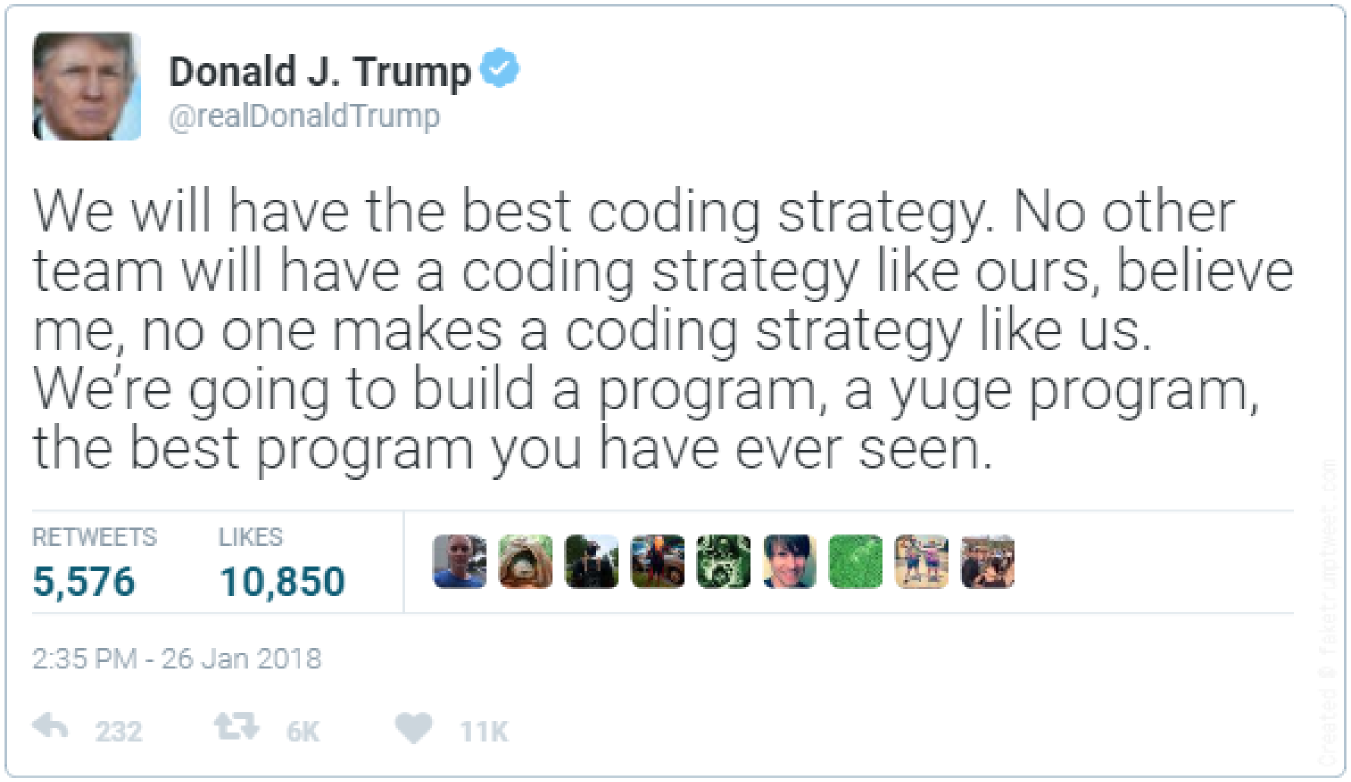
Coding Strategy:

The code will be broken up into two sections: one section will be done by me, REMOVED, which will involve writing the RShell command shell and the Command class, along with the child class Exit that inherits its information from the Command class. The other section will be assigned to me, REMOVED, where I will write the Connector class along with all of its children classes, including, but not limited to, the And class, the Or class, and the Semicolon class. The Rshell will be written to include the calls to the Connector classes for easy merging.

Code sections broken up:

REMOVED
Base
CMD
Exit

REMOVED
Connector
AND
OR
SEMICOLON


Donald J. Trump
@realDonaldTrump

We will have the best coding strategy. No other team will have a coding strategy like ours, believe me, no one makes a coding strategy like us. We're going to build a program, a yuge program, the best program you have ever seen.

RETWEETS 5,576 LIKES 10,850

2:35 PM - 26 Jan 2018

232 6K 11K

Road Blocks:

Time Constraints: In order to overcome our time constraints, we will make use of online tools that allow us to work on our code whenever possible, whether or not the other person is available.

Communication: To best communicate, we will use multiple tools including social media, texting, and comment areas of the various coding and repository platforms.

Difference in location: It is unfortunate to say that my partner and I do not live in the same apartment complex, or even the same city. Therefore, it is not always the easiest for the two of us to meet up in person due to the far distances we both have to travel in order to meet up.

Other coursework requirements: As we both have full class loads, our time is required by multiple other assignments. To remedy this, we will be proactive in completing our various assignments in a timely manner to give us the most time possible to complete our code.

Lack of knowledge on Sys Command usage: Because the world around us is heavily involved with the Windows operating systems, we have not had the luxury to work with Linux for many years. In fact, UCR is our first exposure to the Linux operating system. Therefore, we are still new users and lack the experience of using the operating system.

Issues involving string parsing: As we are used to operating in a different syntax, mostly in C languages, learning to properly parse our strings in the terminal will be a challenge. We will overcome this by viewing tutorials provided online.

Epic:

As the User, we want the code to print a command prompt so we understand what to do.
As the User, we want to input commands so the program does what we want.
As the User, we want the code to parse our input for us so we don’t have to know coding.
As the User, we want the code to tell us if our input is wrong and what is wrong with it so we can fix it without trial and error.
As the SysAdmin, we want to take in the user’s input so the program knows what code to execute.
As the SysAdmin, we want the code to split input into individual commands so it’s easy to parse.
As the SysAdmin, we want the code to handle any number of commands and operator combinations so the code is flexible.
As the SysAdmin, we want the code to view input after the pound sign as a comment so the code doesn’t execute information that isn’t a command.