**Jess Dowd**

**Tanisha Jacks**

**CS 320**

**22 February 2025**

**CS 320 Project Two**

**Summary**

**To what extent was your testing approach aligned to the software requirements? Support your claims with specific evidence.**

My testing approach aligned with the software requirements because I made sure to test all the features that were required in the assignment. For the contact service the requirements said each contact had to have a unique ID and fields like name and phone number had specific rules. I checked for those in my test by making sure exceptions were thrown when any invalid data was used. I did the same thing with the task service. I tested things like creating and updating and deleting tasks to make sure everything worked like it was supposed to. For example in the contact service I tested that creating a contact with a null ID would throw an exception, this confirmed that IDs couldn’t be null.

@Test

public void testInvalidContactCreation() {

// Trying to create a contact with a null ID (should throw an exception)

assertThrows(IllegalArgumentException.class, () -> {

new Contact(null, "Jess", "Dowd", "1234567890", "123 Abc Street");

});

}

**Defend the overall quality of your JUnit tests for the contact service and task service. In other words, how do you know that your JUnit tests were effective on the basis of coverage percentage?**

The quality of my JUnit tests was good because I tested all the important cases and edge cases. The tests I did made sure everything worked correctly on valid and invalid inputs. In the task service I tested things like creating a task with the longest allowed description and making sure an empty string was not accepted. I used assertNotNull(taskService.getTask(taskId)) to make sure that tasks were stored and assertEquals("New Task", task.getName()) to make sure values are set correctly. Here’s an example from my task service tests. This verified that a task could be added and then retrieved correctly.

@Test

public void testAddTask() {

Task task = new Task("12345", "Task Name", "Task Description");

taskService.addTask(task);

assertEquals(task, taskService.getTask("12345"));

}

**How did you ensure that your code was technically sound? Cite specific lines of code from your tests to illustrate.**

To make sure my code technically sound I did my tests so each one was focused on one function. For example in the contact service I used @Test annotations to separate the different cases like this

@Test

public void testContactCreation() {

// Created a contact and check if all values are stored correctly

Contact contact = new Contact("1234567890", "Jess", "Dowd", "1234567890", "123 Abc Street");

assertEquals("1234567890", contact.getContactId());

assertEquals("Jess", contact.getFirstName());

assertEquals("Dowd", contact.getLastName());

assertEquals("1234567890", contact.getPhone());

assertEquals("123 Abc Street", contact.getAddress());

}

**How did you ensure that your code was efficient? Cite specific lines of code from your tests to illustrate.**

I also made sure my tests were efficient by making sure to cut out as much of the repetitive code as possible. I used a @BeforeEach method to handle the setup instead of creating new objects in every test like this

@BeforeEach

public void setUp() {

taskService = new TaskService();

}

**Reflection**

**Testing Techniques**

**What were the software testing techniques that you employed for each of the milestones? Describe their characteristics using specific details.**

For each of the milestones I used unit testing with JUnit to check if the components worked the way they were supposed to. Unit testing focuses on testing small isolated pieces of code like methods or classes to make sure they function correctly on their own. For the contact service I tested that contacts were created with valid constraints, for example making sure the ID wasn’t too long and names weren’t null. For the task service I checked that tasks were added, updated, and deleted and were following rules for ID uniqueness and description limits. For the appointment service I tested that appointments couldn’t be created with past dates, that the ID was unique and that descriptions were the required length. In each case I wrote tests that confirmed that valid inputs worked and that invalid inputs threw exceptions.

**What are the other software testing techniques that you did not use for the milestones? Describe their characteristics using specific details.**

There are several software testing techniques that I didn’t use in these milestones. Integration testing is testing how different parts of an application work together. For example, like checking if a database interacts correctly with the backend. Since the milestones only were standalone services with no database or external dependencies I didn’t need to do integration testing. System testing checks if an entire application meets all the requirements including UI, backend and external services. Since these milestones weren’t a full application then system testing wasn’t needed. Regression testing makes sure that new changes don’t break the existing functions. It is useful in projects that are updated a lot. Since these milestones were built as separate milestones regression testing wasn’t needed so far either.

**For each of the techniques you discussed, explain the practical uses and implications for different software development projects and situations.**

Each of these testing techniques is useful in different situations. Unit testing is good for catching small bugs right away and making sure each part of the code works before even integrating everything together. Integration testing is important when the different services need to communicate and work together, for example like making sure a payment system processes transactions correctly with a bank API. System testing is used before launching an application to make sure everything works together correctly like UI interactions and backend logic. Regression testing is helpful in larger projects where there are regular updates because it makes sure older features still work when the new ones are added.

**Mindset**

**Assess the mindset that you adopted working on this project. In acting as a software tester, to what extent did you employ caution? Why was it important to appreciate the complexity and interrelationships of the code you were testing? Provide specific examples to illustrate your claims.**

While working on this project I had a detail-oriented mindset because small issues could lead to problems in the system functionality. Instead of assuming that the code would work correctly I made sure to test expected and unexpected scenarios.

