Milestone 2 Report

Implementation Plans
Sticking to our Milestone1 implementation plan, we carried out Milestone 2 in mini-sprints, with each member in charge of the integration test for the applications and shells as outlined below.
Project Details and Timeline:

	BF1		**EF1**		**EF2**	
	Shell	**App**	**Shell**	**App**	**Shell**	**App**
Member 1		exit, wc	Globbing	tee	Cmd Sub	rm
Member 2		ls, echo	Pipe	mv	Semicolon	paste
Member 3	Quoting	cd		cat	Cmd Sub	cp
Member 4	I/O Redirection	grep		split	Semicolon	uniq

Milestone 2.1 (15th March Monday 6pm)
• Implementation of EF2 Applications and shell
Command substitution, Semicolon operator, Uniq, Rm, Paste, Cp
Milestone 2.2 (19th Thursday 6.30pm)
Integration and System tests
Milestone 2.3 (20th Saturday 2.30pm)
EF1, (Evosuite and Randoop)
EF2, (Square Test)
Automation frameworks implementation
Milestone 2.4 (21st March Sunday 4pm)
Assumptions and report
PMD + IntelliJ, JUnit style
Testing Plans
TDD Process and experiences using the test cases from other teams:
Using the TDD test cases we created prior in milestone1 and the ones we received from the teaching team, development of EF2 features was faster compared to milestone1 as most of the positive, negative and boundary values have been considered. When a test case failed, we were able to quickly identify the source of the error, be it being from stdout, stdin or input.

In addition to using the TDD test cases provided, we tested for statement, method and path coverage to detect which areas were not covered by TDD test cases. Using these statistics, we came up with more test cases to cover these areas. We considered both positive, negative test cases including exceptional cases, corner cases and cases for error handling that are prone to errors based on our intuition.

As other teams had differing assumptions for the development of our EF2, some assumptions were similar while some of the test cases were against our general assumptions and were not implemented. Our team first read through the test cases and “cleaned” those; either by modifying the test cases to fit our assumptions, or removing them if they were not applicable.

For example, acceptance of consecutive flags. Our team had discussed and decided as per the project description, each flag had to be prefixed with a '-' char and would contain at most 1 flag; meaning -cdD for Uniq would not be accepted. After receiving the test cases, we discussed if decided against following the tdd test cases which accepted consecutive flags for uniformity across our project.

Regression on Milestone1:
The TDD test cases provided were also useful in conducting regression testing on the milestone1 functionalities. This helped us to validate our implementations for milestone 1 before we started integration testing.

Being exposed to other teams' test cases was definitely helpful in general as they helped us to identify any possible edge cases we may have overlooked in the prior milestone and in our own implementation of test cases.

To reduce redundancy, our team merged the test cases we wrote in Milestone 1 with those shared with us in the tdd. Moreover, the given tdd folder to tdd_regression and our program is able to pass all the listed test cases within.

Integration Testing - Plan and Execution:
From milestone1, we had already generated some integration test cases (found in our TEST_CASES file) which came in useful for the integration testing in milestone 2.

The plan for the integration of our components was that we would test an application with at most one shell unless another shell function was needed to generate the input required for the test. We ensured that for every application, we tested them with every relevant shell feature, this ensured that every pair of shell features and application were thoroughly tested.

If an integration test required another application (i.e. pipe, IO-redirection, command substitution), we would use applications that would have minimal impact and not mutate the input such as APP_CAT and APP_ECHO. This was possible as our team followed a bottom-up approach of completing the applications before shell in milestone1.
For example, we would pair the grep command with:

- Quoting
    - Single Quote
    - Double Quote
- Command Substitution (Back Quote)
- Pipe
- Sequence Command (SemiColon)
- IO-redirection
- Globbing

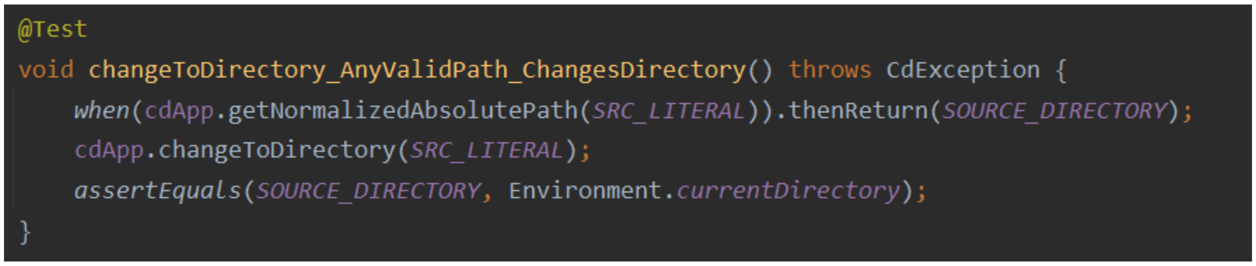
For all of our relevant test cases, we automated the creation and deletion of sample files which were used for testing to reduce human errors and to ensure consistent test results regardless of the number of times we have executed a test case, or the order in which we execute them.

