Implementation Plan

- **Main**
    - The role of main is to accept command strings and pass them to **parseAndEvaluate**

- **parseAndEvaluate**
    - We uses composition and treat **CallCommand** as a blackbox. We then create **PipeCommand** to wrap around it. This is beneficial to our testing since **CallCommand** has little to no modification, therefore minimize the regression failures in our development.
    - Each command string is passed to **PipeCommand** for processing the pipeline before **PipeCommand** calls **CallCommand** to evaluate commands
    - The semicolon operator will eventually be handled in this method

- **LsApplication**
    - Implementation difference: While the Linux shell does not support both -d and -R options, both options can be specified and directories will list folders only recursively
    - Writes to stdout or file if redirected
    - Throws exception when path is not a directory path

- **CatApplication**
    - Reads arguments, but if there are no arguments, read from stdin
    - Execution continues until all arguments have been evaluated even if there is exception thrown from being unable to read a file
    - Does not print a new line if the file does not end with new line

- **EchoApplication**
    - Reads arguments from the console and writes to console
    - Does not read from stdin if redirected
    - Writes to stdout or file if redirected
    - Throws exception when unable to write to stdout, stdout is null or argument is null

- **ExitApplication**
    - Exit Application is implemented and will simply terminate the shell using System.exit
    - No exception is expected to be thrown

- **MkdirApplication**
    - mkdir does not print messages unless an exception is caught
    - Parent folders will be created if they do not exist
    - Ignores stdin and stdout and can handle multiple paths
    - Throws exception when no folders are specified or when path is not a directory path

- **GrepApplication**
    - Grep works with regex, single/multiple files, and piping
    - Tests also includes in context single/double quoting, as well as command substitution with backtick.

- **PasteApplication**
    - Ignores stdin if at least one file is specified

    - Can handle multiple files
    - Writes to stdout or file if redirected
    - Throws exception when stdout is null or stdin is null when there are no arguments
    - Throws exception when arguments cannot be resolved to a files or when unable to merge files

- **DiffApplication**
    - Must have two arguments for file name, otherwise throw exception
    - Options should appear before file names
    - Stdin is represented as '-' for file name
    - Throws exception when stdin is null when '-' file name is in argument

- **PipeCommand**
    - Splits the command string into separate, sequential commands delimited by pipe operators ('|'), commands are then executed sequentially by calling **CallCommand** to parse and evaluate each command
    - This procedure is extracted through a parse stack method in **PipeCommand**. This process is unknown to **CallCommand**, therefore we can separate both unit tests.

- **CallCommand**
    - Parses the command and instantiates the appropriate application that will be run
    - Calls runApp method of ShellImpl to execute the instantiated application
    - **CallCommand** has the responsibility of resolving globing.

- **IO-Redirection**
    - IO-Redirection was implemented in **CallCommand**, so testing of IO-Redirection is done through **CallCommand**
    - IO-Redirection interfaces from the shell have been removed as they are unused

- **Quoting**
    - Quoting is handled by both **PipeCommand** and **CallCommand** in different layers. This is due to the nesting nature of multiple commands.

- **Globing**
    - Works on Linux, but not Windows due to OS-dependent format
    - Can handle deep recursive syntax [double asterisk](**) and wildcard [single asterisk](*)

- **Exception Handling**
    - Exceptions from applications are thrown to ShellImpl through the **AbstractApplicationException** before displaying the error message in the Shell

Workflow

1. Before starting to work on a task, create a branch preferably in the format <type>/<task> e.g feature/cat
2. Create an issue on github to notify everyone that you are working on that task
3. Once the task has been completed together with test cases, submit a pull request
4. Once Travis-CI passes successfully, you may merge the pull request. Otherwise, go back to 3
5. Repeat from 1

Our Travis-CI link: https://travis-ci.com/zavfel/CS4218_team20_2018

Testing Plan and Summary of Test Cases

- **Functional Testing**
  - Blackbox: Done through Linux shell to understand the behaviour of applications for generating test cases

  - Requirements: Interpret project description of command specifications to create test cases that handle the basic requirements, especially those where examples are given

- **Systematic Testing**
  - Test cases for methods that have a non-void return type should test for
    - positive test cases where returned values are compared to expected results
    - negative test cases where an exception is thrown

  - Test cases for methods that have a void return type should include test cases where an exception is thrown

  - Some aspects of Category-partitioning can be used to identify the parameters that affect the expected output
    - e.g. stdin, stdout, arguments

