rshell Design Document

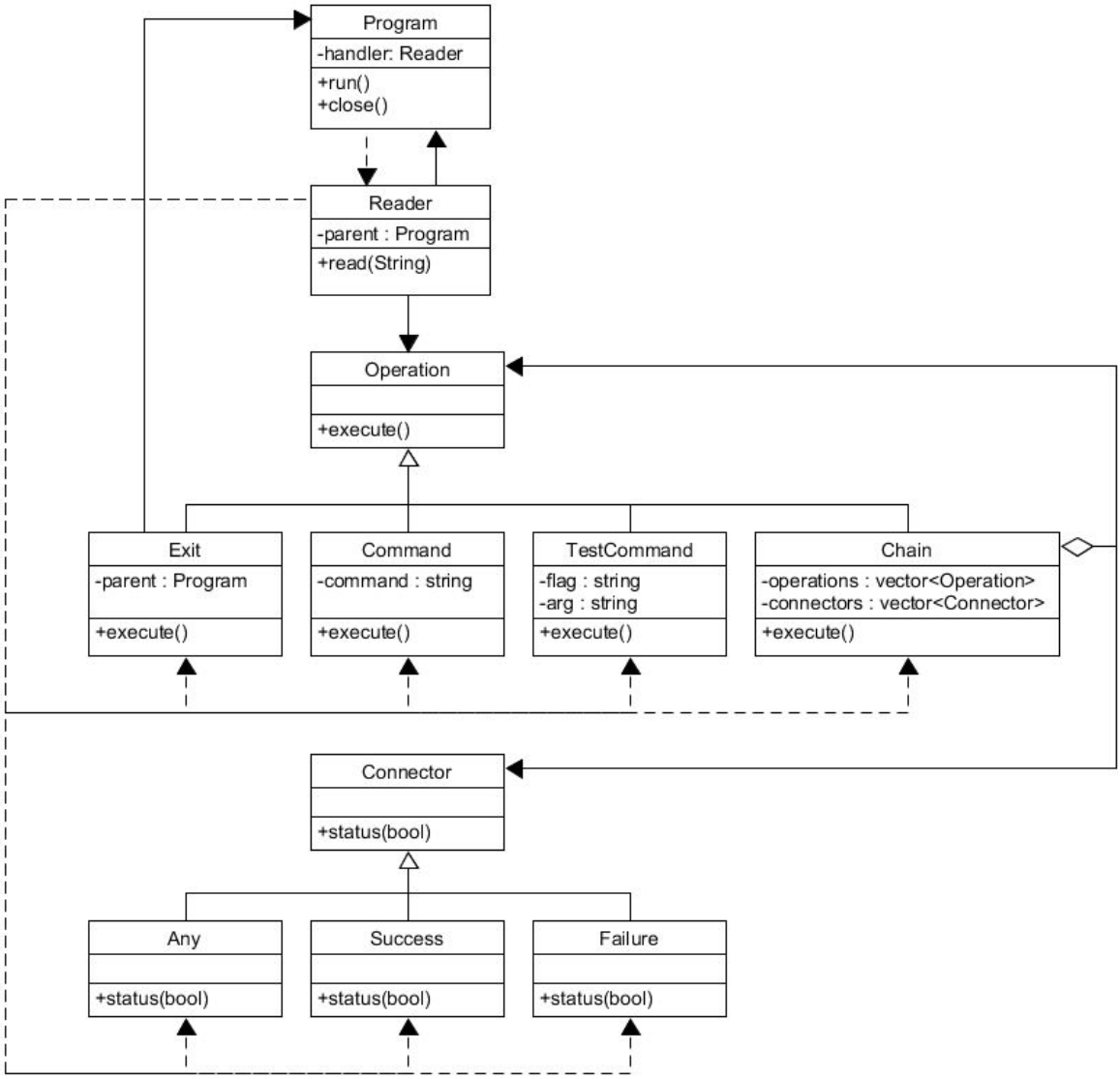
October 19, 2018
Fall 2018

Alex Chen
Kyle Tran

Introduction

The design of our command shell uses the composite pattern to represent our commands and operators. The shell will initialize by creating a **Program** instance that will split each entered line by the semicolon into “statements.” A **Reader** will parse the “statements” to identify individual **Command**s and **Chain**s of **Command**s linked by **Connector**s. **Command**s and **Chain**s are both subclasses of the **Operation** base class to fit the Composite pattern; the **Command** is a leaf of **Operation**s and the **Chain** is a tree of **Operation**s

Diagram


Program
-handler: Reader
+run()
+close()
Reader
-parent : Program
+read(String)
Operation
+execute()
Exit
-parent : Program
+execute()
Command
-command : string
+execute()
TestCommand
-flag : string
-arg : string
+execute()
Chain
-operations : vector<Operation>
-connectors : vector<Connector>
+execute()
Connector
+status(bool)
Any
+status(bool)
Success
+status(bool)
Failure
+status(bool)