Application of Test Generation Testing Tools and Techniques

To supplement our manually crafted test cases, we further leveraged on 4 additional automated testing frameworks to generate test cases for us. The testing efforts are documented as follows:

1) <u>Mockito</u>

As showcased in our lab, we leverage on the Mockito library for automatic spying and mocking to isolate the SUT such that any bugs found can solely be attributed to the SUT - the core essence of unit testing. An example usage is illustrated below.

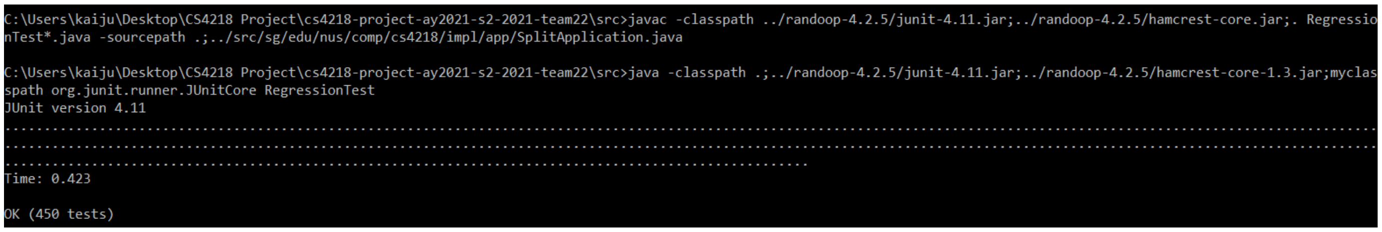


```java
@Test
void changeToDirectory_AnyValidPath_ChangesDirectory() throws CdException {
    when(cdApp.getNormalizedAbsolutePath(SRC_LITERAL)).thenReturn(SOURCE_DIRECTORY);
    cdApp.changeToDirectory(SRC_LITERAL);
    assertEquals(SOURCE_DIRECTORY, Environment.currentDirectory);
}
```

As seen above, to isolate the unit during unit testing, we mocked the behaviour of the getNormalizedAbsolutePath() method to return a predefined string (SOURCE_DIRECTORY) upon its calling. The Spy class was used to achieve so.

2) <u>Randoop</u>

We used Randoop for the generation of automatic test cases for EF1 applications. The commands used to generate, as well as the output of running these test cases, can be found in the randoop/screenshots directory. The generated test cases are also included in the randoop/test-files directory, amounting to more than 35,000 lines using the default Randoop test setting.



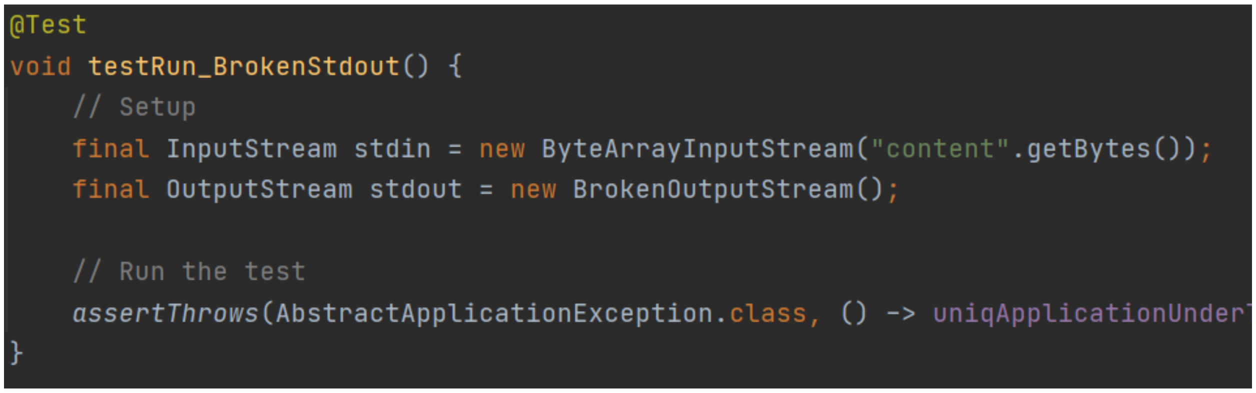
```
C:\Users\kaiju\Desktop\CS4218 Project\cs4218-project-ay2021-s2-2021-team22\src>javac -classpath ../randoop-4.2.5/junit-4.11.jar;../randoop-4.2.5/hamcrest-core.jar;. RegressionTest*.java -sourcepath .;../src/sg/edu/nus/comp/cs4218/impl/app/SplitApplication.java

C:\Users\kaiju\Desktop\CS4218 Project\cs4218-project-ay2021-s2-2021-team22\src>java -classpath .;../randoop-4.2.5/junit-4.11.jar;../randoop-4.2.5/hamcrest-core-1.3.jar;myclasspath org.junit.runner.JUnitCore RegressionTest
JUnit version 4.11
...............................................................................................................................................................................
...............................................................................................................................................................................
....................................................................................................
Time: 0.423

OK (450 tests)
```

