Assignment 1: Design

26 January 2018

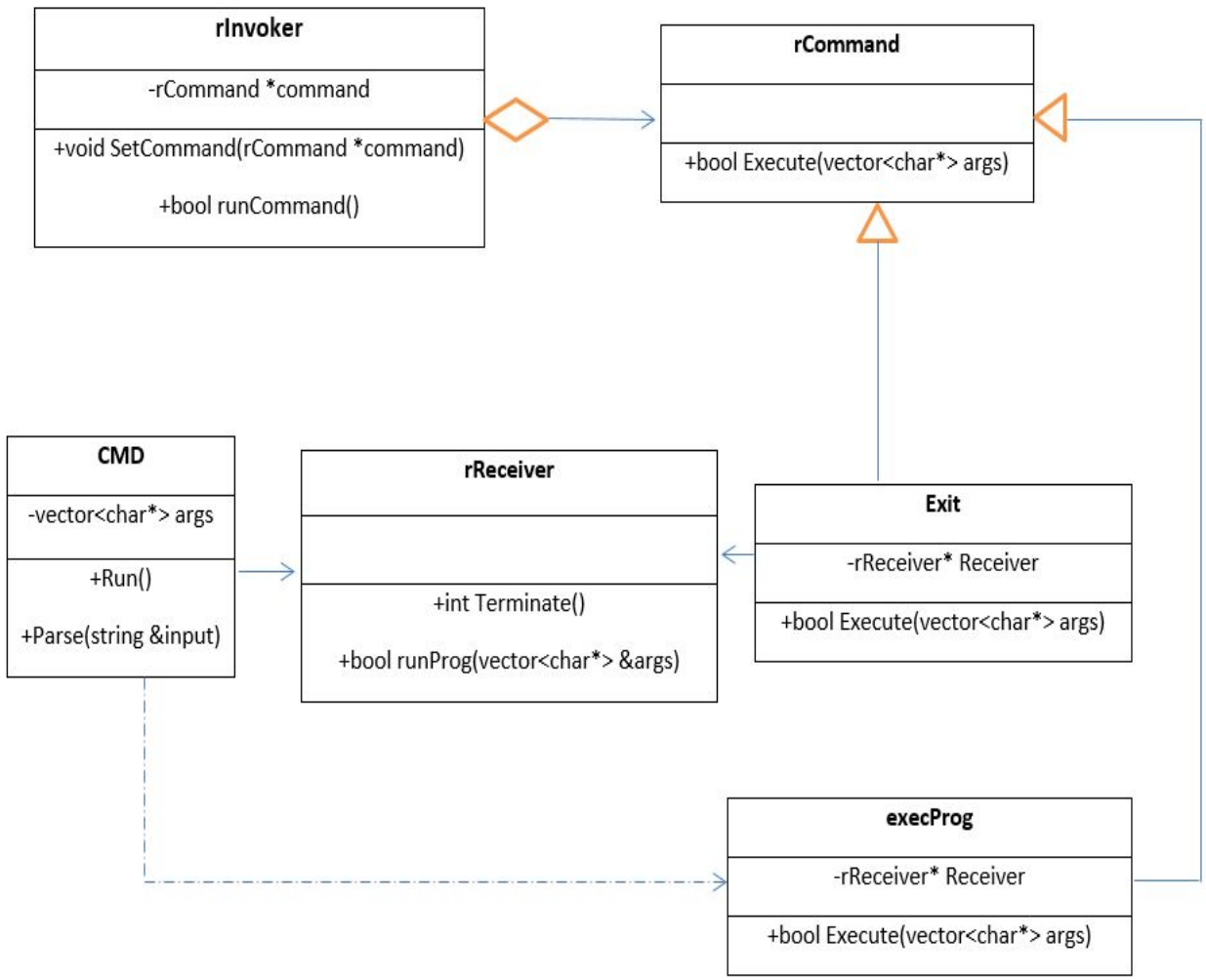
Winter

Benjamin Mao, Johnathan Murad

Introduction

We have chosen to implement a command design pattern for our shell. We will have 3 main classes that will be responsible for invoking and receiving the commands. We have a base class called rCommand that our concrete command classes will inherit from, which are our Exit and execProg classes. Our rInvoker class will be responsible for setting the command and calling execute on each of the concrete command objects. Our rReceiver class is the receiver in our design which will be responsible for implementing all of the methods associated with the concrete commands classes. So in order to call a command we will have a rReceiver pointer in each of our concrete command classes which we can use to call the methods in the rReceiver class, for instance receiver->method().

UML Diagram



Class Descriptions

rCommand

rCommand is the abstract base command class that the concrete command classes will inherit from. They will inherit an Execute function which will be used by the command invoker rInvoker to call the various execute functions the concrete command classes contain.

Exit

Exit is one of the concrete commands in our design that will extend the command interface. It implements the execute function by invoking the corresponding operations on the receiver. So if invoker->setCommand(receiver); invoker->runCommand(quit) then it will call Exit's execute function which will do receiver->terminate() and quit from the program.

execProg

execProg is another concrete command class that inherits from rCommand. It will be in charge of executing the runProgram function. Like Exit, it will hold a receiver pointer which will be used in the invoker to run the execute command for execProg.

rReceiver

rReceiver is the receiver class in our command design implementation of a shell that will be responsible for holding all of the methods that will be called by our command objects' execute function such as terminate() and runProgram().

rInvoker

rInvoker is our command invoker class that will be responsible for invoking commands. In essence it will the ask our command to carry out the request. It will take an rCommand pointer and will call setCommand(receiver) and runCommand(receiver) with receiver being a receiver pointer. The runCommand function will call the concrete command objects' execute functions by doing command->execute().

CMD

Our CMD class is our "client" in the command pattern design that will have a run function responsible for the main loop of the program. The run function will create all the rInvoker, rReceiver, and concrete command objects and will ask for a command to be executed.

Coding Strategy

For our coding strategy, Benjamin Mao will be in charge of implementing the commands and Johnathan Murad will be in charge of implementing the overall structure of the program such as setting up the class and header files and such.

For the input parsing we will utilize a queue and a vector. The queue will take in the connectors ";", "&&", and "||" whereas the vector will take in the commands. We plan to use stringstream to split the input string into tokens which we can then push them into their respective containers.

As for our exit method, we will use the cstdlib's exit function.

Roadblocks

One of the major roadblocks I think we will come across is parsing the input given by the user. Input parsing is a concept that is not really discussed much in detail so it is definitely one of our weaker aspects. In order to solve this problem we have to get acquainted with debuggers and such in order to see where exactly the error is in our problem and refer to documentation to figure out how to solve them.

Another roadblock that we will come across is to ensure that all newly allocated memories will be deallocated in order to prevent memory leak. Since we are working with a lot of pointers we are bound to get a leak somewhere and in order to solve this problem we can use a combination of the standard library's smart pointers which have automatic deallocation for dynamically allocated memory as well as utilizing debuggers to figure out where the leaks are occurring.