7-2 Project Two

Madeline McNeeley

August 12, 2022

CS-320

Table of Contents

[Alignment to Requirements 3](#_Toc111299979)

[Effective Tests 3](#_Toc111299980)

[Technically Sound Code 4](#_Toc111299981)

[Efficient Code 4](#_Toc111299982)

[Techniques Employed 5](#_Toc111299983)

[Other Techniques: Uses and Implications of Techniques 6](#_Toc111299984)

[Caution 7](#_Toc111299985)

[Bias 8](#_Toc111299986)

[Discipline 8](#_Toc111299987)

# Alignment to Requirements

For each portion of this project, we were given a set of customer requirements. I used these requirements as guidelines for implementation and testing to ensure that the intended functionality existed in the application. I accomplished this by going through each line of the requirements before, during, and after the implementation stage. Before the implementation stage, I looked over the requirements to make sure there was an absence of ambiguity, and I had no unresolved questions. During implementation, I used the requirements to verify that what I was writing was meeting expectations. After implementation, I used the requirements again author unit tests and for verification. An example of this would be when I used unit tests to verify that the contact service class did not allow users to create a contact that utilized an already existing contact Id, per the requirements.

# Effective Tests

Coverage percentage is an indicator provided by JUnit that allows developers to see the scope of their unit testing. My test coverage for the Grand Strands Systems application was over 87%, which exceeded the requirement of having at a minimum of 80% test coverage. This means that almost 90% of my implementations were tested thoroughly. 100% test coverage, or exhaustive test coverage is hard to achieve, especially when some methods or other code changes are internal operations that are hard for unit testing to manipulate and verify. To ensure the highest percentage test coverage as possible, I refactored some of my methods, to ensure that they were testable. Some examples of this would be in the contact and task classes. In a previous version, these classes did not utilize methods that set values for the variables inside of the contact or task object. Thus, there set methods were not testable. After I caught the mistake and refactored these, test coverage increased greatly.

# Technically Sound Code

Error handling was utilized in my project to verify that my code implementation was technically sound. By adding error handling, I was able to see exactly what mistake was made when one was made. An example of this would be in the appointment class where I added error handling where I was implementing the requirements for each variable in the appointment object. Since the appointment id could not be larger than 10 digits, an error would be thrown in the event that a user attempted to enter an appointment id that was larger than 10 digits. The also made testing easier as I had a way to negative test specific methods.

Running the application after new code is implemented is a good way to ensure that new changes did not have a negative impact on the performance of the application. I did this frequently when writing the application, so that if there were errors when running the code, I would have a minimal amount of code to go through the fix the problem. When all of my changes were complete and the application could run without error, I could then focus on unit testing. Ensuring my unit tests were passing was the last check to making sure that my code was technically sound.

# Efficient Code

There are multiple approaches I used to ensure my code was efficient. Throughout the project, I used a uniform structure through each of the classes. For example, if a developer were to look at the task service class as well as the contact service class, they would find similarities in how I approached implementations, since they to very similar things. Inside of the contact service class, I have a method that allows users to add a newly created contact object to an array. Inside of the task service class I have a method that allows users to add a newly created task object to an array. These methods follow the same structure for readability.

In addition to making sure my code followed a similar structure throughout the project, I also ensured that I was using the minimal amount of code to fulfill the requirements. Inside of the appointment service class, I used a simple for loops to loop through each item in the array in order to locate a specific appointment id. There are many ways I could have accomplished this using more code, for example, I could have written out if else statements for the objects inside of the array. However, the for loop was the more efficient approach.

Readable naming conventions were also implemented to make my code more efficient. Being able to read what is going on inside of the code is just as important as making sure the code runs. Using a readable naming convention ensures there is no ambiguity as to what a variable is referring to. An example of this would be in the contact class, where variables are clearly named as contact id, contact address, contact phone number, etc. When I felt like code blocks needed more clarity as to what function they served I added commenting. This can be seen inside of the task service class where I used comments to describe the function of each method.

