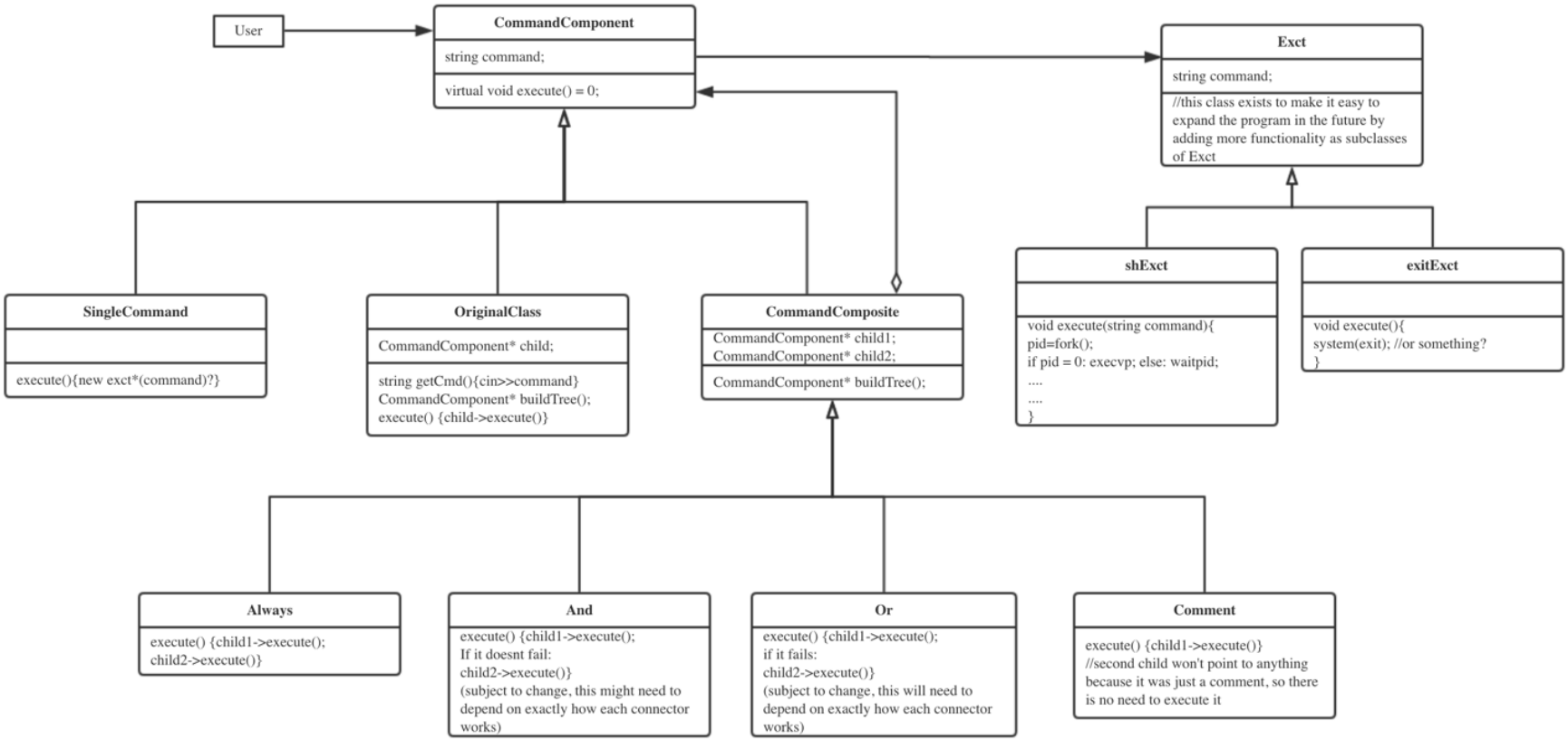
# Assignment 1: Design

October 20th, Fall Quarter, 2017

Junjie Huang and Joshua Wilson

**Introduction:**
We will use the composite pattern to create a command shell that can take multiple commands on a single line as an input and run them all in order. Also some connectors between commands will cause the next command to run or not to run depending on whether the previous command failed or not. Lastly, the user will be able to write comments into their commands using the # sign. This will cause everything after # to be treated as a comment and any commands will not be run.
An explanation of the composite structure:
The user enters a command and that is passed as an argument to a tree constructing function, which does this:
1. searches through command to determine if there are connectors or if it is a single command
2. if it is a single command, the command is simply saved as a single command and then executed with no further action needed.
3. If it finds a connector, it creates a command composite of whatever type of connector it just found, and saves everything that is on the left side of the connector (a single command) as the first child of the command composite
4. Next it takes everything that is on the right side of the connector and and starts over at step 1 by calling the tree constructing function over again, then saving the result of that function as the second child of the command composite
Once this is all done, there will be a tree composed of composite and primitive objects.

