Assignment 1: Design Document

Description

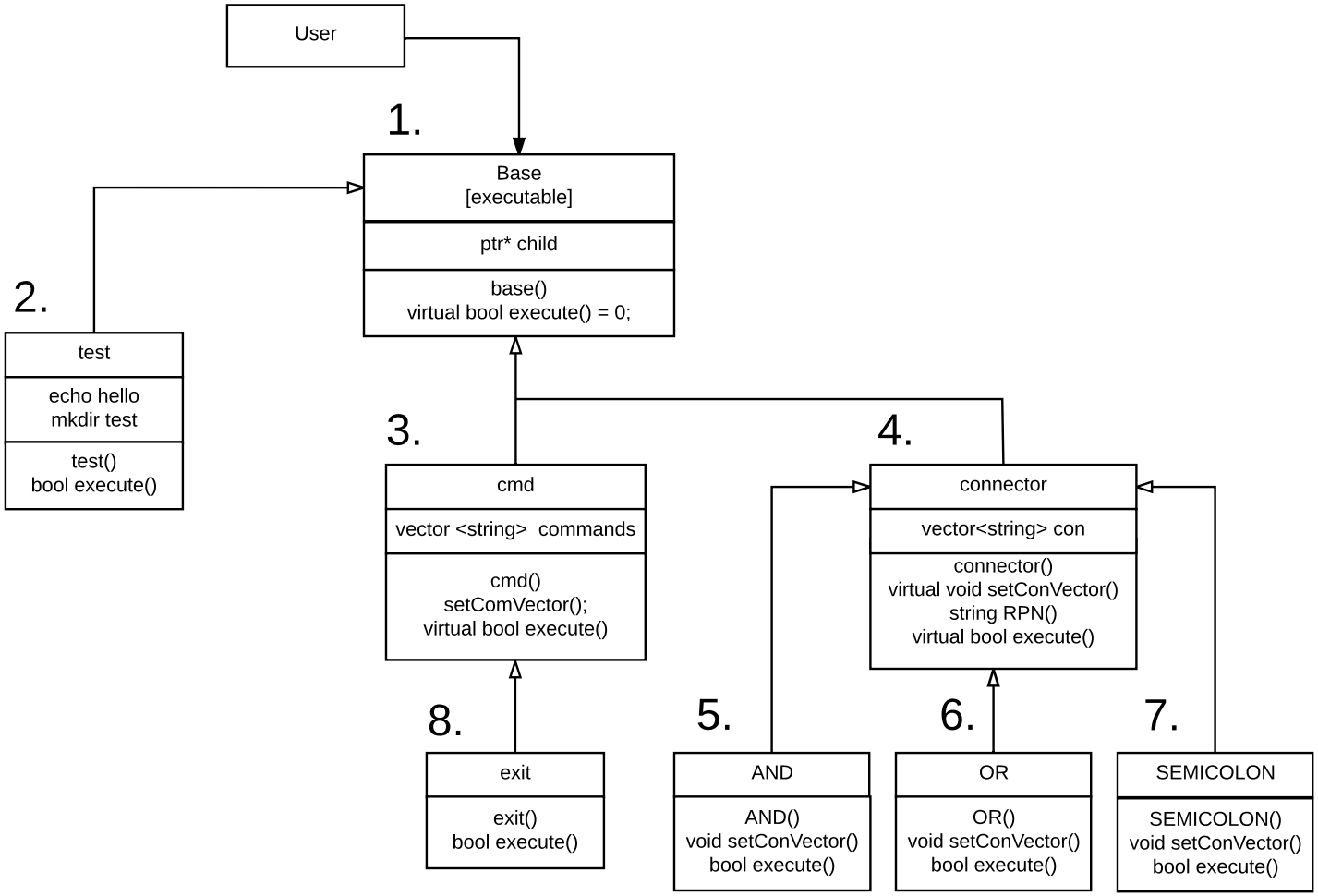
This design document is by Justin Schopick and Neal Goyal for the CS 100 Assignment 2, the rshell program.The rshell consists of executables that performs different aspects of the program. In this design document, there will be four pages consisting of specific descriptions on the project's functionality. The first page will cover the UML diagram with footnotes on the bottom describing each command. The second page will cover an overall project structure for the layout of the repository. The third document will cover the scripts' functionality in order to show how we run through each part of the program. The fourth page will cover our strategy for parsing through each command by using Reverse Polish Notation, or RPN, with diagrams and descriptions. Finally, the last page in the document will cover any roadblocks that may come up and our strategy to overcome them.

Epic

As a couple of CS 100 students, we should create a program that creates executables to parse through commands in order to create an rshell.

