Version 31 January 2024

CS4218 – Software Testing

Project Description

1. Introduction

CS4218 covers the concepts and practices of software testing and debugging. An important portion of CS4218 is the project work. Through this project students will learn and apply testing and debugging techniques followed by quality assessment activities. Teams of students will implement a shell and test its functionality using different testing techniques that will be taught during the course.

Students will learn how to professionally apply testing approaches and techniques with the state of art testing automation framework JUnit. Students will be shown good and poor styles of manual unit-test generation. Students will evaluate the quality and thoroughness of the test cases and project code using different coverage metrics. They will apply testing, debugging and other quality assurance activities in a simulated industrial setting.

2. Project Teams

Students should form teams of 4 or 5 at the beginning of the semester. Once formed, teams would be final for the duration of the course. You are required to create a repository in Github Classroom for your team.

3. Coding

All programming assignments must be completed using JAVA. External libraries/plugins should not be used. The implementation should not rely on network communication, databases. You should **not** rely on any platform-specific functionality. This means that all specifications should work on any platform. Use Java properties “file.separator” and “path.separator” when working with file system. Use “line.separator” property when working with newlines, since different operating system may use different separator.

Your code must conform to CS4218 code conventions. These conventions are checked automatically using PMD tool (see installation instructions in Canvas). Your methods should not be too long (**more than 50 lines of code**). You must have proper Javadoc comments for every method. Follow the Java naming convention. Use naming convention for test classes (*Test.java for unit tests and *IT.java for integration tests). For example, when unit testing a “Foo” (without quotes) class, the test class should be called “FooTest” (without quotes). For test method, in this module we will use the convention **methodUnderTest_Scenario_ExpectedBehaviour**. In other words, indicate the method under test, explain the scenario that you are testing followed by the expected behaviour (e.g., for a test that check for negative value for method foo and a Boolean value true to be returned is expected, the method name would be foo_NegativeValue_ReturnsTrue).

Use the following versions of software:

- JDK 11 or higher
- IntelliJ IDEA
- JUnit 5

4. Assessment

Students can get a maximum of 34 marks for the project, divided as follows:

- 3 marks: Shell and Application Implementation
- 5 marks: Unit Tests
- 5 marks: Test-driven Development
- 5 marks: Integration Tests
- 2 marks: Code Quality
- 6 marks: Hackathon Bugs
- 1 mark: Rebuttal
- 1 mark: Bug Fixes
- 6 marks: Quality Assurance (QA) Report

Please see Submission Instructions section for details.

5. Project Timeline

Week	Lab	Deadline
1&2	No Lab.	
3	Project introduction, Version control systems, PMD tool	
4	Unit testing	
5	Test driven development	
6	Debugging	
Recess		
7	Integration testing Code coverage	Mon, 4 Mar, 2pm: **Milestone 1**
8	Automated testing tools	
9	Complete project development (no content presented)	
10	No class - PH Good Friday Work on Hackathon	Mon, 25 Mar, 2pm: **Milestone 2**
11	Rebuttal – clarification of bug/feature for the bug reports found for your code. Quality Assurance report and its structure	Mon, 1 Apr, 2pm: **Hackathon** Sun, 7 Apr, 2pm: **Rebuttal**
12	No class - PH Hari Raya Haji Work on the QA report	
13	GUI Testing, Testing Distributed Systems	Fri, 19 Apr, 2pm: **Milestone 3**

6. Background

A shell is a command interpreter. Its responsibility is to interpret commands that the user types and to run programs that the user specifies in her command lines. Figure 1 shows the relationship between the shell, the kernel, and various applications/utilities in a UNIX-like operating system:

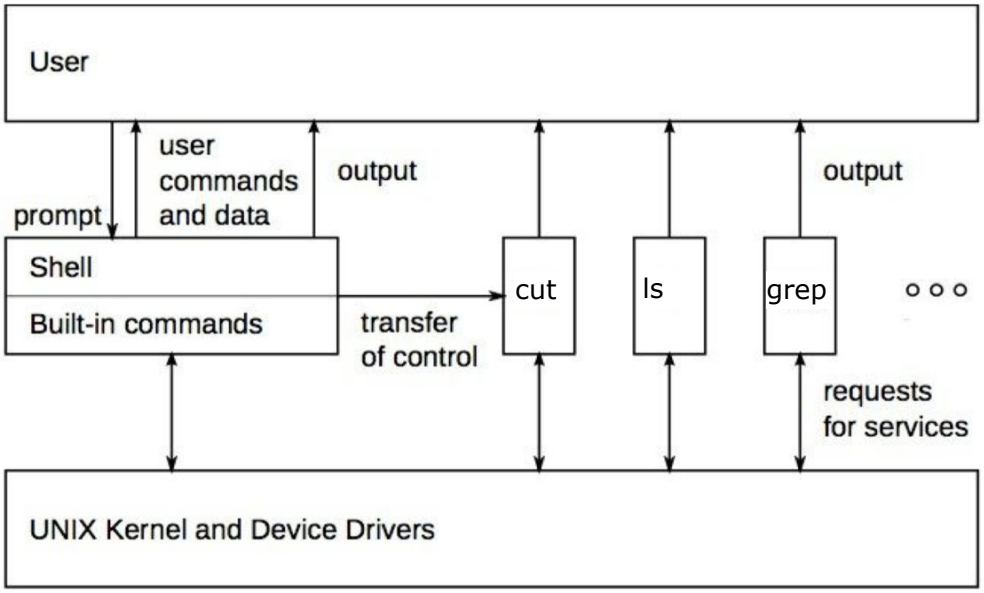

Figure 1: relationship between the shell, the kernel, and various applications

Shell can be thought of as a programming language for running applications. From the user's perspective, it performs the following loop:

1. Print prompt message.
2. Wait for user's command.
3. Parse and interpret user's command, run specified applications if any.
4. Print output.
5. Go to 2.

An application in a UNIX-like system can be roughly considered as a block with two inputs and three outputs, as shown in Figure 2. When an application is run, it reads an array of command line arguments and text data from its Standard Input stream (stdin). During execution, it writes output data to its Standard Output stream (stdout) and error information to its Standard Error stream (stderr). After execution, the application returns Exit Status, a number that indicates an execution error if it is non-zero.

