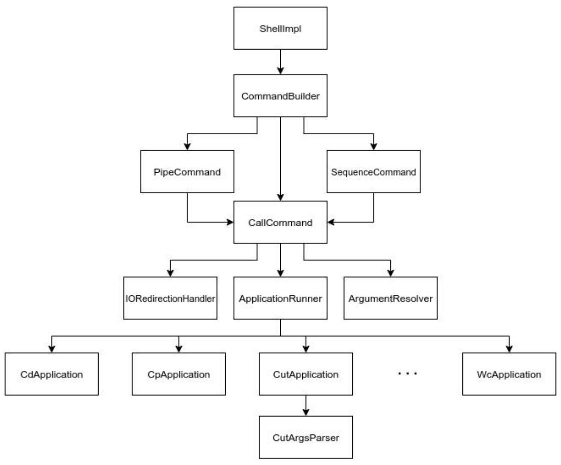
CS4218 - Milestone 2 Report

1. Test-Driven Development Process

- We split our work such that each member of the team was responsible for testing the applications they worked on previously. This allowed us to speed up the process as we could tap on deeper understanding of the commands we had implemented or tested before. We took the TDD suites and analysed each test case, classifying them into one of three categories: passing, bug, and implementation difference. We then opened issues on GitHub for each of the known bugs, and got to work fixing them.
- We found that some teams assumed features beyond what the specification required. One example was in the `rm` command where there were numerous tests related to file permissions. This cost us a lot of time, as we had to go through each test case, and assess whether it was failing due to a bug, or due to the different assumptions of the implementation as we did not want to overlook any potential flaws being exposed. In other cases, such as `exit`, there were also simple implementation differences that rendered the tests unusable.
- It was also clear from the TDD test cases that we had overlooked certain scenarios in our test suites, so the coverage increased. For test cases that did not fit our implementation but were still meaningful and not present in our initial suites, we either modified them to fit our implementation, or disabled them and implemented our own versions, depending on what was less costly.
- Where test cases were disabled, a justification is present in the string passed to the annotation.
- The given test cases were particularly useful to debug certain commands, such as `paste`, where many modifications were made before we felt confident that we had a reasonable implementation with reasonable assumptions, with progressively more and more cases passing.
- For invalid cases where exceptions are thrown it is useful to see what kind of error messages are being thrown and this improves our error messaging to be better for users by handling more exceptions and throwing meaningful error messages rather than letting inbuilt exceptions throw generic messages. This was particularly useful when Java functions like `File.move()` where used and many different exceptions could occur.
- By having a look at test suites of how other teams have written their code, this also helped us in finding better ways we could format and write our test cases, e.g. using mocking stubs, use of `@Display and @Nested` and also the coding style and formatting of test cases which we could also incorporate in our test suites and improve code quality.
- By analysing their test code, we could gain inferences on their functionality specification and compare with our implemented ones to understand which is better. Also it helps find bugs faster and allows the design to evolve and adapt to your changing understanding of the problem. Through TDD we could also know what kind of dependencies it would have and then effectively mock them to make our unit tests isolated and specific.
- Having more TDD test cases gave us confidence in our test suite as any changes in implementation or software could be tested to prevent any regression and ensure the new code works equally well.
- For the implementation of EF1, more specifically of the commands `cp` and `diff`, we based our code on what the tests required, including the tests we had previously written and the ones provided. This was helpful in the sense that it made it very clear what a correct and complete implementation would need to fulfill.
- Since Globbing was part of EF1, we had written tests in MS1 which failed due to bug and TDD also failed, however on understanding specs of TDD, we could easily point that the bug was in relative path like `dir/*` and point to the exact line where error occured due to an empty result. This was a huge benefit of TDD as it was quick and easy to spot bug and understand why it failed as well.
- In hindsight, this helped us understand how TDD (or TFD) can be used alongside or even instead of a requirements document. This would allow for the possibility of having someone familiar with business requirements writing TDD tests that are passed to the development team for potentially better and quicker validation and satisfaction of business needs.

