COMPUTER SCIENCE 131A
OPERATING SYSTEMS

PROGRAMMING ASSIGNMENT 1
UNIX SHELL

Description

You will write a simplified version of a Unix shell, a command-line interpreter. A *shell* is a user-level program that allows users to interact with a computer's operating system via a set of commands. As you have seen in your hands-on assignment, these commands allow to perform file system operations such as read and write files and directories, to search file content, and also to combine commands into larger compound commands, in a convenient way.

There are two things you will have to do to write this simple shell:

1. Implement the command line interface that parses user input and executes commands.
2. Use the *pipes and filters design pattern* to represent an abstract command, and use subclassing to implement the commands in an object-oriented way.

Command Line Interface

The command line interface will be a read-eval-print-loop (REPL). The shell must support the following commands typed into the REPL:

Command	**Description**
`pwd`	Returns current working directory.
`ls`	Lists the files in the current working directory
`cd <dir>`	Changes current working directory to `<dir>`.
`cat <file>`	Writes `<file>`'s contents into pipe or stdout.

`head/tail`	Returns up to the first/last 10 lines from piped input.
`grep <query>`	Returns all lines from piped input that contain `<query>`.
`wc`	Counts the number of lines, words, and characters in the piped input.
`uniq`	Returns all lines from piped input that are not the same as any previous line.
`> <file>`	Redirects output of preceding command, writes it to file <file>
`Exit`	Prints "goodbye" and terminates the REPL.

Terminology and Definitions:

Parameters are the arguments specified within angle brackets <> to the right of the subcommands. All parameters in the above shell commands are required (no optional parameters).

Compound Commands:

A *pipeline* is a set of processes chained together so that the output of each process is passed into the input of the next. In the Unix shell, users can create ***compound commands*** expressed as a pipeline, which is a convenient abstraction used for the purpose of evaluation. These pipelines, defined from left to right are specified using the **pipe** operator **|** (referred to as pipe, for short). For example, in the compound command:

```
command1 | command2 | command3
```

the first pipe operator causes the output of command1 to become the input of command2, the second pipe operator causes output of command2 to become input of command 3. To avoid ambiguity, when a command is part of a compound command, we refer to it as a *subcommand.*

Not all commands can be piped or take parameters. For example, *cat* requires the <file> parameter, but it does not take any piped input. It does produce output. In contrast, > takes piped input, no parameters, and produces output.

Implementation Details

Pipes and Filters Pattern

In order to evaluate a compound command specified as a pipeline using pipe operators, your shell program will first separate the command into subcommands represented as `Filters`. Next, the `Filters` must be connected with input and output `Pipes`. `Pipes` allow `Filters` to pass lines of data between them. When the filters are executed, they take their input from their input `Pipe` (if one exists) and write their output to their output `Pipe` (if one exists). Every pipeline must either end in a `RedirectFilter` or, if the pipeline command does not end in redirection, it should print its output to stdout (i.e. the Eclipse console).

Subcommand Categories

The first category is commands that **cannot have piped input.** These subcommands can only appear at the beginning of a compound command:

```
pwd, ls, cat, cd
```

Next are commands that **must have piped input.** Unlike the previous subcommands these can not appear at the beginning of a compound command, and must be piped into:

```
head, tail, grep, wc, uniq, >
```

Finally, `cd` and > also **cannot have piped output.** They can only appear at the end of a compound command.

```
cd, >
```

Read Eval Print Loop

The REPL runs in the Eclipse Console. The user can terminate the REPL by typing the command “exit”. The shell will exist in a particular directory in your file system. This is known as the current working directory. The shell starts in your Eclipse project directory.

Subcommand Details

The “current working directory” of your Shell is the directory from which you start the Shell. That is, current working directory should be initialized as the current working directory that your Java program is initialized to. In Java, you can retrieve the current working directory using *System.getProperty("user.dir").*
The subcommand cd allows to modify the current working directory of the Shell. Implementing this will be a little more challenging than you might think because you are not allowed to modify the invoking environment's working directory. Your implementation will need to manage a separate working directory for your shell in the static field currentWorkingDirectory defined in SequentialREPL.java. Note, you should initialize this field inside main().
The cd subcommand must support the special directories “.” (the current directory) and “..” (the parent directory). It must allow for nested directories. For example `cd dir1/dir2/dir3.`

The `wc` subcommand counts the number of lines, words and characters in the piped input, and outputs them as a string separated by spaces. A word is a sequence of characters not separated by a space. For example, an input with two lines
`“I am”`
`“input.”`
has 3 words `“I”`, `“am”`, and `“input.”`, 2 lines, and 10 characters. This should be output as `“2 3 10”`.

