**Project Two: Summary and Reflections Report**

Alex Baires

Southern New Hampshire University

CS-320-11359-M01 Software Testing, Automation, and Quality Assurance

Professor Parul Hirpara

August 18th, 2024

**Summary**

**Testing Techniques**

When creating the JUnit testing suite for the Contact and Task services, I was inspired by the requirements-based testing methodology. This methodology involves using the defined software requirements in the document as criteria to test the program against in an effort to identify shortcomings (*Requirement Based Testing,* n.d.).

In both the Contact and Task services, we were given detailed requirements that can be broken down into smaller, more surgical tests. For example, the following task requirement was divided into several tests:

*“The task object shall have a unique Task ID string that cannot be longer than 10 characters. The task ID shall not be null and not updatable.”*

Uniqueness of the Task ID: The ID must be unique because having two different tasks with the same ID can destroy the integrity of each object; it would be impossible to differentiate between them and accomplish the necessary updates. My strategy was to use a 10-character hash function generated by the ‘MessageDigest’ class with SHA-256 to create a unique, unpredictable ID. The uniqueness of the ID was tested with the ‘testUniqueTaskID()’ JUnit test function inside TaskService.java. This test generated two different tasks and compared their generated IDs to ensure they were not equal.

Character Limit: To verify the Task ID length, I used the ‘testUniqueTaskId()’ inside ‘Task.java’ to confirm that the Task object would not accept an ID string longer than 10 characters. I could have improved the testing suite by verifying that the hash function I generated for each object was no more than 10 characters instead of hard coding an 11-character string.

Non-null and Non-updatable Task ID: The final part of the requirement involved verifying that the Task ID is not null and not updatable. For this, I used the ‘testUniqueTaskIdNull()’ and ‘testUpdateTaskID()’ functions. The ‘testUniqueTaskIdNull()’ function verifies that setting the ‘taskID’ to null throws a runtime exception. The setter function ‘setTaskID()’ does not update the task ID because it is not updatable. Therefore, my test involved initializing an object with an ID of 1 and then attempting to update it to a new ID of 2. This throws an unsupported operation exception, verifying that the ID cannot be updated. Breaking down each requirement into smaller pieces demonstrates the importance of using documented requirements as a test basis to write JUnit test cases.

According to Jeff Langr in the book *Pragmatic Unit Testing in Java 8 with JUnit* (2015), code coverage is the percentage of code that the JUnit tests execute. Langr (2015)v writes that a general rule of thumb is to aim for over 70% test coverage to adequately ensure proper verification while aiming for coverage as close to 100% as possible. After running the ‘TaskTest’ and ‘TaskTestService’ unit tests, ‘Task.java’ had 84.4% coverage, and ‘TaskService.java’ had 97.8% test coverage. For the contact service, ‘ContactService.java’ had 84.8% coverage (due to some extraneous placeholder functions that were not removed), and ‘Contact.java’ had 71.7% coverage. The lower coverage scores in ‘Contact.java’ are due to testing only the requirements and not the setters/getters. Additionally, I had a ‘toString()’ method that I was using during development that should not have been in the final product. This was all corrected in the task service, hence its higher coverage scores. Furthermore, some of the conditional statements should have been tested differently. For example, in the ‘verifyMailingAddress()’ function in ‘Contact.java,’ there is a conditional that was partially tested:  
 if(this.mailingAddress == null || this.mailingAddress.isEmpty())

My unit test only tested for the null condition and did not test for an empty string. This was corrected in the ‘TaskTest’ unit test. However, the overall quality of my testing is adequate because the functions and methods in both the contact and task services related to the requirements have the appropriate unit tests written for them, as shown by the green highlights for each method and line and the high coverage percentages. This demonstrates thorough testing, reliability, and quality of the JUnit tests for the contact and task services while ensuring their alignment with the software requirements and their role in maintaining strong software quality.

