CS4218 – Project Description

1. Introduction

CS 4218 covers the concepts and practices of software testing and debugging. An important portion of CS 4218 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 maximum 4 at the beginning of the semester. Once formed, teams would be final for the duration of the course.

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. Use Java properties “file.separator” and “path.separator” when working with file system. Use “line.separator” property when working with newlines.

Your code must conform to CS4218 code conventions. These conventions are checked automatically using PMD tool (see installation instructions in IVLE). 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. For example, test class for the class name “Foo” should be called “FooTest”. For test method, explain the scenario that you are testing (e.g., for a test that check for negative value for method foo, use name like testFooNegativeValue).

Use the following versions of software:

- JDK 7 or higher, compiler compliance level 1.7
- Eclipse (Oxygen)
- JUnit 4

4. Assessment

Students can get a maximum of 40 marks for the project, divided as follows:

- 3 marks: Lab attendance and participation
- 2 marks: Implementation
- 2 marks: Code Quality
- 8 marks: Unit Tests
- 3 marks: Test-driven Development
- 4 marks: Integration Tests
- 7 marks: Hackathon bugs
- 3 marks: Hackathon Fixes
- 8 marks: Quality Assurance (QA) Report

Please see Submission Instructions section for details.

5. Project Timeline

Week	Lab	Deadline
1&2	No Lab. Form teams, register your team, and familiarize yourself with the Project Description (current document).	
3	Project introduction, use of code repositories in the project, PMD tool Start developing first part of functionality	
4	Junit for unit testing	
5	Debugging	
6	Test driven development	
Recess	-	Tue, 27 Feb, 2pm: Milestone 1
7	Integration testing, Code coverage	
8	Randoop	
9	Flexible – Complete development	Thu, 22 Mar, 2pm: Milestone 2
10	Hackathon. Write tests and help to find bugs in each other's code	Thu, 29 Mar, 2pm: Hackathon
11	Explanation of Hackathon results – clarification of bug/feature	Tue, 3 Apr, 2pm: Rebuttal
12	Quality Assurance report and its structure are explained	
13	Flexible – students may use this time to ask questions about QA report	Fri, 20 Apr, 2pm: Milestone 3

6. Background

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

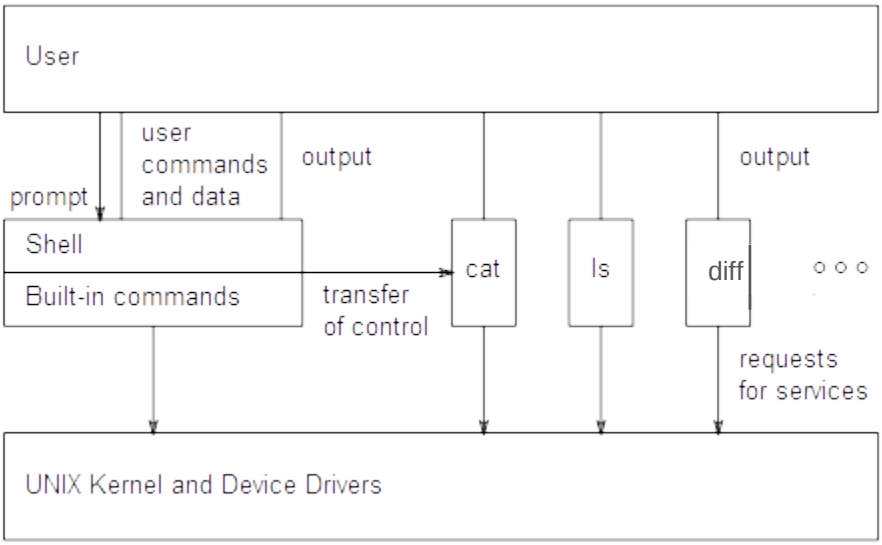

Figure 1

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. Parse user's command.
3. Interpret user's command, run specified applications if any.
4. Print output.
5. Go to 1.

