CS4218 Milestone 1 Report

Team_15
BF+EF2

CAO SHENGZE
KIM HYUNG JON
YAP HAN CHIANG

Plans for implementation and testing

Implementation Plan

- **main**
  - The role of the main function is to read inputs, process each input line to obtain an array of string, and pass each string in the array to the **parseAndEvaluate** function.
  - Notice that the **semicolon** operator is evaluated inside the main `function`. Therefore, the test for the semicolon operator need to relay on the static function instead of using the parseAndEvaluate.
  - It should also be noted that in our implementation the String[] **args array does not include the command** name. (e.g. command “sort -n file.txt” is passed to applications as {“-n”, “file.txt”})

- **parseAndEvaluate**
  - Here, each string is processed for pipes and transformed into an array of string, and each string is passed to **CallCommand**.
  - parseAndEvaluate is the function mostly used in test cases. The behavior of the shell can be performed with the parseAndEvaluate function.
  - The parseAndEvaluate function is also responsible for the processing of the pipeline.

- **CallCommand** helper class
  - We have a **CallCommand** helper class to parse each line into their individual components, which are the application name, the arguments, input file redirection and output file redirection.
  - CallCommand also do the processing of all three kinds of Quotes: The double quotes, the single quotes as well as the back quotes, which is the **command substitution**.
  - CallCommands will call the runApp function of ShellImpl to let the shell implement the parsed command.

- **CatApplication**
  - Can use both stdin and the file reader to read the file.
  - Does not handle multiple files, as of milestone 1

- **CdApplication**
  - Changes the global variable Environment.currentDirectory

    - Also handles special cases such as going to parent directory with command “cd ..”
    - Does not deal with going to home directory (“cd”)
    - Does not print or write to stdout any output: only chages the current directory setting

- **EchoApplication**
    - Can read from stdin only. Cannot read from files directly.

- **HeadApplication**
    - Similar to cat.
    - Only read the first several rows found in the file/stdin. Ignore the rest content.
    - Empty lines are still counted as lines.
    - Does not handle multiple files as of milestone 1

- **TailApplication**
    - Similar to head.
    - Count the number of lines of the input first, then decide which lines to print and which lines not.
    - Does not handle multiple files as of milestone 1

- **SedApplication**
    - We have not changed the interfaces in this function. So all the interface methods are remaining the same. The command would be parsed twice: The first time run function would decide which certain api should be used (read from stdin or file, all or the first one). The second time of parsing is implemented inside each certain interface function and generate the result from the interface function.

- **WcApplication**
    - **Interface changed:** original requirement is that each interface is given the whole command arguments as single string. As we were unable to make this work within first milestone, we temporarily implemented an alternative for milestone 1. In current implementation, the interfaces are invoked with the contents of the file or stdin as parameters. Current tests are also written following this structure.
    - Handles multiple files in milestone 1. One line is printed as output for each file in arguments. For valid files, the output line format is “[chars] [words] [lines] filename”. For invalid files, the output line format is “wc: filename: No such file”.

    - Throws exception for invalid options. Does not throw exceptions for invalid files: as stated above, the error message should be part of ordinary output.
    - Spaces(\s), newlines(\n) and carriage returns(\r) are all counted as characters for option -m. Lines that are empty or contain only whitespaces are counted as lines for option -l. Spaces and empty lines are not counted as words for option -w. This is following the logic of wc in Linux.

- **DateApplication**
    - Invokes the java.util.Date class and prints it out using SimpleDateFormat class with the regex "EEE MMM dd hh:mm:ss z yyyy", allowing the date to be printed in format given in requirements.

- **Semicolon Functionalities:**
    - For semicolon, what we do is before we parse the command line, we look for unquoted semicolon. If found ,then the command would be split to different commands and executed one by one.
    - This is done before the parseAndEvaluate function. All the commands get from processing the semicolon would then be passed to parseAndEvaluate one by one. We want to make sure the parseAndEvaluate would only focus on single command.

- **Quotes Functionalities:**
    - Similar to Semicolon, first look for all special symbols, then for the ones not quoted by other quotes(for example, back quotes would not work when it is inside a pair of single quotes.

- **Pipe Functionalities:**
    - Before passing the commands to the CallCommand function, the parseAndEvaluate functions would detect the valid | and then get a list of commands. With the help of pipeStream, we can easily use the output from one command become the input to the next command.

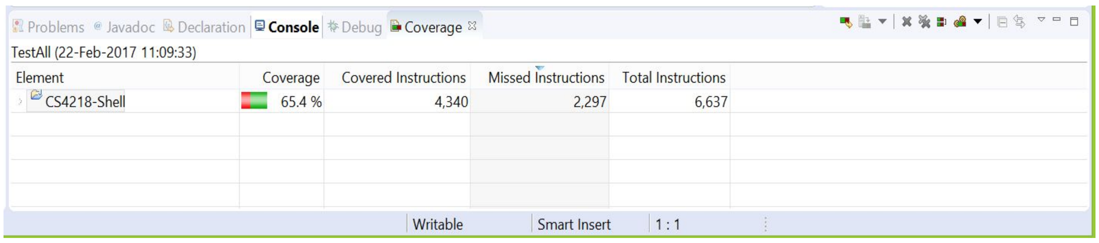
- **Error Handling:**
    - The errors would be handled together under the ShellImpl part. In the application layer, they throw the customized exception which extended the ApplicationException.

