CS320 Project 2

Within the past 3 weeks, I have had the opportunity to create 3 milestones that include a contact service, task service, and appointment service. Using Junit5 testing along with other software testing techniques, I was able to gain confidence in the quality and security of my code. Below I will describe the different techniques, mindsets, and tests performed involving my code. I will also provide rationale for my decisions involving the quality of the code and the tests I chose to execute.

I believe my testing approach aligned to the software requirements. I did this by providing unit tests and conditional checks to ensure the String variables were not null and contained the valid number of characters specified. An example of this is the first name field in the Contact class. In the mutator for this variable, I provided an if statement to check if a given first name was null or was greater than 10 characters. If it was, it threw an exception with an appropriate error message. If it met all requirements of input validation for the first name, it was set to be used by a Contact object.

I also made sure that my data container for the service classes performed the necessary operations specified in the guidelines (add, remove, update). This includes updating only the information specified for each class and using a unique identification (id) for carrying out the operations. An example of this is my use of an arrayList in my TaskService class. Tasks added to the arrayList were given a unique id that could then be used to remove tasks and update task information. If the unique id was not found within the arrayList, an exception was thrown. If it was not thrown, then the operation was carried out.

Finally, I ensured my id field was unique and NOT updateable. I did this by setting the mutator for the id field to private and not providing an update function for id in the respective service class. I also used the UUID class to create a unique id and performed thousands of comparisons in testing to ensure a unique id would not be duplicated.

I know the tests I carried out were effective based on multiple components. The first being that I had 100% function coverage in my code for all classes. I also was able to obtain 92% (24 out of 26) branch coverage for my milestone 3 assignment, 88% (15 out of 17) for my milestone 4 assignment, and 86% (12 out of 14) for my milestone 5 assignment. The only branches that did not execute in each of these tests was the one to check if the unique id was already being used by another element in the arrayList as well as the branch that specifies a remove was unsuccessful after an attempt to remove an element. However, the test for the unique id specifically carried out correct expectations over thousands of times, so this gave me confidence that the uniqueness of my id is high. I also had an outer if statement that searched my arrayList for the matching id before attempting to remove it, which weeded out most, if not all, attempts to that would have procced the removal unsuccessful message. Because of this, I had nearly 100% statement coverage as well for every class, excluding the exception that would have been thrown in the beforementioned conditional branch and the print statement that a removal was unsuccessful.

I ensured my code was sound by performing multiple tests. This included using whitelisting and blacklisting. For whitelisting, I formulated what input would be acceptable (input must align with length guidelines and cannot be null) and used blacklisting to reject bad input I purposely included in my tests. An example of this involves my ContactServiceTest class. In the class, my testOne (lines 56-68) test attempted to add a contact to the arrayList using valid input while testTwo (lines 70-92) used invalid input. Because of the whitelisting techniques I enabled in my Contact class to ensure all String fields met the appropriate requirements, testTwo threw an exception because at least one of the fields did not meet its respective requirements while testOne passed.

Another way I checked to make sure my code was sound was by initializing the fields with null in a class’s default constructor so the tests would fail if operations were not carried out successfully. An example of this is line 24 in my Task class that assigns id, name, and description with null. With this, if my mutator methods do not perform as intended, an exception is thrown for the field being null. This was tested within my TaskTest class during testTwo (lines 45-59) in which I used assertThrows to ensure an exception was thrown if I attempted to create a Task object using at least 1 null field.

Finally, I created test objects and arrayLists to test if they would be fully functional in a live environment. An example of this is within my TaskServiceTest class in lines 39-43. I created multiple Task objects that were added into my arrayList. Each element in the arrayList was tested to ensure the class would stay functional even if the arrayList was scaled to a different size.

During my tests, I used several different techniques to make my code efficient. One way was by creating variables for data that would be used multiple times throughout the test classes. An example of this is in my ContactTest class in my setup method (lines 22-38). I made variables such as GOOD\_F\_NAME so they could be reusable and applied to different tests.

I also used Jupiter annotation to setup variables before all tests and to build my arrayList before every test and then clear it after every test. Examples of this are in my ContactServiceTest class and include my setup method (lines 24-43), my create\_List method (lines 45-49) and my popList method (lines 51-54). With this I made sure that my arrayList was not holding any unexpected elements, but also made sure the necessary elements were loaded for each respective test.

While there were some testing techniques not utilized for these milestones, I will explain their absence along with the used techniques and their implications pertaining to different situations. One testing technique I used was unit testing. When building my code, I created methods that reflected the technical and functional requirements of each assignment. An example of a method I used was setPhone() that ensured the functional requirements were met for the phone number field in the Contact object for Milestone 3. Another example was in Milestone 4 where I used the method updateDescription() that reflected a functional requirement in the guidelines with updating a Task object in its respective data container. A technical requirement I fulfilled was the use of a search() method in all milestone assignments. Though it was not a functional requirement specified in the guidelines, it was a technical requirement that helps to fulfill the requirements of adding, removing, and updating elements in the data container. Creating tests for each of these individual methods in Junit5 allowed me to test the program in units with multiple inputs rather than in a lump sum. For nearly all software projects, a unit test’s practical use involves isolating written code for testing and is often a steppingstone for other tests such as integration testing. It also helps developers fix or identify a problem earlier in the SDLC.