Classes

- **Program**: Prints the prompt, accepts input, sends each line to the **Reader** to create an **Operation**, and executes the operation. Contains a **close()** function to be called by **Exit** for early termination. Contains an **istream** (for input) and two **ostreams** (for output and debugging) for debugging purposes.
- **Reader**: Single-use object that parses a given **string line** by one **Token** at a time. When it is instantiated, it removes the comment and surrounds **line** in parentheses to ensure proper chain parsing. Has an **unsigned index** member to track the parsing progress. Has a reference to the **Program parent** for the **Exit** command. A **Read()** function identifies a token at the current index and **UpdateIndex()** increments the index past the current **Token**. The Reader has several specialized functions to handle specific structures: **ParseChain()**, **ParseConnector()**, **ParseCommand()** (for regular Commands plus Exit and Test), **ParseArgument()**, **ParseQuoted()** (for string arguments), **ParseBackslash()** (for escaped chars), and **ParseTestBracket()**. All functions have specific delimiters and throw exceptions upon reaching unexpected tokens (i.e. double **Connectors** in **ParseChain()**). **ParseChain()**, **ParseQuoted()**, and **ParseTestBracket()** expect the index to be directly after their respective opening delimiter.
- **TokenTypes**: An enum type representing individual types of tokens to parse for: parentheses, brackets, connectors, string arguments, and the line terminator.
- **Token**: Object that contains a **TokenType** and the **string** value of the parsed token.
- **Operation**: Contains an **execute()** method that accepts an input pipe and an output pipe for redirection (by default, both are zero) and returns a **bool** indicating success. Also contains a **print(ostream& out)** method which prints out the equivalent source of what the Operation executes.
    - **Chain**: Contains a **operations vector** and **connectors vector**. When executed, it iterates through **operations** and **connectors**, alternating between executing the current **operation** and checking the status of the current **connector**, and returns a **bool** indicating the success of the last **Operation**. Pipes are passed down to operations. Serves as a composite of **Operations**.
    - **InputOperation**: Contains a **reader** Operation to send input to and a **file** string to get input from. When executed, creates a pipe, reads the file into the pipe, and executes **reader** with the pipe as input.
    - **PipeOperation**: Contains a **writer** Operation to extract output from and a **reader** Operation to send input to. When executed, creates a pipe,

executes **writer** with that pipe as output, and then executes **reader** with that pipe as input.

- **OutputOperation**: Contains a **writer** Operation to extract output from, a **file** string to send output to, and an **append** flag. When executed, creates a pipe, truncates or opens (if **append**) the **file**, and executes the **writer** Operation with pipe as output.
- **Command**: Contains a **string vector** of arguments to be executed. When executed, the Command forks the process, converts the vector to a char string array, calls execvp() on the arguments, exits the child process, and returns a **bool** indicating success. Pipes are duplicated by the child and closed by both sides upon forking. Serves as a leaf of **Operations**.
- **TestCommand**: Contains **strings** for the **flag** and the **argument**. When executed, checks for the existence of a file or directory at the path named by **argument** and checks that its type matches the one specified in **flag**. Pipes are duplicated for the child and closed by both sides upon forking.
- **Exit**: Terminates the **parent Program** by calling **close()** and returning **true** when executed.

- **Connector**: Contains a **status(bool result)** function that determines whether or not to continue based on the **result** of the previous **Operation**. Also contains a **print(ostream& out)** method for debugging.
  - **Success**: **status** returns **true** if the **previous Operation** succeeded (**result** is true). Returns **false** otherwise
  - **Failure**: **status** returns **true** if the **previous Operation** failed (**result** is false). Returns **false** otherwise
  - **Any**: **status** returns **true**.

Coding Strategy

We will work on the initial program and reader together to ensure the foundation is sound. Same with the operation class so we know the functionality and agree how it should be designed. Kyle will write the Connector, Operation, Command, and Exit classes in addition to the CMake and Google Test materials. Alex will write the Program, Reader, and Chain classes. We will collaborate for the entire process however to ensure the correctness of our design. We will use separate git branches and merge when needed. With every completed feature we will commit and push. We will also split the unit test cases for our respective parts.

Roadblocks

- Parsing input. For this we will need to collaborate and think of all possible scenarios for error
- Merging our files. Make good commits so we know what has changed
- Chain class. This can get messy for iterating through commands and checking the status. For this heavy testing will be needed.
- Testing. We will need to think of every situation a user can use the program in and test accordingly.
- Using system calls. We don’t have that much experience using system calls so we will need to do our research beforehand to learn the intricacies.
- Future assignments. We may have to revise the structure to fit new developments