Assignment 1: Design

10/20/17

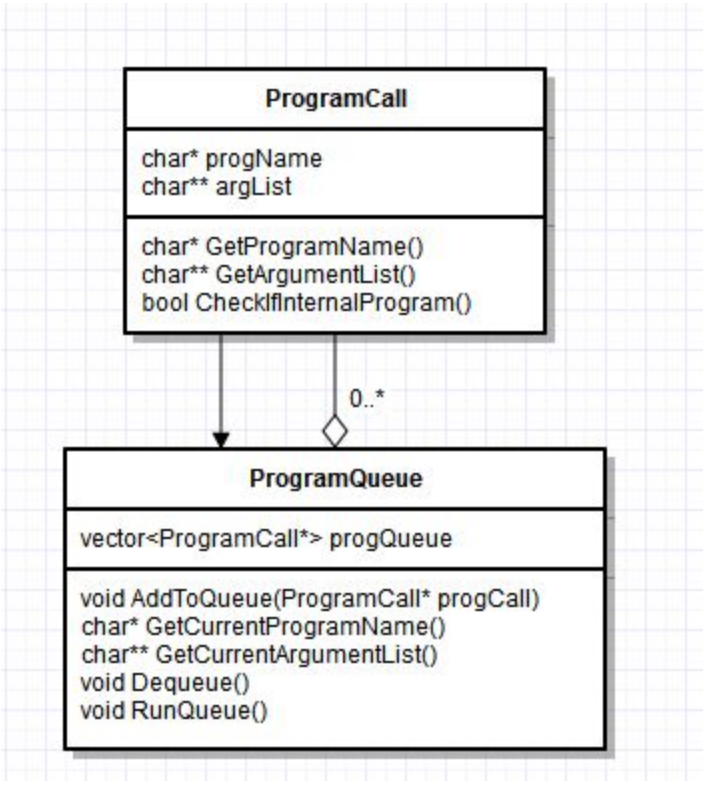
Fall Quarter 2017

By Jake Rodriguez and Lawrence Yuen

Introduction: The end goal of our design is to create a command shell called “rshell”. The function of which can be described in an epic: “As a user, I would like to run as many programs as I’d like with as many arguments as I’d like in just a single command line.” And of course, in a user story, “As a user, I would like to exit the program.”

==========

Diagram:



==========

Class Groups:

ProgramCall class:

private:

char* progName;

char** argList;

public:

ProgramCall(char* programName, char** argumentList);

This creates an object with a program name to run, and an argument list so we can run execvp() with them.

~ProgramCall()

Simply deletes the allocated memory for the variables.

char* GetProgramName()

Returns progName

char** GetArgumentList()

Returns argList

bool CheckIfInternalProgram()

What this does is check if the program name is one of the programs we have implemented, or may implement in the future, such as the “exit” command.

ProgramQueue class: (composite class of ProgramCalls)

private:

queue<ProgramCall*> progQueue;

A queue of ProgramCalls to be run in the order that they were typed in the command line. We chose a queue since we want to run the program calls in the order that they were typed, then a queue is a very convenient structure to store them in. We use a pointer so we can store the program calls in memory until they are ran, where we can then delete them.

Public:

ProgramQueue()

Just creates an empty progQueue.

void AddToQueue(ProgramCall*)

Adds a ProgramCall to the progQueue.

char* GetCurrentProgramName()

Gets the current program name.

char** GetCurrentArgumentList()

Gets the current argument list

void Dequeue()

Removes the top program call from the queue.

void runQueue();

What this function does is effectively run the programs in order according to the progQueue. We run everything forward in a while loop as long as the queue is not empty. First, the program is forked using fork(). The parent waits as long as the child is running with waitpid(). The child executes the ProgramCall in the front of the queue by running the ProgramCall’s progName and argList with execvp() (we will check if the program name entered is “quit” before running execvp with CheckIfInternalProgram(), and if it was, the shell would display a goodbye message before closing). For testing purposes, we’ll print “Success” if execvp returns a non-negative value, and “Error” if it returns a negative value. Then, the child process exits, and the parent process continues. The front ProgramCall is dequeue’d, and the while loop continues running programs as long as the progQueue is nonempty.

main() and its helper functions:

Main by itself will not do much. For main, we need several functions that actually create the shell and allows it to take input.

void RunMenu()

What this does is, in a loop, prompt the user for input, meaning the programs the user intends to run and their arguments. A separate function handles parsing the input and creating the program queue to be run (parseInput()). This function solely prompts the user for input and tells the program queue to run.

ProgramQueue* parseInput(string input)

What parseInput aims to do is get the user's input with a function such as getline(), parse it into separate program names and argument lists, then creates a program queue with the input, returning a pointer to the queue.

==========

Coding Strategy: We plan to split the work between us like this: one person implements main and its helper functions (RunMenu and ParseInput) since parsing the input is a complicated enough task, and the other person implements ProgramCall and it's composite ProgramQueue, since getting the loop to run the programs one after another is a difficult task. Jake Rodriguez will implement the main and its helper functions, and Lawrence Yuen will implement the ProgramCall and its composite ProgramQueue.

==========

Roadblocks: I highly anticipate parsing the input strings in of itself to be a hard task to do. This can be helped by implementing a testing function separate from the program classes that takes in a program name and argument list and attempts to run it. Another roadblock is that the program classes will be difficult to test without fully implementing the main function and its helpers. This can be alleviated by the class implementor also implementing the aforementioned testing function and by working as a tester for the input parser while waiting for the main function to be confirmed as working. I also hope to alleviate future growing pains with the CheckIfInternalProgram function, as we may have to implement functions ourselves rather than run an external program, such as the "cd" command.