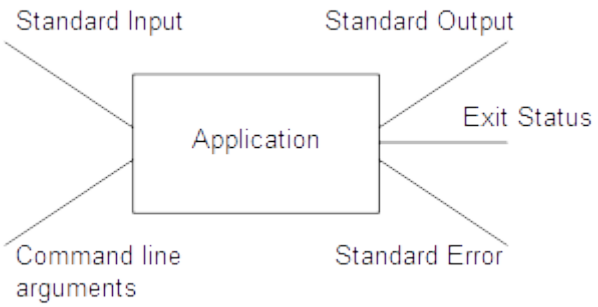

Figure 2

The important feature of shells in UNIX-like systems is the ability to compose complex commands from simple ones. For example, the following command combines the applications “paste” and “grep”:

```
paste articles/* | grep "Interesting String"
```

As shown in Figure 3, Shell expands “articles/*” into the list of all the files in the “articles” folder (directory) and passes them to “paste” as command line arguments. Then, “paste” merges the contents of all these files and passes the results to “grep” using the pipe operator “|”. “grep” finds and displays all the lines that include “Interesting String” as a substring. The connection between these two commands is made using the pipe operator “|” that connects the stdout of “paste” with the stdin of “grep”:

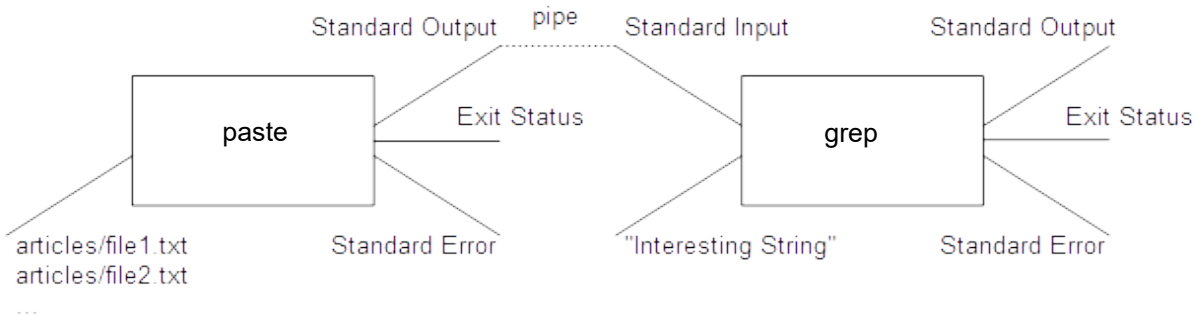

Figure 3

Further information about shells in UNIX-like systems can be found here:

- Explaining Bash syntax: http://explainshell.com/
- The Unix Shell: http://v4.software-carpentry.org/shell/index.html
- Advanced Bash-Scripting Guide: http://www.tldp.org/LDP/abs/html/
- Bash Hackers Wiki: http://wiki.bash-hackers.org

7. Requirements

The goal of the project is to implement and test a shell and a set of applications. The shell and the applications must be implemented in JAVA programming language. The required functionality is a subset (or simplification) of the functionality provided by UNIX-like systems. Particularly, the specification was designed in such a way that it maximally resembles the behaviour of Bash shell in GNU/Linux and Mac OS. However, there are several important distinctions:

1. JVM is used instead of OS Kernel/drivers to provide required services.
2. Shell and all applications are run inside the same process.
3. Applications raise exceptions when they fail during execution. The exception messages should be printed at stderr, if needed. The application should **return non-zero exit code in case of errors**, as shown in Figure 4. The execution of the shell should continue even after an application fails to execute. The error messages should be like the errors provided by bash, but deviations are acceptable if the clarity of the message is improved.

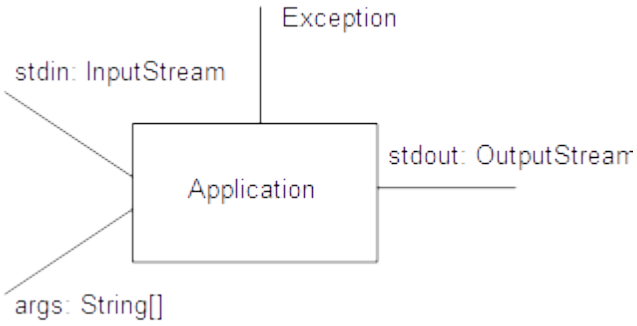

Figure 4

An application in the project implementation is a JAVA class that implements the Application interface and uses InputStream and OutputStream as stdin and stdout respectively. Main JAVA interfaces for applications are provided in a skeleton. Meanwhile, an almost complete implementation of shell is provided, so that students just need minimal work on implementation and pay more attention on the testing. On the other hand, some **bugs are injected** to the implementation of shell, and students should use testing techniques to find and fix those bugs. Specifically, students need to finish the following tasks:

- fix the existing bugs in the provided implementation of shell
- complete the implementation of shell
- implement missing applications
- test the implementation of shell and applications

Note that students need to use the **interfaces provided** and are **not allowed** to modify their definition. The interfaces will be later used in unit testing, so it is important to maintain compatibility among teams. aIt is allowed to add new interfaces, but students should ensure that the given interfaces work according to the requirements.

However, students are allowed to modify the **implementation we provide for the shell** (but not the interface definitions). Note that students are expected to test and fix bugs in our skeleton code assuming they are using the implementation provided.

Since this is a software testing class, students are expected to document all testing activities and submit a report about them at the end of the project (Milestone 3). **Students are advised to check before starting their project the QAReport format.**

The required functionality is split into three groups: Basic Functionality, Extended Functionality 1, and Extended Functionality 2.

The Basic Functionality (BF) includes:

- Shell: calling applications, quoting, command substitution
- Applications: echo, cd, wc, mkdir, sort, cat, exit

The Extended Functionality 1 (EF1) includes:

- Shell: pipe operator, globbing.
- Applications: ls, paste, uniq, mv

The Extended Functionality 2 (EF2) includes:

- Shell: IO-redirection, semicolon operator
- Applications: cut, rm, tee, grep

8. Shell Specification

Shell can be considered as a programming language where applications play the same role as functions in languages like C and JAVA. Shell parses user's command line to determine the applications to run and the data to pass to these applications. Our shell supports two ways to specify input data for applications: by supplying command line arguments and by redirecting output streams.

8.1. Command Line Parsing

User's command line can contain several subcommands. When Shell receives a command line, it does the following:

1. Parses the command line on the command level. Shell recognizes three kinds of commands: call command, sequence command (using semicolon operator), pipe command (using pipe operator).

Command line uses the following grammar:

```
<command> ::= <call> |  <seq> | <pipe>
<seq>  ::= <command> ";" <command>
<pipe> ::= <call> "|" <call> | <pipe> "|" <call>
```

Note: ";" is sequence operator, "|" is pipe operator, | is logic `or`, [text] is used to denote optional text. For example, `<pipe>` can be two `<call>` constructs connected by pipe OR another `<pipe>` construct connected by pipe with another `<call>`. (see documentation about regular expressions to help you understand this representation)
The non-terminals `<call>` and `<quoted>` are described below.

2. The recognized commands are evaluated in the proper order.

8.2. Call Command

Example

```
grep "Interesting String" < text1.txt > result.txt
```

Find all the lines of the file text1.txt that contain the string "Interesting String" as a substring and save them to the file result.txt.

Syntax

Shell splits call command into arguments and redirection operators.

```
<call> ::= <non-keyword>
  [ <whitespace> ] [ <redirection> <whitespace> ]* [<arguments>]
  [ <whitespace> <atom> ]* [ <whitespace> ]
<atom> ::= <redirection> | <argument>
<argument> ::= ( <quoted> | <unquoted> )
<arguments> ::= ( <quoted> | <unquoted> )+
<redirection> ::= "<" [ <whitespace> ] <argument> |
                  ">" [ <whitespace> ] <argument>
```

Note: `a*` is used to denote 0 or more occurrences of `a`, while `a+` is used to denote 1 or more occurrences of `a`.

