CS4218 Software Testing

AY21/22 Semester 2

Milestone 2 Report

Test-Driven Development (TDD)

Introduction

For Milestone 2, the team was given a set of public test cases by the teaching team. Various unit test cases for EF2 functionality were also created during Milestone 1. Using the set of test cases, the team was able to perform Test-Driven Development (TDD) when creating the EF2 functionality.

Process

The team followed the TDD process by first running the test cases provided by the teaching team and written by us in Milestone 1, then proceeded to write code to satisfy a certain feature. For example, the team starts by writing codes to throw an exception when a parameter provided in the method is null. We repeated this process until we could pass most or all the test cases. However, there are some instances where we modified or excluded the test cases to fit our assumptions since we did write the test cases initially without implementing the application. The test cases provided by the teaching team contain different assumptions by other teams. A non-exhaustive list of test cases modified or excluded can be seen below.

Running the test cases written in Milestone 1 and given by the teaching team also helps the team to uncover bugs that were missed out previously. A list of bugs that were uncovered through TDD can be seen here. These bugs are then resolved by the respective programmer in the team and the correctness is then verified through the oracle or test cases or changed if it does not fit the team's assumption. Bugs found are then documented and created as a GitHub Issue to improve collaboration and communication. Examples include Issues IORedirectionIT Bug #89 and Paste Bug #90. Through this, 100% of the test cases that were used passed when testing on Windows and Mac devices.

Bugs Found Through TDD

Location and Bug-type	**Test Input / Test Case that reviewed this error**	**Description**
LsApplication	`Ls emptyFolder emptyfolder2`	It was not displaying anything when there are 2 empty folder
PasteApplication	`paste a.txt b.txt` where a.txt has more lines than b.txt	A 1 B 2 C In an output similar to the one above, there should be an extra tab after C to be aligned with the start of the 2nd column
CatApplication	`cat file1.txt file.2txt` where file1.txt and file2.txt has empty content	In the results, a new line was not returned after the empty content

Sample Modified or Excluded Test Cases

Location	**Test case description**	**Reason for modification or exclusion**
CutApplicationPublicTest	Testing for exception thrown when calling `cutFromFile` method with invalid file input	Does not throw an error due to using `System.out.println` to replicate output behavior in Linux.
PasteApplicationTest	Testing for printing out an empty string by not providing any argument.	The output included an additional line break that is not present in the linux implementation.

LsApplicationPublicTest / LsApplicationTest	Test results that display files, separated by a space.	Our assumption for the Ls command display files, separated by a new line instead

Integration Testing

We have adopted a bottom-up approach for integration testing as we have tested and written the majority of the unit test during iteration 1. We focus on the interaction between different applications via commands, e.g., command substitution, pipe, IO redirection, etc. Since there are many possible combinations of component interactions to test in integration testing and testing all possibilities are not feasible due to potential combinatorial explosion. We have also adopted the strategy we learned and used in iteration 1, such as Category Partitioning and Pairwise Testing.

Pipe Command

While doing integration testing for the pipe command, we looked into these main areas:

1. Pipe with 2 Applications (1 Pipe)
2. Pipe with 3 Applications (2 Pipe)
3. Negative test cases
    a. Application does not exist
    b. Invalid options supplied to application

Although there are a total of 15 applications given for this project, not all of them qualify for testing within Pipe Command. As an example, the echo command cannot be used as the second or third app as it does not take in an input from standard input. As such, testing must be planned in a careful manner as to which application is suitable for the first, second or third application. Therefore, we have collated a list of applications with their respective input and output options (Refer to Appendix). Applications with both input and output can be placed in any position of the pipe while applications with an input and without output will be placed as the last application in a pipe.

Given this approach, we then decided to use combinatorial testing for Pipe with 2 Applications, and pairwise testing for Pipe with 3 Applications. As for negative test cases, non-existent applications and applications with invalid options were tested in all three different pipe positions. The relevant test suites have been attached to the Appendix.

Command Substitution

Following similar semantics as the Pipe Command's integration test, we wrote test cases for

1. Command Substitution with 2 Applications
2. Command Substitution with 3 Applications

Outer Command	Sub Command 1	Sub Command 2
echo	paste	tee
echo	grep	cut
echo	uniq	cat
echo	ls	echo
wc	cat	uniq
wc	paste	cut
…	…	…

Figure: Snippet of Pairwise Combinatorial Testing for Command Substitution Integration Test

The test cases are developed based on the compatibility of the applications (e.g., LS + WC are not compatible with command substitution as there are numbers output by WC). The test cases also emphasize the negative behavior, e.g., the behavior for failing on the outer command and failing on the sub-command should be different.

Tools

Jacoco

In Structural Testing, the team learns about the different types of code coverage available. Based on the Subsumption Relation Figure provided in the lecture, the team understands that Branch Testing subsumes Statement Testing. Although the team focused on Statement Testing in Milestone 1, the team wanted to improve our testing efforts and decided to focus on Branch Testing instead.

