Ben Douglas

10-17-2024

7-2 Project Two Submission- Assignment

1. Summary

a. i. The extent that my testing approach was aligned to the software requirements is that my JUnit ContactServiceTest.java tested for different tasks in my Contact.java code by 1, John, Doe, 1234567890, and 123 Main St. The extent that my testing approach was aligned to the software requirements is that my ContactServiceTest.java tested for contactService, and updateContact 2 with Jane in my ContactService.java. The extent that my testing approach was aligned to the software requirements is that my JUnit TestServiceTest.java tested for different tasks in my Task.java code by adding task1, Another Task, and This is another task. The extent that my testing approach was aligned to the software requirements is that my TestServiceTest.java tested for updated tasks and no tasks in my TaskService.java by Updated Task, task1, Updated Description, nonexistent, Some Name, and Task not found.

The extent that my testing approach was aligned to the software requirements is that my JUnit AppointmentTest.java tested for different tasks in my Appointment.java code by appointmentID A1, A0123456789, Appointment date cannot be null or in the past, and description of Valid description, and Description cannot be null and must not exceed 50 characters. The extent that my testing approach was aligned to the software requirements is that my AppointmentServiceTest.java tested for different tasks in my AppointmentService.java by appointmentService, Appointment with this ID already exists, and Appointment with this ID does not exist.

ii. The overall quality of my JUnit tests for the contact service, the task service, the appointment, and the appointment service are good, because everything passed. My JUnit test was effective on the basis of coverage percentage, because the java project named ModuleFourMilestone had 89.7% coverage, the source folder named src had 89.7% coverage, the task package named task had 87.6% coverage, the code file named Task.java had 79.2% coverage, the code file named TaskService.java had 94.4% coverage, the JUnit package named test had 92.1% coverage, and JUnit code named TaskServiceTest.java had 92.1% coverage. My JUnit test was effective on the basis of coverage percentage, because the project named ModuleFiveMilestone had two JUnit tests, which are AppointmentTest.java, and AppointmentServiceTest.java. The AppointmentTest.java had a coverage of 82.8%. The AppointmentServiceTest.java had a coverage of 95.1%.

b. i. I ensured that my code was technically sound by using techniques of adding task1, and updating the tasks by coding, “public void testUpdateTaskName() {

Task task = new Task("task1", "Test Task", "This is a test task.");

taskService.addTask(task);

taskService.updateTaskName("task1", "Updated Task");

assertEquals("Updated Task", taskService.getTask("task1").getName());

}

” In my TestServiceTest.java.

I ensured that my code was efficient by using efficient testing of task1, and updating the description by coding, “public void testUpdateTaskDescription() {

Task task = new Task("task1", "Test Task", "This is a test task.");

taskService.addTask(task);

taskService.updateTaskDescription("task1", "Updated Description");

assertEquals("Updated Description", taskService.getTask("task1").getDescription());

}

” In my TestServiceTest.java.

My AppointmentTest.java is efficiently coded with “**void** testAppointmentDateInPast() {

// Arrange

String appointmentId = "A1";

Date appointmentDate = **new** Date(System.*currentTimeMillis*() - 100000); // Past date

String description = "Valid description";

// Act & Assert

IllegalArgumentException thrown = *assertThrows*(IllegalArgumentException.**class**, () -> {

**new** Appointment(appointmentId, appointmentDate, description);

});

*assertEquals*("Appointment date cannot be null or in the past.", thrown.getMessage());

}”

My AppointmentServiceTest.java is technically sound with

**“void** testAddDuplicateAppointment() {

// Arrange

Appointment appointment1 = **new** Appointment("A1", **new** Date(System.*currentTimeMillis*() + 100000), "Meeting");

appointmentService.addAppointment(appointment1);

Appointment appointment2 = **new** Appointment("A1", **new** Date(System.*currentTimeMillis*() + 200000), "Another Meeting");

// Act & Assert

IllegalArgumentException thrown = *assertThrows*(IllegalArgumentException.**class**, () -> {

appointmentService.addAppointment(appointment2);

});

*assertEquals*("Appointment with this ID already exists.", thrown.getMessage());

}”

2. Reflection

a. i. The software techniques that I employed for 3-2 Milestone: Contact Service- Assignment are: that I coded the contact ID that is non-null with up to ten characters long, the first name that is non-null with up to ten characters long, the last name that is non-null with up to ten characters long, the phone number that is ten digits long, the address that is non-null with up to thirty characters long, the contacts, and I tested the numbers and names.

