Ethan Klukkert

Professor Gaston

CS-320

11 December 2022

Project 2: Summary and Reflections Report

Summary

The approach I took to implement the requirements was thorough yet concise. At a high level in the context of this application, I ensured that the application requirements were satisfied by breaking down the problem into concepts of accepting and rejecting, searching and returning info, and defining only characteristics within the requirements. I approached the testing by applying constant and defined changes to see if the system properly reacts. For example, the Contact class test first defines what good input is, and how the parameters it uses are supposed to be accepted, such as on line 14 of the ContactTest file. Then, throughout the tests, I only adjust one parameter in the Contact constructor per test function, to ensure that the system is not only denying invalid input, but also that it is only denying it at that error. This is further specified at the ContactServiceTest file, on lines 14 through 44, where I first implement a contact list that successfully creates contacts and successfully adds those unique contacts to the list. Then, the next function only changes the id's to be the same, and so the test then asserts that the list rejects the non-unique contacts based on the same id's. Using this style of technique, I am able to test each and every parameter, to achieve maximum coverage. The Model tests have 100% coverage, while the Service tests have over 80% coverage. This is how I know that the tests were effective, especially since the tests are not technically comprehensive, where I might test every possible combination.

I know that the code is technically sound by doing extra asserts in parts of code that can be “cheated”. For example, in testing the Appointment model, I had to make sure that the update method for the date was successful. This means passing another valid future date so that the system accepts a new date. On lines 67 through 74 of the AppointmentTest file, I create two future dates, where the first one is used in the constructor of an appointment. We know this construction was successful because it would have thrown an error based on our base case on lines 15 through 19, where this is verified to work. Now, I set the new date with the other future date, and we know this works due to no error being thrown, based on line 44 of the Appointment class specifications. Lastly, line 73 of AppointmentTest asserts that the two dates are in fact different. Then we test that the appointment object does have the second date as its attribute. The extra assert on line 73 *proves* programmatically that these dates are different, and I am not accidentally setting the date to the date it already was. By using this technique, I remove the need to manually review whether a logically faulty test is trustworthy. The same approach is taken throughout many tests to prove that updates to attributes were successful, and did in fact change even if they did not through an error and fail the test. This is especially seen throughout the TaskServiceTest file tests, on lines 66 onwards.

I ensured efficiency by adopting industry standard techniques. I used equivalence partitioning at line 27 of the TaskTest file, where I asserted that an error was thrown at the id field. I did not have to test any other longer id input, because more pointless tests need to be made. There is no need to make these tests, especially since the class uses a greater than symbol in the condition that throws an error when true, which is very well defined to reject *any* value above 10 characters. This technique is used across all three features, where character limits are imposed.

Reflection

For each feature, I used equivalence partitioning, boundary value analysis, experience-based testing, and statement/decision testing and coverage. Equivalence partitioning is a black-box technique where we group inputs into one or a few test cases to improve testing efficiency. For example, the “ID” attribute for the task class needed to be 10 characters or less. Here I grouped the “no longer than 10 characters” requirement with a test case of 11 characters. This is too long for the program to handle, so it should be rejected. Testing for any amount larger than 11 is pointless because they are all equally too large, in concept, for 10 characters. I also did a lower bound test at 10 characters, because any input 10 or less is equivalent, so an equivalent test of 10 characters to pass the requirement is sufficient. The 10 and 11 character input for the ID also touches on the boundary value analysis. Most errors occur at the boundaries of inputs, so the value that is directly after the limit needs to be tested. Since the number of characters is a whole number, 11 works for this test.

Statement and decision coverage was a technique I used to test that all lines of code and decision trees were run by at least one test. For each milestone, the decision was either accept or reject, where a bad input on any of the attributes rejected the entire object. I make one case where the input is correct to serve as a basis for what inputs can be accepted. Then, I make a separate test for each attribute where only that one attribute has a failing input. For example, the ID requirement for the task class had the input rejected at the boundary of 11 characters, and the base case of 10 characters served as the accepted input. This covers statements and decisions.

Throughout the application testing, I did not implement decision table testing, state transition testing, and use case testing. I did not create a decision table test, because the decisions were either create an object, or do not create an object based on the input. For the service classes

in each milestone, I did not use a decision table still, because the classes only needed to know if an object was in the data structure (list) or not, and then update, delete, or add. It was a limited number of decisions that only had a few results so a table's usefulness was negligible. State transition and use case testing were not used in these milestones because they relate more to a system that uses these classes. The "service" classes of each milestone came close to needing this technique, but I was able to define the structure quickly as stated above, so the technique was not needed.

The techniques I did and did not use implies that projects should use techniques that are practical for their situation. If decisions and statements are well-defined and small in number, then an exhaustive (but still using equivalence partitioning for efficiency) testing approach may be feasible. For larger and more complex projects, there has to be a lot more analysis of the black-box techniques, and the white-box techniques have to be limited to what is efficient. For example, decision tables have to be simplified, and decision coverage testing has to be prioritized over statement coverage since it automatically guarantees 100 per cent statement coverage (Hambling 2019).

Throughout testing, I confirmed that each step in the testing process was verified programmatically. Instead of just manually reviewing, I also would put extra assert methods to prove that what I think I was doing, was actually what I was doing. This caution ensures the quality and trustworthiness of the code, because despite the fact that a line of code doesn't throw an error, does not mean it works as intended. It is important to understand how and when code does operations in the background, because not all those operations may be expected. Understanding how code works fundamentally allows us to look into the black-box functions and verify that operations are working in the expected manner. With this examination comes bias,

however, and it is crucial to overcome it when you are the tester of your own code. The intention of testing is to find errors, and it can be tempting to avoid finding errors due to a fear of being wrong or making wrong code. But this is what should motivate us to be a more thorough tester, so that we can stand by our code and know that it is in fact working as intended. It is far less embarrassing to expose issues to yourself than to the entire project team and client base when the code goes live.

Being disciplined in software engineering is core to developing great services. Software is used in many critical fields, and supports many industries that affect millions of people. Such as in the disaster of the Ariane 5 rocket, or the Therac-25 radiation machine, code must be rigorously examined in a process before it is used to avoid financial costs, and even legal costs. Software testing is that rigorous process, and it is the process that cannot be prone to technical debt, because it is meant to solve that issue in the first place. An issue must be solved as soon as it is spotted to save costs that only exponentiate as the SDLC progresses. With knowledge of testing code, writing code can be done at a higher level of expertise. And vice versa, testing can be conducted in more thorough ways based on how code is written.

References

Hambling, B. (July 2019) *Software Testing: An ISTQB-BCS Certified Tester Foundation guide-4th Edition* BCS, The Chartered Institute for IT p. 117