A non-keyword character is any character except for newlines, single quotes, double quotes, backquotes, semicolons ";" and vertical bars "|".

Whitespace is one or several tabs or spaces. An unquoted part of an argument can include any characters except for whitespace characters, quotes, newlines, semicolons ";", vertical bar "|", less than "<" and greater than ">".

Semantics

A call command is evaluated in the following order:

1. Command substitution is performed (See section 8.8 on Command Substitution). Note that command substitution can be a call command.
2. The command is split into arguments and redirection operators. The command string is split into substring corresponding to the `<argument>` non-terminal. Note that one backquoted argument can produce several arguments after command substitution. All the quotes symbols that form `<quoted>` non-terminal are removed (see section on Quoting).
3. Filenames are expanded (see section on Globbing).
4. Application name is resolved.
5. Specified application is executed.

When Shell executes an application, it performs the following steps:

1. IO-redirection, if needed (see Section on IO Redirection).
2. Running. Run the specified application (the first `<argument>` without a redirection operator), supplying given command line arguments and redirection streams.

8.3. IO Redirection

Example

```
paste < file.txt
```
Display (merge) the content of file.txt.

```
cat < a1.txt a2.txt
```
Display the content of a2.txt. The redirection is ignored (“cat” application gets input from a2.txt)

Syntax

```
<redirection>::= "<" [ <whitespace> ] <argument> |
                 ">" [ <whitespace> ] <argument>
```

Semantics

- If no files are given, throw an exception.
- If multiple input redirections are used in a command (multiple “<”), only the last file is considered, and the preceding ones are ignored. Same applies for multiple output redirections used in a command (multiple “>”).

  `cat < a1.txt < a2.txt` – Display the content of a2.txt. The first redirection is ignored (“cat” application gets input from a2.txt)

- Open InputStream from the file for input redirection (the one following “<“ symbol).
- Open the OutputStream to the file for output redirection (the one following “>“ symbol).
- If the file specified for input redirection does not exist, throw an exception.
- If the file specified for output redirection does not exist, create it. No exception should be thrown.

8.4. Quoting

Example

To pass several arguments to an application, we can separate them by spaces:

```
echo hello world
```

In this example, "`echo`" gets two command line arguments: "hello" and "world". In order to pass "hello world" as a single argument, we can surround it by quotes, so that the interpretation of the space character as a separator symbol is disabled:

```
echo "hello world"
```

In this case, "`echo`" receives "hello world" as a single argument (without quotes).

Our shell supports three kinds of quotes:

- single quotes (')
- double quotes (")
- backquotes (`)

The first (') and the second ones (") are used to disable interpretation of all or some special characters, the last one (`) is used to make command substitution. Special characters are: \t (tab), * (globbing), ' (single quote), " (double quote), ` (backquote), | (pipe), < (input redirection), > (output redirection), ; (semicolon), space.

Syntax

```
<quoted>         ::= <single-quoted> | <double-quoted> |
                     <backquoted>
<single-quoted>  ::= "'" <non-newline and non-single-quote> "'"
<backquoted>     ::= "`" <non-newline and non-backquote> "`"
<double-quoted>  ::= ""(<backquoted> | <double-quote-content>)*""
```

where `<double-quote-content>` can contain any character except for newlines, double quotes and backquotes.

Single quote disables the interpretation of all special symbols. For example:
`$echo 'Travel time Singapore -> Paris is 13h and 15`'`
would output: `Travel time Singapore -> Paris is 13h and 15`

Double quote disables the interpretation of all special symbols, except for ` (backquote). For example, in the following command:

```
$echo "This is space:`echo " "`."
```

the outer "`echo`" receives one argument rather than two and outputs:
`This is space: .`
The same example, using single quote:

```
$echo 'This is space:`echo " "`.'
```

would output: `This is space:`echo " "`.`

```
$echo "This is space:' '."
```

would output: `This is space:' '.`

```
$echo "'This is space `echo " "`'"
```

The single quote is disabled by double quote, so that the backquote will not be disabled. This command would output: `'This is space  '`

```
$echo '"This is space `echo " "`"'
```

The single quote disables double quote and backquote. It would output: `"This is space `echo " "`"`
Note that we do not use character escaping (\) in our shell.

8.5. Pipe Operator

Example

```
$ paste articles/text1.txt | grep "Interesting String"
```

Find all the line of the file articles/text1.txt that contain “Interesting String” as a substring.

Syntax

```
<pipe> ::= <call> "|" <call> | <pipe> "|" <call>
```

Semantics

Pipe is a left-associative operator that can be used to bind a set of call commands into a chain. Each pipe operator binds the output of the left part to the input of the right part, then evaluates these parts concurrently. If an exception occurred in any of these parts, **the exception is thrown, and the rest of the parts are terminated.**

An example of a scenario with exception

```
$ lsa | echo hello world
```

Where `lsa` is an invalid application.

The expected output:

```
shell: lsa: Invalid app
```

8.6. Globbing

Example

```
ls articles/*
```

Display the names of all the files in the `articles` folder (directory).

```
ls x*
```

Display all files that start with x in the current folder (directory).

```
ls code/article*
```

Display all files under the `code` folder (directory) that starts with `article`.

Syntax

The symbol * (asterisk) in an unquoted part of an argument is interpreted as globbing.

Semantics

For each argument ARG in a shell command that contains unquoted * (asterisk) do the following:

1. Collect all the paths to existing files and directories such that these paths can be obtained by replacing all the unquoted asterisk symbols in ARG by some (**possibly empty**) sequences of non-slash characters.
2. If there are no such paths, leave ARG without changes.
3. If there are such paths, replace ARG with a list of these path separated by spaces.

Note that globbing (filenames expansion) is performed after argument splitting. However, globbing produces several command line arguments if several paths are found.

8.7. Semicolon Operator

Example

```
cd articles; paste text1.txt
```

Change the current folder to articles. Display the content of the file text1.txt.

Syntax

```
<seq> ::= <command> ";" <command>
```

Semantics

Run the first command; when the first command terminates, run the second command. If an exception is thrown during the execution of the first command, the execution of the second command can continue and may return a non-zero exit code.

8.8. Command Substitution

Example

```
paste `ls x*` > all.txt
```

List all files that start with x in alphabetical order and merge their content (see paste). Output of the command is redirected in "all.txt".

Syntax

A part of a call command surrounded by backquotes (`) is interpreted as command substitution if the backquotes are not inside single quotes (corresponds the non-terminal `<backquoted>`).

Semantics

For each part SUBCMD of a call command CALL surrounded by backquotes:

1. SUBCMD is evaluated as a separate shell command yielding the output OUT.
2. SUBCMD together with the backquotes is substituted in CALL with OUT. After substitution, symbols in OUT are interpreted the following way:
    - Trailing newlines are deleted from OUT
    - Whitespace characters are used during the argument splitting step. Since our shell does not support multi-line commands, newlines in OUT should be replaced with spaces.
    - Quotes(`'`) are not interpreted during the next parsing step as special characters. For example:

    ```
    echo `echo "'quote is not interpreted as special character'"`
    ```

    would output: `'quote is not interpreted as special character'`
3. The modified CALL is evaluated.

