Douglas Bowers

CS-320

6/20/2024

7-2 Project Two Submission

**Summary**

My approach was aligned to the software requirements because I was able to implement testing for the required methods in the base code. In the contact class, one of the stated requirements was that both the first and last name could not be longer than 10 characters. With this, I was able to implement JUnit testing within the ContactTest class with the assertion “Assertions.assertThrows(IllegalArgumentException.class…).” This was used to test if the given inputs were too long. The requirements for the task class stated that the task ID could also not be longer than 10 characters. A test class was also made called TaskTest, and JUnit testing was implemented to throw an illegal argument flag if the task ID was greater than 10 characters.

With the available resources within each module, I was able to improve my JUnit tests over time. The overall coverage within the contact test was greater than in the task test, however, in both functions the service tests had a higher coverage percentage. With the indication of a positive percentage, this let me know how much the tests were able to cover the functions. I will be using these in the future to ensure that each statement is being properly implemented in at least one test case.

To ensure that the code was technically sound, I used data structures such as arrays and lists. Within the ContactService class, I used an “ArrayList<>()” to create a unique ID that was set to the required length, and implemented a random ID to each new contact that was made. I also used “List<Contact> to get the available contact list and to be referenced when searching. Within the ContactTest class, I used the JUnit testing method of “@BeforeEach” to ensure that each test was referencing setUp(), which contained a set contactID, firstNameTest, LastNameTest, and addressTest assigned to them. Within this also contained the respective versions that were too long to see if the tests would be able to catch it. When completing my JUnit tests I used “assertEquals”, “assertNotNull”, and “assertThrows” to ensure that what was being tested provided the appropriate assertions.

While I am still learning to code, I attempted to use multiple measures to ensure that the code was as efficient as possible. I used multiple references, videos, and resources within the modules to assist me when completing each assignment. I tried to fix errors as I went, running multiple tests and debugging my code when needed. Within the TaskServiceTest class, I used JUnit testing methods such as when updating the name to get the assigned task ID from the task list, and when correct, retrieve the corresponding name for updating. For each test within the assignments, I tested for both correct and incorrect inputs. For example, when a given contact is updated from within the JUnit test for ContactServiceTest, such as the first name, the corresponding contact ID and first name are referenced from the contact list. If the name is not null, then the test will successfully update the first name and apply it to both the contact list and the corresponding contact ID. Each aspect of the tests can update separate entities and apply them to their respective type.

**Reflection**

For each milestone, the software testing techniques I employed would be black box and white box testing. Since they were all specification or structure-based tests, each test case derives directly from the specification of what the system should do. Black box techniques involve equivalent partitioning to test for both valid and invalid inputs, decision tables to test conditions and actions, and state transition testing to ensure that the test events change state or generate an output. Use cases are also created from each test case, setting boundary values to test the boundaries of each requirement. Structure-based testing was used to analyze components, as well as the if/then statements. This helps break down each test into sections. Structure-based testing includes statement, path, and branch coverage.

One form of software testing techniques I did not use are experience-based techniques. Within this technique, users and testers use their experience to determine the most important areas of a system, exercising consistency with expected use on likely sites of error. This technique also includes error guessing and exploratory testing. Error guessing is where prior experience is used to determine which test would be the most optimal choice to analyze the code. Exploratory testing is used to test areas of the code that are lacking specifications. Having limited experience, this was probably best for me to omit from using.

Black-box testing techniques are used when code segments have clear functionality, generally used for outsourced testing. White-box testing techniques are used when the desired outcome of the product being testing is thoroughly understood. Experience based testing techniques are used to identify special tests that may not be easy to capture through formal techniques. It is also used when specifications are either missing or inadequate. Each technique is implemented based on practicality and implications for different types of software development projects and their situations.

When working on this project, I tried to adopt a mindset that was analytical, experimental, and aimed towards growth. I tried to stay cautious, however, as I would use references whenever there was something that I was stuck on or could not figure out what the best practice would be. When testing your code, it is important to appreciate the complexity and interrelationships because it helps improve both quality and performance. When initially testing my code, the coverage was low but after implementing more tests that covered more of the code, the coverage was much higher. These additional tests produced a more technically sound and higher quality product.

To help eliminate bias, I ran multiple tests on every aspect of the code. If it worked the way I wanted it to, or not, I was working to form a hypothesis rather than an assumption. If I only tested certain aspects such as the length of the name being only 10 characters, and omitted the testing of the ID, I would not be able to say I fully tested my code. It is important to test multiple inputs, both valid and invalid, versus just one or the other.

As a software engineering professional, it is important to be disciplined in your commitment to quality because it helps uphold and advance the integrity and reputation of the profession. To ensure a finished, quality product it is important to not cut corners. The expectations of a software engineer are outlined within the software engineering code of ethics, which states that software engineers shall act in a manner that is in the best interest of the client and employer, consistent with the public interest. To avoid technical debt as a practitioner, I plan to implement agile development methods which test code often. This will help to consistently push for higher quality code, provide demonstratable software production, and keep communication between the client and developers open. Following the software engineering code of ethics as well as practicing the agile development methods will help mitigate technical debt in the long run.

Works Cited

Krysik, A. (2023, December 15). *SDLC GUIDE: Key steps of software testing process*.

Stratoflow. <https://stratoflow.com/software-testing-process/>

*Software development process*. GeeksforGeeks. (2024, February 5).

<https://www.geeksforgeeks.org/software-development-process/#>

*Software engineering code - ACM ethics*. ACM Ethics - The Official Site of the Association for

Computing Machinery’s Committee on Professional Ethics. (2022, June 8). <https://ethics.acm.org/code-of-ethics/software-engineering-code/>