CS 4218 – Software Testing and Debugging

Ack: Tan Shin Hwei for project description formulation

The Project

CS 4218 covers the concepts and practices of software testing and debugging. An important portion of the 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 two components and test their functionality using different testing techniques that will be taught during the course.

In the first four weeks of the project the students will start development and testing activities based on the project specification. The students will learn to professionally apply testing approaches and techniques with the state of art testing automation framework JUnit. The students will be shown good and poor styles of manual unit-test generation. By week four students will develop shell, first component and corresponding test cases.

In the following four weeks, students will develop and integrate all the components of the project and assess the code quality using coverage metrics. The development of the second component will be based on existing test cases in lieu of the project description, applying the concepts of test driven development. Particular attention will be paid to the integration and interaction testing of shell and the implemented components. The students will evaluate the quality and thoroughness of the test cases and project code using different coverage metrics. By the end of week eight students will have integrated both the components with the shell reliably.

In the remaining project weeks, students will take on the role of a professional software tester. In this role, they will apply testing, debugging and other quality assurance activities in a simulated industrial setting with limited communication. Finally, by week 14 students will have submitted all the project artifacts (including source code, test cases, bug reports, quality assurance reports) for final evaluation.

Programming Language

All programming assignments must be completed using JAVA

Project Teams

Students may form a team at the beginning of the semester. Each team may consist of a maximum of 4 team members. Each team will report the team members and the team leader before the second lab session. Once formed, teams would be final for the duration of the course.

Project Constraints

Coding

Must be a standalone JAVA application.
External libraries/plugins should not be used.
Should not rely on network communication.
Should not used databases.
Each method must not be too long. You should refactor your code if the method is too long.
You must have proper Javadoc comments for every method. Comments should explain the purpose of the function in at least one complete sentence.
Do not comment excessively.
Use the given class/method names. For helper methods, follows the Java naming convention:
* Names representing packages should be in all lower case.
* Names representing types must be nouns and written in mixed case starting with upper case(e.g., Line,

AudioSystem)
* Variable names must be in mixed case starting with lower case. (e.g., line, audioSystem)
Use meaningful names for method and variables. This rule applies to test method as well. 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 naming convention for test classes. For example, test class for the class name "Foo" should be called "FooTest".
Classes should be declared in individual files with the file name matching the class name.
Format your code with proper indentation for better readability.

Submission
All submission must be made through the IVLE
Each submission should be named in the following format XX-YY.zip, where XX is the assignment name and YY is the team name
Do not include your name in any of the submitted code. This rule is made for the purpose of grading.

4218 Project Timeline

Week 1
Hands-on hour: Basic introduction to the course project.

Week 2
Hands-on hour: We describe the shell that needs to be implemented (different file utilities, see project description). The class is divided into two groups to allow exchanging of code and test cases. Group 1 gets assigned grep tool and piping (extended functionality 1). Group 2 gets assigned the text utilities (extended functionality 2)
Homework: Develop shell and start with your extended functionalities

Week 3
Hands-on hour: Setup Junit as per our tutorial website
Homework: Continue work on shell and extended functionality. Setup JUnit environment for your project.

Week 4
Hands-on hour: Continuation of JUnit, and its usage on the code. Students are encouraged to start using Junit in their projects, as they are developing the code.
Homework: Finish development of shell and your extended functionality.
Milestone: Submit the test cases for the shell and extended functionality before week 5

Week 5
Hands-on hour: *Test Driven Development*. Swap the test cases between the groups
Homework: Develop the other extended functionality based on the newly acquired test cases

Week 6
Hands-on hour: Complete the code development in the project.
Homework: Write integration tests. Make sure that integration test invokes all the utilities in combinations with at least one other utility.

Week 7
Hands-on hour: Discuss code coverage tools. Determine the statement, branch, and MC/DC coverage of your code
Homework: Add relevant test cases (if needed)
Milestone: Submit all code, test cases and coverage reports by week 8

Week 8
Hands-on hour: left flexible this week, can discuss course materials outside course project specifically for this week.

Week 9

Hands-on hour: **Hackathon**. Hand-over of one group's code to another. *Limited interaction in explanation across teams to simulate industrial practice*. Generate test cases that fail due to bugs in others code (if any). Determination of testing strategies to test other's code.

Week 10
Hands-on hour: Quality Assurance. Comparison of testing strategies, measures of confidence
Homework: Write a quality assurance report
Milestone: Submit all test cases with bug reports, from the Hackathon week (specifically, the failing ones)

Week 11
Hands-on hour: Debugging strategies for the failed test cases that were found earlier
Homework: Debug and fix your code such that all test cases pass

Week 12:
Hands-on hour: Finish debugging, and generate fixes for failed tests.

Week 13:
Hands-on hour: Finalization and validation of fixes, by re-running tests. Discussion of how test selection was done during the re-running

Week 14: (Reading week)
Milestone: Finalization and Submission of Project Report, Initial Code, Tests, Final Code, QA report plus any other artifacts (such as debugging logs)

Implementation Requirements

How to begin?

