CS4218 - Milestone 2 Report

Test-Driven Development (TDD) Plan and Execution

The adoption of Test-Driven Development (TDD) within our project played a crucial role in ensuring the early detection of potential flaws and the validation of feature behaviour.

These pre-written tests served as an initial blueprint, guiding the development of features by specifying expected outcomes in the development cycle. This method ensured a consistent focus on fulfilling functional requirements. Not only did we use the tests provided to us for this TDD exercise, we also used the TDD tests written by us in MS1 in order to develop the applications.

These pre-written tests served as an initial blueprint, guiding the development of features by specifying expected outcomes in the development cycle. This method ensured a consistent focus on fulfilling functional requirements. Upon reviewing the TDD cases, we identified opportunities to bolster our testing suite further. Additional unit test and integration TDD test cases were crafted, extending the coverage beyond the initial scope.

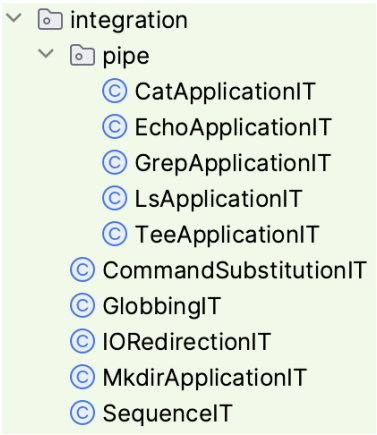
The execution of TDD tests was instrumental in unveiling potential bugs within our implementation. A notable observation was that certain TDD tests failed due to discrepancies between the tests' assumptions and our own.
In response to the identified discrepancies, we undertook a careful revision of the TDD tests. The modifications were aimed at realigning the tests with our project-specific assumptions, ensuring that they accurately reflected the intended behaviour of our shell application. In the end, we ensured that all the test cases (the TDD tests we created in MS1 as well as the tests provided to us) were passing.

Integration Tests Plan and Execution

For integration tests, our approach was to test comprehensively with multiple units based on the project specification. One way we did this is to look at how two units are used together. For example, there is usually no dependency among echo and wc, hence one way to test these two units together is to make use of operators such as pipe and sequence operators etc.

Our Integration Tests covered multiple units from different applications units, shell units and other utility units, and was done by running Shell.parseAndEvaluate(commandString, outputStream). By including the shell as part of the integration test, we were able to get a more comprehensive test that included more than 2 units. Some of the integration tests also included end-to-end testing (aka system test cases).

We also tried the Big Bang Integration approach where we combined many shell operators with commands to see how they interact. This was good for small systems and also did stress testing for big commands with a lot of tasks and combinations to handle.



The above diagram is the structure of our integration tests. As you can see from the folder, the pipe has multiple test files and the other operators will have one test file each for the integration tests. This is because pipe operators rely heavily on the combination of different applications, hence we made use of pairwise testing.

Pairwise testing for pipe command

As there are many possible combinations of components interactions to test in integration testing and testing all possibilities is not feasible due to potential combinatorial explosion.
As such, we first identified all possible pairs of commands that are reasonable to be connected with a pipe. Considering a pipe command in the form of “app1 | app2”, a pipe command that is worth testing is where app1 writes to standard out and app2 reads from standard in as this ensures there will be interaction between app1 and app2. On the other hand, it is not so meaningful to extensively test commands like as“cd|exit”as there is no interaction between the two apps. Besides the positive cases, we also tested on negative cases to verify that “app2” will not be executed if “app1” throws an exception.

For example, in GrepApplicationIT under /Integration/Pipe folder, it contains pairwise testing applications where grep is the first command in the pipe (e.g.. grep “th” input.txt | wc), and it will be piped to other commands that have input streams involved (e.g. echo, tee, wc etc.)

Application Modules Categorisation	
Involves IO streams	**No IO streams involved**
EchoApplication, PasteApplication, GrepApplication, WcApplication, CutApplication, LsApplication, SortApplication	RmApplication, ExitApplication, CdApplication, MvApplication

Negative testing

In our testing strategy, we maintain a balanced approach, with approximately 80% of our tests focusing on positive scenarios and 20% on negative scenarios. Negative testing involves intentionally introducing invalid commands or combining invalid commands with valid ones. Each negative test is meticulously designed to pinpoint only one intended point of failure. This ensures that any exceptions or errors encountered are solely attributed to the invalid input under examination.

Automated Testing Tools - PIT Test

The project integrated the PIT Test as an automated testing tool for mutation testing. PIT Test introduces mutations to the program source code and it triggers our existing test suites to see if our tests can detect those mutations.

Below here shows the results of our PIT Test (found in /pit-reports folder).