Note that command substitution is performed after command-level parsing but before argument splitting.

9. Applications Specification

Applications can take input from the console. The provided skeleton allows applications to accept input from the console. The input must end with Ctrl+D (^D). If expected stdin is not provided, the application must raise an exception. If required command line arguments are not provided or arguments are wrong or inconsistent, the application should raise an exception as well.
If the applications specification is not comprehensive enough, you are supposed to further specify the requirements in your **Assumptions.pdf** file AND code comments. When in doubt or when the specification is not clear, **you must follow the Bash shell specification of version 3.0 and above of the applications.** Note that you need to use a Unix-based system with a bash shell to check on the behaviour expected for each application. You might use:

- WSL 2 in Windows with bash as the default shell
- Linux-based OS (such as Ubuntu) with bash as the default shell
- MacOS with bash (the default shell in MacOS is zsh, not bash)

Clarifying (specifying) requirements beyond the project description is part of the project. Check with the teaching assistant assigned to your team when in doubt.

You are required to mention in your Assumptions.pdf the operating system that you would be using when you implement and test your project. You should try to make your implementation independent of the operating system used. Your testing should aim to be independent of the operating system, but you might need test cases that test specific cases in an OS-dependent manner.

File and Folder Manipulation

Your applications should handle "." (current directory) and ".." (parent directory), but it is not necessary to handle "~" (home directory). If you use Java Path, such symbols should be correctly handled on any operating system.

Files cannot be removed or modified if certain permissions are not present: e.g. write permission in parent folder, and write permission in file are needed to remove a file. Your applications need to handle cases where permissions do not allow certain operations on the file and should return errors for such cases. Use Java utilities to handle permissions and permission-related issues.

Multiple Dashes Used with Different Applications

When `Ctrl+D` is used to end the input for a command, the execution of the shell will end as well. Using 2 or more dashes (-) with some applications means that the applications read from standard input (stdin). If stdin is not redirected (using IO Redirection or pipe), `Ctrl+D` is used in the shell to add `EOF` to stdin. In IntelliJ, the console will not allow you to continue inputting at stdin after the first `Ctrl+D` is used. However, you might be allowed to continue inputting after the first `Ctrl+Z` (instead of Ctrl+D) if you run your Java shell from console (not from IntelliJ).

In IntelliJ console, your Java shell will not be able to continue to read input for a command or other commands after a `Ctrl+D` (as it would happen in a Linux shell). This is the expected behavior. This means that when `Ctrl+D` is used, the execution of the shell will end as well When running your shell from a console (terminal), you may continue to read input after a `Ctrl+D (`or `<enter>` and `Ctrl+Z)`. You are expected to implement this behaviour for ease of testing.

For example, if you run `wc -l - -` in a Linux shell, the first input file (first dash, stdin) would end after the first EOF (Ctrl+D). The second input file (second dash, stdin) would end after the second EOF (Ctrl+D).
Hence the expected output for `wc -l - -` would be:

```
<user inputs lines>
no_of_lines_before_Ctrl+D -
0 -
total_no_of_lines total
```

The applications that use dash (-) will consume input from stdin in the same way they would consume input from a file. If the application reads all content of the files in sequence, it will consume input from stdin until EOF is met. This behaviour is similar to the Unix shell behaviour.
For example:
`$ wc 1.txt - 2.txt - < a.txt`
Would consume the whole content of files 1.txt, stdin, 2.txt, and stdin in sequence. Since we only have one stdin for each process, the second (and subsequent) dash will not have anything to consume from stdin because the first dash reads stdin until EOF (and all input is consumed).

`$ paste – A.txt - < B.txt > AB.txt`
Would consume one line at a time from each file and merge them together. Hence the next (first) line from stdin (content of B.txt), next (first) line from A.txt, and next (second) line from stdin (content of B.txt) are read and merged into one line. Next, the next (third) line from stdin (content of B.txt), next (second) line from A.txt, and next (fourth) line from stdin (content of B.txt) are read and merged into one line. At the next step, if B.txt has only four lines, EOF is observed at stdin, and only the lines of A.txt are used in the merge. A sample expected output is shown in the project description:

A.txt	B.txt	AB.txt
A B C D	1 2 3 4	1 A 2 3 B 4 C D

9.1. echo

Description

The echo command writes its arguments separated by spaces and terminates by a newline on the standard output.

Command format

`echo [ARG]`

`ARG` – list of arguments

Examples

Display A B C (separated by space characters)
`$ echo A B C`
Display A*B*C (separated by * and A*B*C not wrapped with double quotes)
`$ echo "A*B*C"`

9.2. cd

Description

The cd command changes the current working folder.

Command format

`cd PATH`

`PATH` – relative or absolute folder path. If the PATH does not exist, raise exception. State your assumptions about your absolute path (highly dependent on the operating system you are using).

Examples

Change current working folder to A/New folder (relative path). The following shell commands will be run in the new current working folder.
`$ cd A/New`

9.3. wc

Description

The wc utility displays the number of lines, words, and bytes contained in each input file, or standard input (if no file is specified) to the standard output. A line is defined as a string of characters delimited by a <newline> character. Characters beyond the final <newline> character will not be included in the line count. A word is defined as a string of characters delimited by white space characters. If more than one input file is specified, a line of cumulative counts for all the files is displayed on a separate line after the output for the last file. The application should behave as expected for both text and non-text files.

Command format

`wc [Options] [FILES]`

By default, wc will display lines, words, and bytes in the following format, separated by tab character.

`lines words bytes filename (filename is empty for standard input)`

`Options` - One or more of the following arguments can appear. The arguments can appear together `(-cl)` or separated `(-c -l)`

`-c` The number of bytes in each input file is written to the standard output.

`-l` The number of lines in each input file is written to the standard output.

`-w` The number of words in each input file is written to the standard output.

Note that if more than one argument appears, it will display in the order: lines words bytes.

`FILES` – the name of the file or files. With no FILES, or when FILES is -, read standard input.

Examples

Display `3 3 21 test.txt` (separated by tab characters)

```
$ wc test.txt
```

Display `3 21` (separated by tab characters)

```
$ wc -c -l < test.txt     #input redirection is regarded as standard input
```

Display

```
       3       3      21 test.txt
       3       3      21 test2.txt
       6       6      42 total
$ wc test1.txt text2.txt
```

9.4. mkdir

Description

Create new folders, if they do not already exist. The mkdir command is used to create one or more folders specified by the user.

Command format

`mkdir [Option] DIRECTORIES…`

The command creates the specified folder or folders. If multiple folders are specified, each is created separately.

`Option` – The following option may appear:

`-p` – Create parent folders as needed. If a specified directory already exists, no error is reported.

Examples

Create a single folder named “example” that does not exist (without quotes):

```
$ mkdir example
```

Create multiple folders at once, “dir1”, “dir2”, “dir3” (without quotes):

```
$ mkdir dir1 dir2 dir3
```

Create a directory along with its parent folders, for instance, create “parent/child/grandchild” folders (without quotes), where some or all parents might not exist:

```
$ mkdir -p parent/child/grandchild
```

9.5. sort

Description

The sort command orders the lines of the specified files or input and prints the same lines but in sorted order. Compares each line character by character. A special character (e.g., +) comes before numbers. A number comes before capital letters. A capital letter comes before small letters, etc. Within each character class, the characters are sorted according to their ASCII value.