Above is an example test run of the automatically generated test cases for the Split application. Overall, Randoop did not produce any error revealing tests for us. We note that these test cases are computationally expensive to run, so we only intend to run them once before every release to ensure we do not break any existing functionalities.

3) <u>Evosuite</u>

Our team utilized Evosuite for the generation of automatic test cases for EF1 applications as well. Similar to the labs in class, our team utilized evosuite-1.1.0.jar, which only supports Java 8 in full (since Java 9+ support comes with a bit of caveats), run time environment 52 to generate the test cases for the respective EF1 applications. Unfortunately however, we encountered dependencies issues and were unable to run the generated test cases. We resorted to manual observation to identify test cases that we missed out, and included them into our own set of test cases. The generated test cases can be found in evosuite-tests directory in the project root.

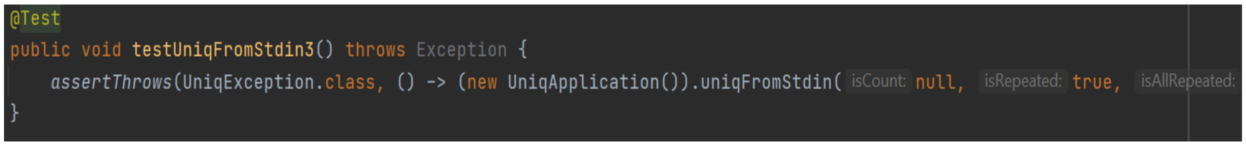
4) <u>SquareTest</u>



```
@Test
void testRun_BrokenStdout() {
    // Setup
    final InputStream stdin = new ByteArrayInputStream("content".getBytes());
    final OutputStream stdout = new BrokenOutputStream();

    // Run the test
    assertThrows(AbstractApplicationException.class, () -> uniqApplicationUnder
}
```

We used SquareTest to automatically generate test cases for EF2 applications. All the test cases which are generated are merged together with our original test files in [App]ApplicationTest files. SquareTest effectively presented error-revealing test cases to us. The tests produced simulated null and broken streams in stdin and stdout, that allowed us to program our project defensively and omitted null checks on inputs. These inputs would have rendered us NullPointerException or IOExceptions had we ignored them.

5) DiffBlue Cover



```
@Test
public void testUniqFromStdin3() throws Exception {
    assertThrows(UniqException.class, () -> (new UniqApplication()).uniqFromStdin( isCount: null,  isRepeated: true,  isAllRepeated:
}
```

This was a testing framework our team discovered and ran on our EF2 class that auto generated tests cases. Similar to SquareTest, the cases were merged with our original test files. DiffBlue Cover generated tests for methods which would set the flags as null or where the combinations should error. This helped to catch possible NullPointerExceptions and prevent possible invalid combinations being called by the methods.

PMD rules

God class: Possible God Class

Implementation of TODO in GrepApplication had already triggered this warning. Refactoring private methods to a Utils file and using GrepArgs still triggered PMD. Triggered PMD after refactoring file to be 4 methods (run, files, stdin and files&stdin). Violation of rule is also not clear on how to resolve, even with googling.

ClassNamingConventions:

Environment class violating but, environment does not act as util and does not require helper or Constants.

PreserveStackTrace:

Application specific exception messages require new exceptions to be thrown causing stack trace to be lost.

Close Resource:

In order to solve this PMD, we would need to close the opened streams whenever an exception is caught and thrown. As this project is IO intensive in each method, this PMD rule would conflict with the ExcessiveMethodLength PMD rules.

Libraries Used:

JUnit, Mockito (for stubbing), System-lambda (for testing of output)