Error Handling

The shell should not crash when an error is encountered, and instead display an appropriate error message. Each error message can be found in the `Message` enum and must be called from there. The shell must detect the following errors and display the appropriate message:

Message	**Trigger**
`COMMAND_NOT_FOUND`	Executing an unlisted command.
`FILE_NOT_FOUND`	Reading from a file that does not exist
`DIRECTORY_NOT_FOUND`	Changing current directory to one that does not exist
`REQUIRES_PARAMETER`	Failing to pass a parameter to a command that requires one
`REQUIRES_INPUT`	Failing to pipe into a command that requires input
`CANNOT_HAVE_INPUT`	Piping into a command that does not take input
`CANNOT_HAVE_OUTPUT`	Piping from a command that does produce output

Note: You may assume that your commands will never be provided extra (too many) parameters. That is, a command that receives no parameters (like `pwd`) will not be provided with any and a command that receives 1 parameter (like `grep`) will not be provided with more than one.

For this reason, you can parse a command with a parameter as: [command] [space] [parameter].

Design Strategy

SequentialCommandBuilder

The skeleton of this class is provided to you. You are not required to implement this class, but it is recommended you do. This class has five methods which provide an outline of how you could go about converting a command `String` into a `List` of linked `SequentialFilter`s:

createFiltersFromCommand: String → List<SequentialFilter>

The goal of this method would be to take a `String` command and produce a `List` of `SequentialFilters`.
Hint: Your REPL should use this method to evaluate commands.

constructFilterFromSubCommand: String → SequentialFilter

The goal of this method would be to take a `String` subcommand and return an instance of the appropriate `SequentialFilter`. For example, "`cat file.txt`" would return an instance of `CatFilter`.

Hint: You can use the constructors of your `Filters` to your advantage.

linkFilters: List<SequentialFilter> → boolean

The goal of this method would be to take a `List` of `SequentialFilter` objects corresponding to subcommands and link them together. To understand what linking means, make sure to read `SequentialFilter`.

Hint: Some errors can be caught in `constructFilterFromSubCommand` while others can be caught in this method. Correctly deciding which errors should be caught at this stage will lead to an elegant solution.

Note on Exceptions

As noted above, your REPL cannot crash if the user types in an incorrect command. However, you can throw an exception in one class and catch it in another, allowing your REPL to continue. Throwing an exception in one class A and catching it in class B provides an easy way to pass a message from an instance of A to an instance of B.

Implementing Error Handling

One could try to detect every type of user error in `SequentialCommandBuilder` and/or `SequentialREPL`. However, this will likely involve detailed code addressing each command separately for both syntax and piping errors. A syntax error is an error unrelated to piping, say something like failing to pass in a required parameter or passing in a nonexistent file. Exceptions would allow to localize the detection of these types of errors in the `Filter`s which could throw an exception to the class that creates them. The creating class would then catch the exception and display the appropriate error message.

Implementation constraints and Grading

(I) We provide you with the test files below that contain tests to examine your implementation of the various commands. You are not allowed to modify these files (besides for debugging purposes).

```
AllSequentialTests.java, RedirectionTests.java, REPLTests.java,
TextProcessingTests.java, WorkingDirectoryTests.java
```

There are no other hidden unit tests that will be used for grading this assignment. You should use these tests to make sure you have the correct behavior and are printing the appropriate `String`s for each of the error cases. For example, you will see that the tests make the assumption mentioned in the note on Error Handling (no extra parameters).

(II) You cannot modify `SequentialFilter` and `Filter` files.

(III) Make sure you import and export the Eclipse project properly. You need to import through Import > Existing Projects into Workspace, otherwise you won't be able to refactor your project name correctly. The submitted project should have a structure that looks exactly like the skeleton. An improperly structured project will receive a 0.

(IV) The project needs to be renamed FirstnameLastnamePA1 (make sure to use Eclipse's Refactor > Rename). The zip file you submit should be named FirstnameLastnamePA1.zip.

Notes on Tests

- To run the tests run `AllSequentialTests.java`. Do not run each of the other test files individually.
- The files will create files to pass into your shell and your shell will output files as part of the tests. To save all the files that are created over the course of the tests, set `AllSequentialTests.DEBUGGING_MODE` to `true`. Your code should still work if it is set to `false`.

- You may see `AllSequentialTests.java` throw an `Exception` when you have `AllSequentialTests.DEBUGGING_MODE` set to `false`. This is most likely due to the fact that you have left a Java reader/writer object open on a test data file which did not allow the tests to properly delete the file. To fix this issue, check that you open and close your reader/writer objects properly in the related `Filters`.

Remarks

This is an individual assignment. Any sign of collaboration will result in a 0 score for the assignment. Please check the academic integrity policies in the syllabus of the course.

Late policy: As explained in class, this project has a special policy, no late submission is allowed.