Mikayla-Joy Botha

CS-320-11465-M01 Software Test, Automation QA

7-2 Project Two Submission

**Summary**

Throughout the development of the mobile application for Grand Strand Systems, I took a methodical approach to unit testing for the ContactService, TaskService, and AppointmentService components using JUnit 5. Each service underwent rigorous testing to ensure it met the specified requirements and performed reliably under various conditions, establishing a solid foundation for the application.

In the context of ContactService, I implemented unit tests to validate the core CRUD operations. For example, I created the `testAddContact` to confirm that a contact could be successfully added and retrieved from the system. Similarly, the `testDeleteContact` ensured that contacts could be removed as expected. Additionally, I included the `testCreateContactWithInvalidId` to verify that the system handled invalid contact IDs appropriately. These tests directly aligned with the requirement to manage a unique set of contacts and execute CRUD operations effectively.

Within TaskService, I focused on CRUD operations with particular emphasis on updates. Tests such as `testAddTask` confirmed the correct addition of tasks, while `testDeleteTask` ensured that tasks could be removed from the service. Crucially, the `testUpdateTask` was essential for validating that task updates were handled correctly. Moreover, I utilized `testCreateTaskWithLongDescription` to address input constraints, ensuring that descriptions

beyond allowed lengths were appropriately managed. This comprehensive testing approach ensured the effective management of task information and alignment with the required functionalities.

Moving on to AppointmentService, I designed unit tests to create, delete, and handle duplicate appointments. The `testAddAppointment` checked if appointments were added correctly, while the `testDeleteAppointment` ensured they could be removed. In addition, the `testAddDuplicateAppointment` verified that the system handled duplicate appointment IDs correctly, maintaining the uniqueness requirement. This approach guaranteed that appointment operations were effectively managed.

The quality of my JUnit tests is evident from their comprehensive coverage and ability to capture typical scenarios and edge cases. Tests such as `testCreateContactWithInvalidId` and `testCreateAppointmentWithLongDescription` provided effective boundary and error condition testing. My thorough validation of normal operations and exception handling reflected the effectiveness of these tests in ensuring adherence to requirements and identifying potential issues, providing a robust safety net for the software.