Testing Plan & Summary of test cases

- **Structural Testing**
    - When designing the test cases, we applied the structural testing. More specifically, we try to look at the code and write the test cases try to cover all statements. For example, when I write test cases of the function of call a command, I would come to take a look at my code where such a statement found:

```
public static void runApp(String app, String[] argsArray,
                InputStream inputStream, OutputStream outputStream)
                throws AbstractApplicationException, ShellException {
        Application absApp = null;
        if (("cat").equals(app)) {// cat [FILE]...
                absApp = new CatApplication();
        } else if (("echo").equals(app)) {// echo [args]...
                absApp = new EchoApplication();
        } else if (("head").equals(app)) {// head [OPTIONS] [FILE]
                absApp = new HeadApplication();
        } else if (("tail").equals(app)) {// tail [OPTIONS] [FILE]
                absApp = new TailApplication();
        } else if (("cd").equals(app)) {
                absApp = new CdApplication();
        } else if (("pwd").equals(app)) {
                absApp = new PwdApplication();
        } else if (("date").equals(app)) {
                absApp = new DateApplication();
        } else if (("wc").equals(app)) {
                absApp = new WcApplication();
        } else if (("sed").equals(app)) {
                absApp = new SedApplication();
        }else { // invalid command
                throw new ShellException(app + ": " + EXP_INVALID_APP);
        }
        absApp.run(argsArray, inputStream, outputStream);
}
```

Then, in order to cover all the statements here, I need to design test cases for app equal to cat, echo, head, cd, etc. I also need to have one more test case for app not equal to any of the above words to see whether the throw exception operation is correctly executed or not.



TestAll (22-Feb-2017 11:09:33)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
CS4218-Shell	65.4 %	4,340	2,297	6,637

Besides manually reading through the code, we can also try to use some test case analysis tool to help us take a look whether we already cover all the statements. If no, then we can design the specific test case to cover the uncovered part.

- **Functional Testing**

  - Beside using structural testing to try to cover all instructions, we also implement test cases based on functional testing. For functional testing, we are using a hierarchical structure: Start from the unit test and end by a overall test. For example, for the application sed, we first test the helper function used in the apis as unit test. Then we generate unit test for the apis. Then we test by the interface of application: the run function. After finishing all of the above test, we then have test cases that call the application by the **parseAndEvaluate** function from ShellImpl.

  - Manual testing: a.k.a. random testing. In order to first understand the required behavior of Shell and the Applications and to ensure that our implementation is correct, we ran many random testing with Linux shell by preparing several test cases, running them under Linux shell, then using those expected values to build systematic test cases. Differences between actual Linux shell behavior and our project requirements were observed and applied when creating the test cases.

  - Systematic testing: The Shell and each Application are extensively tested with JUnit tests, with as many corner cases as possible. Some of the procedures followed while generating test cases are:
    - Testing with empty files, files containing only newlines and containing only whitespaces. This is to test corner cases for Applications for which counting the lines is important, such as HeadApplication and WcApplication.
    - Testing with invalid files

        - Multiple invalid files. (As of milestone 1, wc can handle multile files.)
        - Invalid files mixed with valid files - wc should still process the valid files, and for invalid files print error message as output. The outputs for the files should be tested to be printed the same order that the files appear in the arguments.
        - Place the valid and invalid files in different order. wc should be able to process the valid files and output error for invalid files regardless of order.
    - Testing with invalid options, for Applications that provide options.
        - Invalid options mixed with valid options - wc should throw exception, with indication of which option caused the exception.
        - Place the valid and invalid options in different order. wc should throw exception at first invalid option encountered. e.g. “wc -lxm file.txt” should throw exception “wc: invalid option -- x”.
    - Testing with both invalid files and options
        - Testing with valid files and options mixed with invalid files and options, with as much variation in the permutation as possible.
    - Testing with valid but complex cases
    - Case sensitivity: for grep and sort (not implemented in milestone 1).

- **Pair-wise Testing**

  ○ This is a more detailed testing method. When generating test cases, as there are many different applications and different functionalities of the shell, sometimes I need to pair these two different features and see whether it works. For example, when writing test cases for sed, after testing all the apis, we would choose to generate some test cases like

  sed with pipe: cat test.txt | sed s/[1]/one/

  sed with semicolon: cat test.txt | sed s/[1]/one/ ; cat test.txt | sed s/[1]/one/g

  sed with quotes: sed s/[1]/”one”/

  This would not only help sed, but also test the paired functionalities of the shell.

- **System Testing**

○ Besides all the methods above, we also performed system testing. We just use our program and play it with all kinds of test cases. This test case may not be strictly designed. It is just more random. Once a test case fail, then we would add it to the relative test files as a new test case.