Command format

`sort [Options] [FILES]`

`Options` - One or more of the following arguments may appear. The arguments can appear together `(-nr)` or separated `(-n -r)`.

`-n` If specified, treat the first word of a line as a number. For instance, sorting file.txt:

```
10
1
2
```

when `-n` is specified, -- the order will be `1, 2, 10.`

Otherwise, consider numbers as normal characters -- the order will be `1, 10, 2.` If the first word is not a number, treat it as a zero, and sort those lines according to their ASCII value.

`-r` – Sort in reverse order.

`-f` – Convert all lowercase characters to their uppercase equivalent before comparison, that is, perform case-independent sorting.

When `-n` and `-f` are specified together, `-n` takes precedence over `-f`.

`FILES` – the name of the file or files. If not specified, use stdin.

9.6. cat

Description

The cat command concatenates the content of given files and prints on the standard output.

Command format

`cat [Option] [FILES]…`

`Option` – The following option may appear:

`-n` – Prefix lines with their corresponding line number starting from 1.

`FILES` – the name of the file or files. With no FILES, or when FILES is -, read standard input.

Examples

Display a file
`$ cat myfile.txt`

Display all .txt files
```
$ cat *.txt
```

Concatenate two files
```
$ cat File1.txt File2.txt > union.txt
```

Display the file contents with line numbering (presence of n flag).
Assuming myfile.txt contains the following contents:
```
Hello World
Welcome to CS4218 module
```

```
$ cat -n myfile.txt
     1 Hello World
     2 Welcome to CS4218 module
```

Display the contents of two files with line numbering (presence of n flag).
Assuming myfile.txt contains the following contents:
```
Hello World
```

Assuming myfile2.txt contains the following contents:
```
Welcome to CS4218 module
```

```
$ cat -n myfile.txt myfile2.txt
     1 Hello World
     1 Welcome to CS4218 module
```

9.7. exit

Description

Exit command terminates the execution.

Command format
```
exit
```

9.8. ls

Description

List information about files.

Command format

`ls [Options][FILES]`

`FILES` – the name the files (non-folders or folders). If no files are specified, list files for current folder. Hidden files should not be listed. If multiple files are provided and one of them produces an error (for example, file does not exist), you might stop the executions of the application, or continue to list the remaining files (mention your assumption in Asumptions.pdf)

`Options` – One or more of the following options may appear:

- `-R` – List files and subfolders recursively. You may assume that there are no cyclic recursive links in the folder where we call ls.
- `-X` – Sort folder contents alphabetically by file extension (characters after the last '.' (without quotes)). Files with no extension are sorted first.

Options can be used in any order or combination.

Examples

run ls on the expansion of all entries in the current folder
`$ ls *`
Due to expanding *, `ls` might receive arguments that are non-directory files. This should not be considered invalid input for `ls`. This means that `ls` is expected to successfully handle non-directory files as well.

9.9. paste

Description

Merge lines of files, write to standard output lines consisting of sequentially corresponding lines of each given file, separated by a TAB character.

Command format

`paste [Option] [FILES]...`

`Option` – The following option may appear:
`-s` – Paste one file at a time instead of in parallel

`FILES` – the name of the file or files. With no FILES, or when FILES is -, read standard input.

Examples

Merge two files A.txt and B.txt (lines from the two files will be merged and separated by TAB)

```
$ paste A.txt B.txt >AB.txt
```

A.txt	B.txt	AB.txt
A B C D	1 2 3 4	A 1 B 2 C 3 D 4

Merge two files A.txt and B.txt with -s (serial) flag

```
$ paste -s A.txt B.txt >AB.txt
```

A.txt	B.txt	AB.txt
A B C D	1 2 3 4	A B C D 1 2 3 4

Merge stdin, A.txt, stdin (stdin is B.txt):

```
$ paste – A.txt - < B.txt > AB.txt
```

A.txt	B.txt	AB.txt
A B	1 2	1 A 2 3 B 4

C D	3 4	C D

9.10. uniq

Description

The uniq command filters **adjacent** matching lines from INPUT_FILE (or standard input) and writes to an OUTPUT_FILE (or to standard output). With no options, matching lines are merged to the first occurrence.

Command format

```
uniq [Options] [INPUT_FILE [OUTPUT_FILE]]
```

`Options` – One or more of the following options may appear:

- `-c` – Prefix lines by the number of occurrences of **adjacent** duplicate lines
- `-d` – Print only duplicate lines, **one for each group**
- `-D` – Print all duplicate lines (takes precedence if used with d flag)

If INPUT_FILE is not specified or INPUT_FILE is '-', read standard input.

If OUTPUT_FILE is not specified, write to standard output.

Examples

Assume 1.txt contains the following content:

```
Hello World
Hello World
Alice
Alice
Bob
Alice
Bob
```

Display output where adjacent duplicate lines are removed

```
$ uniq 1.txt
Hello World
Alice
Bob
Alice
Bob
```

Display output where adjacent duplicate lines are removed with prefix indicating the number of occurrences of **adjacent** duplicate lines

```
$ uniq -c 1.txt
     2 Hello World
     2 Alice
     1 Bob
     1 Alice
     1 Bob
```

Display only duplicate lines, one per group

```
$ uniq -d 1.txt
Hello World
```

```
Alice
```

Display all duplicate lines
```
$ uniq -D 1.txt
Hello World
Hello World
Alice
Alice
```

9.11. mv

Description

Move files or directories from one place to another. By default, it will overwrite an existing file.

Command format

`mv [Option] SOURCE TARGET`

`mv [Option] SOURCE … DIRECTORY`

In the first form, renames the file named by the source operand to the destination path named by the target operand. This form is assumed when the last operand does not name an already existing directory.
In the second form, moves each file named by a source operand to a destination file in the existing directory named by the directory operand. There can be multiple sources.

`Option` – The following option may appear:

`-n` – do not overwrite any existing file.

Examples

move all files with a .txt extension to the folder “test”

```
$ mv *.txt test
```

9.12. cut

Description

Cuts out selected portions of each line (as specified by list) from each file and writes them to the standard output. If no file arguments are specified, cut from the standard input. Column numbering starts from 1.

Command format

`cut Option LIST FILES...`

`Option` – One of the following options may appear:

`-c` – Cut by character position
`-b` – Cut by byte position

`LIST` - can be a list of comma-separated numbers, a range of numbers or a single number.

`FILES` – 'FILE1 FILE2…'. With no FILES, or when FILES is -, read standard input.

Examples

Display 'a'

```
$ echo "baz" | cut -b 2
```

Display 'Ts'. Suppose the file contains one line: "Today is Tuesday."

```
$ cut -c 1,8 test.txt
```

Display 'Today is'. Suppose the file contains one line: "Today is Tuesday."

```
$ cut -c 1-8 test.txt
```

9.13. rm

Description

The rm command attempts to remove the non-directory type files specified on the command line.