A shell is used to interpret and execute user's commands. Following sequence explains how a basic shell can be implemented in Java
Do Forever
1. Wait for a user input
2. Parse the user input. Separate the command and its arguments
3. Create a new thread to execute the command
4. Execute the command and its arguments on the newly created thread. Exit with the status code of the executed command
5. In the shell, wait for the thread to complete execution
6. Report the exit status of the command to the user

The functionality in the shell will be implemented (and tested) in two phases: Basic Functionality and the Extended Functionality

***Basic functionality*:** As a part of basic functionality the shell should be able to interpret the following commands. Note that none of the commands in this category require additional arguments.

pwd : report present working directory
cd : change directory
ls : list the contents of a directory
copy : copy a file to a given location
move : move a file to a given location
delete : delete a file
cat : cat copies each file or standard input (denoted by '-') if no files are given to the standard output
echo : echo writes its arguments separated by blanks and terminated by a newline on the standard output
Ctrl + z : stop the execution of present command (thread). If no command is being executing this

key combination has no effect on the shell

All other inputs : Any inputs other than the ones explicitly mentioned above should return an error message

***Extended Functionality*:** This set of functionalities is further subdivided into two parts. Each team will have to implement both the sets of Extended Functionality before the final submission.

***Extended functionality, Set I*:** As a part of extended functionality set I, teams will implement the *grep* and the *pip*e tool.

Grep : The grep command searches one or more input files for lines containing a match to a specified pattern. The grep tool must work on all characters in UTF-8 encoding.

Command Format - grep [OPTIONS] PATTERN [FILE]
PATTERN - This specifies a regular expression pattern that describes a set of strings
FILE - Name of the file, when no file is present (denoted by "-") use standard input
OPTIONS
-A NUM : Print NUM lines of trailing context after matching lines
-B NUM : Print NUM lines of leading context before matching lines
-C NUM : Print NUM lines of output context
-c : Suppress normal output. Instead print a count of matching lines for each input file
-o : Show only the part of a matching line that matches PATTERN
-v : Select non-matching (instead of matching) lines
-help : Brief information about supported options

Pipe : The pipe tools allows the output of one program to be sent to the input of another program. With the help of pipe tool multiple small (and simple) programs can be connected to accomplish large number of tasks.

Command Format - PROGRAM-1-STANDARD_OUTPUT | PROGRAM-2-STANDARD_INPUT
Where "|" is the pipe operator and PROGRAM-1-STANDARD_OUTPUT is the standard output of program 1 and PROGRAM-2-STANDARD_INPUT is the standard input of program 2.

***Extended functionality, Set II*:** As a part of extended functionality set II, teams will implement the *Text utilities* that are listed below

cut : prints a substring that is specified in a certain range

Command Format - cut [OPTIONS] [FILE]
FILE - Name of the file, when no file is present (denoted by "-") use standard input
OPTIONS
-c LIST: Use LIST as the list of characters to cut out. Items within the list may be separated by commas, and ranges of characters can be separated with dashes. For example, list '1-5,10,12,18-30' specifies characters 1 through 5, 10,12 and 18 through 30.
-d DELIM: Use DELIM as the field-separator character instead of the TAB character
-help : Brief information about supported options

paste : writes to standard output lines consisting of sequentially corresponding lines of each given file, separated by a TAB character

Command Format - paste [OPTIONS] [FILE]
FILE - Name of the file, when no file is present (denoted by "-") use standard input
OPTIONS
-s : paste one file at a time instead of in parallel
-d DELIM: Use characters from the DELIM instead of TAB character
-help : Brief information about supported options

comm : Compares two sorted files line by line. With no options, produce three-column output. Column one contains lines unique to FILE1, column two contains lines unique to FILE2, and column three contains lines common to both files.

Command Format - comm [OPTIONS] FILE1 FILE2
FILE1 - Name of the file 1
FILE2 - Name of the file 2
-c : check that the input is correctly sorted, even if all input lines are pairable
-d : do not check that the input is correctly sorted
-help : Brief information about supported options

sort : sort lines of text files

Command Format - sort [OPTIONS] [FILE]
FILE - Name of the file
OPTIONS
-c : Check whether the given file is already sorted, if it is not all sorted, print a diagnostic containing the first line that is out of order
-help : Brief information about supported options

uniq : Writes the unique lines in the given input. The input need not be sorted, but repeated input lines are detected only if they are adjacent.

Command Format - uniq [OPTIONS] [FILE]
FILE - Name of the file, when no file is present (denoted by "-") use standard input
OPTIONS
-f NUM : Skips NUM fields on each line before checking for uniqueness. Use a null string for comparison if a line has fewer than n fields. Fields are sequences of non-space non-tab characters that are separated from each other by at least one space or tab.
-i : Ignore differences in case when comparing lines.
-help : Brief information about supported options

wc : Prints the number of bytes, words, and lines in given files

Command Format - wc [OPTIONS] [FILE]
FILE - Name of the file, when no file is present (denoted by "-") use standard input
OPTIONS
-m : Print only the character counts
-w : Print only the word counts
-l : Print only the newline counts
-help : Brief information about supported options