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 text data from its Standard Input stream (stdin) and an array of command line arguments. 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.

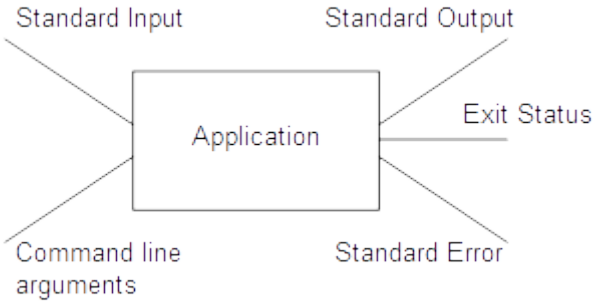

Figure 2

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 “cat” and “grep”:

```
cat articles/* | grep “Interesting String”
```

As shown in Figure 3, Shell expands “articles/*” into the list of all the files in the “articles” directory and passes them to “cat” as command line arguments. Then, “cat” concatenates 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 “cat” with the stdin of “grep”:

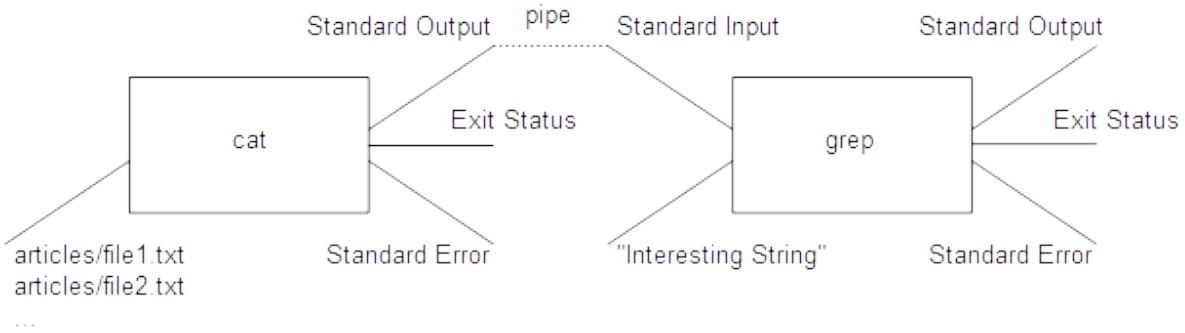

Figure 3

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. Implementation 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 behavior 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 instead of writing to stderr and returning non-zero exit code in case of errors, as shown in 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 shell and applications are provided in a skeleton. Students need to use the interfaces provided, and are not allowed to modify them. The interfaces will be later used in unit testing, so it is important to maintain compatibility among teams.

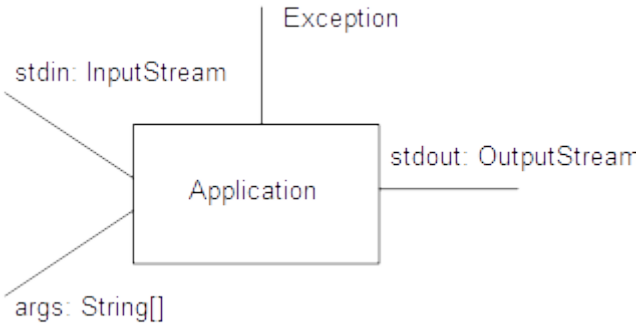

Figure 4

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, pipe operator, IO-redirection
- Applications: ls, cat, echo, exit.

The Extended Functionality 1 (EF1) includes:

- Shell: quoting, globing.
- Applications: mkdir, grep, paste, diff.

The Extended Functionality 2 (EF2) includes:

- Shell: semicolon operator, command substitution.
- Applications: cd, sed, split, cmp.

7.1. 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 input streams.

7.1.1. 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 kind of commands: call command, sequence command (using semicolon operator), pipe command (using pipe operator).

