**7-2 Project Two Submission**

Christine R. Emerson

Department of Computer Science, Southern New Hampshire University

CS-320: Software Test Automations

Dr. Toledo Lopez

April 10, 2023

**Summary and Reflections Report**

1. **Summary**
   1. Describe your unit testing approach for each of the three features.
      1. To what extent was your approach **aligned to the software requirements**? Support your claims with specific evidence.

The testing phase is essential in detecting defects/bugs, ensuring software requirements are met before delivering the software to the client and guaranteeing the quality of the software. In addition, it makes the software more reliable and easier to use. Software that has been thoroughly tested ensures a high level of performance, reliability, and efficiency. Testing with JUnit ensures developers' code meets the client's requirements. I divided the requirements into more manageable and smaller sections to create my test cases. For example, in Task.java, I started by ensuring that the task Id, name, and descriptions were not null, had correct lengths, and were updatable (if applicable). The process begins by throwing illegal arguments (error messages) when the length is incorrect or if the argument is null. Next, setting the variables in Task.Java enables them to be updated because the only one that should not be updated is the task id which was not set. For example:

Graphical user interface, text, email

Description automatically generated

In TaskTest.java, ContactTest.java, and AppointmentTest.java is where I used JUnit tests to ensure those requirements were being properly implemented, for example, with setting the task name:

***TaskTest.java***

Graphical user interface, text, email

Description automatically generated

* + 1. Defend the overall quality of your JUnit tests. In other words, how do you know your JUnit tests were **effective** based on the coverage percentage?

JUnit test quality can be evaluated based on their coverage results. It is always necessary to analyze the entire code for all test cases. As a result, we have definitive and statistical proof that any code that can be tested has been tested. In general, 80% or higher test coverage is considered reasonable. When running the JUnit test on the ContactService, TaskService code, and AppointmentService code, the JUnit and Coverage tests were conducted to measure how much of the code ran during the unit test. As the completed test indicates for all classes, the JUnit tests were completed quickly, with all files running with no errors or failures. As shown below:

Graphical user interface, text, application

Description automatically generated

Indicating that the code is intact, efficient, and error-free; however, these JUnit tests do not cover incorrect implementations. I am confident that my JUnit tests were effective based on the coverage percentage of every class presented below:

Text

Description automatically generatedText

Description automatically generated

You can see all methods have over 100% coverage (which is over the acceptable 80%), indicating that the percentage of tested code was effective. I created effective testing by checking that my work aligned with the requirements. Each part of the test case and each test case were covered within the requirements given. For example, in AppointmentService.java, once I finished one method, I would go to AppointmentServiceTest.java and write the test for that method. As shown:

***AppointmentService***

***Graphical user interface, text

Description automatically generated***

To ensure that the appointment gets properly deleted it is tested here:

***AppointmentServiceTest***

***Text

Description automatically generated***

Then once all was done, I went through the contactService to ensure all requirements were met, such as the contact service was able to add contacts with a unique ID, delete contacts per contact ID and update contact fields per contact ID (firstName, lastName, phoneNum, and address updatable).In addition, I did the same for the taskService class; the task service was able to add tasks with a unique ID, delete tasks per task ID, and update task fields per task ID (name and description updateable). The same goes for appointmentService as well. As a result, this ensured that the requirements were met, and the JUnit tests were effective.

* 1. Describe your experience writing the JUnit tests.
     1. How did you ensure that your code was **technically sound**? Cite specific lines of code from your tests to illustrate.

I ensured my code was technically sound by making it syntactically accurate, logically concise, and modular. For example, below, the code searches the dictionary to get a appointment id instead of going through every id individually and uses Boolean to minimize unnecessary for loops which helps keep code minimal and organized, as shown: Text

Description automatically generated

I kept the code simple as possible and met each requirement without error. The code has been tested and functions as expected. For example, in the contact class, I used an if statement to iterate over each field to ensure they were not null and over the required length and throw an exception if the field does not meet those requirements:

Text

Description automatically generated

Then each value is assigned, and the getter methods are established to access each variable from other classes. As well as setter methods, certain variables that should be updated can be updated. This was done for the following classes: Contact.java, Task.java, and Appointment.java.

* + 1. How did you ensure that your code was **efficient**? Cite specific lines of code from your tests to illustrate.