Command format

```
rm [Options] FILES...
```

`Options` – One or more of the following options may appear:

`-r` – Traverse recursively to delete directories and their contents

`-d` – Remove empty directories

Examples

Remove the file "1.txt" (without quotes) from current folder

```
$ rm 1.txt
```

Remove the empty folder named "test" (Without quotes)

```
$ rm -d test
```

9.14. tee

Description

The tee command reads from standard input and writes to both the standard output and FILES.

Command format

`tee [Option] [FILES]...`

`Option` – The following option may appear:

`-a` – Append the standard input to the contents of the files specified in FILES

`FILES` – the name of the file or files. With no FILES, or when FILES is -, read standard input.

Examples

Read from standard input and write to both the standard output and 1.txt.

```
$ tee 1.txt
Hello world      // (input by user)
Hello world      // (output in stdout)
How are you      // (input by user)
How are you      // (output in stdout)
CTRL + D (Note: This is a user action, ^D, no need to key into stdin)
```

It is acceptable to print the content from stdout after Crl+D, as follows:

```
$ tee 1.txt
Hello world      // (input by user)
How are you      // (input by user)
CTRL + D (Note: This is a user action, ^D, no need to key into stdin)
Hello world      // (output in stdout)
How are you      // (output in stdout)
```

After the command, 1.txt's content should be:

```
Hello World
How are you
```

(Use of pipe operator) Display "hello" (without quotes) in standard output and store the content "hello" (without quotes) in the file 1.txt.

```
$ echo hello | tee 1.txt
```

(Use of input redirection operator) 2.txt contains "hello" (without quotes) as its file content. Display "hello" (without quotes) in standard output and append the content "hello" (without quotes) to the file content of 1.txt.

```
$ tee -a 1.txt < 2.txt
```

9.15. grep

Description

The grep command searches for lines containing a match to a specified pattern. The output of the command is the list of the lines matching the pattern. Each line is printed followed by a newline.

Command format

```
grep [Options] PATTERN [FILES]
```

`PATTERN` – specifies a regular expression in JAVA format (https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html).

`Options` – One or more of the following options may appear:

- `-i` - Perform case insensitive matching. By default, grep is case sensitive.
- `-c` - Only a count of selected lines is written to standard output.
- `-H` - Print file name with output lines.

`FILES` – the name of the file or files. With no FILES, or when FILES is -, read standard input.

Examples

Search the file example.txt for the string “hunting the shark”:

```
$ grep “hunting the shark” example.txt
```

Search the files A.txt for the string “CS4218 module” with -H flag:

```
$ grep -H “CS4218 module” A.txt
A.txt:This is CS4218 module

$ grep -H “CS4218 module” < A.txt
(standard input):This is CS4218 module
```

10. Additional Information

PMD temporary exclusion

In the event you faced overwhelming PMD violation warnings from both source code and test cases, you can **temporarily** disable PMD checks for test cases. This will result in PMD only checking for violations in your source code (excluding test cases).

This would allow you to resolve PMD violations in the source code before moving on to resolve violations in test cases.

1. Please ensure you have installed IntelliJ's **PMDPlugin** and have setup the provided rule set.
2. Go to IntelliJ's **Settings > PMD > Options > check "Skip Test Sources" > Apply**
   Note: The Skip Test Sources setting does not persist whenever IntelliJ is restarted.

A screenshot of the PMD Option panel can be seen below:

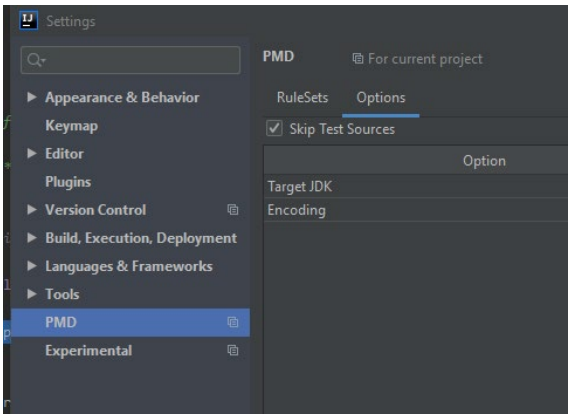

Figure 4: IntelliJ PMD Plugin's Skip Test Sources Feature

Your code and test cases might still have PMD violations at submission time. Briefly explain in the report the reason that prevents you from fixing the remaining violations. You can temporarily disable some rules. Violations in the provided TDD test cases and violations related to the length of the test cases method's names will not receive deductions.

If you encounter PMD violations which you are unable to resolve and/or find that it does not make sense to resolve the PMD violation, you can highlight the specific PMD violation, the code which violates PMD and your justification in your report. Marks will not be deducted if you have a few justifiable PMD violations.

10.1. FAQ

Before asking your questions, check the following page with FAQ: https://docs.google.com/document/d/e/2PACX-1vRBsWgnTHsK3Ju9-khUt61wEX5XvJNgUWmoGDqhDcY8WbyxX7FPiVk7hh5d5InckUEb866o2OtaKoI2/pub

Check this file regularly for any updates.

Any questions you might have about the project you can post them on the Canvas discussions. Alternatively, you may contact Cristina (ccris@comp.nus.edu.sg) and your TA assigned to your team with the mail subject prefix [CS4218] for our easier reference. An example is Mail Subject: *[CS4218] Questions.*

10.2. Instructions on creating team repository using GitHub Classroom

Step 1. Ensure that your team formation through Canvas has been confirmed.
Do not proceed to step 2 otherwise!

Step 2. Visit the assignment link at https://classroom.github.com/a/P1sHqH3y

The first member of the team to access this link will be prompted to accept the assignment that gives your team access to the new repository (as shown in Figure 5).

Create a new team by typing **2023-TeamXX** , where XX is the Team number as noted in Step 1 above. **(Note that the naming convention must be followed strictly, e.g. capitalisation, dash, and spacing. If your group number is a single digit, i.e 2023-Team1 is fine as well.)**

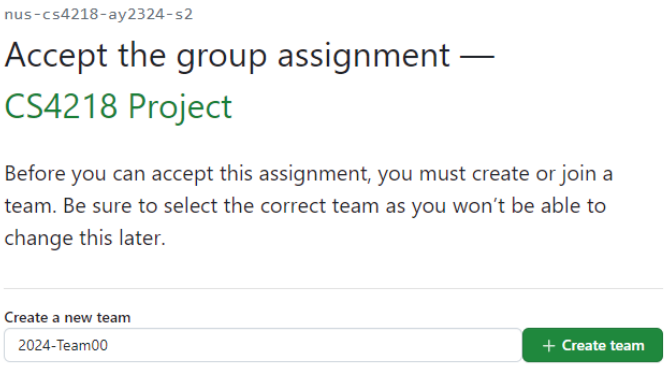

Figure 5: Create a New Team in GitHub classroom

The other members in the team will be able to see an existing team with your team number in the **"Join an existing team"** section. Click **Join**.

Step 3. All of you should be able to see the acceptance page (as shown in Figure 6). Click on the assignment link to see the project on GitHub.