**UML Diagram:** (on next page)


User
CommandComponent
string command;
virtual void execute() = 0;
Exct
string command;
//this class exists to make it easy to expand the program in the future by adding more functionality as subclasses of Exct
SingleCommand
execute(){new exct*(command)?}
OriginalClass
CommandComponent* child;
string getCmd(){cin>>command}
CommandComponent* buildTree();
execute() {child->execute()}
CommandComposite
CommandComponent* child1;
CommandComponent* child2;
CommandComponent* buildTree();
shExct
void execute(string command){
pid=fork();
if pid = 0: execvp; else: waitpid;
....
....
}
exitExct
void execute(){
system(exit); //or something?
}
Always
execute() {child1->execute();
child2->execute()}
And
execute() {child1->execute();
If it doesnt fail:
child2->execute()}
(subject to change, this might need to depend on exactly how each connector works)
Or
execute() {child1->execute();
if it fails:
child2->execute()}
(subject to change, this will need to depend on exactly how each connector works)
Comment
execute() {child1->execute()}
//second child won't point to anything because it was just a comment, so there is no need to execute it

**Classes/Class Groups:**

*CommandComponent (base class):*
It's the base class for commands. It contains a string to store the commands that the user types in. It also has a function called execute that is pure virtual and is implemented differently for primitive classes and composite classes. Currently There is only one primitive class, but we may add more in the future depending on the requirements that we learn about in future assignments.

*SingleCommand (primitive which inherits from CommandComponent:*
This class will hold a single command in the form of a string. It redefines execute to create a new object from the Exct class. We chose not to implement the execvp() and other related functions in this class to allow for different functionality later (see Exct class for more on this)

*OriginalClass (This might be considered a composite, inherits from CommandComponet):*
This class has a single child which will be a CommandComponent pointer. It will be created at the beginning of the program before the user enters any commands, then it will be the responsibility of this class to initiate the composite tree. This is so that we don't need to rewrite some of our code multiple times. First getCmd() will be called, which will get a command which will be stored as a string, then buildTree() will be called, which will start splitting the string of commands into single commands (if applicable), which will build a tree of commands. The top level of the tree will be saved as this class's child. Then execute() will be called which will simply call execute on the first component in the tree.

*CommandComposite (composite class, inherits from CommandComponent):*
Has two children, child1 and child2. This class should never be directly created, it will define buildTree() for all of the other composites which will be important because the tree will be build recursively. When the buildTree() function finds a connector inside of a command, it stores everything to the left of the connector as a SingleCommand as child1. Then buildTree() will be called again and saved as child2. This will allow the tree to build itself only using one function recursively. Note: Subclasses of CommandComposite all redefine the execute() function.

*Always (composite class, inherits from CommandComposite):*
An object of this particular class is created when a semicolon is found, this class differs from the other subclasses of CommandComposite by always running execute on both of it's children.

*And (composite class, inherits from CommandComposite):*
An object of this particular class is created when && is found, this class differs from the other subclasses of CommandComposite by running execute() on its first child, and either passing information about the status of the first child's execute() on to the second child, or completely skipping the execute() on the second class, depending on the exact meaning of && and ||. For example, if we have Cmd1 || Cmd2 && Cmd3, does Cmd3 run if Cmd1 passes? We would all agree that this would cause Cmd2 not to run, but then would Cmd3 run because the most recently run command passed? What about Cmd1 || Cmd2 ; Cmd3?

*Or (composite class, inherits from CommandComposite):*
An object of this particular class is created when || is found, this class differs from the other subclasses of CommandComposite by running execute() on its first child, then determining what to do next by checking the pid code from child1. (see above for specifics, currently we have a known issue that will be resolved by getting some clarification on the specifications for assignment 2)

*Comment (composite class, inherits from CommandComposite):*
An object of this particular class is created when # is found, this class differs from the other subclasses of CommandComposite by running execute() only on its first child no matter what. The second child may be left empty, or the implementation of buildTree() may force us to create a dummy second child (new primitive class?). We don't think it will be a problem right now, but we are prepared to fix it if such a problem comes up.

*Exct (DOES NOT inherit from anything):*
This class will be used to execute individual commands from the CommandComponent tree. This particular class only contains a string and exists mostly for the purpose of expansion in the future. For example if more custom commands not found in bin are required (other than just exit), this class will become very useful because each one can just get its own class.

*shExct (inherits from Exct):*
This class is specifically for commands described in assignment 2 (the ones found in bin that we need to use execvp() for). It has a function execute() that will create a fork and run the command then return.

*exitExct (inherits from Exct):*
This class is specifically for the exit command that we are supposed to hard code into our shell. When SingleCommand creates a new exct object we will make it go to slightly different classes depending on what type of command is being executed (this is mostly for expandability, and isn't really needed right now). The exit command will quit the rshell process in some way (are we allowed to use system(exit) or something like that?).

**Coding Strategy:**
We will divide the epic into smaller user stories and issues and then take turns picking user stories to work on, slowly working our way through until everything is done. We are not very worried about one person doing too much or not enough of the work because we are working pretty well as a team so far.

**Roadblocks:**

- When the requirements for the program change (because the next assignment is released), we may have to change large parts of our program if we don't make room for different functionality to be added.
- Some parts of the assignment are confusing because I've never written a program like this before
- Creating different types of Exct class from SingleCommand might be annoying and turn out to be more trouble than it it worth
- We may need to create another primitive class for comments just because of the implementation of the buildTree() function