Project Two

David Carmichael

In project one, I worked to deliver three services. The Contact, Task, and Appointment services were written to meet requirements, and tests were made to ensure functionality and special use case scenarios. I understood throughout development that tests would be written so each method was written with this in mind. I specifically tried my best to make methods that have parameters and those without parameters. This allowed for testing of calls to these methods with and without parameters, which prevents users from causing errors or finding potential bugs. The code for this project was written in the Java coding language, and tests were written using the JUnit 5 framework. In TaskService, I created tests to make sure that task creation with valid input worked flawlessly, and that it throws exceptions for invalid task names or descriptions over 50 characters. An example of one of these tests can be seen in the screenshot below. 

This screenshot shows a test that checks for valid input when a user initiates a new task within the TaskService. I used this approach to align with the software requirements. The requirements specifically stated that there needs to be exceptions for each form of user input. Each service contained specific data validation rules, for example, contact phone numbers must be 10 digits. I validated these constraints with test cases that checked both expected success and failure scenarios. JUnit tests were required to have 90% coverage and I made sure the coverage was greater than 90%. This ensured little to no room for bugs/errors, and was not only a requirement, but also good practice for actual development in the real world. I focused on branch coverage and tested exception handling thoroughly. For example:

@Test

void testInvalidPhoneNumber() {

Exception exception = assertThrows(IllegalArgumentException.class, () -> {

contactService.addContact(new Contact("002", "Alice", "Smith", "999", "test@email.com"));

});

assertTrue(exception.getMessage().contains("Phone number must be 10 digits"));

}

I made sure technical soundness was achieved by using a consistent structure when developing. The tests were made efficient by only testing one thing per test.

I used multiple software testing techniques during the development of this project. For example, unit testing, boundary testing, exception testing, and negative testing techniques were used. Unit testing was used to focus on individual methods. Boundary testing was used for testing inputs. Negative testing was used to test for when users put in invalid inputs. Exception testing was used to make sure these inputs were rejected properly with the correct exception messages. I did not use system testing throughout the development of this project. System testing is used to test the full system in an almost complete environment. It was not applicable here due to the project’s scope, but this technique is crucial for connected systems. I found that unit testing is great for development phases when isolating specific logic errors, while system testing is used before production release on larger programs. My mindset when writing test was to be as careful and cautious as possible. I recognized the interdependencies across services like shared validation logic. I checked each test to make sure no logic was missed in testing. For example, in AppointmentService, I tested both valid future dates and invalid past dates.

I tried to limit bias by writing tests in order to break the program. I realize, as a developer, that users will break the software if it is possible. Most of the time, these errors are found by accident by users trying to use the software as intended, but there are also cases where users try to break the software for any multitude of reasons. Even when I thought I understood what a user might input, I checked a more absurd input, like combinations of letters, numbers, and special characters. As a software engineer, it is important to keep a disciplined commitment to quality. In this course, we talked about the dangers of defects in code and testing, like the ones that were done in this project, which are critical to preventing such defects from going into production. Cutting corners could cost a company millions of dollars so it is important to keep quality in mind while developing software. I plan to avoid these problems by integrating continuous integration pipelines, keeping test coverage thresholds, and keeping my code modular and testable.

Overall, this project emphasized the significance of good unit testing in delivering intricate software. The testing approach I used, through high coverage percentage and thoughtful test design, helped to ensure that the application's core services met all requirements. I had to make sure all three services delivered the requirements laid out by the client, which included checks for valid inputs. This approach is crucial for developing larger projects as it allows the developer to prepare for normal and abnormal interactions from users.   
  
  
  
  
**References**Simmonds, I., & Hills, T. (2023). *Software testing: An ISTQB-BCS certified tester foundation guide* (4th ed.). BCS Learning and Development Ltd.

Gamma, E., Beck, K., & Van Bennekum, A. (2005). *Test-driven development by example*. Addison-Wesley.