Brad Peterson

CS 320 – Project 2

August 25, 2024

Testing Reflections

1. **Summary**

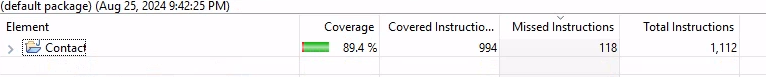
I took each software requirement we were given and made a JUnit test for that requirement to be sure that the code I wrote would align with what I set out to do. A good simple example of this is something like testing that the requirement for the Contact classes LastName was less than 10 characters long. The key I found to this approach was to be meticulously sure the requirements were understood thoroughly and have a one-to-one test in place.

A close-up of a website

Description automatically generated

For instance, during the first two milestones I failed to ensure that the Contact/Task classes had a unique ID. Instead, I approached this requirement without a test, but having the ContactService/TaskService classes iterate through the list and not add a new instance of the individual class if there was a unique ID already. However, the requirements were slightly different and were directly applicable to the Contact/Task classes. My approach meant you could create a duplicate class ID for them, even if you couldn’t add them to the service class list, which isn’t quite what the requirements were and if I had specifically made a test for the appropriate class I would have caught that error sooner.

In the end this requirement to test case conversion was a thorough way to ensure the code is doing exactly what I wanted it to do. It also meant that each part of the code base was covered by JUnit tests, resulting in an overall test coverage of 89%. I think there are diminishing returns to shooting for 100% test coverage for the sake of it. But shooting for a test coverage of 90% or so seems like a reasonable bar to feel confident you have a good suite of unit tests that are thoroughly testing your code base.

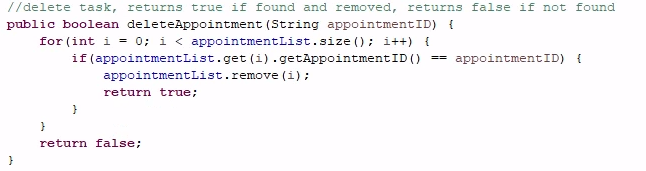


In the process of writing the code for these requirements there were several new tricks I learned to make sure the code I wrote was following best practices for technically sound and efficient code. The first one was the “final” key word in Java. This enabled me to set an initial value for the ID field of each class but then lock it down so it cannot be modified once the constructor has generated that initial ID, fulfilling the requirement that they be immutable. When I first set out to do this, I thought it would be enough to make it a private variable and write a setter function to go with it so learning this intricacy of Java was useful.

A close up of a white background

Description automatically generated

To make sure the code was efficient, I tried to not iterate more than was strictly necessary to fulfill my requirements. The best example of this is found in the deleteAppointment function of the AppointmentService class. Instead of iterating through the entire list, it only iterates until a matching ID is found. It then deletes that instance from the list and returns true to indicate a successful deletion.



1. **Reflection**

Over the course of this project, I used static testing and unit testing to assess the requirements given and create the tests required to ensure they were met. Static testing being manually reading through the code and requirement documentation to look for any obvious flaws or bugs. This, I think was the most intuitive part of the testing process because it’s how most of us begin writing code. However, when you really learn about static testing you start to approach the same process of understanding requirements a little more deeply and I found my self breaking each piece of the requirements more as I was going than before because of this mindset. Next came the actual JUnit tests we wrote. These were the main focus of this project and tested those requirements directly by trying to either break the requirements specifications or ensuring that if followed, an appropriate class was created.

I didn’t use any further testing techniques like integration or system testing for larger picture testing techniques on this project. Our requirements were quite narrow and largely focused on unit testing, however, we could have done some integration testing by writing tests to ensure the creation of individual classes and then adding them to the class lists. Instead, my tests focused only on either the individual class or the list itself and didn’t test both simultaneously for integration between one another. Since there was no other overall use of tying together the appointment/contact/service classes there really weren’t enough requirements to test these as a part of a larger whole.

While this project was narrow and purpose built for unit testing, most software isn’t and serves some larger purpose. In those cases, employing integration and system testing is essential to make sure each smaller piece is doing its job to contribute to the larger requirements of the software. Integration testing to tie each piece of code together and ensure they are properly functioning with one another and can communicate by accessing the data from one another as appropriate. Systems testing is the final check that the product is fully functional as the requirements dictated once all the pieces of code have been integrated together. This is really key as it’s what your end users will be doing and if they encounter problems then your software isn’t serving your users.

My mindset for writing tests was to focus solely on one requirement at a time while writing code and solve that particular problem before moving on to the next. It’s easy to read a quick requirement line that states something like “first name is required, less than 10 characters, not be empty” and process that as one requirement for the field but, really it’s 3 different requirements that have their own problems to solve, the smaller the problem the better.

By breaking them into such small pieces that sometimes felt trivial you avoid missing small bugs that might happen if you solve all three at once. This provides a cautious approach that says it’s better to take your time and be certain you’re doing exactly what it is you think you are then potentially allowing edge cases to exist that will be harder to track down later as the complexity increases in the code.

I hadn’t really thought much about the idea of bias of writing our own tests for code. But it’s absolutely something I ran into over the course of this project, especially when something seems very simple it’s easy to write that off in your head as a verified functional line of code. This is also why I think code reviews are so critical to guard against inadvertent biases of one individual programmer. Another set of eyes is likely to see a different angle on what the code is doing, especially if they aren’t actively involved in the thought process of solving the problem initially so they can come with a fresh perspective.

Ultimately, taking a cautious approach to testing by breaking everything into the smallest pieces possible and keeping in mind our own cognitive biases result in a very disciplined testing approach that will do well to guard against rogue bugs that will pop up later down the road and be far more difficult to unwind and resolve than they are in the moment. Most of the time we will be writing our own unit tests for the code we write so it’s important to stay sharp while doing it and be open to feedback during code reviews. Following these practices helps us avoid building more technical debt of code that really needs reworked later but it’s such a mess that it’s potentially a far bigger job than it ever really should have been in the first place.

Margheim, S. (2020, December 17). *How to combat bias in software testing*. TEST IO. https://test.io/resources/blog/how-to-combat-bias-in-software-testing