I utilized error-handling techniques, particularly conditional statements and try-catch blocks, to ensure technically sound code. This came in handy when verifying a phone number—the requirements called for the phone number to be a string. However, phone numbers typically utilize numbers rather than letters (and especially not special characters). Additionally, a phone number is ten digits, and someone incorrectly entering an incomplete phone number must be accounted for. Below is the verifyPhoneNumber() function I created to handle the above errors:

public void verifyPhoneNumber() {  
 if (this.phoneNumber == null || this.phoneNumber.isBlank()) {  
 throw new RuntimeException("Phone number is null or empty");  
 }  
  
 if (this.phoneNumber.length() != 10) {  
 throw new RuntimeException("Phone number length must be 10 digits");  
 }  
  
 try {  
 Double.*parseDouble*(this.phoneNumber);  
 } catch (NumberFormatException e) {  
 throw new RuntimeException("Phone number has invalid characters");  
 }  
}

Additionally, to ensure that the string only has numbers, I used the ‘parseDouble()’ function on the ‘phoneNumber’ object to parse the string and convert it to a double. If an invalid character is found, a RuntimeException is thrown showing an invalid input.

Again, I also utilized unit tests to verify that my code meets the client's stated requirements while behaving correctly.

For example, below is the unit test that ensures that the phone number: has exactly 10 characters, is not null, and only has digits while verifying that any errors are caught early and handled appropriately:

@Test  
@DisplayName("Phone Number String Field must have exactly 10 characters")  
public void testPhoneNumber() {  
 *contact* = new Contact("1",  
 "Bart", "Simpson", "8005557246", "742 Evergreen Terrace");  
  
 Assertions.*assertThrows*(RuntimeException.class, () -> { *contact*.setPhoneNumber("80055572461234");  
 });  
 Assertions.*assertThrows*(RuntimeException.class, () -> {  
 *contact*.setPhoneNumber("1");  
 });  
}  
  
@Test  
@DisplayName("Phone Number cannot be NULL")  
public void testPhoneNumberNull() {  
 Assertions.*assertThrows*(RuntimeException.class, () -> {  
 *contact* = new Contact("1",  
 "Bart", "Simpson", null, "742 Evergreen Terrace");  
 });  
}  
  
@Test  
@DisplayName("Phone Number can only have Numbers")  
public void testPhoneNumberChars() {  
 Assertions.*assertThrows*(RuntimeException.class, () -> {  
 *contact* = new Contact("1",  
 "Bart", "Simpson", "800555BOOT", "742 Evergreen Terrace");  
 });  
}

Finally, I used object-oriented programming principles and naming conventions in my ‘Task,’ ‘Contact,’ and ‘Appointment’ objects to avoid ambiguity; the function names describe the code's functionality without the need for excessive comments.

public class Contact {  
  
 final String contactID; // contactID not updatable  
 String firstName;  
 String lastName;  
 String phoneNumber;  
 String mailingAddress;  
  
 public Contact(String contactID, String firstName, String lastName, String phoneNumber, String mailingAddress) {  
 this.contactID = contactID;  
 this.firstName = firstName;  
 this.lastName = lastName;  
 this.phoneNumber = phoneNumber;  
 this.mailingAddress = mailingAddress;  
  
 verifyContactID();  
 verifyFirstName();  
 verifyLastName();  
 verifyPhoneNumber();  
 verifyMailingAddress();  
 }  
}

public class Task {  
  
 final String taskID;  
 String taskName;  
 String taskDescription;  
  
 public Task(String taskID, String taskName, String taskDescription) {  
 this.taskID = taskID;  
 this.taskName = taskName;  
 this.taskDescription = taskDescription;  
  
 verifyTaskID();  
 verifyTaskName();  
 verifyTaskDescription();  
 }

public class Appointment {  
 final String appointmentID;  
 Date appointmentDate = new Date();  
 String appointmentDescription;  
  
 public Appointment(String appointmentID, Date appointmentDate, String appointmentDescription) {  
 this.appointmentID = appointmentID;  
 this.appointmentDate = appointmentDate;  
 this.appointmentDescription = appointmentDescription;  
  
 verifyAppointmentID();  
 verifyAppointmentDate();  
 verifyAppointmentDescription();  
 }

The above code snippet shows the ‘Contact,’ ‘Task,’ and ‘Appointment’ objects with the attributes that comprise a contact, task, or appointment, the behaviors used to verify each piece of data, and a getter that retrieves their unique ID. Anyone looking at the code automatically knows what each function does because of the clear and concise nomenclature used throughout the project.