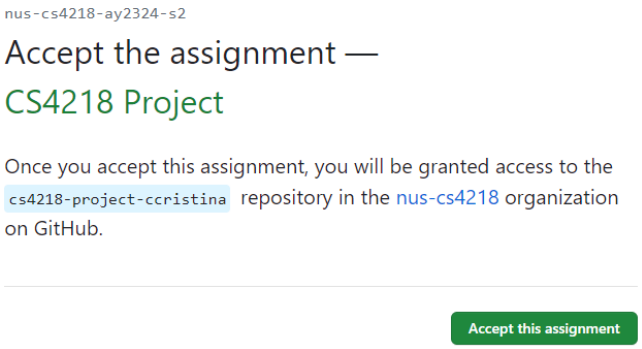

Figure 6: Acceptance Page

You're all set now. A team repo has been created for your team and this will be your team's main repo.

11. Submission Instructions

All submission must be made through Canvas. Apart from that, for each submission, there must be a corresponding tag in your git repository.

11.1. Milestone 1

Due date is Mon, 4 Mar, 2pm
Important note: The team's name, number, and members' names **should not appear** in any of the files submitted on Canvas (only the ZIP file should have the team number)! Your tests will be sent to another team for usage in their test-driven development.

1. Upload a ZIP file to Canvas -> Assignments -> Milestone 1:
    - Zip file format: **<teamNumber>_MS1.zip**
    - Example: TEAM5_MS1.zip
2. In the repository of your team, tag the submitted commit with a tag name "ms1" (meaning milestone 1). Note that the *same files* as those in the submitted zip files should be in the repository.
    - Create a tag, e.g., **git tag -a ms1 -m "Milestone 1"**
    - Push the created tag into the repository, e.g., **git push origin ms1**
    - More details are in: http://git-scm.com/book/en/v2/Git-Basics-Tagging

Your ZIP file submitted on Canvas should contain the following (basically, all your IntelliJ project files):

- A document (**Assumptions.pdf**) containing the assumptions you have made in your implementation. Include here any clarifications you might have made to the applications specification.
- A **report file (named Milestone1.pdf)** in bullet-point format containing (max of 4 pages, 10pt font):
    - Your plans for implementation and testing covering the process you followed for implementation and testing, how test cases were generated to ensure full coverage for individual components and the integration of components, tools and techniques used for testing etc.
    - Summary of test cases provided (what have you covered during testing, did you have any plan for generating tests?, etc.)
- The test cases that helped you find some initial faults in the template (skeleton) code.
- The clean, documented **source code for basic functionality** (BF)
- The clean, documented **source code for one of the extended functionalities (EF1 or EF2)**
    - Please check on Canvas Files -> Project Topics if you have to implement EF1 or EF2. Half of the teams have been assigned to implement EF1 and the other half EF2. Implement the functionalities according to this assignment.
- **Unit tests** for basic and extended functionalities: **BF and EF1 and EF2.**
- **Test cases for all functionalities.**
    - No marks deductions when the test cases for the functionalities that you did not implement are not passing.
- You can separate the test cases for unimplemented functionalities in different folders. You should also include documentation for all the test cases.
- Indicate how to run all your test cases. The tests for the unimplemented functionality may fail.
- Any resource files used in the project.

After Milestone 1, the teaching team will publish a set of test cases to be used by teams in TDD in Milestone 2. These test cases might not be used as they are by all teams. Adjustments, changes, and re-implementation might be needed to match the assumptions your team has made.

Assessment

- **Unit Tests [5 marks]**
  - Unit tests for BF, EF1, and EF2 are graded for all teams.
- **Integration tests [1 mark]**
  - Whether various chains of interactions and integration upon shell state are tested properly
  - Whether both positive and negative tests (exceptional case/corner case/error handling) are written.
- **Implementation and code quality [3 marks]**
  - The implementation for the requested functionalities and applications is successfully completed.
  - Code quality: whether source code and test cases are successfully tested with PMD. However, if there are PMD errors due to missing functionalities, they can be ignored.

11.2. Milestone 2

Due date is Mon, 25 Mar, 2pm.

Important note: The team's name, number, and members' names **should not appear** in any of the files submitted on Canvas (only the ZIP file should have the team number)! Your code and tests will be evaluated and tested by our TAs.

1. Upload a ZIP file in the Canvas -> Assignments -> Milestone 2
    - Zip file format: **<teamNumber>_MS2.zip**
    - Example: TEAM3_MS2.zip
2. In the repository of your team, tag the submitted commit with a tag name "ms2" (meaning milestone 2). Note that the *same files* as those in the submitted zip files should be in the repository.
    - Create a tag, e.g., **git tag -a ms2 -m "Milestone 2"**
    - Push the created tag into the repository, e.g., **git push origin ms2**
    - More details are in: http://git-scm.com/book/en/v2/Git-Basics-Tagging

Your ZIP file submitted on Canvas should contain the following (basically, all your IntelliJ project files):

a. A document (**Assumptions.pdf**) containing the assumptions you have made in your implementation (include here any additions you might have made to the applications specification).
b. A report file (**Milestone2.pdf**) in bullet-point format containing (max of 4 pages, 10pt font):
    - Details about TDD process. Please document briefly in your report the experience you had using the testcases from other teams.
    - Integration testing – plan and execution
c. The clean, documented **implementation** of Basic Functionality (**BF**) and both Extended Functionalities (**EF1 & EF2**)
d. **All your test cases! You should include unit, integrations and system test cases**
e. Document any assumptions that you have made to ensure that features do not get classified as bugs.
f. Any resource files used in the project.

Document your code! Write positive and negative test cases! Your code will be tested by the TAs with an extended suite of test cases, and you will receive the bug reports on your Github repository.

Assessment

- **Integration tests [4 marks]**
    - Whether various chains of interactions and integration upon shell state are tested properly
    - Whether both positive and negative tests (exceptional case/corner case/error handling) are written.
- **TDD [5 marks]**
    - Whether all test cases provided for TDD exercise are passing (mention this in the report). No deductions will be applied in case you decide not to use some (all) of the test cases provided. Do not include the test cases that fail in your Junit test cases.
    - Whether more test cases are written.
- **Implementation and Code quality [2 marks]**
    - Whether source code and test cases are successfully tested with PMD.

- Whether source code and test cases are properly formatted using the IntelliJ formatter.
- Whether JUnit tests are properly written in a good style

- **Effective use of automatic testing tools [*up to* 2 bonus marks]**
  - If the tool discovers a bug, a bug report must be provided that includes (1) generated test, (2) buggy code fragment, (3) applied fix, (4) command line options and other resources
  - However, if the tools did not produce any error revealing tests, then you can explain how you used the regression tests generated by these tools in your project.
  - In the submission you should only include the tests that were relevant with a brief description of tests that were not considered and why they weren't.
  - 1 bonus mark is allocated for each tool that you successfully use on your project.
  - CI/CD tools such as Travis do not qualify for bonus marks.

11.3. Hackathon

Due on Mon, 1 Apr, 2pm.
Your team will receive two other project implementations. You must test these projects and find as many bugs as possible in a short time.

You will receive 2 Github Classroom links corresponding to each of the of the two projects you must test. Create a repository for each project, and submit your bug reports on the repository. You should try to reuse the test cases you have written for your own implementation or prepare new testcase.

