Carmen Kingery

Bryant Moscon

Software Test Automations & QA

Project Two

04/16/2021

Summary & Reflections Report

**Summary:**

*To what extent was your testing approach aligned to the software requirements?* In short, I would say my testing was in direct alignment with the software requirements. Before writing out the tests, I had to figure out what I am testing for specifically. To do this, I looked at the requirements my classes needed to fulfill, and I made sure to include tests for each requirement. For example, in the ContactService class, I tested the update methods to make sure they followed the expected procedure from the requirements; Namely, that the methods properly updated the first/last name, phone number, or address when given the unique ID. Another example can be seen in the Contact class. As per the requirements, each String attribute had a set maximum length. Therefore, I had to write tests that would verify the program was checking these lengths and properly handling cases where the Strings were not the acceptable length.

*Defend the overall quality of your Junit tests for the contact service and task service.* When I run the coverage test on each class, I was able to get 85.1% coverage for the AppointmentService class,84.8% for the ContactService class, and 87% for the TaskService class. This tells me I have successfully and effectively tested a safe majority of the statements and branches in the program. There is definitely room for improvement, but over 80% coverage with effective tests is acceptable for a beginner tester. All projects could benefit from more testing if the time allowed for it, the trick is in figuring out when your testing is sufficient to move on to the next step in the project.

*How did you ensure that your code was technically sound?* Some things I tried to strive for in hopes of producing more professional-like and “technically sound” code were simple logic and clean/concise classes. Being a beginner programmer, it is easy for me to over-think methods, and I tend to add things I don’t necessarily understand from examples I find. However, for this project I tried to keep the logic within my understanding, which entails simple logic. Along with that, I have tried to keep my methods short and sweet to enhance their readability. In terms of clean and concise code, I made sure to break up the ContactService project into two smaller classes: the Contact class and the ContactService class (same for the TaskService project and the AppointmentService project). This allowed for less clutter and more concise and efficient classes.

*How did you ensure that your code was efficient?* For efficient code, I made sure to keep the project modular by breaking it up into smaller classes (as mentioned above). I also kept maintenance in mind while coding. I wanted to make sure that, in the case of minor changes in the requirements, I or anyone else would be able to look at my program and be able to find and adjust the proper sections without too much hassle. For example, as mentioned previously, each of the String attributes under the Appointment, Task, and Contact classes have a specific length requirement they have to meet or be under. At some point later down the road, those max or min lengths could change. Therefore, instead of hard coding those numbers into all my length checks throughout the program, I created variables to hold those values and placed them at the top of each class. That way they are easily accessible and simple to updated, and therefore efficient to maintain.

**Reflection:**

*Techniques employed:* For most of my tests, I used the technique of analyzing the boundary values of my methods. That is, I tested specific inputs that were right on the edge of acceptable and non-acceptable values, in order to validate that the proper inputs were being either rejected or accepted. As an example, these kinds of tests were used in my class constructors. In all three of my projects, the attributes under the object classes had specific acceptable lengths. So, for my tests, I had to check the lengths of the inputted values for these attributes and make sure the proper denial messages were presented if those lengths were not followed, hence the boundary tests. I also employed a few integration tests. That is, I tested that each of the individual pieces of the project worked well with each other as one whole project. For example, after verifying my object class, I would create my method class and verify that it properly called and created objects from my object class.

*Other techniques (not specifically employed):* A few of the other testing techniques that we have learned about include equivalent class partitioning, decision tables, state transitions, and error guessing. Equivalent class partitioning is similar to boundary value analysis for it also checks the boundary logic for errors. However, it also broadly checks each type of input. For equivalent class partitioning, you would try to group together like inputs (i.e. negative numbers, numbers under a given range, or numbers over a given range) and choose a single input from each group to test. Decision tables and state transitions are two similar testing techniques as they are focused on the different paths and branches the program can follow. According to my understanding, decision tables are for tracking the branches of a program to make sure the logic is solid, each branch is accessible, and each branch handles their specific inputs properly. Whereas state transitions track and test various stages the program should contain (more detailed explanation below). The last technique we have learned about is error guessing. This technique is best performed by those with decent prior testing experience because it requires knowing the common places errors occur, errors that would otherwise slip past other testing techniques.

