CS 320 SOFTWARE TEST, AUTOMATION QA

PROJECT TWO

NII AMATEY TAGOE

JUNE 2024

AWS.AMAZON.COM defines unit testing as the process where you test the smallest functional unit of code and software testing aids to guarantee code security and quality, and it's a vital portion of software development. One of the best industry practices is to develop software by writing software in smaller bits, and functional units and then write a unit test for each code unit.

My first feature was to create an appointment class that has appointment objects that have a unique appointment ID that cannot be longer than 10 characters, not null and not updatable, an appointment date field not null and not updateable, and finally a description field not longer than 50 characters and not null. The feature also included an appointment service that enables the addition, update, and deletion of unique appointment IDs. To answer the question of what extent my approach aligned with software requirements, for instance, the Appointment class delivers methods to set appointment dates and descriptions, ensuring they are valid. For example, the setAppointmentDate method allows updating the appointment date.

“public void setAppointmentDate(Date appointmentDate) {

validateAppointmentDate(appointmentDate);

this.appointmentDate = appointmentDate;

}”

Another example is the AppointmentService class where I made use of a HashMap to store and manage Appointment objects and the result is allowing for efficient addition, deletion, and retrieval of appointments.

“public void addAppointment(Appointment appointment) {

if (appointments.containsKey(appointment.getAppointmentID())) {

throw new IllegalArgumentException("Appointment ID already exists");

}

appointments.put(appointment.getAppointmentID(), appointment);

}”.

Secondly, my JUnit tests were effective based on the coverage percentage because it showed 100% which means it covered all main functionalities and edge cases for both the Appointment and AppointmentService classes.

My second feature was to create a task class that has task objects that have a unique task ID string that cannot be longer than 10 characters, not null, and not updateable, a name string field that cannot be longer than 20 characters and not null, and a description string field that cannot be longer than 50 characters and not null. The feature also included a task service capable of adding, deleting, and updating unique task IDs. My approach is aligned with the software requirements because, for instance, the Task class confirms the creation of task objects with names, valid and unique IDs, and descriptions with these lines of code below.

“public Task(String taskID, String taskName, String taskDescription) {

validateTaskID(taskID);

validateTaskName(taskName);

validateTaskDescription(taskDescription);

this.taskID = taskID;

this.taskName = taskName;

this.taskDescription = taskDescription;

}” a constructor that authenticates inputs ensuring alignment with constraints of the task objects. The TaskService class certainly delivers methods to delete, add, update, and retrieve tasks and also guarantees no duplicate tasks can be added and tasks can only be updated or deleted if they exist. The line of code below makes it evident.

“public void addTask(Task task) {

if (taskMap.containsKey(task.getTaskID())) {

throw new IllegalArgumentException("Already added task");

}

taskMap.put(task.getTaskID(), task);

}”. In defending the overall quality of my Junit testing for this feature, my tests cover all the constraints and creation of the task objects, addition, updating, and deleting task IDs. This is evident because after testing for the coverage percentage it showed 100%.

My third feature was to create a contact class that has contact objects that have a unique contact ID string that cannot be longer than 10 characters, not null and not updateable, and a phone number, first and last name string field that cannot be 10 characters, not null, and an address field that is no longer than 30 characters and not null. The second part of the feature includes a contact service class with the ability to update address, number, first and last names, and delete, and add contacts with unique IDs. Again, my approach aligns with the software requirement evident because the Constructor in the Contact validates inputs ensuring alignment with constraints associated with the lengths of the address, phone number, and first and last name, “public Task(String taskID, String taskName, String taskDescription) {

validateTaskID(taskID);

validateTaskName(taskName);

validateTaskDescription(taskDescription);

this.taskID = taskID;

this.taskName = taskName;

this.taskDescription = taskDescription;

}”. The overall quality of my Junit tests is wide-ranging and efficient in covering numerous scenarios, including duplicates, and valid and invalid inputs. The coverage percentage for this feature was recorded as 100%.

Furthermore, I ensured that my code was technically sound by avoiding code duplicating and confirming that my test covered every input validation, duplication, and constraint. For instance, this code of line “

@Test

public void testInvalidAppointmentId() {

Date futureDate = new Date(System.currentTimeMillis() + 86400000);

assertThrows(IllegalArgumentException.class, () -> {

new Appointment(null, futureDate, "Description");

});

assertThrows(IllegalArgumentException.class, () -> {

new Appointment("12345678901", futureDate, "Description");

});

}” tests for constructor parameters in the Appointment class to safeguard the handling of invalid IDs, dates, and descriptions. Again, avoiding code duplication, making use of the OOP principles, and a data structure to contain the unique IDs for all the features. For instance, this line of code “private final Map<String, Task> taskMap = new HashMap<>();” ensures the usage of HashMap for task storage and confirms average O(1) complexity for deleting, adding, and update tasks.

The software testing techniques that I employed in this project include boundary value analysis, equivalence portioning, statement testing and coverage, and decision testing and coverage. For instance, testing the validity of an Appointment ID to ensure that it adheres to constraints or requirements because of the line of code below

“@Test

public void testInvalidAppointmentId() {

Date futureDate = new Date(System.currentTimeMillis() + 86400000);

assertThrows(IllegalArgumentException.class, () -> {

new Appointment("12345678901", futureDate, "Description"); // 11 characters

});

}”. The software techniques that I did not use for this project include security testing, dynamic testing, and usability testing. My testing techniques did not include identifying vulnerabilities and ensuring that data is safeguarded from external threats as well as testing to find defects by executing the software.

The practical uses and implications of boundary value analysis and equivalence partitioning testing techniques, for instance, guarantee that inputs at the edge of acceptable ranges are handled correctly, dividing up tests into multiple sections with similar values and then performing a test on each section. They are critically used for systems that rely heavily on user input an example is Project One. The next testing technique is statement testing and coverage, this technique ensures that all the statements of the source code are executed at least once. I used this technique to ensure that the constructor correctly initializes an Appointment object with valid inputs and throws exceptions for invalid inputs in the AppointmentTest class.

The mindset I adopted while working on this project includes thoroughness, avoiding code error, and code duplication, and enforcing code high code readability. Appreciating the complexity and interrelationships of the code aids in recognizing possible parts of the code where errors or bugs could occur due to communications between components. For instance, testing the TaskService methods and ensuring that tasks can be managed correctly and that the service performs as projected when interacting with the Task objects.

One way I tried to limit bias in my review of the code was by basically using Junit or automated testing. On the software developer side, I imagine that bias would be a concern if I were responsible for testing my code because I could overlook certain test cases and also low coverage percentages. For instance, in the initial stages of developing the Contact Service feature, I overlooked a coverage percentage of 67.5% which was not ideal.

Finally, in evaluating the significance of being disciplined in my commitment to quality as a software engineering professional believe it promotes customer satisfaction, code reliability and robustness, cost efficiency, and validates professional integrity. Not cutting corners when writing code ensures good code readability as well. I plan to avoid technical debt as a practitioner in the field by implementing continuous integration and continuous deployment as well as using other industry best practices such as code reviewing and pair programming.

Work Cited

*What is Unit Testing? - Unit Testing Explained - AWS*. (n.d.). Amazon Web Services, Inc. https://aws.amazon.com/what-is/unit-testing/