CS-320-H7099

Project Two

Kasey Myers

09 June 2022

**Summary**

My unit testing approach for each of the three segments was the same. I intended to test each functionality requirement that was presented for each segment. For example, in the ContactService.java, we were required to be able to add, delete, and update contacts per the contact ID. Based on these requirements, I tested each functionality individually. I did the same for each of the other java files and their presented requirements. Below is an example of the alignment between my testing approach and the requirements:

Requirement:

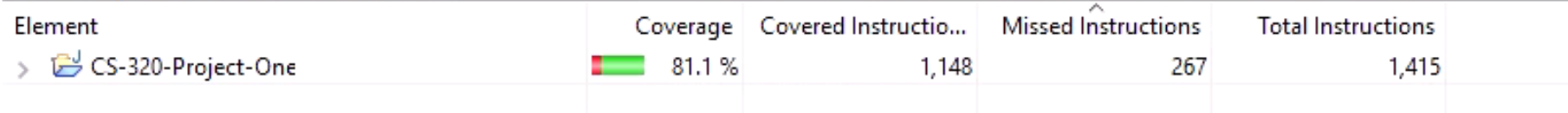


Test:

Text

Description automatically generated

The average Junit test coverage is between eighty and eighty-five percent. As you can see below, for the project as a whole, I was right on par with the average:



However, when I expand this coverage to be broken down by each individual file, there are a few that I missed the mark on:

Table

Description automatically generated

There are at least three sections that could use some extensive work on them to ensure coverage meets expectations.

Ensuring that your code is technically sound and efficient increase the chances that you are providing the client with a quality and accurate product. This means that your code should be bug and error free, simple in construct, and well tested. For this project, the entirety of my code is debugged and error free, has a simple and easy to read structure, and is mostly (see above screenshot of coverage as reference) well tested. The testing that was completed was accurate and effective per the example below:

Code:

Graphical user interface, text, application

Description automatically generated

Test:

Graphical user interface, text, application

Description automatically generated

**Reflection**

For each of our milestones and project, we have used Junit to utilize unit testing for our software. Unit testing is a form of white-box testing, or structural testing, that develops tests for the software on a structural basis. According to Brian Hambling and his associates, “At the component level, the structures of interest will be program structures such as decisions; at the integration level we may be interested in exploring the way components interact with other components (in what is usually termed a calling structure); at the system level we may be interested in how users will interact with a menu structure.” In our case, we focused on the integration level where we tested the units of the program and how they interacted with the call methods to ensure proper functionality.

We also utilized black-box, or specification based, testing techniques. Black-box testing creates test cases based directly on specifications or from a model of what the system should do functionally (Hambling et al., 2015, pp. 85). Although our unit testing was testing our call methods’ functionality, it was simultaneously testing the functionality of the system based on the specifications provided in the assignment rubrics. In our Junit tests, any errors that presented themselves showed a flaw in the code, whereas a failure or success highlighted the specification functionality.

One testing technique category that I did not utilize is experience-based testing. This is mostly due to my lack of experience. An example of experience-based testing is error guessing. In error guessing, there are typically not specifications to compare to and developers and users will test the software for assumed functionality based on their experience and make “guesses” as to the errors and how to rectify them based on their idea of the most important features. Since we were provided with specifications, experience-based testing was not required for our assignments.

When it comes to my mindset working through the project, I tried to approach it as I would any other project at my job. I like to have a list of tasks that I prioritize and work through based on that list. In this case, my highest priority was to ensure test coverage of 80% or higher. Writing the tests required an abundance of caution due to the code complexity, interrelationships, and requirements. Trying to ensure the proper coverage while writing the tests gives an appreciation of the code that you may not have otherwise (if you weren’t the one testing the code). For example, if we look at Contact and ContactService, or any other pairing, the service file calls on the other file multiple times.

I imagine that bias would be a major concern in testing your own code. As the developer, if your program is free of bugs and errors and runs smoothly, it is easy to assume that everything functions properly. However, just because your code functions, does not mean that it produces the desired result. It is easy to assume that your code is efficient and functional if you are the one who wrote it, whereas an outside tester will be more objective about the testing process. For this project, bias became an issue once or twice. I found myself confused by why I was getting errors and failures in my testing when the code was bug and error free. You have to strip away your pride when testing your own code. A failure in your code is not a failure in your ability.

Discipline is vital to ensuring code quality. It helps the developer in creating code that on average has fewer bugs and errors. Cutting corners in testing can have massive financial implications as well as major integrity and functional implications. Cutting corners can lead to missed functionality requirements, code structure issues, and ultimately incurred costs of fixing the issues that were not caught in testing.

**References**

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter.

(2015). *Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition) - 5.1.1.1 Test Organisation (K2).* (pp. 77-139). BCS The Chartered Institute for IT. Retrieved from  
https://app.knovel.com/hotlink/pdf/id:kt00UC2JS2/software-testing-an-istqb/test-organisation-k2