To ensure my code was efficient, I removed any unnecessary code or code that goes to redundant processing. I ensured the best speed or run time for completing the algorithm, as shown:

Graphical user interface, text, application

Description automatically generated

I made use of reusable components wherever possible. I made use of error and exception handling at all layers of software. The exception handling of code can be traced using JUnit. The code can be tested to see if it throws a desired exception and adds in the @BeforeAll annotation so redundancy in the code isn't present. @Test annotation is used along with the expected parameter, as shown:

***TaskServiceTest.java***:

Text

Description automatically generated

In addition to developing programming code compatible with the design logic and flow, I also followed appropriate coding practices. For example, I used the best keywords, data types and variables, and other available programming concepts to implement the algorithm.

1. **Reflection**
   1. Testing Techniques
      1. What were the **software testing techniques** that you employed in this project? Describe their characteristics using specific details.

The modules in this course have been tested with a combination of white-box and black-box approaches, including system, unit, and static testing. When creating an application or software, black-box, and white-box testing should almost always be combined. They analyze your software from different perspectives to better find any issues before it is released. By comparing code to specifications, static testing identifies bugs in the code. After one of my JUnit tests failed, I often used this technique to locate and fix the offending code. Creating the Contact, Task, and Appointment classes before anything else was necessary to meet the client's requirements. Each of which had service classes, ContactService, TaskService, and AppointmentService. Each class was tested against the specifications with J-Unit tests to ensure its functionality met the customer's requirements. Each class was tested using J-Unit directly. Before testing could be performed, the code had to be written according to the specifications.

For example, in the Task.java and TaskService files, requirements needed to be met, such as the ability to add tasks with a unique task Id that cannot be null, the length cannot be longer than ten characters, and it cannot be updated. A task object must have a required name String field no longer than twenty characters, not null, and updateable. The task object shall have a required description of no longer than 50 characters and shouldn't be null and be updatable. In addition, tasks can be deleted according to the task Id. For example, in the *Task.java* file:

Text

Description automatically generated

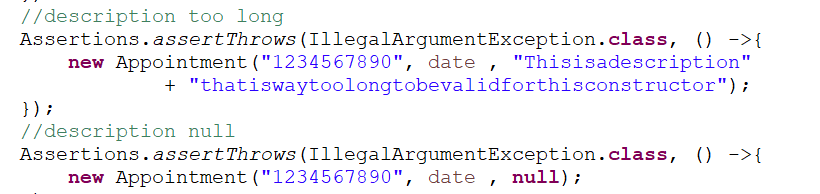
A line of code like the one above ensures that the taskId, taskName, and taskDescription do not fall within the parameters of being null or invalid length. If they do, they will throw an IllegalArgumentException, which was hardcoded in the method. The assertTrue method from the assertions API built into Java was chosen to test for the thrown during JUnit testing. It is the idea that if the class is properly constructed, there will be no exceptions, and the methods will work when passed the correct parameters is true. For example:

***ContactTest.java***Text

Description automatically generated

Furthermore, there was a method that tested whether an exception was thrown when incorrect data was passed. An exception should be thrown if the method works correctly. The tests were also run with coverage to ensure that all J Unit tests covered as much code as possible. For example:

***AppointmentTest.java***



In addition, testing the functionality of updating objects using assertTrue, set, and get and the functionality of deleting objects by adding new objects and using a method to delete the contact by getting the object by the Id, which all tests were successful in each class-tested for this test. For example:

***ContactServiceTest.java***

Text

Description automatically generated with medium confidence

The simple act of throwing an exception when necessary is not enough. It is also crucial to catch the correct and expected exceptions.

* + 1. What are the **other software testing techniques** that you did not use for this project? Describe their characteristics using specific details. **<Write your answer>**

There were no integration tests performed. Integration testing would test the entire application, bringing all the systems together for testing. In this case, the system and integration tests would be the same. I could, in theory, validate that the TaskService, AppointmentService, and any other classes all work together well at their integration layer – I have not. Typically, this testing is conducted to detect defects in an application's interface, but this project did not have an interface, so it was unnecessary. In addition, I did no automated testing – all my JUnit tests were manually run. In theory, I could have had a service that ran the JUnit tests after each build-event in the IDE. This is more important for large applications. There was also no security scanning performed. Libraries and components need to be scanned for security vulnerabilities, e.g., SPRING or even the Java components themselves. In this case, I am not extensively using any libraries, databases, or shared components so security testing could wait.

* + 1. For each of the techniques you discussed, explain the **practical uses and implications** for different software development projects and situations.

**Automatic unit** testing isolates codes to be tested independently in a test environment by replicating them, revealing dependencies between code units. Automatic Testing relies heavily on unit testing tools and frameworks. Automated tests should be built into the continuous integration pipeline when continuous delivery is being used - especially for large applications. Automated tests can be run at any stage of the development process, from development to deployment. Most of my projects run Unit tests, System tests at build time, and Integration tests just before deployment.