**Validating CRUD Operations for ContactService**

* **Adding a Contact**

**@Test**

**void testAddContact() {**

**Contact contact = new Contact("1", "John", "Doe", "1234567890", "123 Elm Street");**

**contactService.addContact(contact);**

**assertEquals(contact, contactService.getContact("1"));**

**}**

* **Deleting a Contact**

**@Test**

**void testDeleteContact() {**

**Contact contact = new Contact("1", "John", "Doe", "1234567890", "123 Elm Street");**

**contactService.addContact(contact);**

**contactService.deleteContact("1");**

**assertNull(contactService.getContact("1"));**

**}**

* **Updating a Contact**

**@Test**

**void testUpdateContact() {**

**Contact contact = new Contact("1", "John", "Doe", "1234567890", "123 Elm Street");**

**contactService.addContact(contact);**

**contactService.updateContact("1", "Jane", null, null, "456 Oak Street");**

**Contact updatedContact = contactService.getContact("1");**

**assertEquals("Jane", updatedContact.getFirstName());**

**assertEquals("456 Oak Street", updatedContact.getAddress());**

**}**

* **Handling Duplicate Contact ID**

**@Test**

**void testAddDuplicateContact() {**

**Contact contact = new Contact("1", "John", "Doe", "1234567890", "123 Elm Street");**

**contactService.addContact(contact);**

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

**contactService.addContact(new Contact("1", "Jane", "Doe", "0987654321", "456 Oak Street"));**

**});**

**assertEquals("Contact is null or ID already exists.", exception.getMessage());**

**}**

* **Handling Invalid Contact ID**

**@Test**

**void testCreateContactWithInvalidId() {**

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

**new Contact(null, "John", "Doe", "1234567890", "123 Elm Street");**

**});**

**assertEquals("ID cannot be null and must be 10 characters or less.", exception.getMessage());**

**}**

**Validating CRUD Operations for TaskService**

* **Adding a Task**

**@Test**

**void testAddTask() {**

**Task task = new Task("1", "Task1", "This is a description.");**

**taskService.addTask(task);**

**assertEquals(task, taskService.getTask("1"));**

**}**

* **Deleting a Task**

**@Test**

**void testDeleteTask() {**

**Task task = new Task("1", "Task1", "This is a description.");**

**taskService.addTask(task);**

**taskService.deleteTask("1");**

**assertNull(taskService.getTask("1"));**

**}**

* **Updating a Task**

**@Test**

**void testUpdateTask() {**

**Task task = new Task("1", "Task1", "This is a description.");**

**taskService.addTask(task);**

**taskService.updateTask("1", "Updated Task", "Updated description.");**

**Task updatedTask = taskService.getTask("1");**

**assertEquals("Updated Task", updatedTask.getName());**

**assertEquals("Updated description.", updatedTask.getDescription());**

**}**

* **Handling Task Description Length**

**@Test**

**void testCreateTaskWithLongDescription() {**

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

**new Task("1", "Task1", "This description is definitely longer than fifty characters, which should cause an error.");**

**});**

**assertEquals("Description cannot be null and must be 50 characters or less.", exception.getMessage());**

**}**

**Validating CRUD Operations for TaskService**

* **Adding an Appointment**

**@Test**

**void testAddAppointment() {**

**Date futureDate = new Date(System.currentTimeMillis() + 100000);**

**Appointment appointment = new Appointment("1", futureDate, "Meeting with team.");**

**appointmentService.addAppointment(appointment);**

**assertEquals(appointment, appointmentService.getAppointment("1"));**

**}**

* **Deleting an Appointment**

**@Test**

**void testDeleteAppointment() {**

**Date futureDate = new Date(System.currentTimeMillis() + 100000);**

**Appointment appointment = new Appointment("1", futureDate, "Meeting with team.");**

**appointmentService.addAppointment(appointment);**

**appointmentService.deleteAppointment("1");**

**assertNull(appointmentService.getAppointment("1"));**

**}**

* **Handling Duplicate Appointment ID**

**@Test**

**void testAddDuplicateAppointment() {**

**Date futureDate = new Date(System.currentTimeMillis() + 100000);**

**Appointment appointment = new Appointment("1", futureDate, "Meeting with team.");**

**appointmentService.addAppointment(appointment);**

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

**appointmentService.addAppointment(new Appointment("1", futureDate, "Another description."));**

**});**

**assertEquals("Appointment is null or ID already exists.", exception.getMessage());**

**}**

* **Handling Past Appointment Date**

**@Test**

**void testCreateAppointmentWithPastDate() {**

**Date pastDate = new Date(System.currentTimeMillis() - 100000);**

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

**new Appointment("1", pastDate, "Meeting with team.");**

**});**

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

**}**

**Validating Contact Creation**

* **Creating a Valid Contact**

**@Test**

**void testCreateValidContact() {**

**Contact contact = new Contact("1", "John", "Doe", "1234567890", "123 Elm Street");**

**assertEquals("1", contact.getId());**

**assertEquals("John", contact.getFirstName());**

**assertEquals("Doe", contact.getLastName());**

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

**assertEquals("123 Elm Street", contact.getAddress());**

**}**

* **Handling Invalid Phone Number**

**@Test**

**void testCreateContactWithInvalidPhone() {**

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

**new Contact("1", "John", "Doe", "12345678", "123 Elm Street");**

**});**

**assertEquals("Phone must be exactly 10 digits.", exception.getMessage());**

**}**

* **Handling long address**

**@Test**

**void testCreateContactWithLongAddress() {**

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

**new Contact("1", "John", "Doe", "1234567890", "This address is definitely longer than thirty characters.");**

**});**

**assertEquals("Address cannot be null and must be 30 characters or less.", exception.getMessage());**

**}**

**Reflection**

Throughout this project, I implemented various software testing techniques, each with its specific focus and approach. I primarily utilized Unit Testing to validate individual components or methods in isolation. This involved testing specific functionalities and exception handling, as evidenced by tests like `testAddContact` and `testCreateTaskWithLongDescription.`

Boundary Testing was also crucial, allowing me to validate edge cases such as maximum description lengths and invalid inputs, ensuring the application could handle constraints effectively. However, I acknowledge that several other testing techniques, such as Integration and System Testing, are essential for comprehensive testing.

While Unit Testing effectively isolates and debugs individual code units, Integration Testing is essential for verifying that combined components work seamlessly. On the other hand, System Testing is crucial for assessing the complete system against user requirements in real-world scenarios. My meticulous and detail-oriented mindset during the project focused on thoroughly validating each component's functionality.

Understanding the complexity and interrelationships of the Task, Contact, and Appointment objects was crucial in this process. For example, managing duplicate IDs requires careful test design to avoid service conflicts. To limit bias, I approached testing from potential end-user perspectives and scenarios beyond developer assumptions, aiding in identifying issues that might be overlooked from a developer's perspective.

Maintaining high-quality code and avoiding technical debt is crucial, and cutting corners can lead to unresolved issues and reduce software reliability. Therefore, my comprehensive testing of edge cases and exception handling reflected a commitment to delivering robust software and preventing future issues. Adhering to best practices such as thorough testing, code reviews, and continuous integration has ensured long-term code reliability and maintainability. In conclusion, the testing approach and techniques I employed for the ContactService, TaskService, and AppointmentService were designed to ensure that each service component functioned correctly, covering typical and edge cases and adhering to best practices, reflecting my unwavering commitment to delivering reliable and robust software.

**Citations**

1. Grant, J. (n.d.). *11 Tips for Unit Testing in Java. Code Intelligence.* https://www.code-intelligence.com/blog/11-tips-unit-testing-java
2. GeeksforGeeks. (2024, May 13). *Software testing boundary value analysis*. GeeksforGeeks. https://www.geeksforgeeks.org/software-testing-boundary-value-analysis/
3. Schmitt, J. (2021, December 3). *Unit testing vs integration testing*. CircleCI. https://circleci.com/blog/unit-testing-vs-integration-testing/#:~:text=While%20unit%20tests%20always%20take,works%20in%20an%20integrated%20way.