During my unit testing, I also performed manual regression testing. Regression Testing is practical in making sure a change in the code did not affect the integrity or functionality of another part of the code. I performed manual regression testing by running all my tests whenever I made a change (even a small one I found unimpactful) to the code. This type of testing paid off during my incremental development and refinement of my code. As I built the program up, I was testing each method such as setName() in milestone 4 to ensure it met requirements. When I added the product of that method to an object (this.name = name;), I ran the test again. Then, as the object was modified in different forms in the data container, I ran all tests again. Also, as I refactored code to make it readable and usable, I ran tests to make sure the refinement process did not bring about a defect.

Finally, I used some exploratory testing to create negative test cases and conditions to support the logic of the program. An example of this is the last name field in the milestone 3 assignment. In the requirements, the last name could not be null and could not exceed 10 characters. While testing against the functional requirements of this field, I noticed that an empty string (“”) would pass tests. This is illogical as a last name is required and everyone has a last name. Also, while not being technically null, an empty string also does not give value besides being a string of length 0. This enticed me to add this condition to the last name along with other applicable fields such as description. In my Junit5 tests, I also used edge cases and random test cases to make sure expectations of the output met the actual output. An example of this is making the test case involving the name field in the task object during milestone 4 a combination of letters, digits, and special characters. Exploratory testing’s practical use of finding bugs that would not be found during the structured phases of testing paid off, as I was able to improve upon the quality of the program and enhance my test case possibilities.

One testing technique that was omitted from this assignment was automated testing. Automated testing is the use of testing scripts automatically within an application instead of it being manually tested by humans and the system. They are specifically used to save time and money that would be consumed by manual tests. They can also be used to provide periodical assessments of the program such as testing a portion of the program daily at midnight to ensure it upholds its integrity. I do not have the software necessary for automated testing and the situation does not involve spending on tests, so though it can be very beneficial for automated testing, it was not utilized within this program.

In acting the role of software tester, I had to employ caution with my mindset on how to handle testing. One caution, that regards to the first principle of Software Testing, is that tests show defects, not their absence (*7 Principles of Software Testing — Learn with examples*, n.d.). Basically, even though I can feel confident that my tests were able to detect many defects due to input invalidation, improper setting of a date for an appointment, etc., I must take caution in believing that my program is defect free. This is because the complexity of providing a wider range of user input and having this input contain relationships with my data structure and objects makes many possibilities for defects to occur that could not be caught even if testing was exhaustive.

On the note of exhaustive, I also had to understand that exhaustive testing is impossible. Instead, according to the second principle of Software Testing, it is often better to optimize tests using black and white box testing strategies (*7 Principles of Software Testing — Learn with examples*, n.d.). This means using effective and varying test cases to avoid the pesticide paradox. It also means analyzing which part of my software will be in most use by users to understand the likely areas a defect occurs, which aligns with the Pareto Principle (*7 Principles of Software Testing — Learn with examples*, n.d.).

In my testing I also had to limit bias. In other words, I had to avoid the phenomenon of Confirmation Bias to where I am more likely to verify my own prediction of output with the real output by creating test cases and/or procedures that would bring about positive tests rather than negative ones (Calikli, G., Uzundag, B., & Bener, A., 2010). There are ways I went about limiting the bias in my code. The first step I took was sharing my tests with tutors for recommendations. With this, I was able to receive feedback on providing different tests and strategies that would lower the bias in my project. One specific example of this was a tutor suggesting I make more comparisons for my Unique ID test. Even though my test compared the uniqueness of a created ID with ones in the data container a thousand times, they believed that increasing that number at least another thousand would increase the confidence of its uniqueness. I also sought my instructor who sent test cases to use for my date field in the AppointmentServiceTest to ensure I set my date field up correctly. Another additional way I tried to prevent bias was creating failed tests as I conducted my successful tests. This meant checking not only that my tests passed with the proper conditions, but also failed with the wrong ones.

Finally, there is importance in being disciplined as a Software Tester. Unfortunately, some company structures and lifecycles may make it harder for discipline to be upheld. One example of this is the Big Bang Model which some acquaint with a strict Waterfall Method. In these models, testing is usually performed when the software is “done” and just about ready to be sent out. Testing can be considered problematic for many reasons in this situation: 1) Because the model is linear, finding defects at this stage mostly equates to making the defects known instead of resolved and 2) the longer you take to test, the longer you are withholding the customer from the product and possibly being viewed as wasting money in the eyes of management (*Testing Axioms*, 2000). Though opinionated, I feel that cutting corners for testing can potentially waste even more money than if the testing phase occurs longer than what other parties intended. In fact, according to the Cost Escalation Model, the earlier a defect is known and fixed, the more likely you are to save money and time in the overall project. However, due to the diversity of Software companies and their procedures/policies, this model is not always upheld. In tackling this, I plan to research the company I am using and ensure that their principles of Software Testing align with the best modern practices in the industry. I also plan to educate myself with new modern practices, even if they contradict my previous practices, to discover which ways are going to support my progression and quality of deliverables best.

References

*7 Principles of Software Testing — Learn with examples*. (n.d.). <https://tuskr.app/learn/software-testing-principles>

Calikli, G., Uzundag, B., & Bener, A. (2010, September). *Confirmation Bias in Software Development and Testing: An Analysis of the Effects of Company Size, Experience and Reasoning Skills*. Research Gate. https://www.researchgate.net/publication/235430372\_Confirmation\_Bias\_in\_Software\_Development\_and\_Testing\_An\_Analysis\_of\_the\_Effects\_of\_Company\_Size\_Experience\_and\_Reasoning\_Skills

*Testing Axioms.* (2020, November). InformIT. https://www.informit.com/articles/article.aspx?p=19796