Kerrian Offermann

CS-320: Software Test Automation & QA

Module Seven: Project Two

August 11, 2022

When unit testing for contacts, tasks, and appointments, my approach was to examine the software requirements laid out for the customer. Contacts, for example, had specific requirements laid out in terms how many characters were allowed for each string. The contact ID, first name, and last name fields needed to be 10 characters or less while the phone number field needed to be exactly 10 characters. The final contact field—address—needed to be 30 characters or less. Each field was not allowed to have a null input. There is also the request that all ID strings for contacts, tasks, and appointments be prevented from updating. When it came to tasks and appointments, there were restrictions on software requirements as well. Tasks needed non-null strings for ID with a 10-character limit, name with a 20-character limit, and description with a 50-character limit. With appointment, there were limits on the strings requested as well (10 characters for ID and 50 for description), but the appointment feature had a unique software requirement compared to the others. While these other strings only required checking if fields were properly rejecting anything that was null or over a certain string length, the appointment feature had one involving dates. Appointment dates entered could not based on a past date which is logical. A user should not be able to make an appointment for yesterday since it has already passed; therefore, testing had to be done with past, present, and future dates to ensure that dates entered were in chronological order. These claims and the evidence can be found in the customer’s request to Grand Strand Systems.

When it came to the overall quality of my JUnit tests, I would stay that it was not perfect, but it functioned as it was supposed to despite the fact that the coverage did not quite meet the 80% coverage recommendation (I made it to 72%). Outside of this unfortunate shortcoming, I was able to test the quality of my code by having JUnit tests that all passed. These tests passing informed me that the code I wrote was properly accepting inputs that met the software requirements and rejecting inputs that did not. I tested both the length of characters and null inputs during my tests, and periodically performed coverage tests to increase the coverage as much as I could.

When writing JUnit tests, my experience ranged from gratifying to frustrating depending on what I was working on. Oddly enough, I assumed I would struggle most with the appointment featured; however, once I researched how to enter and test dates it was easy to perform a test. The most difficult part for me came with null testing. Even when I managed to make null testing work properly in JUnit tests, the coverage test often highlighted anything null-related in red to signal to me that something was not connecting as it should. Once things began to work, though, I was able to learn a lot of new things about how JUnit tests work and how useful it is when it comes to programming new software. I cannot imagine testing code in the future without introducing JUnit tests to the process now. It seems too useful to forget.

I ensured that my code was technically sound by comparing each test file to its corresponding file of methods, and checking for everything from data entry errors to errors in logic. Thankfully, Eclipse does a great job of highlighting things that do not come off as correct. For example, if a typo is entered when writing a code, Eclipse will flag it and recommend that correct term that I meant to use as well. An example of this is with ContactTest.java. When I attempted to delete a contact that did not exist in line 38, Eclipse flagged it as incorrect when I ran a test afterwards. As a result, I ended up including this mistake into my JUnit test to make sure that if the customer ever made the same error that the software itself will flag it the way that Eclipse did.

The first step that I took to ensure that my code was efficient was scanning the code for anything that was unnecessary. Once more, the Eclipse software does a great job of flagging things that are extraneous in the coding, but sometimes it is a matter of scanning the code itself and ensuring that anything that does not need to be there is taken out in order to make the program itself run faster in the long run. Specific lines that I scanned and updated for efficiency include lines 22-23 of ContactServiceTest.java. Initially, I wanted to add a multitude of different contacts, but after making about five different examples of sample contacts I realized that it might not be necessary to do so much. Regardless of how many contacts and subsequent tasks and appointments I added, the prime focus of the test was to examine software requirements. Having so many different contacts, tasks, and/or appointments all doing the same tests seemed unnecessary; therefore, I just chose a sample or two to make the test speed faster. Tests should last a few seconds at most (Garcia, 2017), so the less complications, the better.

The first software testing technique that I employed during this project was old-fashioned checklists. I knew that every section I was testing had specific requirements that had to be met, so when testing I would rely on a checklist to ensure that the test covered everything that the customer was requesting. As discussed by Boni Garcia (2017) in *Mastering Software Testing with JUnit 5: A Comprehensive, Hands-on Guide on Unit Testing Framework for Java Programming Language*, what the customer wants and what the programmer provides often does not match up due to a missing link in communication. Therefore, tracking everything requested and regularly checking it can assist in eliminating this issue. The next technique that I used was simply running inputs that both met and did not meet software requirements, and seeing if they were being accepted or rejected respectively.

I generally avoided any software testing techniques that did not align with the software requirements. For example, I did not test if integers were being entered in text fields or vice versa because even though it seems logical to me, the customer may prefer it this way (perhaps to allow the customer to write text such as “N/A” or “none” in phone number fields, as an example). I also did not test anything that dealt with the speed or performance of the program itself since it was not a requirement of the customer.

Based on other software that I worked on in the past, I can say with confidence that a checklist can go a long way when it comes to testing. In fact, a checklist is essentially used during the Agile process when developing and testing programs because it keeps track of what must be accomplished. It never hurts to copy the software requirements down and use them as a reference during testing instead of relying on unreliable methods such as memory or assumption. As for testing different inputs, this is also a reliable technique that can be used in various projects and situations since testers always want to emulate user actions in order to examine how future users might behave when the final product is released.

As a software tester, I employed caution by always returning to double and even triple-checking things in order to ensure that I did not miss anything. It is important to appreciate the complexity and interrelationships of the code being tested because carelessly altering code without understanding the delicate connections between them can cause more problems in the long run. For example, changing an object name in the class will immediately cause errors with everything that used that object name; therefore, it is important to think about the consequences of changing things before actually doing it.

I tried to limit bias in my code by relying on code-checking techniques like coverage tests to tell me whether my code is reliable or not. As a software developer, I can imagine bias being a problem when testing your own code. For example, if a person has a habit of making the same coding errors and no one ever corrects them, then how will they know that they are doing anything wrong? It often takes another party with different perspectives—or at least a program made to indiscriminately check errors—to examine code for what it is rather than we one believes that it might be.

It is important not to cut corners when it comes to writing and testing code because software operates on the idea of countless moving parts working together to create one program. Therefore, if one part is not operating as it should then the entire unit is compromised. Cutting a corner is no different that sabotaging the success of the software itself. I plan to avoid technical debt as a practitioner in the field by making every effort imaginable to test and eventually approve acceptable code. For example, now that I understand a new testing technique (JUnit 5) that is beyond manually checking code for shortcomings, I can add it to my list of testing techniques to use. I believe that the more testing techniques I learn, the more knowledge and skill I will gain to find and eliminate errors in the software before it is released.

**References**

Boni Garcia. (2017). *Mastering Software Testing with JUnit 5: A Comprehensive, Hands-on Guide on Unit Testing Framework for Java Programming Language*. Packt Publishing.