Command line uses the following grammar:

```
<command> ::= <call> |  <seq> | <pipe>
<call> ::= ( <non-keyword> | <quoted> ) *
<seq>  ::= <command> “;” <command>
<pipe> ::= <call> “|” <call> | <pipe> “|” <call>
```

A non-keyword character is any character except for newlines, single quotes, double

quotes, backquotes, semicolons “;” and vertical bars “|”. The non-terminal <quoted> is described below.

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

7.1.2. 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> ::=
  [ <whitespace> ] [ <redirection> <whitespace> ]* <argument>
  [ <whitespace> <atom> ]* [ <whitespace> ]
<atom> ::= <redirection> | <argument>
<argument> ::= ( <quoted> | <unquoted> )+
<redirection> ::= “<“ [ <whitespace> ] <argument> |
                  “>“ [ <whitespace> ] <argument>
```

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 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 Globing).
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.

7.1.3. Pipe operator

Example

```
cat 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 execution of the other part must be terminated with a non-zero exit code.

7.1.4. IO Redirection

Example

```
cat < file.txt
```

Display the content of all the files in the articles directory.

Syntax

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

Semantics

- 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 several files are specified for input redirection or output redirection, throw an exception.
- If no files are given, use the NULL value.
- 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.

7.1.5. Quoting

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), * (globing), ‘ (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: ‘ (single quote) and ` (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 “ “`.`

Note that we do not use character escaping (\) in our shell.

7.1.6. Globing

Example

```
cat articles/*
```

Display the content of all the files in the articles directory.

Syntax

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

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

7.1.7. Semicolon operator

Example

```
cd articles; cat text1.txt
```
Change the current directory 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.

7.1.8. Command substitution

Example

```
cat `ls x*` > all.txt
```
List all file that start with x in alphabetical order and print their content. Output of the command is redirected in all.txt.

Syntax

A part of a call command surrounded by backquotes (`) is interpreted as command substitution iff the backquotes are not inside single quotes (corresponds the 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:
    - 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.
    - Other characters (including quotes) are not interpreted during the next parsing step as special characters.
3. The modified CALL is evaluated.

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

7.2. Applications specification

Applications that require stdin stream do not read it directly (from console), but only use input streams provided by shell through redirections. 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, follow the UNIX shell specification of the applications.

7.2.1. ls

Description

List information about files.

Command format

`ls [-d][FILE][-R]`
`FILE` – the name of the file or files. If no files are specified, list files for current directory. Hidden files should not be listed.
`-d` – list the directories only
`-R` – List subdirectories recursively

Examples

List the directories in the current directory
```
$ ls -d */
```

list ALL subdirectories
```
$ ls *
```

7.2.2. cat

Description

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

Command format

`cat [FILE]...`

`FILE` – the name of the file or files. If no files are specified, use stdin. If FILE and stdin are not specified, raise an exception.

Examples

Display a file:
```
$ cat myfile.txt
```

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

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

7.2.3. 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)

```
$ ls A B C
```

Display “A*B*C” (separated by *)

```
$ ls “A*B*C”
```

7.2.4. exit

Description

Exit command terminates the execution.

Command format

```
Exit
```

7.2.5. mkdir

Description

Create new folder(s), if they do not already exist.

Command format

```
mkdir FOLDERS
```

FOLDERS – the name of the folder or folders to be created.

Examples

Create a folder named New in the current path

```
$ mkdir New
```

Create a folder names New in folder A

```
$ ls A/New
```

7.2.6. 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 [-v] PATTERN [FILE]...
```

PATTERN – specifies a regular expression in JAVA format (https://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html).
-v – invert the sense of matching, to select non-matching lines.
FILE – the name of the file or files. If not specified, use stdin.

Examples

Search the file example.txt for the string ‘hunting the snark’:

```
$ grep ‘hunting the snark’ example.txt
```

Search the file wordlist.txt for any lines that don’t include at least one vowel:

```
$ grep -v [aeiou] wordlist.txt
```

7.2.7. 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 [FILE]...
```
FILE – the name of the file or files. If not specified, use stdin.