The primary strategy I used to make my code efficient is to store multiple, unique objects using a hash map of contacts or tasks. Hash maps have a speed of O(1), meaning no matter the number of tasks or objects stored inside the hash map, each element can be instantly retrieved using the key (Wengrow, 2020). In the Contact, Task, and Appointment services, the key is the uniqueID, which retrieves the values: the attributes comprising each object.

HashMap<String, Task> taskList = new HashMap<>();

Using a hashmap makes it easy to add multiple objects quickly, delete an object, or find that an object exists.

public void addTask(String taskName, String taskDescription) {  
 String taskID = getHash(taskName);  
 Task task = new Task(taskID, taskName, taskDescription);  
 checkExistingTask(task.getTaskID());  
 taskList.put(taskID, task);  
}

**Reflection**

Developing the ‘ContactService,’ ‘TaskService,’ and ‘ApplicationService’ files for the customer’s mobile application required white-box static and unit testing techniques while implementing requirements analysis and design to create a test plan. These are under the white-box testing umbrella because my process required testing code whose inner workings were familiar to me. Static testing involves analyzing requirements and specifications and reviewing code and documentation without running the program (Morgan & Thompson, 2019). I examined each software requirement during the initial requirements analysis stage to resolve any inconsistencies before entering the formal unit testing phase. I also ensured my ‘Contact,’ ‘Task,’ and ‘Appointment’ objects would meet the functional requirements by parsing through my code line-by-line in the form of a manual code review. Unit testing consisted of a battery of JUnit5 functions that tested the smallest components of the code. An example of an inconsistency that I needed to resolve during development is the first requirement of the ‘ContactService’ application:

“The contact object shall have a required unique Contact ID string that cannot be longer than 10 characters.”

The semantics of the requirement to be tested is essential. A required contact identifier means that one must ensure that a ‘Contact’ object cannot be generated without an ID. The function to test this would be ‘testUniqueContactIDNull()’ where the unit test ensures that having a null value for the ID results in an exception being thrown:

@Test  
@DisplayName("Unique Contact ID should not be NULL")  
public void testUniqueContactIDNull() {  
 Assertions.*assertThrows*(RuntimeException.class, () -> {  
 *contact* = new Contact(null,  
 "Ivan", "Baires", "8008675309", "123 Sesame Street");  
 });  
}

Additionally, the word ‘unique’ in the requirements must be considered when designing the test suite; no two ‘contact’ objects can be identical. I wrote about the hash function I created in a previous assignment. However, the uniqueness of the ID requires designing a test that verifies two objects that do not have the same identifier. This is done by generating two ‘contact’ objects and asserting that their identifiers are not the same in the below unit test:

@Test  
@DisplayName("Testing whether generated ID is Unique")  
public void testUniqueContactID() {  
 *contactService*.addContact("Aaron", "Judge", "7188675309",  
 "Yankee Stadium, NYC");  
 String firstUniqueIDTest = *contactService*.getContactList().keySet().iterator().next();  
 *contactService*.addContact("Luke", "Skywalker", "8009992345", "Imperial City, Coruscant");  
 String secondUniqueIDTest = *contactService*.getContactList().keySet().iterator().next();  
 *assertNotEquals*(firstUniqueIDTest, secondUniqueI

Statement coverage is a primary white-box testing technique used to test this project. It shows how much of the executable code base has been tested (Langr, 2015). Verifying that all conditional branch predicates have been appropriately tested significantly impacts the coverage percentage. Eclipse automatically has EclEmma installed, displaying code coverage and highlighting conditional statements in yellow if they need to be sufficiently tested. Green statements are thoroughly tested, while red statements do not have an accompanying unit test or are inadequately tested.

A close up of words

Description automatically generated

The code coverage percentage is calculated with the following formula:

Statement Coverage = (Total # of Statements / # of Executed Statements) x 100

A screenshot of a graph

Description automatically generated

Utilizing requirements analysis to inform test design allows developers to account for potential inconsistencies and pitfalls arising from requirements elicitation. Thoroughly understanding and resolving inconsistencies ensured that, during white-box testing, I could write practical unit tests that verified that each functional unit of my software met each of the client’s requirements (Garcia, 2017).