*Uses and implications of techniques:* Boundary value analysis is extremely useful anytime you have a set length of characters that are accepted such as the projects we have been working on these last few weeks. They are great for verifying acceptable lengths, as well as for verifying if loops and branches have been set up with the intended number of iterations or comparative values. Equivalent class partitioning would be excellent for minimizing the testing of projects requiring things like an email or password. Anything that would have a large number of restrictions for inputs would benefit from using equivalent class partitioning because of its ability to test all similar inputs with only one or two test cases. I could see decision tables being heavily used in the gaming industry as every decision a player makes is another branch the program needs to predict. The tables would help the programmers not only work through but also keep track of and properly test all the intricate paths and pieces of such a large project. State transitions map out all the different states the user might be in and helps visualize to the programmer what needs to be tested. A common example I have seen for using state transitions is in online banking. You want those cites to be secure. Therefore, you program in different states a user can be in when on the login screen. Some states might be: have not tried logging in, tried but failed, tried twice and failed. Depending on which state the user is in, the program will treat a failed login attempt differently. Lastly, error guessing can be used at any time in combination with all of these techniques. It is unique as it is not a set method for testing. It takes all the experience you have acquired in previous projects and all the lessons you learned in those projects and applies them to the current project. If you have had trouble with password verification before, maybe you should make some extra test cases for that method. Maybe you are working on a calendar app, did the programmers format the date in proper MM/dd/yyyy form or did they brain-fart and format in descending order dd/MM/yyyy? It takes a lot of experience to know where to look for potential errors but if you have an experienced coworker on the team error guessing can be an extremely beneficial testing technique.

*Assess the mindset that you adopted working on this project.* In acting as a software tester, I had to take on the mindset of finding errors. In comparison to making the program work as the developer, the tester is trying to find where the program does not work properly. This was difficult as I, the developer of the code, was personally attached to the program. This caused some caution or hesitation when testing, although I tried to limit this as much as possible in order to find and correct as many errors as possible. It was important and helpful to appreciate the complexity and interrelationships of the code I was testing because it helped me realize how complex and in-depth my testing needs to be. This appreciation was also beneficial while developing because it helped me accept the thought that my code will not be perfect and that errors will be found.

*Assess the ways you tried to limit bias in your review of the code.* Testing your own code without bias is difficult because of the fact that you are the one who wrote it. You are personally connected to the code and therefore are invested in seeing that it is a success. However, when testing that code, you must flip that mindset and go out of your way to find errors in it. I tried to limit my bias when testing in a couple different manors. First, I split up my development time and testing time. Instead of doing it all in one sitting, I would develop the classes, take a break, and then come back with a tester’s mindset. Second, since the tester’s mindset was more difficult for me to don, I tried to do multiple sessions of testing before moving on to develop the next class. In one session I would write out as many efficient tests I could think of, run them, fix the errors found, and take a break. Then I would come back and do it all again.

*Finally, evaluate the importance of being disciplined in your commitment to quality as a software engineering professional.* Being committed to quality takes a lot of discipline. Without discipline you tend to procrastinate doing certain items which can lead to cutting corners and cheaping out on certain things later just to get the product done on time. For example, say you were going to build your mother a new table as a Mother’s Day gift. You have spent all month planning it out and now you only have a week to make the thing. Cutting corners might include things like: not planing your wood before using, buying cheap stain from Walmart because it would take too long to drive 30 mins to the nearest Menards or Lows, making due with an improper tool because it would also take to long to go buy what you need. Despite cutting corners, the table might turn out just fine, but more likely than not it will turn out with significantly less quality than it would have had you taken the time to do things properly. The same concept applies to programing with the biggest corner-cutting temptation being to delay certain work in order to hit a deadline, also known as technical debt. The most common type of technical debt a project tends to put themselves in is testing debt. Instead of testing as they go, they focus on developing the program with the hopes of getting to the testing sometime later. This can cause major setbacks for a product’s publication as serious issues are not being caught early in development and are instead being caught (if at all) right before publication. Therefore, deadlines might be met, but the quality of the program is severely compromised. My solution: discipline yourself. Do not settle for anything less than your best. Cutting corners and accumulating technical debt is you settling for less than your best and not practicing discipline.