Assignment 1: Design

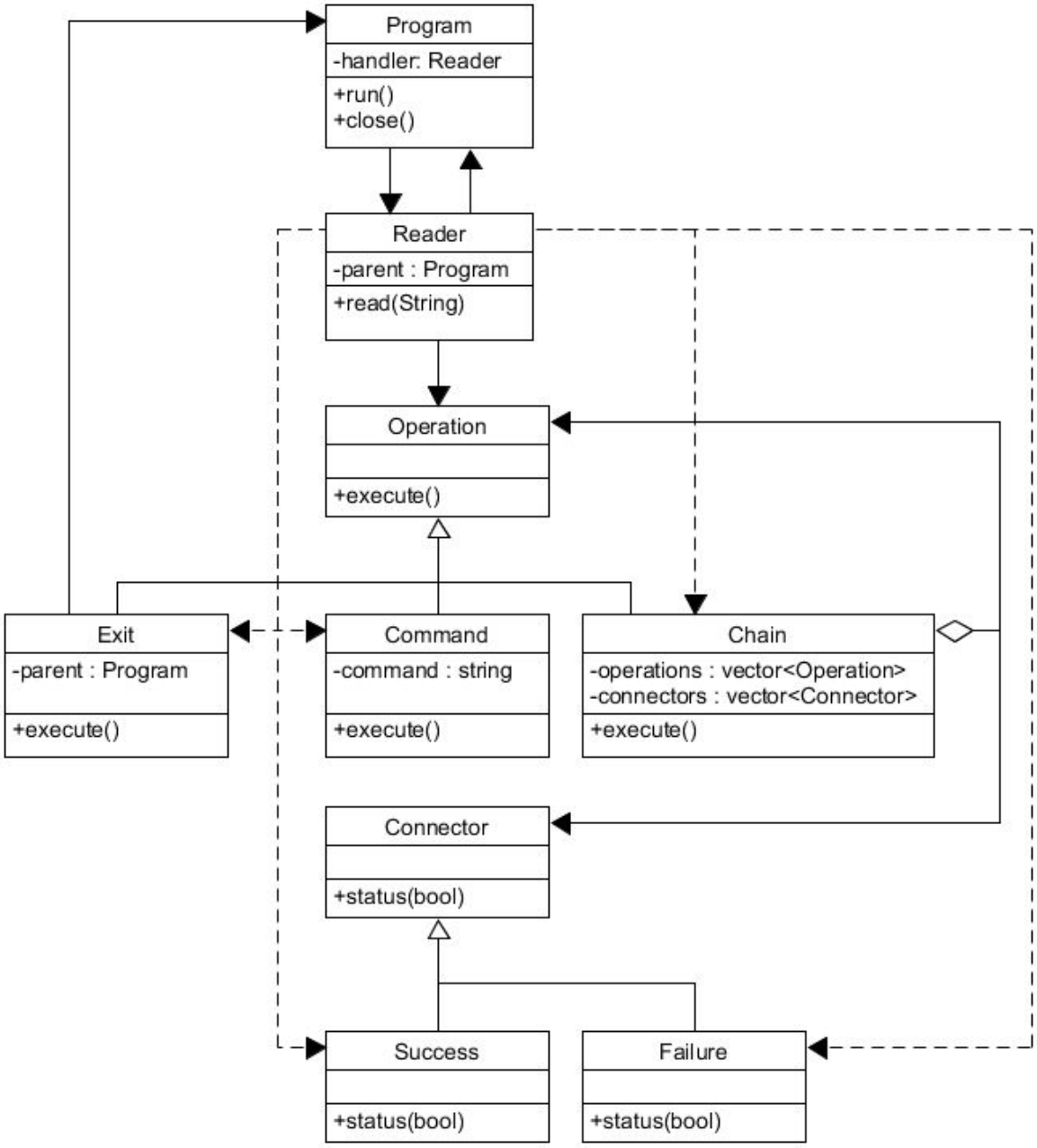
October 19, 2018
Fall 2018

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



[Umlet File]

Classes

- **Program**: Prints the prompt and reads input to be sent to the **Reader**. 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**: Contains a **readLine** function that splits lines by the semicolon into "statements" (after removing comments) and evaluates each statement separately. Parses statements for tokens using space as a delimiter. Contains a reference to the parent **Program** so that it can recognize and create the **Exit** command. Splits the line by the semicolon and then parsing each piece as a **Chain**. When reading a statement, it keeps a **vector of Operations** and **vector of Connectors** in case it has a chain. It assumes that each substring that does not match a **Connector** is an argument to a **Command**. If it reaches a **Connector**, then it stops parsing the current Command, adds the **Command** to a **vector of Operation**, and adds the **Connector** to a **vector of Connectors**. When it reaches the end of the statement, it adds the trailing Command and detects whether it has a Chain or single Command by checking the count of Operations and Connectors.
- **Operation**: Contains an **execute()** method that returns a **bool** indicating success. Also contains a **print(ostream& out)** method for debugging.
    - **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**. Serves as a composite of **Operations**.
    - **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. Serves as a leaf of **Operations**.
    - **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

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