The team made use of Jacoco to make sure that we achieve an overall branch coverage of at least 80%. Results of our testing efforts are summarized in the table below:

Application	Method Coverage (%)	Line Coverage (%)	Branch Coverage (%)
CatApplication	100	96	97
CdApplication	100	96	95
CpApplication	100	94	91
CutApplication	100	97	89
EchoApplication	100	86	100
ExitApplication	50	50	100
GrepApplication	100	98	98
LsApplication	100	92	96
MvApplication	100	90	95
PasteApplication	100	99	96
RmApplication	100	92	88
SortApplication	100	96	95
TeeApplication	100	97	90
UniqApplication	100	95	96
WcApplication	100	96	96

Othey Key Classes	Method Coverage (%)	Line Coverage (%)	Branch Coverage (%)
app/args	99	95	94
cmd	81	94	90
parser	91	94	91
util	98	95	89

Evosuite

To boost our test quality and branch coverage, the team used Evosuite to generate additional test cases for all our application implementations. Evosuite was also used to generate test cases for other classes with low coverage.

Making use of Evosuite improves our overall coverage as summarized in the table below.

	Method Coverage	Line Coverage	Branch Coverage

Without Evosuite	93% (277/296)	92% (2007/2172)	90% (1116/1238)
With Evosuite	94% (281/296)	94% (2050/2172)	92% (1147/1238)

Evosuite undoubtedly helped the team include comprehensive test cases and improve test coverage for certain classes. However, the team kept our previous test cases in the project as we felt that there were some limitations in using Evosuite and did not generate more tests from it as most of them were not as helpful.

Based on the team experience, Evosuite limitation includes:

- The test cases do not understand the requirements of the application
    - For example, the test case generated do not know that running the MvApplication should move a file from one the given source to destination
    - Hence, the test cases generated are mostly based on the return values of the method only
- Increasing the time to generate the test cases does not necessarily translate to better coverage
    - Generating the tests with the recommended command for CutApplication only yields about 50-60% coverage. Even when running the command with `-Dsearch_budget 600` flag, it still generated the same amount of test cases with the same coverage.
    - Hence, even when increasing the search time for generating tests, it did not increase our test coverage.
- Evosuite might create test cases that are not dynamical enough
    - For example, it created a test case for LsApplication in which it checks directly for the string of the current directory of the user that runs evosuite. Hence when passing this over to the other teammates, this test case will definitely fail.
    - Test cases that pass on an OS might not be able to pass on a separate OS.
- Evosuite's test cases cannot consider the "deeper" edge cases
    - For example, in Paste, the test cases generated do not consider the content of the files that are being pasted. So, key behaviors of paste that involve pasting files with content of different length will not be tested. Hence, we have to manually create those test cases.
- Increases the overall testing runtime significantly
    - For example, if our test cases are run without Evosuite's test cases, it takes 2.6 seconds to complete. With Evosuite's test cases, it doubles the runtime to 5.31 seconds. This is presumably due to the overhead in setting up Evosuite's test scaffolding.

The team was not able to find any bugs through generated test cases by Evosuite as well. Hence, the team still continues to perform manual testing to ensure that our test cases are effective in detecting buggy implementation.

Pit Mutation Testing

The team decided to use Parallel Isolated Test (PIT) mutating testing introduced in the lab session to increase our confidence in the test cases we wrote.

We decided only to run PIT mutation testing only for our applications due to the amount of time needed to run this test. However, we ran into some problems where test cases were failing even when no mutations were introduced, so we decided to exclude cat and cp application from the mutation testing as it was causing those problems. In the end, we did manage to run the PIT mutation testing and generated the report for it. We did not further improve our test coverage due to the time limit and were also satisfied with our mutation test coverage.

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
73	90%	8480/9382	36%	1265/3498	38%	1265/3322

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
sg.edu.nus.comp.cs4218.impl.app	52	93%	7244/7831	38%	1067/2828	39%	1067/2723
sg.edu.nus.comp.cs4218.impl.app.args	11	97%	907/933	35%	181/519	35%	181/519