**Unit Testing** ensures that every unit works properly and can cause problems if not performed, significantly affecting the program's performance. Code can also be hardened by unit tests to prevent missed logic branches, like what if the value is null, and missed specifications, like how long a string can be.

**Static Testing** **checks software application defects without executing the code. Therefore, it is easier to identify and solve errors at an early stage of development with static Testing. It also helps find errors that Dynamic Testing may not see.**

**System Testing**is always essential to ensure the final product meets all client specifications, especially where data is concerned, which is accomplished by integrating different components into a mockup to examine how the completed system will operate. Comparatively, **Integration Testing** ensures that connected units collaborate effectively. The more interconnected the components are within the code base, the more critical integration testing becomes. Integration testing typically involves top-down, bottom-up, ad-hoc, and backbone integration.

**Security Testing** identifies vulnerabilities and risks associated with applications, systems, and networks. During security testing, potential threats to the system are assessed. The system's security is tested to identify potential security threats by performing both positive and negative tests. Identifying the dangers in the system and measuring its potential vulnerabilities is the primary goal of security testing so that the threats can be encountered, and the system doesn't stop functioning or cannot be exploited.

* 1. Mindset
     1. Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ **caution**? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.

Adapting my mindset was a necessity for this project. To apply tests to my code, I had to develop a new way of thinking. At first, I had some trouble getting the hang of this way of thinking, but I eventually got the hang of it. To ensure I wasn't missing anything, I treaded cautiously throughout the modules. Due to my inexperience with testing, I would write a line of code in a specific class, then go into the class's testing class and write a corresponding code to test it. I did this for every module. For example, in Contact.java and ContactTest.java ensuring the ID was not null:

Graphical user interface, text

Description automatically generatedText

Description automatically generated

 Since all of the code needs to work together **cohesively**, it is essential to appreciate the complexity and interrelationships of the code. A large application can become difficult to maintain, and if one section has problems, it can cause many other sections to develop similar problems. The coverage test will also help you figure out precisely what is being tested, even though you may think you are testing everything. For instance, my contactService and taskService classes had additional methods that were not needed or used. Through the coverage test and Professor's help, I realized I could cut those methods out because I already had a more efficient code to update the task and contacts, which improved my coverage from 85% and 86% to 100%.

* + 1. Assess the ways you tried to limit **bias** in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.

Due to the nature of this project, I don't believe my code contains any bias, as it was more about following directions to meet simple requirements. In this project, I kept my bias to a minimum by not getting personally involved and focusing solely on the requirements. Whenever I had a question, I asked the Professor for clarification instead of assuming or using my bias to judge. Defining bias can be done based on its conscious and unconscious nature. Laws and commissions often protect people's rights due to conscious biases. Being able to distinguish between rewriting code due to a legitimate need versus your own biases distinguishes between a good developer and a great one. During my research on bias in coding, I discovered an algorithmic bias, where a developer might use an algorithm that favors one group of users or discriminates against another, such as artificial intelligence. For example, facial recognition software has been linked to racist bots.

You might be concerned about bias if you are responsible for testing your code because, often, unconscious bias will prevent you from fixing it if you don't know about it. Everyone subconsciously has biases, whether they are aware of them or not. Parents, for instance, do not see faults in their children because they made them, so they are biased toward them. The same goes for coding. When you create a piece of code, it is almost like your child. Without someone else pointing out any biases and faults in your creation, seeing them is sometimes impossible. With no prior experience and little testing knowledge, I completed this assignment as efficiently as possible. I did my best with the resources I had at my disposal; as stated above, the coding, to my knowledge, was not biased.

* + 1. Finally, evaluate the importance of being **disciplined** in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.

1. It is absolutely important not to cut corners when writing or testing code because you are
2. learning how to write it for future employers. Lack of care in programming shows the personality
3. and work ethic of the developer. Which reflect negatively on them, whether intentional or
4. unintentional. I think the best way to avoid practitioner in the field is to practice.

In writing or testing code, you should never cut corners, especially as you learn how to write it for a prospective employer. A developer's work ethic and personality can be determined by their lack of care in programming. Whether intentional or unintentional, they reflect negatively on themselves and the company they represent. As discussed this week, minor oversights and conditions can have significant repercussions, ranging from a failed link, lost lives, and injuries to millions of dollars in damage. A company's reputation and profitability can be destroyed by cutting corners. Practitioners should be thorough in their testing and then automate it so they can focus on new methods to test for and encompass all previous testing methods to avoid technical debt. Over time, improving quality and reducing technical debt will be achieved by finding new scenarios and methods to test for. Learning is never complete for developers and testers. We are responsible for paying attention to every detail and developing new testing strategies.