For the appointment service I focused on testing date validation and uniqueness of appointment IDs. I made sure that appointments followed the constraints and stopped ay incorrect data from being accepted.

One test I checked for duplicate appointment IDs and making sure that the system rejected adding a second appointment with the same ID:

@Test

public void testAddDuplicateAppointment() {

AppointmentService service = new AppointmentService();

Calendar futureDate = Calendar.getInstance();

futureDate.add(Calendar.DATE, 2);

Appointment appt1 = new Appointment("123", futureDate.getTime(), "First appointment");

Appointment appt2 = new Appointment("123", futureDate.getTime(), "Second appointment");

service.addAppointment(appt1);

// Adding another appointment with the same ID should fail

assertThrows(IllegalArgumentException.class, () -> service.addAppointment(appt2));

}

This test made sure that the appointment service had a unique ID and did not allow duplicate entries.

**Assess the ways you tried to limit bias in your review of the code. On the software developer side, can you imagine that bias would be a concern if you were responsible for testing your own code? Provide specific examples to illustrate your claims.**

When reviewing my code I tried to **limit my bias** by testing scenarios that I might assume would never fail. It’s easy to miss failure points and overlook things. To try and limit my bias I made sure to test both **valid and invalid inputs** and making sure the appointment service handled errors the correct way.

For example instead of just testing if an appointment was successfully added I also wrote a test to check that **attempting to add an appointment with a duplicate ID would fail**:

@Test

public void testAddDuplicateAppointment() {

AppointmentService service = new AppointmentService();

Calendar futureDate = Calendar.getInstance();

futureDate.add(Calendar.DATE, 2);

Appointment appt1 = new Appointment("123", futureDate.getTime(), "First appointment");

Appointment appt2 = new Appointment("123", futureDate.getTime(), "Second appointment");

service.addAppointment(appt1);

// Adding another appointment with the same ID should fail

assertThrows(IllegalArgumentException.class, () -> service.addAppointment(appt2));

}

This helped make sure that **I didn’t just assume the system worked** and instead I tested how it failed.

When testing I also made sure to **use structured assertions (**assertEquals**,** assertThrows**,** assertNull**) instead of print statements**. This helped stop **confirmation bias** where I might unconsciously accept an incorrect result just because it looked right to me.

**Finally, evaluate the importance of being disciplined in your commitment to quality as a software engineering professional. Why is it important not to cut corners when it comes to writing or testing code? How do you plan to avoid technical debt as a practitioner in the field? Provide specific examples to illustrate your claims.**

Being disciplined in maintaining code quality is important as a software engineering professional because having low quality can cause problems like unstable software, hidden bugs and even raise the maintenance costs. Writing clean and well-tested code helps make sure that that the software is maintainable and reliable.

One risk of cutting corners is technical debt which happens when a project doesn’t do proper testing and best practices so they can have a short-term speed boost. Skipping tests or rushing development saves time at first it leads to more debugging and more rework and higher costs later on.

For example, in my appointment service I could have only tested the basic functionality of adding and deleting appointments and not tested any edge cases like duplicate IDs, past dates or missing entries. But I wrote tests that caught these thigs early so I stopped issues that could have caused data corruption or system crashes in a real-world application.

One of my tests made sure that appointments couldn’t be created with duplicate IDs:

@Test

public void testAddDuplicateAppointment() {

AppointmentService service = new AppointmentService();

Calendar futureDate = Calendar.getInstance();

futureDate.add(Calendar.DATE, 2);

Appointment appt1 = new Appointment("123", futureDate.getTime(), "First appointment");

Appointment appt2 = new Appointment("123", futureDate.getTime(), "Second appointment");

service.addAppointment(appt1);

// Adding another appointment with the same ID should fail

assertThrows(IllegalArgumentException.class, () -> service.addAppointment(appt2));

}

If I had ignored this test and a duplicate ID was later introduced then it could have led to conflicting appointment data and make it hard to track or manage schedules.

To avoid technical debt I plan to try and follow best practices like writing unit tests for every new feature so it is correct from the start and I will also use structured test cases instead of manual testing or assumptions. I will also use automated testing in future projects so that test coverage will be high.

Reference:

Collins, T. (2024, August 22). *Different types of testing in software | BrowserStack*. BrowserStack. <https://www.browserstack.com/guide/types-of-testing>

GeeksforGeeks. (2024, September 26). *Software testing techniques*. GeeksforGeeks. <https://www.geeksforgeeks.org/software-testing-techniques/>

*JUNiT - Test Framework*. (n.d.). <https://www.tutorialspoint.com/junit/junit_test_framework.htm>

Hamilton, D. (2024, July 21). *JUNit tutorial with examples: Setting up, writing, and running Java unit tests*. Parasoft. <https://www.parasoft.com/blog/junit-tutorial-setting-up-writing-and-running-java-unit-tests/>