# Techniques Employed

The first type of testing I used when approaching this project was static testing. For this type of testing, I read through the requirements to verify there was no ambiguity or error. If there had been any ambiguity, missing requirements, or possible errors, I would have asked questions to gain more clarity. By using the review technique of the requirements documentation, I was able to verify there were no errors prior to implementation changes and beginning development. One exact example of where this testing technique became useful was in inside of the appointment object when we needed to use the Date java structure to create appointments for specific dates and times. I was unsure how this was done initially. So, I did research of how this could be implemented before I began developing this feature. In the grand scheme of things, I ended up saving time by researching how I could add a custom date into the Date structure before developing that feature.

Whitebox testing is another type of testing I utilized when working through each milestone. Whitebox testing is a testing strategy used when the author of the tests is aware of the code implementations and expectations (Hambling, et al, 2015, Ch 4.) Since the application I created for this project had no UI or backend operations, unit testing was my main technique for testing my code changes. Statement testing is the technique I used most when writing the unit tests. To achieve this, I tested specific executable statements when given specific input values. For example, when testing the task service class, I needed to verify that a task would not be added to the task service array if a task that had the same id already existed inside the array. To do this, I added one task and then I attempted to add another task of the same id, with the expectation being that an exception would be thrown.

# Other Techniques: Uses and Implications of Techniques

For static testing, one technique of testing that I did not use for these milestones was the technique of using a flowchart or UML diagram. This technique is useful when a developer needs to visualize how the program will flow or function when users are at different stages of use. One specific way I have used a UML diagram was for a previous project and the application was an admin system to be used by school administrators. On the UML diagram, the flowchart showed what options an administrator could do with the program when they went to a specific part of the system. This ultimately could be used a road map for functionality for the implementation stage of the SDLC.

One white box testing technique that I did not include in my project was integration testing. Integration testing adds value by testing how different components of functionality interact with each other and verifying that no errors occur when they do. I have used integration on several projects in my current development role. These tests are particularly useful when new code changes are being added to an existing project. Integration testing is performed at this stage to make sure new functionality and features work well in the already established project.

Black box testing is another type of testing that has not been implemented in my project. Black box testing focuses on specifications and requirements and does not require the tester to have any knowledge of how the code changes have been implemented. Two techniques of black box testing that were not used in any of the milestones is regression testing and smoke testing (build verification testing). Smoke testing is used to verify the build stability and main components of the application. Regression testing is used to continuously run non-functional and functional tests to ensure that new changes have not affected previous states of the application.

# Caution

I knew that unit tests would need to be written after implementing my changes, so I approached this project with the mindset that I needed make sure that my code was testable. Additionally, I had to work with the mindset that my code might fail until this was proven otherwise. In doing so, I took my time and abundance of caution when creating the unit tests for this project. Many of the classes depended on other classes to function, which created complexity in the project. Understand the relationship between these classes were imperative to creating test cases that covered the full scope of what I had implemented.

# Bias

I was the author of the project, which means I had inherent bias when it can to the authoring of the tests. Bias in terms of software testing becomes an issue when a developer might think their code changes do not need to be tested, because they implemented the change. Similarly, bias can be problematic in testing when the developer might not consider different edge case scenarios, so the code is only tests one way. An example of this would be a developer only writing unit tests that cover the happy path scenario and not thinking negative testing is needed. Therefore, it is important to have other developers review and test code changes. I did not have other developers testing my code, so I had to consciously mitigate testing bias as much as possible when it came time to write tests. I did by test planning. In the test planning phase, I was able to think about different scenarios that would test if my code was throwing errors if errors were present.

# Discipline

Cutting corners where testing is involved can be costly to companies, annoying or even dangerous to users, and can be hard to fix. A certain level of discipline when it comes quality assurance is necessary because it helps ensure that bugs are not making it to the production environment of an application. I think in some cases technical debt can be hard to predict and avoid. For example, a customer requirement that would change previously implemented code or when code/frameworks deprecate. However, there are ways to decrease the amount of technical debt. To ensure a minimal amount of technical debt, it is important to research the best approach to a problem, take time going over requirements, code in a way that makes space for future implementations to reduce refactoring time, and have an open mind to other developers reviewing changes and providing input.

References:

Hambling, Brian Morgan, Peter Samaroo, Angelina Thompson, Geoff Williams, Peter. (2015). Software Testing - An ISTQB-BCS Certified Tester Foundation Guide (3rd Edition) - BCS The Chartered Institute for IT. Retrieved from https://app.knovel.com/hotlink/pdf/id:kt00UC2IE3/software-testing-an-istqb/reviews-test-process