- Use RPN to create a stack of the connectors and commands
- Use boolean executes to determine whether each class function is true to parse through the text
- Create a test class to test each function
- Test the functionality further using scripts to create specific test cases with each class

UML Diagram

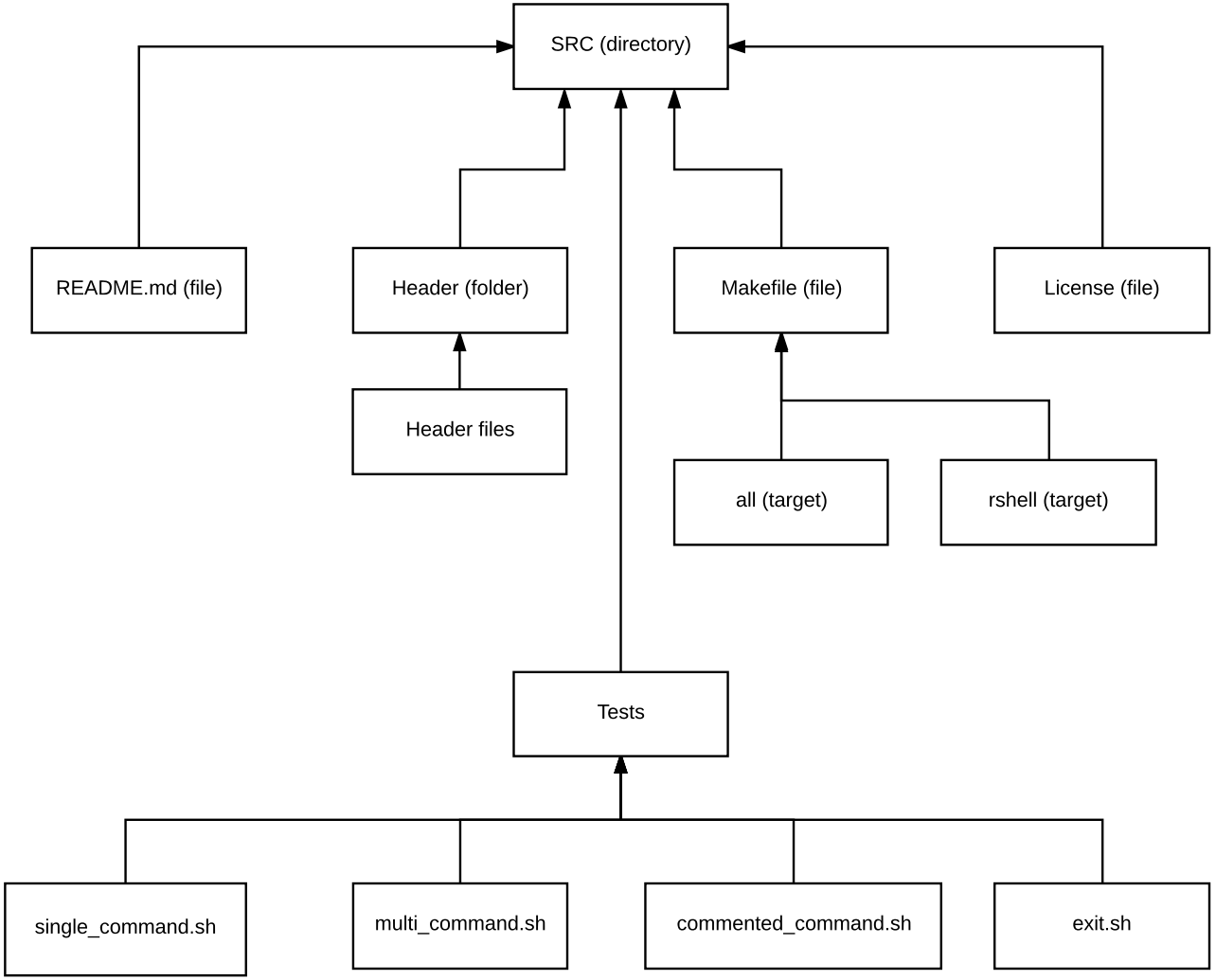


Class Descriptions:
1. The base class is the executables. This is the parent class for all the commands. It has a virtual void execute function because each class is inheriting from the base class. Likewise, each class has a bool execute for their specific executions. Therefore, each class can override the base class' execute function. We are also including a child pointer for forking the parent class. This will be useful when dealing with memory management.
2. The tester class is created with using echo and then creating another director to test each aspect of the code. Therefore, there is also a bool in order to determine if each aspect of the code is true/false.
3. The cmd class creates the actual commands. In order to collect all the commands and add new ones, we will create a vector of strings.
4. The connector focuses on the &&, ||, and ; to connect the commands. There will be a RPN function to parse through the connectors and create them in a stack. From the description, we learned that each connector executes depending on if the previous one fails (ie. if/else statements). Therefore, we are using a bool as well to execute each aspect of the code.
5. If the AND function returns true, then the next command will succeed. We will parse through the && connector by using bool and the string RPN function.
6. If the OR function returns true, then the next command will succeed. We will parse through the || connector by using bool and the string RPN function.
7. If the SEMICOLON function returns true, then the next command will always return true. We will parse through the ; connector by using bool and the string RPN function.
8. This is an exit class that will return true when there are no longer any other commands from the vector.

Project Structure

Description

This page is focused on the overall project structure. There are five main parts: README.md file, header files, scripts, Makefile, and license. The main parts of the assignment are the Makefile and scripts. The Makefile targets the program with all and the rshell. The scripts test the functionality of the program, covering each test case, including edge case. The purpose of this assignment is the create a functional Kanban board with an epic to portray agile methodology.



