CS-320

Professor Alexander

Nicholas Altemoos

12/14/2024

**Final Project**

1. **Summary**  
   My focus on unit testing was adjusted on each feature to fit software requirements of each and focused on trying to include edge cases. In ContactService I focused on each individual methods such as add, update, and delete and that they handled duplicate and invalid IDs. For example I used testAddContactIdDuplicateThrowsException to limit any IDs that would try to be added that would be duplicated. In TaskService I focused on the creation, updates, and retrieval and wanted them to handle constraints like the character limits on name and description. I used *assertThrows*(IllegalArgumentException.class, () ->task.setDescription("012345678901234567890123456789012345678901234567891));")) in TaskServiceTest to test character limit for the description. In AppointmentService I focused on the proper scheduling and that it doesn’t have duplicate instances and valid formats. The quality and effectiveness of my JUint tests was validated when tested for coverage percentage rate and it came back over 80% which showed that most of the code paths and edge cases were tested. An example would be with the updateTask method was tested for valid updates and how it tested when updating a task that wasn’t there such as when I used public void testUpdateTaskNonExistent() { assertThrows(IllegalArgumentException.class, () -> service.updateTask("999", "New Name", "New Description")); }. I worked with technical soundness for example in ContactServiceTest I used assertThrows(IllegalArgumentException.class, () -> service.addContact(null)) to make sure that null input was dealt with properly and used assertEquals tests to make sure the lines had the expected outputs. I worked on efficiency by using @BeforeEach public void setUp() { service = new TaskService(); service.addTask(new Task("123", "Sample Task", "Sample Description")); } to reduce making a new instance for each test reducing having redundant code.
2. **Reflection**
   1. Testing Techniques  
      For techniques used I used Unit testing and Boundary testing. I used Unit testing when it comes to individual methods focused on correctness and robustness. An example is the tests of addContact and updateTask I checked for both successful and failed cases with unique IDs. Unit testing helps us build reliability earlier in the development process and to catch bugs and errors before we integrate. The next technique used was Boundary testing when I tested the input validation with the software requiring character limits. I tested it for example in TaskService for name and description and tested when the exact amount was reached and over. This technique helps us validate edge cases and adds even more robustness and helps reduce risks where input errors can have severe consequences. Some techniques not used were Integration and Performance testing. Integration testing was not used but it would help validate how the features would interact when linked such as making sure the tasks linked to appointments are synched correctly. Performance testing was not used but this would test the whole system when it handles a large number of instances, this helps to identify bottlenecks when the system is stressed.
   2. Mindset  
      In terms of mindset I employed caution when I created testing to check for empty and null values in the updateContact to handle complex input. It’s important to be cautious because it can lead to any number of issues including runtime errors or corruption tables. It’s also important to appreciate the complexity and interrelationships of the tested code because overlooking this can lead to integrity issues when systems and features must interact with each other and can avoid redundant testing and hidden bugs. As a developer we need to watch and try to minimize bias. We reduce it by having a structured testing mindset to avoid creating tests we only believe should work and by writing and developing tests before implementing features such as deleteTask and by using JUnit coverage as a guide we can have stats to back up what we test. We also need to commit to quality and avoid cutting corners. Cutting corners leads to technical debt and future bugs where we have to redo or rework features later in the developmental process. We can avoid technical debt by following disciplined practices with thorough tests and refactoring such as the tests in TaskServiceTest that helped perform repetitive checks.