CS4218 Milestone 2 Report

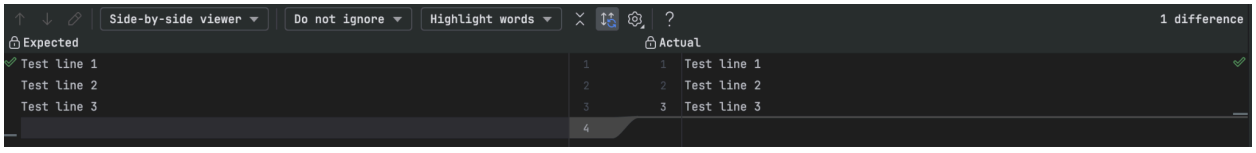
TDD Testing Process

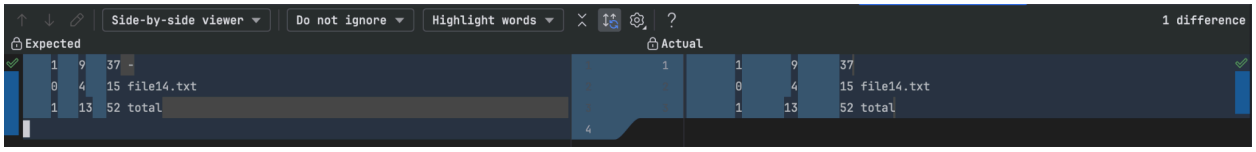
Writing Tests

Usually in TDD, we would write tests to define how an application should behave according to its requirements. Although there were some test cases written for EF2 in Milestone1, the majority of the test cases discussed below will be the public test cases received by the teaching team. These test cases are expected to fail as the requirements given for the features were not very specific and some specifications are left for the team to decide such as output formatting and error handling. This highlights the difficulties in translating feature requirements to actual code implementations and shows that this step is a very important step in TDD.

Implementing Functionality & Finding Bugs

The first thing that we had to do with the provided public test cases for TDD was to filter through the false negative failing test cases. We performed a manual verification on the discrepancies in outputs for failing test cases, accounting for the assumptions that we made. Majority of the output were additional line separators, or tab separators, as shown below, which were due to different assumptions. For instance, in the TeeApplication, it's presumed that the content written to files does not conclude with a newline character, leading to adjustments in the corresponding test cases. Moreover, attempting to write to an unwritable file triggers an exception, as opposed to executing the write operation and displaying the results.



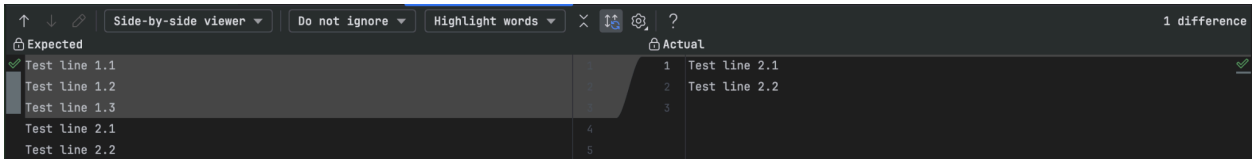


We also handled some erroneous inputs differently from the public test cases, which resulted in some of the failures of the test cases. We deem these to be false alarms and choose to ignore these test cases. These assumptions on error handling are documented in Assumption.pdf separately. We used bash as our default reference to resolve the differences in output.

There were also cases whereby the test cases were not possible due to our way of implementation. For example, one of the test cases for CutApplication had a null input stream for the stdin. However, this was

not a reachable case in our implementation since we would have thrown an exception for a null input stream from the run method inside CutApplication.

However, by using the test cases provided by the teaching team, we did uncover some bugs, for example in CatApplication.



`catFileAndStdIn` is supposed to read from both file and stdin, rather than just one of them, which was the wrong assumption that we had made. Hence, we made the fixes and added regression tests for it.

Another bug we found using the test cases provided by the teaching team was in the `MvApplication`. Initially, our `performMoveOperation` method did not consider the edge case when the destination file has unwriteable permissions. Fortunately, these two test cases `run_UnwritableDestFileWithoutFlag_ThrowsException`, `run_UnwritableDestFileWithFlag_NoChange` assisted us to discover these edge cases.

We apply the same steps to our own crafted TDD tests. In general, most of the test cases were passing, but we had to fix several of the failing test cases due to the wrong assumptions made, as well as several miscalculations when crafting the test cases, which led to a wrong expected output. But overall, our team deems the test driven development to be highly effective. This is because it eliminates developer bias, since we do not write our own test cases, which was very helpful in uncovering edge cases. The TDD tests also served as a checklist against the requirements of the application and uncovered any missing functionalities in our features.

Integration Testing Plan And Execution

Our testing plan closely resembled the bottom up sandwich approach that was taught in the lab. This approach blends the bottom-up approach's incremental nature with the high-level perspective of the top-down approach. We decided that this was a good approach since the modules are decoupled from each other, which allowed us to test individual modules in parallel.

For instance, an example of the bottom layer tests would be the *ParserTests, since the parser parses the argument first, before feeding these arguments into the respective applications. On the other hand, the top layer tests would be the higher level tests such as the CommandBuilder tests and PipeCommand tests.

Pairwise testing is a powerful technique that was integrated into our testing plan to ensure comprehensive coverage of interactions between input parameters while optimizing the efficiency of our testing efforts. This method proved particularly valuable in our evaluation of the WcApplication, given the diverse array of flag permutations.

Before commencing on any integration testing, it was important that we verified all related unit test cases have been thoroughly tested and any identified bugs have been fixed. As mentioned above, we started with integrating the lowest level modules that have any dependencies. These were typically utility functions. These utility functions were then used for each application's unit and integration testing. Further integration testing was done with the use of CallCommand, PipeCommand and SequenceCommand to test the overall functionality with handling multiple application commands. During integration testing, we constantly executed regression tests after fixing bugs to ensure new defects have not been introduced during integration. Lastly, this is followed up by system testing to validate the overall system functionality from receiving shell input to output.

Automatic Testing Tools

We have tried using **Randoop**, documented in randoop.sh script, generating a total of **974** test cases. We observed that the generated test cases were generally not very readable and could not identify the intuition behind the test cases. It was also very messy as each test case tested multiple applications at once. Even though Randoop was able to generate a large number of test cases, the majority of the assertions were not very meaningful and did not offer any deeper insight for our testing.