Tanner Hunt

CS320 Software Test, Automation QA

Feb 13, 2025

Software Testing and Automation QA Retrospective

Throughout the last three milestone assignments, I used a variety of testing techniques to ensure the product met the software requirements. I mostly made use of dynamic testing techniques, including black box and boundary testing techniques. The unit tests that made use of black box testing techniques tested expected outputs of modules without knowledge of the inner workings of the code. Boundary testing used my knowledge of the code to test decision points within the software. To that extent, every decision point within the code should have been tested – as well as every data input and output point.

My testing approaches were mostly in alignment with the software requirements, with some errors in between. Most module requirements outlined in the bullet points were explicitly supported by a test unique to that requirement. For example, Appointment objects have a requirement to accept IDs of length 10 or less. There is a corresponding test in the Appointment Test class called AppointmentIDTooLong on lines 35 through 41. There were also mistakes in the test cases that likely could have been prevented by static testing. One such disambiguation I could have made was whether the project should accept empty strings for object names, descriptions, IDs, etc. I went ahead and tested against empty string objects, an example of which is the test EmptyContactID in the ContactTest class, which created scope creep.

I ensured the quality of my unit tests in a few ways. I didn’t consider my test suite to be complete until I had tested every decision point within the code, as well as any lines of code that handled I/O. The Eclipse IDE also has a tool that shows test coverage within the project. After my first pass in writing unit tests, I looked at Eclipses estimated test coverage percentage to ensure I caught everything I thought would be important to test.

I also took steps to ensure the code was both technically sound and efficient. I regularly tested my unit tests while I wrote them to ensure my assertions were correct. These tests did not make it to the final production but were spot checks I made when I felt uncertain. For example, if I made an assertion that passing a null value to a function threw an exception, I would ensure the test failed if I didn’t pass a null value. I also used small, manageable test cases with clear responsibilities. For example, the DuplicateContactID() test only checks if an exception is thrown when duplicate Contact IDs are created. I used Big O analysis when designing my tests to make sure they are efficient. My tests only validated one or two methods, so they used a simple algorithm with a time complexity of O (1). This is usually done by instantiating an object and making assertions about the object. For example, the NullAppointmentID test only creates an object with a null ID and ensures it throws an exception. I also restricted my use of tests using the @BeforeEach tag because it added to the time complexity. To improve the efficiency of @BeforeEach methods, I restricted them to nested classes where the method was pivotal to each test. For example, the InitTest method on line 160 of the TaskTest class only applies to tests that use the member variable it creates.

My software testing techniques were primarily boundary testing techniques. For example, the AppointmentID class should accept an ID of length 10 or less. The unit tests for this requirement then asserted an ID of length 10 would be accepted and an ID of length 11 would be rejected. This technique was expanded for every decision point within the code and covered every outcome of those decision points. The practical use of this testing technique is that it creates higher test coverage, which may be a legal requirement of some forms of software. I also used some static testing techniques while writing the main code. This involved looking at points in the code I know to be prone to errors. The benefits of static testing techniques like this are that they can prevent errors before they become more costly down the line. There were many testing techniques I did not make use of. For example, I didn’t make use of other static testing techniques like document review, where project requirements are disambiguated before any work-products are created. The benefit of this kind of testing is cheaper correction of software defects. I also did not make use of specialized testing techniques, like security testing or performance testing. The benefits of these are increased confidence in the product, testing better reflects the live environment, and fulfillment of possible legal requirements. I also did not perform any acceptance testing or usability testing. The practical importance of these tests is that they help the product reach a wider audience and ensure the product fulfills the user’s needs.

In donning the role of software tester, I had to change my mindset from one of creating a product that delivered on functionality to one of “how many ways could I break this code?” I still had to exercise caution in a few ways though. For instance, I had to keep in mind that the defects I find are only a subset of the defects – I can’t find them all. I couldn’t write any security tests because I am not knowledgeable about the subject. There were also tests I thought could benefit the project, but I had to respect the time constraints that I am under. For example, the performance of the app could be tested by pushing hundreds or thousands of new contacts through the contact service but writing tests like these would take more of my limited time, were not in the project requirements, and would slow the performance of the testing suite.

I tried to limit bias in my tests by having testing procedures in place and setting standards for code coverage. Some of these testing procedures included writing tests that validated expected outputs for every method, and testing the boundaries of every decision point. Bias in the testing phase could negatively impact the quality of code because the required mindset between development and testing is different. When I have my developer hat on, I tend to think of my code in terms of proofs, which may make me blind to the unexpected errors the code may have.

Discipline is an especially important quality to have while writing tests. I have found that testing is a repetitive task, but that should not mean shortcuts should be taken. There is a joke that captures the essence of why discipline is important in software testing: “A doctor can only kill people one at a time while an engineer can kill them hundreds at a time.” Poorly executed and tested code can cause harm to humans. An example of this was a defect in the Therac-25, a machine that delivered radiation therapy to patients. A race condition in the software and absence of hardware controls led to deadly radiation doses to at least three people (Lynch, 2017). While not all software defects can cause harm on this scale, software defects can also cost significant amounts of money and affect the public’s opinion about a product. Developers have an ethical responsibility to protect the interests of the public, the company, and the profession – which means writing quality tests. To write quality tests in the future, and prevent technical debt, I will use Agile principles to maintain a quality code base. This includes strong communication practices, such as estimating the time requirements of tests and communicating with the team when a test suite takes an unexpected amount of time to develop. This also means insisting on quality, because it is my ethical responsibility to do so.

References:

Lynch, J. (2017). The Worst Computer Bugs in History: Race Conditions in Therac-25. *Insight Hub*. Retrieved from https://www.bugsnag.com/blog/bug-day-race-condition-therac-25/