Assignment 1: Creating a Design Document

11-29-18

Fall 2018

Jeff Trang (861261717)

Mitchell Doan (862022288)

Introduction

We will be designing an rshell program using the composite pattern. It will consist of a base component which will contain child classes to store the command prompt and its connectors. The Command class will store a string that consists of the command line. The TestCmd class will also store a string that consists of the command line. The Connector class will store two Base* objects and will or will not execute the passed in commands based on the previous bool value from execute().

Classes/Class Groups

Our design consists of three classes/class groups which are the Base class, Command Class, and Connector class. Additionally we will create a main to ask the user for an executable. Additionally the main will parse the user input and push the commands and connectors into their appropriate vectors.

The abstract Base class will consists of a virtual function, execute, which will be implemented in each of its child classes.

The Command class, a child of the Base class, consists of a private string variable, which will hold a command based on the user input. The execute function will run the passed in command and return a bool on whether or not the command failed or passed. The execute function will use certain functions fork(), wait_pid(), and execvp() in order to successfully run the command without creating errors or undefined behaviors. The fork function will create a duplicate child to the parent executable. Using wait_pid() assures that the child runs all its processes before the parent. Using execvp() actually executes the command that was passed in.

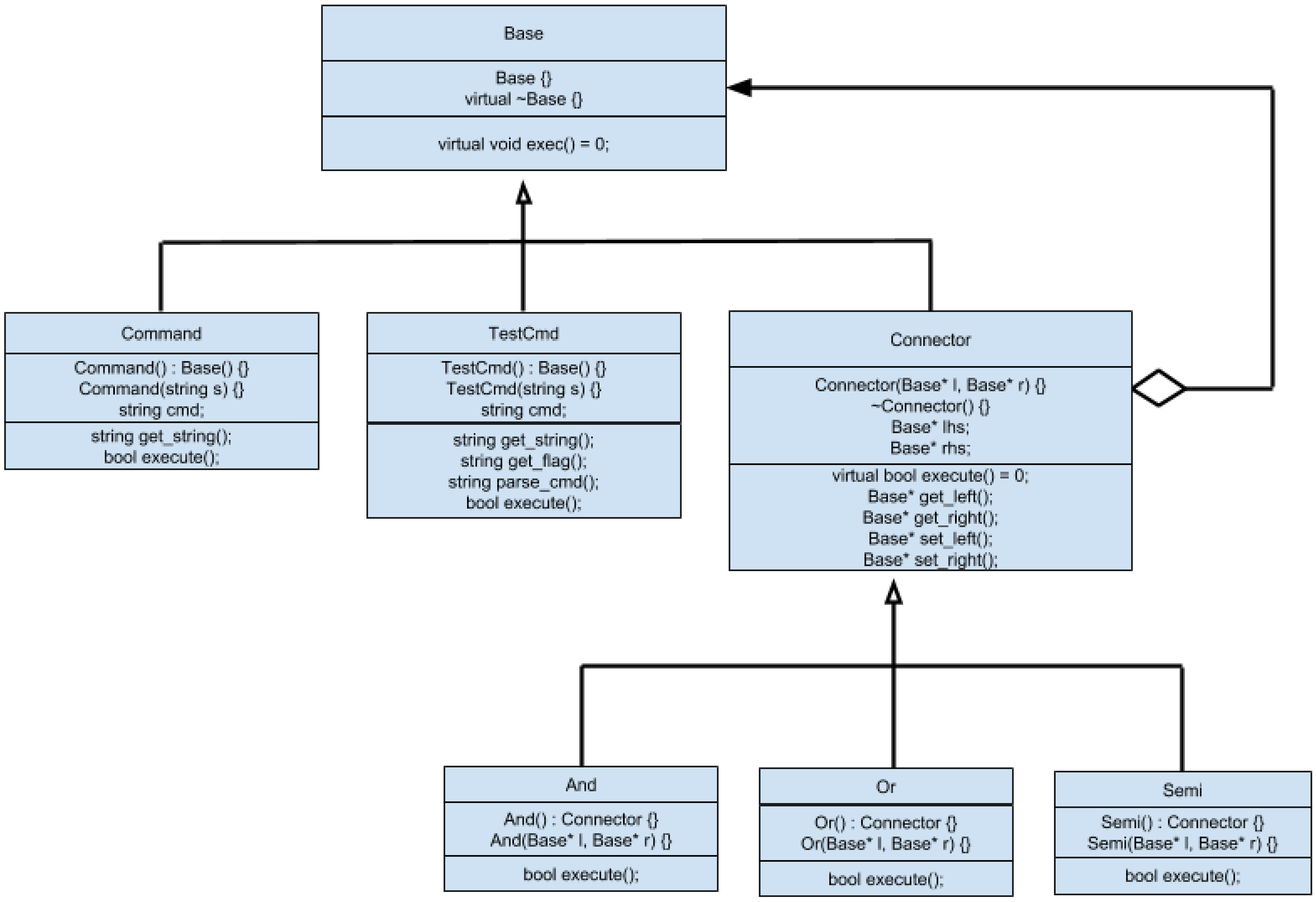
i.e. if user inputs : echo hello world
The “hello world” will be outputted.

