CS 320 Project Two
Chris Marrs
02/22/2024

Grand Strand Systems Project Summary and Reflections Report

1. Summary

a. Approach to Unit Testing

i. Alignment to the Software Requirements

The unit testing approach for each of the three features - contact, task, and appointment services - was carefully planned to align with software requirements. Tests were structured to evaluate both expected behavior as well as boundary conditions within the software for an exhaustive assessment of all the system functionality.

Contact Service

The unit tests for the contact service focused on validating its constraints, such as maximum length of a contact ID or format of phone numbers. The test *testContactConstructorWithInvalidID()* in the ContactTest class checks for null or overly long contact IDs that do not adhere to our requirement that contact IDs cannot exceed 10 characters long and must not contain null characters, thus fulfilling compliance with this rule. Similarly, *testSetPhoneInvalid()* verifies whether the application can accurately handle invalid phone numbers by testing this requirement that all phone numbers must contain exactly 10 digits long.

Task Service

The unit tests for the Task Service were focused on validating constraints associated with task objects, including maximum length for ID, name, and description fields as well as errors detected during construction/setting up operations of its constructor and setters.

Example tests include *testTaskConstructorWithInvalidTaskId()* from the TaskTest class which verifies compliance with task ID length requirements: no null or overly long task IDs should be accepted, and compliance should not exceed 10 characters without exception (*testSetNameInvalid()* and *testSetDescriptionInvalid()* respectively).

Appointment Service

The unit tests for the appointment service aimed to verify the unique IDs assigned, valid dates for appointments made through it and length of descriptions given out for service appointments.

The test in AppointmentTest class - *testPastAppointmentDate*() - ensures that the system accurately rejects appointment dates that fall in the past based on requirements stipulating that appointment date fields cannot be null or in the past, which ensures users cannot create appointments using invalid dates which is necessary to maintaining integrity within an appointment scheduling functionality.

The test *testLongDescription()* verifies that the system enforces its maximum length constraint on description fields - in this instance ensuring they do not surpass 50 characters for consistency and readability of appointment details. This step ensures an improved user experience when managing appointments details.

ii. Overall Quality of JUnit Tests

Overall, the JUnit tests used in this project can be considered of high quality based on factors like coverage percentage, thoughtful design of test cases and their efficiency at detecting defects as well as asserting correct behavior of software. Tests provide a solid platform to ensure software meets requirements while functioning perfectly under various scenarios. There were a variety of metrics to ensure that the code was of high quality, was technically sound, and efficient.

Coverage Percentage

JUnit tests' coverage percentage is an essential metric of their quality, and in this project, they were tailored to achieve high coverage - guaranteeing that a significant portion of codebase was covered by tests. For instance, testing of contact service objects covered all possible scenarios of their creation and validation including edge cases like null values and maximum length violations to ensure maximum code path coverage which decreased risks of undetected bugs. The overall coverage percentage exceeded 80% which is the normally acceptable threshold for this type of validation.

Test Case Design

The design of test cases is also key in creating high-quality JUnit tests, and in this project, they were meticulously created to be both thorough and concise. Parameterized tests could further increase efficiency by testing multiple sets of inputs with one piece of code; and grouping the tests into logical groups and using descriptive names for methods facilitated easy understanding and maintenance of the suite of tests.

Asserting Correct Behavior

Beyond finding bugs, tests also validate that software performs properly under different circumstances. For instance, *testTaskConstructorWithValidData()* from the TaskTest class verifies that a task object was properly created with valid input parameters; similarly, *testAppointmentConstructorWithValidData*() in the AppointmentTest class verifies an appointment object was correctly created with future date and description information.

b. Junit Testing Experience

i. Technically Sound

To ensure the technical soundness of my code, attention was made to write clear and concise test cases that directly corresponded with software requirements. For instance, in the ContactTest class *testInvalidFirstName()* specifically checks for the requirement that first names cannot contain nulls or be longer than 10 characters - by aligning my test cases closely with these needs, I was able to effectively validate its functionality.

ii. Ensuring Efficiency

Efficiency was achieved in the test code by eliminating redundancy and prioritizing critical test scenarios. For instance, instead of writing multiple separate tests for every invalid phone number format possible in Contacts, I implemented one single test, *testInvalidPhone()* to do just this task - keeping the test suite manageable and focused at once.

2. <u>Reflection</u>

a. <u>Testing Techniques</u>

i. Employed Techniques

In this project, various software testing techniques were employed to ensure the robustness and reliability of contact, task, and appointment services.

Black-Box Testing - This technique focused on testing software's functionality without considering its internal structure. For instance, tests for the contact service validated its behavior based on various input scenarios (for instance invalid phone numbers or lengthy names) without delving too deeply into implementation details.

Boundary Value Testing - This technique was employed to examine edge cases of input values. Tests were written to examine system behavior when input values approached or approached specific boundary limits, such as an ID of exactly 10 characters or task descriptions with exactly 50 characters.

Equivalence Partitioning - This technique used equivalence partitioning to divide input data into equivalent partitions that could be tested simultaneously, for instance the phone number field in a contact object was divided into valid and invalid categories, then representative values from each partition were tested for quality.

ii. Techniques Not Employed

White-box Testing - involves inspecting the inner workings and structures of an application, although not explicitly employed here in this project. Instead, this technique could be implemented during future iterations to investigate internal logic and paths present within its software.

Decision Table Testing - provides an effective means of simulating complex decision-making logic by visualizing different input combinations with their expected outcomes in tabular form.

State Transition Testing - can be helpful for testing how software transitions between various states, especially complex applications with many states and transitions.

iii. Practical Uses and Implications

The choice of testing techniques depends on various factors, including the complexity of the software, the development stage, and the testing objectives.

Black-box Testing - is widely used for functional testing, where the focus is on verifying the functionality of the software against its specifications or requirements. It is applicable across various levels of testing, including unit, integration, system, and acceptance testing.

This technique is particularly useful in scenarios where the tester does not have access to the internal structure of the software, such as in third-party application testing or when testing from an end-user perspective.

Boundary Value Testing - is particularly useful in scenarios where input values have defined limits or ranges. It is commonly used to test fields that require numerical inputs, date ranges, or any other input with clear boundary conditions.

Equivalence Partitioning - is used to reduce the number of test cases by dividing the input data into equivalent classes or partitions. Each partition is considered to represent the same behavior of the system, so testing one value from each partition is often sufficient.

b. Mindset

i. Caution and Complexity

In this project, it was important to maintain a cautious mindset when considering the interrelationships among various parts of code. For instance, correct functioning of AppointmentService class depends upon validating logic from the Appointment class; and with such intricate interactions present throughout, tests were designed specifically to detect any possible errors caused by these interactions.

ii. Limiting Bias

To mitigate bias when reviewing code, tests were conducted that covered both expected and unexpected scenarios. Testing one's own code can lead to bias due to potential oversights; peer reviews or automated testing tools may help alleviate this concern.

iii. Disciplined

Being disciplined when it comes to quality is critical in avoiding technical debt. This means not cutting corners when writing or testing code under tight deadlines; for example, ensuring all edge cases are covered during tests as done here can contribute significantly to long-term maintainability of code. Avoiding technical debt requires taking an active approach towards quality assurance for sustainable software development projects.