During the milestones, I did not intentionally use black-box testing because the codebase was only exercised through the unit testing suite; it was never directly executed and did not have a user interface for a tester to interact with directly. The vital aspect of black-box testing is that it seeks to separate the resulting behavior from the program's implementation. In contrast, during this project, the tester is tasked with verifying that the code implementation results in the expected behavior per the specifications (white-box testing). However, I can argue that I did use the black-box method of “boundary value analysis” when verifying that the program will not accept a string greater than ten characters for its unique identification when choosing an 11-digit string to test with. A classic black-box implementation would have a user interface that allows the tester to enter any digit number, such as a blank (minimum), 11 digits (edge case), or 100 digits (maximum), to see how the program would behave without seeing the internal structure of the code (Ahmad, 2024; Morgan & Thompson, 2019).

Another black-box testing method not implemented in this project is equivalence class partitioning, where inputs are divided into groups of equivalence classes. The objective is to group inputs that produce the same output, such as invalid and valid inputs. This consolidates the number of test cases into those checking validity and those checking for invalidity. Additionally, decision table-based testing was not utilized. This testing method lists every combination of inputs and outputs determined by requirements and business rules where various combinations of test inputs result in unique outputs. An example would be inputs being the conditions, such as an appointment ID being in the past, with the action “add appointment” being false because the specifications state that the user should not be able to add an appointment if they are attempting to use a past date (Morgan & Thompson, 2019).

|  |  |  |
| --- | --- | --- |
|  | Appt. Date in Past | Appt. Date in Future |
| Add Appointment | F | T |

*The above table helps verify that an appointment cannot be added if the date is in the past.*

The project did not utilize state transition testing, which verifies that the system behaves predictably when changing from one state to another. Typically, a slate of states is listed, and the tester tests inputs that change between them. For example, a traffic light would have a “green (go),” “yellow (caution/prepare to stop),” and “red (complete stop)” state to change between. States can be tested through cause-effect graphing, where input combinations are tested to see the resulting events, Stateflow which models the systems used to develop algorithms and logic; state transition diagrams, which model the program’s states and the actions that change them; and the state transition table, which is similar to a decision table, but instead shows the states, events, transitions, and their outputs. Finally, the project did not use experience-based testing techniques, which are methods of last resort where there are little to no specifications to draw tests from or little time to run a full test battery. Error guessing uses the tester’s experience developing similar programs to create test cases that seek to identify known vulnerabilities or find errors that are consistently known to the team in prior projects. Exploratory testing involves writing a test charter of objectives and quickly time-boxing concurrent test design, execution, and logging while using the objectives to focus only on priority issues. Checklist-based testing is derived from different sources and standards for functional and non-functional testing. They are used to inspire new forms of tests and may be reused from previous projects (Ahmad, 2024; Morgan & Thompson 2019).

To reiterate, static testing techniques are used in the system without executing a single line of code. The practical use of static testing is to apply root cause analysis to failures, identifying the sequence of events leading to the impact rather than rectifying the actual failure. Products that can be analyzed through static testing are specifications like business, functional, non-functional, security requirements, epics, user stories, acceptance criteria, architecture/design specifications, user guides, web pages, models, and the codebase itself. The implications of utilizing static testing are beneficial in that the earlier it is applied, the sooner defects are found. Furthermore, its benefits include increasing productivity, reducing the amount of dynamic testing conducted, reducing cost, identifying defects missed during dynamic testing, and finding and repairing defects more efficiently. Dynamic testing techniques, on the other hand, execute the code. The method I discussed and utilized in the project is component or unit testing. The practical usage of unit testing is to test each individual function and isolate it from the rest of the system while using mock objects, function stubs, and drivers, depending on the project’s completion stage. These tests verify that the code meets functional requirements (like the ability to add or remove a task), non-functional requirements (like checking for memory leaks), and structural testing, which goes hand-in-glove with white box testing and code completion rates. The implications of proper unit testing include risk reduction, functional verification, quality assurance, locating defects in the code, and preventing an error from creeping up further along in the development cycle. A real-life use case for unit testing is iterative test-driven development, where a developer writes their unit tests based on requirements that fail until they write code that passes each unit test (Ahmad, 2024; Morgan & Thompson 2019).