The TestCmd class, a child of the Base class, behaves similar to that of the command class in that it will take in a similar form to that of a command object. A string is stored as its private variable. It has the functions get_flag() to determine which type of flag is stored in the string or if there is none at all. The string will then be sent into parse_cmd(), where the “test” keywords as well as any flag indicated by ‘-’ will be cut off from the string. The new string will then be sent in the execute() function where we will use the stat() function call as well as the macros, S_ISREG and

S_ISDIR to determine if the string will correctly see if the string is a file or directory and then return the proper bool value.

The Connector class, also a child of the Base class, will hold protected variables (l & r) and has three child classes: &&, ||, and ;. We decided to have two protected variables in order to store the executables and arg lists of both the right and left side of the connector. Additionally, the Connector class private variable “lhs” or left hand side has the ability to store other connector objects. This is in order to allow us to know whether or not we must execute a command based on whether the previous command passed. The “And”, “Or” and “Semi” classes will each have their own exec() functions that will execute the command based the private variables lhs and rhs. The execute() function will return a bool so that the next connector knows whether or not to execute its right hand side command.

Diagram



Base
Base {}
virtual ~Base {}
virtual void exec() = 0;
Command
Command() : Base() {}
Command(string s) {}
string cmd;
string get_string();
bool execute();
TestCmd
TestCmd() : Base() {}
TestCmd(string s) {}
string cmd;
string get_string();
string get_flag();
string parse_cmd();
bool execute();
Connector
Connector(Base* l, Base* r) {}
~Connector() {}
Base* lhs;
Base* rhs;
virtual bool execute() = 0;
Base* get_left();
Base* get_right();
Base* set_left();
Base* set_right();
And
And() : Connector {}
And(Base* l, Base* r) {}
bool execute();
Or
Or() : Connector {}
Or(Base* l, Base* r) {}
bool execute();
Semi
Semi() : Connector {}
Semi(Base* l, Base* r) {}
bool execute();

Coding Strategy

We will have a main file which will cout “$ “ to prompt the user to provide their input executable. Within the main file, Jeff will parse the string input which the user provided. He will also create functions which will parse the user input and push the commands into a command vector and connectors into the connector vector. Additionally, Jeff will create multiple functions which help the parsing process such as where to end (#) and if the user passes in exit. Jeff will create an abstract base class.

Under the Base class will be a child class, Command. Mitchell plans on working on the Command class which will consist of a string along with two functions, exec() and get_string(). The get_string() function will simply return the command in the private variable. The Command class will also have an exec() function which takes the string command (i.e. echo hello world) and convert this into a vector of characters. With the vector of characters, we will separate the command and argument in order to push them into an array. The array will call fork() in order to create a pid for the child and parent. If fork fails, we will return a perror. Otherwise, the parent will wait for the child to finish its processes including when it runs execvp(). This function will actually execute the command, and if an invalid command is passed in, it will return negative.

Then we will create a Connector class. Both Jeff and Mitchell plan to work on this class which will consist of private variables (lhs & rhs) along with a public virtual exec() function. We will then create three child classes of the connector class (And, Or, and Semi). Each class will hold an exec() function which will decide whether or not to run the rhs variable (command).

Jeff plans on testing certain commands with and without connectors to assure that our methodology works as expected.

Mitchell will then create gtests which will assure that our functions work correctly before testing our entire rshell.

Roadblocks

- Finding a way to output the contents of the stack of vector<string> properly
- Using execvp() function call correctly on the vector of strings
- Parsing the user input correctly and getting rid of white spaces
- Knowing when to stop parsing the user input
- Knowing when to use fork() and waitpid() function calls
- Determine how the exit function interacts with connector statements
- Make sure to detect when a user will input a message during the command line input during execution.
- Preventing memory leaks and fixing bugs