# Assignment 1: Design

November 1st, 2018

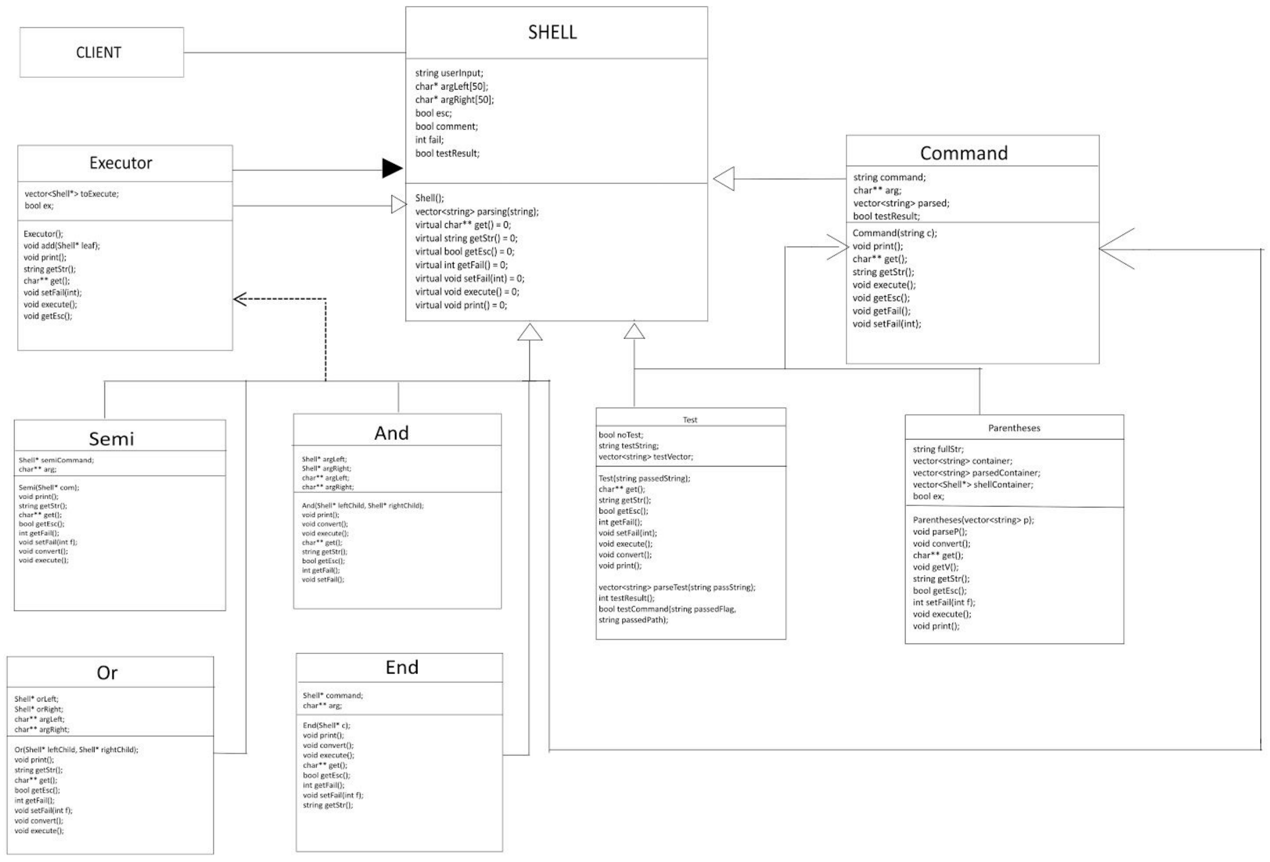
Fall 2018

Bryan Nguyen & Brandon Cai

## Introduction:

Our design breaks the program down into a few simple steps. We have an interface “Shell”, we take user input, we parse the input in order to find how the user would like to have their commands executed, and then we will execute those commands in the order that it was parsed. We created a parser class which parses and stores the parsed strings in a vector of strings. We separated them by connectors and words. After that, we created our composite class, which will take the parsed object into a vector of our base class pointer which will then unpackage the string into char** and finally execute to make it appear like a real, basic command shell.

## Diagram:



High Quality Link of UML

# Classes/Class Groups

## CLASS GROUP SHELL

This is our abstract base class that all our classes will inherit from.

**Shell();** - Shell constructor.

**vector<string> parsing(string); -** Parses string by breaking up the string based on connectors such as ; && ||. As well as other chars of interest such as ( ) and [ ].