Submission:

1. Submit your bug reports on the Github Classroom repository for each of the two projects.
2. Upload a ZIP file in the format described below Canvas -> Assignments -> Hackathon
    - Zip file format: **<teamNumber>_Hackathon.zip**
    - Example: TEAM3_Hackathon.zip

Your ZIP file submitted on Canvas should contain the following (basically, all your IntelliJ project files):

a. A file, named **<teamNumber>_Testing_plan.pdf**, in bullet-point format containing:
    - General method used in testing the two project implementations as part of a test plan
b. The **implementation** you have received for testing (pertaining to the other teams)
c. **Testcases that generate errors** (please do not submit all test cases, but only those that show the presence of bugs for each team, in a separate folder)
d. **Any other resource files used.**

Assessment

- **Hackathon Bugs [6 marks]** - Teams can get up to 8 marks for finding and documenting bugs in the implementations provided. Marks will be awarded after the bug reports are checked by our TAs.

11.4. Rebuttal

Due on Sun, 7 Apr, 2pm.

The bug reports discovered after testing your code will be added to your team's GitHub repository under Issues. You will be able to classify the bug reports as valid or invalid. If needed, you will be discussed with the tutor during the lab session on Thursday or Friday.

Submission:

1. The bug reports will be filed on your project repository. Use labels to classify the bugs reports. Exact details will be provided when the rebuttal phase will start.
2. You have to analyse each bug and label them according to your resolution: "bug accepted", "invalid", "duplicate".

Assessment

- **Rebuttal [1 mark]** - Teams can get up to 1 mark for preparing the rebuttal files.
- Milestone 2 marks may be deducted if the team is found to have an excessive number of bugs. If you find that many bugs come from the same fault, mark them accordingly in the rebuttal file.

11.5. Milestone 3

Due date is Fri, 19 Apr, 2pm.

1. Upload a ZIP file in the format described below to Canvas, under Assignment 'Milestone3'
    - Zip file format: **<teamNumber>_MS3.zip**
    - Example: TEAM3_MS3.zip
2. In the repository of your team, tag the submitted commit with a tag name "ms3" (meaning milestone 3). Note that the *same files* as those in the submitted zip files should be in the repository.
    - Create a tag, e.g., **git tag -a ms3 -m "Milestone 3"**
    - Push the created tag into the repository, e.g., **git push origin ms3**
    - More details are in: http://git-scm.com/book/en/v2/Git-Basics-Tagging

Your ZIP file submitted on Canvas should contain the following (basically, all your IntelliJ project files):

a. The quality assurance report file (**Milestone3-QAreport.pdf**)
    - The QA report will be described in detail during the lab session in week 12.

b. Label the bugs as "fixed" in your Github repository to identify those that have been solved in your final submission.

c. The clean, documented **implementation** of Basic Functionality (**BF**) and both Extended Functionalities (**EF1 & EF2**). This code should pass all test-cases written by your team as well as at least 30% of the test cases for the bugs.

d. **All your test cases!** A separate file (folder) named "Bugs" should be included. This file should contain all test cases for the bugs filed on your repository. Your implementation should pass at least 30% of these valid bugs found by our TAs. Only testcases for the fixed valid bugs should be included.

e. Any resource files used in the project.

Assessment

- **Quality Assurance (QA) Report [6 marks]**
- **Bug Fixes [1 mark]** - Fixes for 30% of the **valid bugs** found for your implementation.
- The final code must adhere to code quality standards. If the code is found not adhering to code quality standards or the code is found to be failing for many test cases, marks may be deducted.

12. Software Quality Assurance (QA) Report

Analysis across project artefacts and milestones

For Milestone 3 of the project, you are required to write a report covering the following topics and questions. Refer to MS1, MS2 and Hackathon/Rebuttal in your answers. Please provide numbers, plots, and explanations with few lines of text (say 1-2 paragraphs) for each question. Note that you should not simply answer each question, but rather create a well-rounded report about your testing efforts during the project. Feel free to add any other information you consider useful regarding your testing efforts.

Topics and questions to be covered in the report

1. How much source and test code have you written?
Test code (LOC) vs. Source code (LOC).
Give an explanation if you observe any unexpected values.

2. Give an overview of the testing plans (i.e., timeline), methods and activities you have conducted during the project. What was the most useful method or activity that you employed?

3. Estimate and analyse the distribution of fault types versus project activities (note that you do not need to provide exact numbers if you did not track your bugs throughout the development and testing efforts; estimates are enough)

3.1. Use diagrams and/or explain the distribution of faults over project activities.

Types of faults: unit fault (algorithmic fault), integration fault (interface mismatch), missing functionality. Add any other types of faults you might have encountered.

Activity: requirements review, unit testing, integration testing, hackathon, coverage analysis. Add any other activities you have conducted.

Discuss what activities discovered the most faults. Discuss whether the distribution of fault types matches your expectations.

Mention if you tracked your bugs or your explanations are based on estimates.

3.2. Analyse the causes of bugs found in your project during Hackathon.

Possible causes: Error in constants; Error in identifiers; Error in arithmetic (+, -), or relational operators (<, >); Error in logical operators; Localized error in control flow (for instance, mixing up the logic of if-then-else nesting); Major errors (for instance, ‘unhandled exceptions that cause application to stop’). Add any other causes you might have identified.

Is it true that faults tend to accumulate in a few modules? Explain your answer.

Is it true that some classes of faults predominate? Which ones?

4. Provide estimates on the time that you spent on different activities (percentage of total project time):

- requirements analysis and documentation %
- coding %
- test development %
- test execution %
- others %

5. Test-driven Development (TDD) vs. Requirements-driven Development. What are advantages and disadvantages of both based on your project experience?

6. Do coverage metrics correlate with the bugs found in your code during hackathon (and not only)?

For example, what 10% of classes achieved the most branch coverage? How do they compare to the 10% least covered classes in terms of code?

Provide your opinion on whether the most covered classes are of the highest quality. If not, why?

7. What testing activities triggered you to change the design of your code? Did integration testing help you to discover design problems?

8. Automated test case generation: did automatically generated test cases (using Evosuite or other tools) help you to find new bugs?

Compare manual effort for writing a single unit test case vs. generating and analysing results of an automatically generated one(s).

9. Hackathon/rebuttal experience: did testing another project help you to improve your own project quality? What about the bugs discovered for your own project?

10. Debugging experience: What kind of automation would be most useful over and above the IntelliJ debugger you used – specifically for the bugs/debugging you encountered in the project?

Would you change any coding or testing practices based on the bugs/debugging you encountered in the CS4218 project?

Did you use any tools to help in debugging?

11. Propose and explain a few criteria to evaluate the quality of your project, except for using test cases to assess the correctness of the execution.

12. Which of the above answers are counter-intuitive to you?

13. Describe one important reflection on software testing or software quality that you have learnt through CS4218 project in particular, and CS4218 in general.

14. We have designed the CS4218 project so that you are exposed to industrial practices such as personnel leaving a company, taking ownership of other's code, geographically distributed software development, and so on. Please suggest a new topic for the project that would bring similar or more benefits to the students.