Clayton Wyman

Professor Toledo

CS-320-T1181 Software Test Automation & QA

October 16, 2022

Project 2

In order to test each of the three features (add, delete, and update), I used return cases to make testing with JUnit tests far easier than using an inputStream and a printStream. By using return cases in this manner, I could test each feature directly, rather than having to manipulate them through automating command line inputs. I believe these tests were also entirely in line with the assignment requirements. The base classes (Contact, Task, and Appointment) were all tested using JUnit to ensure their variables all worked with the limitations set forth by the requirements. As such, each variable had a JUnit test that checked for values outside of the accepted range, either values longer than the string’s acceptable length, or dates before the current date. In addition, this test would also test null values to ensure they threw properly, as outlined in the requirements.

In my testing of each individual feature of the drivers, each test tested both the valid return and invalid return results of each function. In addition, there were different functions for updating each field when updating was an expected requirement, as while this required writing several more functions and tests, it made the overall program more concise both in use and testing. This overall system made testing coverage easier, as tests could be written to only test the function, and therefore made testing individual pieces far easier. Between the simple testing for base classes, and the concise testing using return values, this created an overall effective way to write tests that resulted in having good test coverage for much of the program.

Using JUnit tests ensured my code was technically sound by ensuring it reacted correctly to both valid and invalid input. Code like **Assertions.assertThrows(IllegalArgumentException.class, () -> {new Appointment("12345678901", "Zoom.", date);}); Assertions.assertThrows(IllegalArgumentException.class, () -> {new Appointment(null, "Zoom.", date);});** ensures that my code is technically sound by ensuring that invalid values are thrown, in this case being an id that is too long or null. Using separate functions for updating different fields ensures my code is efficient by reducing the amount of “if else if” decisions required in making my program function correctly, and avoids requiring input while inside of a function.

The main techniques I used for testing my program were input partitioning, statement testing, and decision testing. Input partitioning is fundamental to testing what values are valid and invalid. Therefore, by testing what strings are too long, what dates are too early, and if the system correctly deals with null values, I used input partitioning in my testing. Boundary value testing is a part of input partitioning that I neglected to utilize in my testing, as while the strings I tested were only just above the allowed length, I didn’t use any strings that matched the allowed length in testing, or were just under it. Therefore, I personally consider boundary value partitioning a form of testing I should have utilized during my testing.

Statement testing was utilized heavily in all testing, as it’s defined as testing that focuses on executing as many lines of the program as possible. Therefore, by testing all possible functions and their results, there were no lines of the program that were missed while testing. As such, this qualifies as a form of statement coverage for my testing. In addition, this involved utilizing decision testing and coverage, as in order to test as many lines of the program as possible, I also had to test as many decisions as possible. By testing both success and failure for the different functions, I effectively utilized decision testing, and achieved full decision coverage.

Input partitioning is a standard part of testing in almost all projects, as most programs require some kind of input, and therefore can have both valid and invalid inputs. This means that some kind of input partitioning can be applied, and by extension boundary value partitioning can be applied. Similarly, all programs can have statement testing and coverage applied, as thoroughly testing any program is always best done by ensuring you test as much of the program as possible. Decision testing is not necessarily as widely applicable, as there are some programs that have no decisions to make, and do the same thing with all valid inputs. However, most programs have some kind of decisions to be made, and therefore decision testing and coverage can be applied.

While writing my code from the perspective of both a developer and a tester, caution became more of a concern as I continued, as I realized the difficulty of both testing, and later correction of my code. As a result, by the second milestone, I had developed my code with testing in mind, utilizing tools like return cases to make testing easier, rather than utilizing cumbersome tools like inputStreams to compensate for poor coding practice. Limiting bias however, was harder, as it’s difficult to spot my own blind spots in my testing and code. As such, in order to limit bias I attempted to utilize a consistent method of doing tests, and maximizing coverage to ensure my testing was comprehensive enough to attempt to limit personal bias.

Finally, my consistent testing also was an attempt to work with disciplined testing. By using a consistent method of testing, I could ensure my testing remained effective regardless of the program it was being run on, because my methodology was effective in testing. By cutting corners early in writing my code, I ended up having to do more work later to compensate for an inflexible system. In my original version of the ContactDriverTest, I had to utilize an InputStream and PrintStream which affected testing by making it difficult to write further tests. As a result, I applied discipline by utilizing return cases in future testing.