Examples

Merge two files A.txt and B.txt (lines from the two files will be merged sequentially 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

7.2.8. diff

Description

Display the differences between two files, or each corresponding file (with differences) in two directories. For two directories, diff compares corresponding files in both directories, in alphabetical order; this comparison is not recursive. For files that are identical, diff normally produces no output.

Command output convention in displaying the videos:

< - denotes lines only in the first file
> - denotes lines only in the second file

Command format

```
diff [Options] FILES...
```

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

`-s` - Report when two files are the same by printing a message “Files are identical”
`-B` - Ignore changes with all blank lines
`-q` - Output only whether files differ by printing a message “Files differ”

`FILES` – ‘FILE1 FILE2’ or ‘DIR1 DIR2’. If a FILE is ‘-’, read standard input.

Options can be used in any order or combination: all following options should be allowed for `diff –sBq …, diff -s –B …, diff -qs –B …`

Examples

Differences for two files A.txt and B.txt

```
$ diff A.txt B.txt
```

A.txt	B.txt	Output
line1 line2 line3 line5	line1 line3 line5 line6	<line2 >line6

Differences for two directories A and B

```
$ diff –q A B
```

---?

7.2.9. cd

Description

The cd command changes the current working directory.

Command format

```
cd PATH
```

`PATH` – relative or absolute directory 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 directory to A/New folder (relative path). The following shell comments will be run in the new current working directory.
```
$ cd A/New
```

7.2.10. sed

Description

The sed command copies input file (or input stream) to stdout and performs string replacement. For each line containing a match to a specified pattern (in JAVA format), replaces the matched substring with the specified string.

Command format

```
sed REPLACEMENT [FILE]
```

`REPLACEMENT` – specifies replacement rule, as follows:

- `s/regexp/replacement/` – replace the first (in each line) substring matched by regexp with the string replacement.
- `s/regexp/replacement/X` – X is a number. Only replace the Xth match of the regexp.

Note that the symbols "/" used to separate regexp and replacement string can be substituted by any other symbols. For example, "s/a/b/" and "s|a|b|" are the same replacement rules. However, this separation symbol should not be used inside the regexp and the replacement string.

`FILE` – the name of the file. If not specified, use stdin.

Examples
add a leading angle bracket and space to each line (quote a message) in file.txt
```
$ sed s/^/> / file.txt
```

delete leading angle bracket & space from each line (unquote a message)
```
$ sed s/^> //1 file.txt
```

7.2.11. split

Description

The command splits a file into fixed-size pieces, creates output files containing consecutive sections of FILE(standard input if none is given). By default, ‘split’ puts 1000 lines of INPUT (or whatever is left over for the last section), into each output file.

The output files’ names consist of PREFIX (‘x’ by default) followed by a group of letters ‘aa’, ‘ab’, and so on, such that concatenating the output files in sorted order by file name produces the original input file. If more than 676 output files are required, ‘split’ uses ‘zaa’, ‘zab’, etc.

Command format

`split [`*`Options`*`] [`*`FILE`* `[`*`PREFIX`*`]]`

`Options` – One of the following options can appear:

`-l` *`LINES`* – Put *LINES* lines of *FILE* into each output file.

`-b` *`BYTES`* – Put the first *BYTES* bytes of *FILE* into each output file. Appending ‘b’ multiplies *BYTES* by 512, ‘k’ by 1024, and ‘m’ by 1048576.

`FILE` – the name of the file. If not specified, use stdin.

Examples

Split up the file demo.zip into multiple 100 MB files:

```
$ split -b 100m demo.zip
```

The output files will be named with 3 letters starting `xaa`, `xab`, … to reassemble them, cat the files in alphabetical order:

```
$ cat `ls x*` > demo2.zip
```

7.2.12. cmp

Description

The cmp command compares two files, and if they differ, tells the first byte and line number where they differ. For files that are identical, ‘cmp’ produces no output. When the files differ, by default, ‘cmp’ outputs the byte offset and line number where the first difference occurs.