**virtual char** get() = 0;** - This function allows us to return data type at any given method.

**virtual string getStr() = 0;** This function allows us to get the string that is currently being used by a class. This is useful for testing.

**virtual bool getEsc() = 0;** Allows us to check if the user has typed ‘exit’ or not within classes.

**virtual int getFail() = 0;** Allows us to check if the command has failed within classes.

**virtual void setFail(int) = 0;** Allows us to manually set the fail state if we need to.

**virtual void execute() = 0;** Execution command based on connector/class needed.

**virtual void print() = 0;** Debugging function we use to see what variable a class is working with.

**virtual void convert() = 0;** Generally used to convert a string to a char in order to pass to argument.

## CLASS GROUP COMMAND

The command class is charge of taking in a string from our vector of parsed strings and assigning it to a the proper “factory” if you will, that being and, semi, or, etc.

**Command(string c);** The constructor of the class takes in the string and assigns it as a command within the class.

**void print();** Simply prints out the command.

**char** get();** This function converts the string that we input into a char** which can be used as an argument for execution.

## CLASS GROUP EXECUTOR

This class is in charge of creating a tree so that we can traverse it and execute it when it is called. It essentially holds a tree of commands and arguments that we can later call to execute in order.

## CLASS GROUP CONNECTORS

**And:** This class/object will be created when the program sees an && within the parsed vector and will execute the right command only if the left command runs.

**Semi:** This class/object will be created when the program sees a ; within the parsed vector and will execute the left and right commands always.

**Or:** This class/object will be created when the program sees a || within the parsed vector and will execute the right command only when the left command fails.

## CLASS GROUP TEST

This is the testCommand object that is called when the program sees the keyword for the command “Test”, when this object is created, we will simply execute the command along with the flag and path, and whatever is returned we will replace what is inside the original vector of strings to either testPass, testFail, or testError depending on what was returned. The connector classes are programmed so that it will recognize these strings execute those strings accordingly.

## CLASS GROUP PARENTHESES

The Parentheses Class allows us to utilize order of operations to our code. The Parentheses constructor deals with much of the work, since it parses the vector of strings that is sent to it. **parseP()** further parses the vector, by parsing spaces and the parentheses themselves. Though it may not be fully utilized such as (echo x) does not work, we believed that it was not the most important to deal with and went to complete more important tasks. The **getV()** method parses the things inside the parentheses such as &&, ||, and ;. After parsing like so, we are able to parse it given some logic to the executor class so the order of operations are being followed.

## Coding strategy:

We will break up the work as 50/50 to the best of our ability. Brandon will take care of the Composition part of the code and the Bryan will be in charge of the strategy implementation. We will integrate the segments together by pushing our code to the github repository and notifying each other of changes so we can make sure that the code is integrated along the way instead of waiting till the end when the code is done to implement in which the code may not be compatible. This way we won't run into catastrophic errors because of the incompatibility in our parts.

**UPDATE:**

The coding ended up being about half and half. Brandon took care of the parsing, while Bryan took care of the composition and execution. We also took care of everything related to our responsibilities such as gtests.

**UPDATE: ASSIGNMENT 3**

We rewrote the entirety of assignment 2. Bryan was in charge of connectors and execution, while Brandon was in charge of parsing and the test commands. We worked together in regards to integration tests and merging our code together.

## Roadblocks:

Even though we have planned this so that adding new nodes and classes should not be much of a problem, we still expect to have difficulties implementing new strategies and compositions into our code. Also, both of us are new to using the excvp(), wait(), and fork() commands and have little to no understanding on how to use them. We plan on just using the trial and error technique and researching how and when to use these methods. Furthermore, other than file i.o, we have never worked on executing commands that affect things outside of the compiler. One of the main roadblocks will definitely be in the limited knowledge of implementation and lack of experience in building shells and the bash environment. We plan on overcoming this roadblock by working on the project in a timely manner and being able to seek help regularly from external sources.

Another roadblock would be straying from our design as the project calls for it. We think that that would be a challenge to realign our thought process for the project as it comes together. We can overcome this roadblock by consulting the professor and teacher assistants for guidance in this regard.

One roadblock that can also occur is conflicting schedules on when one person can work on the project. We plan on solving this roadblock by setting up dedicated meeting times where we can group code and collaborate.

Another roadblock is that we feel as if our code is not what it should be. Even though it works, our code is most likely inflexible in its design. We strongly believe that we will have to redo this project in hopes to make it more flexible.