The software techniques that I employed for 4-1 Milestone: Task Service- Assignment are: that I coded the task ID doesn’t have no more than ten characters, the task ID isn’t null, the task ID isn’t updateable, the name string doesn’t go past twenty characters, the description string doesn’t go past fifty characters, the task service has tasks added with unique ID’s, the task service has tasks deleted by tasks ID’s, the name and description has updateable task ID’s, and I tested the unique ID’s, I tested the deletion by task ID’s, I tested the name, and I tested the description.

The software techniques that I employed for 5-1 Milestone: Appointment Service- Assignment are: that I coded the appointment ID that is unique without going past ten characters, the appointment date isn’t null nor in the past, the description isn’t null and doesn’t go past fifty characters, and the appointments that do exist and don’t exist, and I tested the meeting, I tested the appointment that does exist, I tested the appointment that doesn’t exist, the appointment type, the doctor’s appointment, I tested whether the appointment ID was valid or not, I tested whether the appointment date wasn’t null nor in the past, I tested whether the appointment description was too long or not, and I tested whether the appointment was unique with up to ten characters.

ii. The other software techniques that I didn’t use for the milestones are Integration testing, Functional testing, Regression testing, and Performance testing. The characteristics of Integration testing are that it detects interface problems, and it improves code quality. The characteristics of Functional testing are that it has the perspective of the end-user, and it manages test data. The characteristics of Regression testing are that it detects errors, and it ensures stability. The characteristics of Performance testing are that it analyzes the results, and it can be recovered.

iii. The practical uses and implications for different software development projects and situations of the Integration testing are tests how different software modules interact with the system. The practical uses and implications for different software development projects and situations of Functional testing are tests that make sure the application’s basic functionalities meet the specifics. The practical uses and implications for different software development projects and situations of the Regression testing verifies changes to the codebase don’t affect the application’s functionality. The practical uses and implications for different software development projects and situations of the Performance testing are tests for software’s speed, stability, response time, reliability, scalability, and resource usage.

b.i. The extent that I employed caution was all throughout the project, because I wanted to get the coding right, and I wanted to get the JUnit tests right to make sure that the contacts, tasks, and appointments work for the customers. It was important to appreciate the complexity and interrelationships of the code that I was testing, because I wanted to be able to debug the code better, I wanted to maintain the code longer, and I wanted to identify bottlenecks and optimize my code. An example is “**public** Appointment(String appointmentId, Date appointmentDate, String description) {

**if** (appointmentId == **null** || appointmentId.length() > 10) {

**throw** **new** IllegalArgumentException("Appointment ID must be unique and cannot exceed 10 characters.");

}

**if** (appointmentDate == **null** || appointmentDate.before(**new** Date())) {

**throw** **new** IllegalArgumentException("Appointment date cannot be null or in the past.");

}

**if** (description == **null** || description.length() > 50) {

**throw** **new** IllegalArgumentException("Description cannot be null and must not exceed 50 characters.");

}”

ii. The ways that I tried to limit bias in my review of my code are that I tried to use third person coding, and I tried to code what the consumer wants. I couldn’t imagine that bias would be a concern if I were responsible for testing my own code, because I would try to use third person coding, and I would try to code what the consumers wants. An example is

**“public** **void** deleteContact(String contactId) {

**if** (!contacts.containsKey(contactId)) {

**throw** **new** IllegalArgumentException("Contact ID not found.");

}

contacts.remove(contactId);

}”

iii. It’s important not to cut corners when it comes to writing or testing my own code, because my code would be less likely to have bugs in it, my code would be more secure, and the consumer would trust my code more. I plan to avoid technical debt as a practitioner in the field by not cutting corners when it comes to writing or testing my own code, I would prioritize code quality, and I would document my decisions. An example is

“**public** **void** setPhone(String phone) {

**if** (phone == **null** || !phone.matches("\\d{10}")) {

**throw** **new** IllegalArgumentException("Phone must be a 10-digit number.");

}

**this**.phone = phone;

}”

Another example is

**“public** **void** testAddDuplicateContact() {

Contact contact1 = **new** Contact("1", "John", "Doe", "1234567890", "123 Main St");

contactService.addContact(contact1);

Contact contact2 = **new** Contact("1", "Jane", "Doe", "0987654321", "456 Elm St");

*assertThrows*(IllegalArgumentException.**class**, () -> contactService.addContact(contact2));

}."