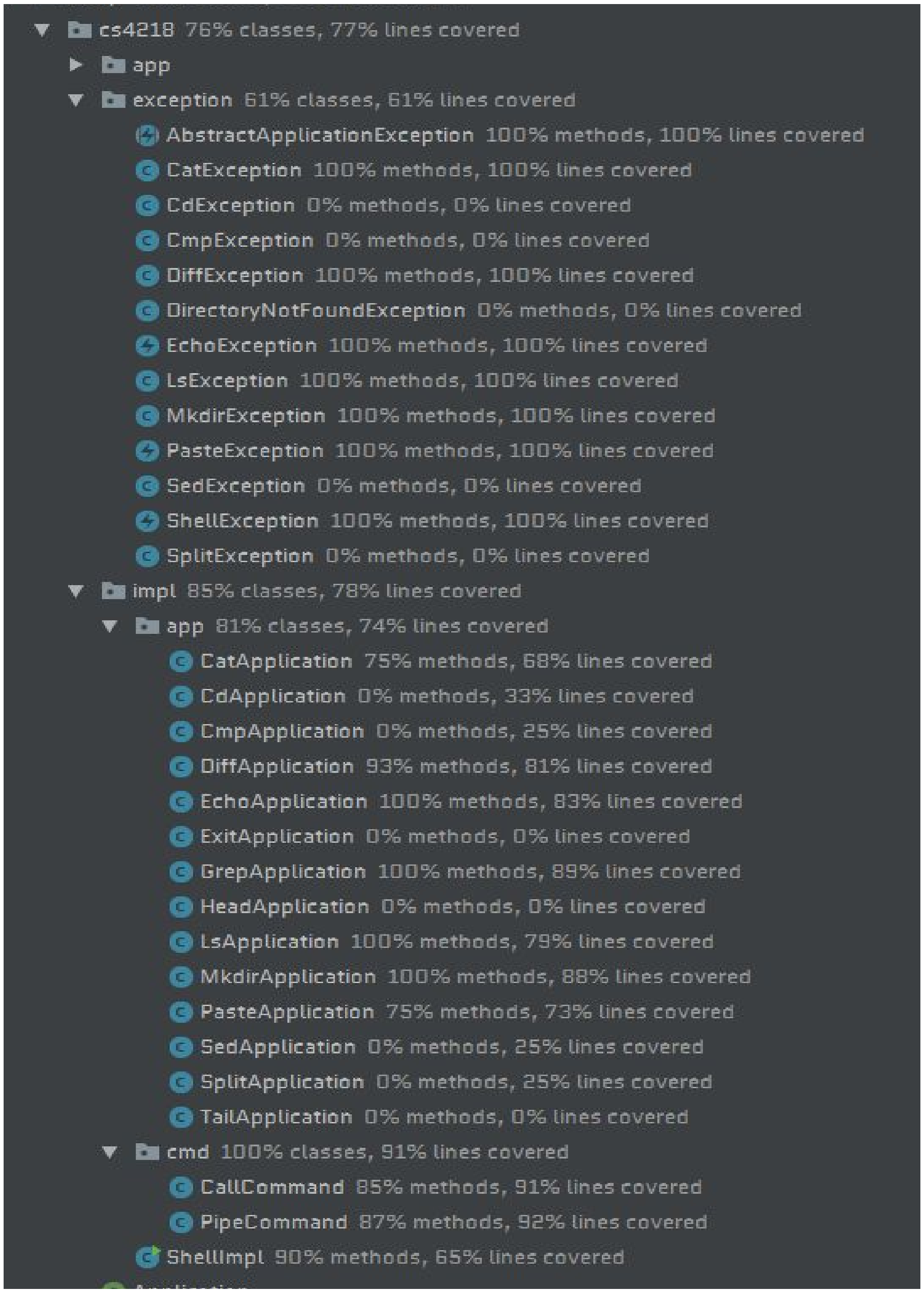
  - Test cases for relevant and boundary values can be generated where applicable from Category-partitioning such as
    - arguments array: null, not null
    - arguments size: zero, one, many
    - argument: valid, invalid
    - index: 0, 1, -1 like in the case of sed replacementIndex

  - Try to apply MC/DC in test case generation, especially for combination of options for application with option flags such as diff, cmp, sed
    - Repeated option flags
    - Invalid and valid option flags together
    - Relevant and boundary values from category-partitioning

- **Coverage**
  - EclEmma plugin is used to give us an idea of how much code coverage our test cases provide
  - The overall coverage statistics are:

Class	Method	Line
76%	72%	77%

These statistic also includes unimplemented application which is out of our project scope (e.g: sed, cmp, split). The actual percentage should be higher if we exclude them.


```
▼ cs4218 76% classes, 77% lines covered
    ▶ app
    ▼ exception 61% classes, 61% lines covered
        AbstractApplicationException 100% methods, 100% lines covered
        CatException 100% methods, 100% lines covered
        CdException 0% methods, 0% lines covered
        CmpException 0% methods, 0% lines covered
        DiffException 100% methods, 100% lines covered
        DirectoryNotFoundException 0% methods, 0% lines covered
        EchoException 100% methods, 100% lines covered
        LsException 100% methods, 100% lines covered
        MkdirException 100% methods, 100% lines covered
        PasteException 100% methods, 100% lines covered
        SedException 0% methods, 0% lines covered
        ShellException 100% methods, 100% lines covered
        SplitException 0% methods, 0% lines covered
    ▼ impl 85% classes, 78% lines covered
        ▼ app 81% classes, 74% lines covered
            CatApplication 75% methods, 68% lines covered
            CdApplication 0% methods, 33% lines covered
            CmpApplication 0% methods, 25% lines covered
            DiffApplication 93% methods, 81% lines covered
            EchoApplication 100% methods, 83% lines covered
            ExitApplication 0% methods, 0% lines covered
            GrepApplication 100% methods, 89% lines covered
            HeadApplication 0% methods, 0% lines covered
            LsApplication 100% methods, 79% lines covered
            MkdirApplication 100% methods, 88% lines covered
            PasteApplication 75% methods, 73% lines covered
            SedApplication 0% methods, 25% lines covered
            SplitApplication 0% methods, 25% lines covered
            TailApplication 0% methods, 0% lines covered
        ▼ cmd 100% classes, 91% lines covered
            CallCommand 85% methods, 91% lines covered
            PipeCommand 87% methods, 92% lines covered
        ShellImpl 90% methods, 65% lines covered
```

- **OS-dependent tests / Unimplemented functionality**
    - Due to differences between Windows and Linux OS such as file naming restrictions, some OS-dependent test cases may be skipped using JUnit Assume.assumeTrue
    - The same applies to unimplemented functionalities such as EF2