Jasmine Zeng

Dr. Ramsey Kraya

CS 320

18 June 2023

**7-2: Summary and Reflections Report**

**Summary**

Unit Testing Approach

*Alignment to Requirements*

My approach was aligned with the customer’s software requirements. This was because my back-end service programs were modeled after the customer’s requirements. For each service, the customer required that we construct a data set consisting of certain data store variables. Each data store variable must not exceed specific character limits and values. The customer also required that the system should be able to update a given data store variable when needed. The data sets for the Appointment, Contact, and Task services were constructed in the Appointment.java, Contact.java, and Task.java files, respectively. Each data store variable in the data set was also checked for character limits and values in the same files. The system was able to update each data store variable for each service in the AppointmentService,java, Contact.java, and Task.java files.

*Effective Tests*

My JUnit integrations ensured that the program was tested comprehensively and effectively. As mentioned in the *Alignment to Requirements* section, I developed the services programs based on the customer’s requirements. In effect, I developed a JUnit test for each of these requirements once the program was ready for testing; JUnit tests were made for character limits and update functions for each data store variable. All tests passed, indicating complete coverage.

JUnit Testing Experience

*Technically Sound Code*

I ensured that my code was technically sound by considering other factors of the program that the customer may not have requested.

One instance was that the customer requested a dataset of specific data store variables for each of the services. To ensure that the dataset would not be corrupted from null values, I incorporated code that added filler values. This can be inspected in lines 22 through 23 and lines 31 through 32 in Appointment.java for the appointment date and description variables, respectively. I followed suit for other data store variables in the Contact Service and Task Service projects.

Another instance was that the customer requested character limits of specific data store variables for each of the services. The program first tests if the variable is null, per lines 31 to 32 in Appointment.java. If not, the program moves onto lines 33 to 34, which tests for the character limit of the data store variable. If it exceeds the customer-specified limit, the program will take the first X number of characters in the variable, with X indicating the customer-specified limit. In the case of this example, the program only takes the first customer-specified 50 characters of the appointment description if it exceeds the limit.

*Efficient Code*

I ensured that my code was efficient by safeguarding the program and data with layers of logic. Using the second example in the *Technically Sound Code* section, lines 31 to 37 in Appointment.java tested the intake of data for the appointment description variable. The program first checks if the variable is null before proceeding with data intake. If it is null, the program provides a filler value to prevent that entire dataset from being lost or corrupted. If it is not null, the program then checks if it exceeds the customer-specified 50 character limit. If it does, then the program only takes the first 50 characters of the string. If it does not exceed the limit, then the program stores that string in the appointment description variable. Providing data checks in this order allow data to be considered for customer-specified factors and factors of data integrity. I replicated this decision branching for all other data store variables requiring a character limit in the Contact Service and Task Service programs as well.

**Reflection**

Testing Techniques

*Techniques Employed*

I used the unit testing software testing technique in this program to ensure that my program passed all customer requirements. Unit testing allowed me to test each component of the application. More specifically, I was able to test the character limits of each data store variable. This can be shown in the first few tests of the AppointmentTest.java, ContactTest.java, and TaskTest.java files. In the AppointmentServiceTest.java, ContactServiceTest.java, and TaskServiceTest.java files, I integrated unit tests for each function that updated each data store variable. This allowed me to pinpoint which functions were not working, if any, and target what may have caused the error. White box testing was another software testing technique I used for the service programs. I was able to utilize customer requirements and my background knowledge on Java and object-oriented programming to develop these service programs. This made me better equipped to run unit tests and anticipate any additional logic that may need to be incorporated.

*Other Techniques*

Integration testing was a technique I did not use when developing this program. However, I felt this might have made the software testing process stronger. Integration testing involves running a test against a real database. This would help ensure that the program runs properly and in accordance with the customer’s database. Additionally, the database may provide additional scenarios or behaviors neither the customer nor I may have considered.

*Uses and Implications of Techniques*

Unit testing is practical for complex programs that require testing on all levels and layers of a program. From data store variables to functions, unit testing would be a good fit for customers and developers looking to construct and update vast sets of data. However, other testing techniques may be required to test the program altogether. Integrated testing helps test programs against real data and sees how the functions work together.

White box testing is a good method for developing programs that involve layers of complexity. Having prior knowledge of the programming language and its integrations help developers find exactly which tools are necessary for developing a robust program that fits customers’ needs. Implications of using white box testing are the opposite, which is black box testing. It is not guaranteed that developers will know everything that should be incorporated into a system or program, so it is better to go in assuming there are other unknown elements. Programmers need to go in with a flexible mindset and be prepared to address and/or learn about issues when they arise.

Mindset

*Caution*

I employed caution throughout all phases of development. Phases included collecting requirements, designing the program, coding the program, and testing the program. I made sure to use the right Java tools and integrate object-oriented programming so the program could use the right functions and remain organized. It was important to be conscious of the complexity and interrelationships of the code I was testing because there was a lot of data handling and management occurring simultaneously. For example, in the Appointment.java, Contact.java, and Task.java files, I made sure to construct the dataset variables for each one so it contained an array of the proper data store values and variables. I also ensured that getters and setters were used for each variable so the system would be able to retrieve, set, and/or update information when needed. If I did not incorporate these components into the program, datasets could be corrupted or lost, affecting data integrity overall and may cause larger issues down the road.

*Bias*

I eliminated bias from my code by adhering to customer requirements and the assignment rubric. In this manner, I would be working in the interests of those parties and avoid adding any of my personal embellishments that may affect the functionality or integrity of the program. I also deployed software testing techniques and Java tools that would enhance the efficiency of the program, such as the JUnit tests in the AppointmentTest.java, ContactTest.java, TaskTest.java, AppointmentServiceTest.java, ContactServiceTest.java, and TaskServiceTest.java files. I also kept the best programming practices in mind to ensure I was writing clean code, which made latter development phases easier since I resolved most known potential issues before running JUnit tests. Bias would definitely be a factor of consideration when software developers test their own code. They may abridge functions or decision branches when other components may need to be incorporated to preserve data integrity, as such in lines 31 to 37 in Appointment.java. Having other parties review code also eliminate fatigue of developers from repeatedly checking their own code. Different perspectives offer fresh eyes to the code, possibly picking up errors that may not have been detected in previous rounds of testing and editing.

*Discipline*

Discipline is of utmost importance when software engineers develop programs. Being comprehensive throughout each stage ensures that the program is planned, designed, developed, and tested to the highest caliber. Preparation prevents errors that may snowball into larger issues if the opposite were done. As such, software engineers must not cut corners; this minimizes human error and helps to detect as many issues as possible before the development timeline progresses further. This prevents technical debt as a practitioner in the field – when you maximize best practices, you minimize technical risk. For example, if I did not implement the decision branching in lines 31 to 37 in Appointment.java, the system may not be able to collect data for appointments, contacts, or tasks for the respective service programs. It would run into errors and be unable to decide what to do with input. Having this decision branch for each of the data store variables helps the program “screen” the data and decide what to do for each instance. The program will be more prepared in this manner when it goes live, compared to the opposite.