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
sg.edu.nus.comp.cs4218.impl.app	15	80%	955/1188	81%	425/522	90%	425/470
sg.edu.nus.comp.cs4218.impl.cmd	3	91%	75/82	93%	26/28	93%	26/28
sg.edu.nus.comp.cs4218.impl.parser	13	92%	115/125	83%	76/92	93%	76/82
sg.edu.nus.comp.cs4218.impl.util	8	94%	320/341	90%	120/134	93%	120/129

Based on the results of PIT Tests, we identified test cases that did not adequately cover the mutations and also the different test coverage of our program. Based on the result, we tried to improve and create more test cases that can catch these mutations and iteratively ran the PIT Test to try to improve our scores/results.

PIT Test helped us ensure our test suite quality as our tests are not just passing but actually effective. It also uncovered edge cases and conditions that were not covered by existing tests, which were subsequently resolved.

Automated Testing Tools - Randoop

In our continuous effort to maintain high-quality software standards, we explored various testing methodologies to identify and rectify potential weaknesses in our codebase. This report delves into our experiences with employing Randoop, an automated testing tool. Randoop is an open-source automated testing tool designed to generate unit tests for Java programs.

In our project, tests created from Randoop can be viewed in the ‘randoop’ folder within the root directory. It contains tests for 2 parser classes and 2 application classes.

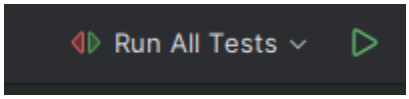
Acknowledging the challenge of achieving comprehensive test coverage, we employed Randoop with the strategic aim of enhancing the testing suite for classes that previously exhibited low coverage metrics. A significant achievement through the use of Randoop was the identification of errors within our code that had eluded traditional testing methods. These errors, once rectified, contributed to the overall integrity and performance of the applications.

The culmination of our efforts with Randoop was a notable increase in test coverage. By addressing undercovered classes and incorporating the insights gained from the generated tests, we significantly expanded the scope of our testing suite.

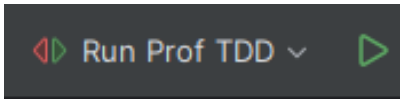
APPENDIX:

How to run the tests

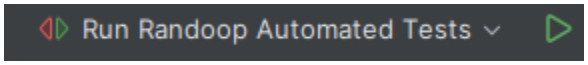
We are using JUnit, the tests can be found in the /public_tests folder, you can run the tests using a run configuration that is configured to run all the test files we created.



In addition, to run the TDD tests given by the professor (found in /public_tests_from_prof folder), you can trigger the configuration:



To run the automated tests created by randoop (found in /randoop folder), you can run this configuration:



Note: You may need to go into File -> Project Structure -> Modules and make sure /public_tests, /public_tests_from_prof and /randoop are configured to be test folders.

Additional Dependencies

- Added mockito for mocking classes during JUnit tests.
- Added Pitest for mutation testing
- Added Randoop for automated tests

PMD Justifications

Justifications of Ignoring PMD Warnings

NOTE: All Critical Violations are resolved, this table only justifies why we suppress certain warnings.

PMD Bugs Warnings	**Justification**
CompareObjectsWithEquals IOUtils.closeInputStream() And	PMD suggests we use equals() method to compare the inputStream and System.in instead of using == sign, but in our case, we are

IOUtils.closeOutputStream()	actually comparing the variable and the actual System stream, hence we will ignore this error.
AvoidDuplicateLiterals pmd violation	There is no getting around this violation since we have to create multiple tests, for example if I were to call echo in different tests, this violation will show up. In some cases where duplicate strings are used in different test cases, the team thinks this is a lot more readable than having a global variable at the top of the test case file where a developer has to constantly reference to see what string is contained in that variable.
CloseResouce PMD Violations in IOUtilsTest	The streams was mocked using mockito, and using mockito we already verified that mockedStream was closed. PMD is not able to detect and understand the mocked streams and hence this violation cannot be avoided. PMD is not able to recognize stdin.close() and throws the warning even though my input stream is closed.
CloseResouce PMD Violations in CallCommand.evaluate() And PipeCommand.evaluate()	It says the streams are not closed. It is already closed in the logic by calling IOUtils.closeInputStream and closeOutputStream. We think PMD is not able to detect this.
LongVariable warning	PMD is warning us for long variables, but these variables are not extremely long. It is longer because we want to better portray to the coders what they are used for. Hence this warning is ignored.
PreserveStackTrace	Stack traces are not preserved since exceptions are thrown. We are following the implementation of the existing exceptions, hence this warning is ignored.
Possible god class	Shown in CutApplication. We have only implemented the methods shown in CutInterface. Nothing can be done to avoid this PMD warning. All our commands comply with the single responsibility principle.

PMD Violations from the TDD files provided by professor	A ruleset was added in our PMD.Rules.xml to exclude the violations from prof provided TDD tests since these are not test cases we created.
PMD Violations from Randoop	These are suppressed since they’re automatically generated by randoop.