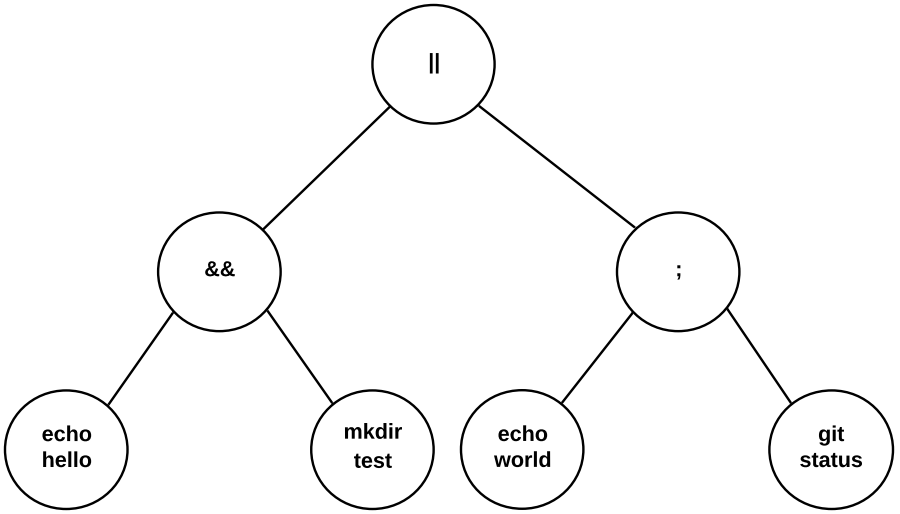
Script Functionality

single_command.sh	The purpose is to create a shell to test a single command. This will be a simple echo parameter ($1) command to determine whether the function outputs correctly.
multi_command.sh	The purpose is to create a shell to test a mutli command. We will code in bash to create a for loop determing whether the script follows the RPN code. If successful, the script should create an if statement echoing a correct statement or a error response.
commented_command.sh	The purpose is to create a shell to test commands with comments. The code will be a script using a for loop with comment commands (ie. # or //) stored as variables, and go through commands to determine whether they exist or not and determine whether the code recognizes the comments by creating an if statement similar to the multi_command.sh if statement.
exit.sh	The purpose is to create a shell to test whether the function will exit. The main functionality will be determining whether the bool execute will return correctly. We will code in bash to create an if statement and use echo to output whether the function returned correctly.

Coding Strategy

Example: `echo hello && mkdir test || echo world; git status`

Commands	Connectors
echo	&&
mkdir	||
echo	;
git	



Description

We will be using Reverse Polish Notation, or RPN, to parse through the commands. This will be like a infix traversal as demonstrated in the tree diagram above. Our goal is to parse each command and connector in order to output each command in order by precedence. We will use the vectors in the UML diagram to store the commands and connectors. Then, we will use the bool execute() to determine what command it is and then follow the guidelines of AND, OR, and SEMICOLON precedence to finish each statement on the command line. In terms of runtime, this tree should be an O(n) runtime because once the vector is parsed, there will be a single for loop searching for an element through the vector.

Partner Breakdown

Both:
- Create the Base class and Test class.

Justin:
- Create the command class with the exit subclass.
- Create the comment and exit scripts

Neal:
- Create the connector class with the AND, OR, and SEMICOLON child classes.
- Create the single and multi command scripts

Roadblocks

- Unfamiliarity with Languages and Concepts
  - Neither of us are really familiar with bash, shells, or git. We need to look at tutorials and practice writing code in order to be successful in this project. As a result, this will be a slower process at first, but we will learn over time.
- Working as a Team
  - We have a lot of commitments, both academic and extracurricular, in addition to this project. Scheduling time to work together, whether we simply discuss our plan or work on code, will be difficult.
  - We have exchanged our standard weekly schedules with each other and will be in contact throughout the week to spread the work out and make sure it is done in a timely manner.
  - Communication - We need to be on the same page with this assignment. We will be meeting in-person to discuss or work on it at least once a week and will be updating each other as we progress.
- Splitting up Tasks
  - Based on our coding strategy, we will be splitting up how we tackle each section of the assignment. If one of us writes code that the other needs for another piece of the assignment, the other person may need to wait for that code to be complete in order to use its functionality. One way to avoid this is to have one person handle a specific class and all of the subclasses so that they understand the functionality of all of the code they write and can tackle it in the necessary order. We should still be able to understand all of the code written in the assignment.
  - Also, when working on code together, making sure there are not merge errors when working on different branches on git. A potential error is for one code to overwrite another's.
- Segmentation Faults
  - We will be using pointers and need to make sure we do not try to access memory that is not allocated. Our plan to avoid segmentation faults is to throw exceptions; this will also help us find where the errors may be.
- Memory Leaks
  - This is a larger project; if we do not free memory after we are done with it, we may come across unknown errors or find the execution of our code impaired. The plan to avoid these is to just really keep in mind that whenever we allocate memory(i.e. in a function), we need to free it when we no longer need it(i.e. at the end of a function).