Command format

`cmp` *`Options... FILE`*

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

`-c` - Print the differing characters.

`-s` – Output only whether files differ by printing a message “Files differ”

`-l` – Print the (decimal) offsets and (octal) values of all differing bytes.

`FILES` – ‘FILE1 FILE2’. If a FILE is ‘-’, read standard input.

Multiple single letter options (unless they take an argument) can be combined into a single command line word: ‘-csl’ is equivalent to ‘-c –sl’ or ‘-l –c-s’.

Examples

Compares file1 to file2, reading each file byte-by-byte and comparing them until one of the byte pairs is not equal:

```
$ cmp file1.txt file2.txt
```

The output is:

```
file.txt file2.txt differ: char 1011, line 112
```

Compares file1 to file2 by should all differences:

```
$ cmp –l file1.txt file2.txt
```

The output is:

```
1 155  40
2 145 156
3  40 141
4 151 155
5 163 145
```

8. *Submission Instructions*

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

8.1. Milestone 1

Due date is Tue, 27 Feb, 2pm.

Important note: The team's name, number, and members' names **should not appear** in any of the files submitted on IVLE (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 in the format described below to IVLE Workbin, under the folder 'Project-Milestone1'
    - Zip file format: **<lab_slot>_<teamNumber>_MS1.zip**
    - Example: 10_TEAM5_MS1.zip (10 stands for the lab slot from 10am-11am).
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 IVLE should contain the following (basically, all your Eclipse project files):

- 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 **fixed template code** without any of our faults. To be on the safe side, you can also submit the test cases for each of our faults in the template 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 IVLE Project->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 or EF2)**)
    - If you are have implemented EF1, you should submit unit tests for EF1.
    - If you are have implemented EF2, you should submit unit tests for EF2.
- T**est cases for the other extended functionality** (**EF2 or EF1**)
    - If you are have implemented EF1, you should submit tests for EF2.
    - If you are have implemented EF2, you should submit tests for EF1.
- All other test cases!
- 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.

Assessment

- **Unit Tests [8 marks]**
  - Unit tests for BF, EF1, and EF2 are graded for all teams.
- **Implementation [1 mark]**
  - The implementation for the requested functionalities and applications is successfully completed

8.2. Milestone 2

Due date is Thu, 22 Mar, 2pm.

Important note: The team's name, number, and members' names **should not appear** in any of the files submitted on IVLE (only the ZIP file should have the team number)! Your code and tests will be sent to another team during the Hackathon.

After Milestone 1, the teaching team will publish a set of test cases to be used by teams in TDD. 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.

1. Upload a ZIP file in the format described below to IVLE Workbin, under the folder 'Project-Milestone2'
   - Zip file format: **<lab_slot>_<teamNumber>_MS2.zip**
   - Example: 10_TEAM3_MS2.zip (10 stands for the lab slot from 10am-11am).
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 IVLE should contain the following (basically, all your Eclipse project files):

a. A document (Assumptions.pdf) containing the assumptions you have made in your implementation (include here any additions you might have made 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 (in the Hackathon).
f. Any resource files used in the project.

Document your code! Write positive and negative test cases! After the Hackathon you will need to defend your code and reduce the number of bugs issued against your code.

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 [3 marks]**
  - Whether all test cases provided for TDD exercise are passing.
  - Whether more test cases are written.
- **Implementation and Code quality [3 marks]**
  - Whether source code and test cases are successfully tested with PMD.
  - Whether source code and test cases are properly formatted using the Eclipse formatter.
  - Whether JUnit tests are properly written in a good style
- **Effective use of Randoop or other tools [*up to* 2 bonus mark]**
  - If Randoop 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 used to generate Randoop
  - Randoop generates both regression tests and error revealing tests. However, if Randoop did not produce any error revealing tests, then you can explain how you used the regression tests generated by Randoop 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.