Appendix

Acceptable PMD Violations

PMD Violation	Explanation
Closure Resources	● Overall, there are various PMD issues regarding CloseResources that were suppressed. ● This is largely due to using an external util method to close the resources which PMD does not recognize.
Long Variables	● Found in the ErrorConstant class. Long and descriptive constants help the programmer understand the error easily. Hence, the team decided to keep the constant naming, prioritizing understandability.
Excess Method Length	● Found in classes such as ArgumentResolver, CommandBuilder, GrepApplication, RmApplication, ApplicationRunner, and IORedirection. ● **ApplicationRunner**: As we introduce more applications, it is inevitable that the length of ApplicationRunner's switch statement increases and leads to the runApp method getting excessively long. To ensure that it continues to work with the given interfaces, we will not make any significant modification. ● **Others:** Results due to the interface methods given or the method are rather complex, and many parameters are required.
Class Naming Conventions	● Found in CommandBuilder and Environment, the class name given seems intuitive and descriptive which the team feels is reasonable
God Class	● Despite the team's best efforts, the team feels that the complexity in the classes is high, and having a larger class is reasonable. ● **CutUtilsTest:** The test file requires a relatively large amount of constants due to repeated usage of certain values for testing and a large number of methods to test the methods of a class. ● **GrepApplication:** The original skeleton (with TODO) had already triggered this warning. However, an attempt to refactor the methods persisted with the warning. Since we are not allowed to remove any predefined method from the skeleton codes, usage of auxiliary files such as GrepArguments introduced even more PMD warnings due to redundant assignments and repeated logic that violates Software Design Principles. ● **PasteApplication**: The necessary process to merge the files and stdins into one output is fairly complex, resulting in methods being fairly long. In order to keep the class readable, we have extracted certain operations of the methods into smaller methods which increases the length of the class and the number of classes. Even though this triggers the god class error from PMD. We believe that it is reasonable to prioritize readability of the methods.
Avoid String Buffer Field	● Found in RegexArgument, using string builder gives the developer more flexibility when dealing with strings.
UseVarargs	● Using string array instead of varags allows for better flexibility to the developer and the team feels that it is easier to be used.
PMD violations from Evosuite generated tests	● As mentioned in the forum, it is not required to resolve PMD violations from auto-generated tests. ● Forum post: https://luminus.nus.edu.sg/modules/efe37db4-83fb-4090-a0d4-c86d3cdba61c/forum/categories/b371264e-d699-409e-8a20-bfbc923bd2ca/d4405086-cd6f-4fa1-ba73-ce5cf9a1e4ac

PMD violations from TDD test cases by teaching team	• As mentioned by the teaching team, it is not required to resolve PMD violations from the TDD test cases provided by the teaching team.

Pipe Command: Applications and Input/Output

Application	**Input**	**Output**
echo	N	Y
ls	Y	Y
wc	Y	Y
cat	Y	Y
grep	Y	Y
exit	N	N
cut	Y	Y
sort	Y	Y
rm	N	N
tee	Y	Y
cp	N	N
cd	N	N
paste	Y	Y
uniq	Y	Y
mv	N	N

Pipe Command: Application with 2 Pipe (Combinatorial)

Application 1	**Application 2**
ls	ls
ls	wc
ls	cat
ls	grep
ls	cut
ls	sort
ls	tee
ls	paste
ls	uniq

wc	wc
wc	cat
wc	grep
wc	cut
….	…..

(total of 90 test cases)

Pipe Command: Application with 3 Pipe (Pairwise)

Application 1	**Application 2**	**Application 3**
ls	ls	ls
ls	wc	wc
ls	cat	cat
ls	grep	grep
ls	cut	cut
ls	sort	sort
ls	tee	tee
ls	paste	paste
ls	uniq	uniq
wc	wc	cat
wc	cat	grep
wc	grep	cut
wc	cut	sort
wc	sort	tee
wc	tee	paste
….	….	….

(total of 97 cases)

Quoting: 2 sets of quotes (Unnest and Nested)

Application	**Argument1**	**Argument2**	**HasInvalidArg**	**HasNesting**
cat	Backquotes	Double quotes	FALSE	TRUE
cat	No quotes	Backquotes	TRUE	FALSE
cat	Double quotes	Single quotes	TRUE	FALSE

cat	Single quotes	Double quotes	FALSE	TRUE
cd	Single quotes	Backquotes	FALSE	TRUE
cd	Backquotes	Single quotes	FALSE	TRUE
cd	No quotes	Double quotes	TRUE	FALSE
cd	Double quotes	No quotes	FALSE	FALSE
cp	Backquotes	Double quotes	FALSE	TRUE
cp	No quotes	Single quotes	TRUE	FALSE
cp	Single quotes	Double quotes	FALSE	TRUE
cp	Double quotes	Backquotes	FALSE	FALSE
cut	Single quotes	Double quotes	FALSE	TRUE
cut	Double quotes	Single quotes	FALSE	FALSE
cut	Backquotes	Double quotes	FALSE	TRUE
cut	No quotes	Backquotes	TRUE	FALSE
grep	Backquotes	Single quotes	FALSE	TRUE
grep	Single quotes	Backquotes	FALSE	FALSE
grep	No quotes	Double quotes	TRUE	FALSE
grep	Double quotes	No quotes	FALSE	FALSE
ls	Single quotes	Double quotes	FALSE	TRUE
ls	Double quotes	Backquotes	FALSE	FALSE
ls	No quotes	Single quotes	TRUE	FALSE
ls	Backquotes	No quotes	FALSE	FALSE
mv	No quotes	Single quotes	TRUE	FALSE
mv	Double quotes	Single quotes	TRUE	FALSE
mv	Backquotes	Double quotes	FALSE	TRUE
mv	No quotes	Backquotes	FALSE	FALSE
…	…	…	…	…

(Total of 67 test cases)