2. Integration Testing



The main aim with unit testing was to verify if dependent classes worked well with each other and testing the interfaces between components. Also not all specifications of software can be tested through unit testing only, hence having functional integration tests ensure acceptance tests as well. This step was done after unit testing and was preceded by System testing. Having a good understanding of component dependencies and data/control transfer between the components. We used two different methods to integrate the components which are explained below. Since we had to choose good test cases, we explored how different components can fit together functionally and structurally.

2.1. Structural Integration

- We decided to start with a bottom-up approach for integration testing. Based on our understanding of the shell and the interaction of different modules with each other, we derived the above diagram, and based on it, we added tests somewhat incrementally.
- We already had unit tests for the individual applications and utilities like the IO redirection handler (where JUnit acted as the driver). If we were following a strict bottom-up integration we would then test from `ApplicationRunner` but we considered that this would not add much value to our test suite, as the interaction with that class was trivial, providing very little return for the extra effort it would require. For this reason, we passed this and wrote some tests at the level of the classes implementing the `Command` interface.
- The next step in strict incremental integration would be to test from `CommandBuilder` but once again we skipped into `ShellImpl` directly. This was done because we considered that it would not significantly ease fault localization, and so once again we would not get much return for the extra effort.

2.2. Functional Integration

- Once we were done with the structural approach to integration, we decided to test the interaction of different command applications with each other. For this purpose, we tried to come up with different test cases using the shell operators (globbing, quoting, piping, etc).
- In this phase we made an effort to include corner cases and negative cases for these tests, where for example one of the commands in a pipe fails, aiming to fulfill the suggestions presented in the lab session about integration.
- When it came to testing different commands together we encountered the difficulty that each of us had somewhat specialized in the commands they had tested or implemented before. This ended up setting us back slightly as we had to regularly refer back to the project specification and test cases to clarify how other commands could and should be used, in order to create the interactions. However, this also helped

each team member understand the system as a whole better and find bugs that the person originally working on the command would not have noticed.

- At a final stage we created some cases attempting to combine as many commands and shell operators as possible. This led to us finding a few bugs that although not related specifically to integration were not detected by the unit tests.

2.3. General Considerations

- This approach turned out to be really helpful as it helped us in finding bugs which we were not able to find through our Unit Tests. One of the main bugs we found this way was in command substitution. When we only unit tested Command Substitution we went with mocking and assuming Echo Application is bug-free. This allowed us to test functionality of command substitution like nested, multiple and ivalid substitutions could be tested. However, when we started doing integration tests with other commands which for e.g. gave outputs in multiple lines and had to be determined on how to tokenize, our integration test could spot these bugs and lead us to which part of code needed to be fixed. Being integration tests, this pointed us to the module ArgumentResolver, a dependency of command and shell. This helped in finding the specific place where the fault was to be found.
- Having a thorough test suite ensured us that when bugs were being fixed we could trust it to find regressions, and so it enabled us to move quickly and not be too afraid of breaking things, as we could count on the tests to alert us of most mistakes that were made.
- We found it, at times, rather difficult to come up with meaningful cases that would target the right interactions between applications and shell operators. It was also hard to judge if some of the test cases being written would actually be helpful, as this application allows for an extremely large number of interactions among modules, but many of them are of the same type, meaning the same kind of interfaces are used.
- 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.