8.3. Hackathon

Due on Thu, 29 Mar, 2pm. Your team will receive the source code implemented by another team. You have to test this source code and find as many bugs as possible in a short-time. This activity should take place during the lab session and points will be giving for your activity. You should reuse the test cases you have written for your own implementation.

Submission:

1. Upload a ZIP file in the format described below to IVLE Workbin, under the folder 'Project-Hackathon'
    - Zip file format: **<lab_slot>_<teamNumber>_Hackathon.zip**
    - Example: 10_TEAM3_Hackathon.zip (10 stands for the lab slot from 10am-11am).

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

a. A report file (BugReports.pdf) in bullet-point format containing:
    - General method used in testing
    - A table containing the bug reports:

Bug Report Number	**Description**	**Testcase**	**Comments (this column should be left empty; for marking usage)**
1	…	…	

b. The **implementation** you have received for testing (pertaining to another team)
c. **Testcases that generate errors** (please do not submit all test cases, but only those that shows the presence of bugs)
d. **Any other resource files used.**

Assessment

- Teams can get up to 7 marks for finding and documenting bugs in the implementation provided. Marks will be awarded after rebuttal.

8.4. Rebuttal

Due on Tue, 3 Apr, 2pm. Your team will receive the bug report produced by another team for your implementation. You will be able to classify the bug reports provided by the hacking team as valid or invalid. The bug rebuttal file will be discussed with the tutor during the lab session on Wed, 4 Apr.

Submission:

1. Upload a ZIP file in the format described below to IVLE Workbin, under the folder 'Project-Rebuttal'
    - Zip file format: **<lab_slot>_<teamNumber>_Rebuttal.zip**
    - Example: 10_TEAM3_Rebuttal.zip (10 stands for the lab slot from 10am-11am).

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

a. The report file "BugReports.pdf" received by the team describing the bugs found in their code by another team.
b. A report file named "Rebuttal.pdf" containing the following table. The numbers of the bug reports should correspond with those found in "BugReports.pdf":

Bug Report Number	**Status**	**Reason if invalid**	**Comments (this column should be left empty; for marking usage)**
1	Valid/ Invalid	(e.g. duplicate of bug report #X, bug report is not in accordance with project specification on page Y line Z "quote line", …)	

8.5. Milestone 3

Due date is Fri, 20 Apr, 2pm.

1. Upload a ZIP file in the format described below to IVLE Workbin, under the folder 'Project-Milestone3'
    - Zip file format: **<lab_slot>_<teamNumber>_MS3.zip**
    - Example: 10_TEAM3_MS3.zip (10 stands for the lab slot from 10am-11am).
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 IVLE should contain the following (basically, all your Eclipse project files):

a. The quality assurance report file (**Milestone3-QAreport.pdf**)
    - The template for the QA report will be posted on IVLE. The QA report will be described in detail during the lab session on 11 Apr.

b. A report file named "**Fixes.pdf**" containing the following table. The numbers of the bug reports should correspond the valid bugs found in "BugReports.pdf" and "Rebuttal.pdf":

Bug Report Number	Status	Testcase (filename)	Comments (this column should be left empty; for marking usage)
1	Fixed/Not fixed	…	

c. The report file "**BugReports.pdf**" received by the team describing the bugs found by another team, received before the Rebuttal phase (Note: you should not submit the BugReport file you have generated for another team's implementation).

d. The report files previously submitted in the Rebuttal phase "**Rebuttal.pdf**"

e. 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 Hackathon test cases (bug reports).

f. **All your test cases!** A separate file (folder) named "Hackathon" should be included. This file should contain all test cases for bugs found during Hackathon. Your implementation should pass at least 30% of these test cases found during the Hackathon. Note that this file will be evaluated.

g. Any resource files used in the project.

Assessment

- QAReport – 8 marks
- Fixes for bugs found during the Hackathon. Teams can get up to 3 marks.
- 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.