3. Automated Testing Tools

- For Automated Tests, we experimented and used the following two tools: SquareTest and EvoSuite. Due to limited time, we only used PIT to see how mutation testing worked but could not exploit it any further. However, it did assure us of the thoroughness of our test suite, since we could see quite a few test cases failing in a limited number of runs.
- PIT was tougher to set up and configure as test states were dependent on each other and it was not always the cause of mutants. Having a Testing framework like NGTest would have proven to be useful as it provides a good framework to run specific tests with forking. We mainly worked heavily with EvoSuite and SquareTest for automated test generation.
- Using SquareTest did not help us much in finding bugs or increasing coverage. It helped in generating test cases for possible functions and paths; however, most of them failed and had to be modified heavily based on functionality to actually make them useful. It was also not able to decipher the dependencies and mock them. This would have been useful for automated generation of unit tests in the previous milestone but now that we had already done good functional and structural testing it did not provide us with any benefits.
- With EvoSuite we had a lot of good takeaways as we were able to find bugs and also easily generate unit test cases for trivial classes like ArgsParsers, `CommandBuilder`, `IOUtils`, and `StringUtils`. Moreover, they also served as good regression tests as when we updated implementation some of these tests failed. Even though some were a result of poor test assertions which were hardcoded they helped in understanding how evolving software caused our auto generated tests to fail due to regression.

Bug	TestGenerated	Fix	Remarks

Providing <=1 arguments in `MvArgsParser`	`MvArgsParser_ESTest#test11()`	Condition check before doing sublist to see if it has enough elements	`mv one-file`
Invalid Regex in `RegexArgument`	`ArgumentResolver_EsTest#test03()`	Handling via `PatternSyntaxException` instead of null exception message	`sed s|[|b| file.txt`

- We tried creating Unit tests for all possible classes and kept those which helped in achieving coverage or finding bugs in our test package. Most of these can be found in the test.automated package. However, since a lot of effort was required to make them compilable and get them running only those which seemed useful and modified were kept. It also helped in adding some random input test cases especially for `RegexArgument` and `ArgumentResolver` as it required a lot of effort to generate regex with all possible scenarios which we felt would be good regression tests useful when further implementation is evolved for the software.
- All generated tests with Evosuite were pushed to package `test.automated` in the package structure like src. More than 200 test cases were generated taking less than 1 minute for each class. Since these required dependencies of Evosuite and its dependency jars which we could not add, an effort was made to modify the tests for Junit and the rest were commented out. The virtual file system provided by EvoSuite seemed very useful for testing software like a shell, but unfortunately we could not exploit it.
- Out of the ones that could work without dependencies, we chose the ones we thought were useful and fully integrated them with our test suite. The heuristic used was that it should cover a new path, increase coverage, expose structural bugs, be a good regression test, reduce explosion, consider randomness of input and based on value addition. It helped us achieve 100% class coverage and 100% coverage for all util classes. Having hybrid testing with our functional and auto-generated structural tests gave us a good unit test suite which helped increase statement coverage by 8% and find boundary case bugs.
- Test cases in our suite with the name `test{num}` are ones taken directly from automated tools. These mostly increased coverage but also helped a lot in finding `NullPointerException`, `ClassCastException`, `InvalidInputException` (for parsers) among others, and finding the paths which handles boundary inputs, for instance in `LsApplication`, and `RegexArgument` (Find Parser). An issue faced here was that Evosuite always generated passing test cases so each had to be manually inspected to understand the underlying root cause. Many bugs were null checks or interface bugs like in IO redirection, globbing, or parser's edge cases.

4. Appendix

- We have a 100% class coverage, 98% method coverage and 97.2% statement coverage. The uncovered lines are mostly streams and IOExceptions, OS related code, file permissions and we didn't want to generate an enormous number of invalid cases. We have over 400 unit tests, 140 integration tests and system tests for end to end testing. The coverage webpages can be found in `MS2-Cov`.
- `test.sg.*` => Our main test suite for unit tests [has some chosen automatically generated tests and some inspired from TDD]
  - `test.automated` => Generated by EvoSuite
  - `test.integration` => All integration tests using two different integration strategies
  - `test.system` => System test making direct call to ShellImpl main using stdin
  - `test.tdd` => Provided TDD test cases modified and disabled (reasoning provided) so all pass
- When JUnit runs multiple test classes, the test methods from different classes are interleaved. Since we are using the Environment class to store the current directory as a static field, this causes several errors to occur, so we have just been running each test class separately to avoid it. For this, we have already set up a configuration `All Test Class Forking` in IntelliJ which can run the tests sequentially. Since this is a high level report, more details can be found in `NOTES.md`.