Black-box testing techniques seek to ensure that the system behaves according to the requirements. Practically, the team separates what the software does from how it accomplishes the objectives. The implication is that the team seeks to create test cases that model system behavior while testing it thoroughly and efficiently. Boundary value analysis is a form of black-box testing that is useful where specific input ranges are specified, like the project’s ‘appointment’ description field that requires 50 characters or less. In general, boundary value analysis can also be used to find faults with the number of loop iterations or the number of times before a loop finishes executing. The goal of this testing is to verify strict input constraints and to test whether it can adequately handle extreme input values. A practical use case of the equivalence class partitioning black-box testing technique that can logically group inputs and outputs by simplifying the number of test cases while maintaining adequate test coverage. An example would be having a program that takes any valid integer. These inputs can be partitioned as negative, zero, and positive inputs that can be tested. Invalid inputs like special characters can also be partitioned to verify that they can be handled appropriately. An implication of utilizing this method is the reduction in the number of test cases that need to be written, particularly for more extensive, complex software projects. A practical use of the decision table-based technique is to list business rules to define system functionality and list the logical conditions that both satisfy and dissatisfy the rule. This allows a tester to vet whether requirements and accompanying software functionality work hand-in-hand accurately. A project that benefits from such a testing table would be a logistics workflow program with numerous functions, such as inventory control, transportation management, and informational queries. The implication of this form of testing ensures that software behaves correctly for many different scenarios and moving pieces while negating the likelihood of unintended consequences. The final black-box testing method is state transition testing. Practically, this involves analyzing the changes to input conditions that result in a change in the program’s state and the transitions involved. This form of testing can be applied to more complex software systems. For example, an order kiosk that communicates with an inventory system. A client can press “start order” from the home page, which transitions to the menu. After selecting an item, the state changes to a “shopping cart,” where the customer double-checks their order and is transitioned into the “payment processing” state. State transition testing ensures that these operations are performed seamlessly, accurately, and securely (Morgan & Thompson, 2019).

Experience-based testing techniques are practical because they utilize programmer and tester experience to determine what to test. These tests are helpful in iterative software development, where requirements are not set in stone and constantly evolve. The tester uses their experience and intuition to find and quickly negate any issues. The implication of using experience-based techniques to test software is to use it as a tool among the other thorough testing techniques, as it can find defects that may have been missed (Morgan & Thompson, 2019).

**Mindset**

Exercising caution is essential to writing comprehensive unit tests. Langr (2015) writes about the ease with which users find bugs and errors when running a program and believes it is vital to predict common defects cautiously by asking the correct questions about what to test. Firstly, a software tester must ask whether the application displays the correct results (Langr, 2015). Langr (2015) refers to this as the “happy path,” where the program operates as it should, and users can accomplish their goals with the right results based on the requirements. The battery of unit tests for the mobile application has many such tests. An example is testing the behavior of the ‘AppointmentService’ class to verify that a user can add and retrieve appointments:

@Test  
@DisplayName("Should Add an Appointment and Retrieve its Individual Details")  
public void testAddAndRetrieveAppointment() {  
 *appointmentService*.addAppointment(currentDate, "Dental Exam");  
 *assertFalse*(*appointmentService*.getAppointmentList().isEmpty());  
 *assertEquals*(1, *appointmentService*.getAppointmentList().size());  
  
 String uniqueID = getAppointmentIDByDate(*appointmentService*, currentDate);  
 Appointment appointment = *appointmentService*.getAppointment(uniqueID);  
 Assertions.*assertNotNull*(appointment);  
 *assertEquals*(currentDate, appointment.getAppointmentDate());  
 *assertEquals*("Dental Exam", appointment.getAppointmentDescription());  
}

We assert that the hash map size storing the list of appointments is incremented when a user adds a new appointment, such as a dental exam scheduled for the current date and time. I also verified that the hash map of appointment objects is the correct size to ensure we add objects to the proper data structure and not the hash maps that hold contacts or tasks. Moreover, I validated whether users can retrieve their appointments through the unique ID generated for each object. Cautiously, I double-checked that the appointment description was for a “dental exam” because a collision or inadvertent generation of a duplicate unique ID can pull incorrect appointment data.

To test cautiously, one must ask whether the boundary conditions are correct (Langr, 2015). These edge-case tests involve finding defects generally not found when testing for positive outcomes. Langr notes that these consist of testing the program with sham inputs, poorly formatted data, and empty or missing parameters. I followed his methodology in the ‘ContactServiceTest JUnit files by creating a testEdgeCases() JUunit test that creates an input of 1000 characters, attempts to create an object, and tests special characters for each contact parameter.

@Test  
@DisplayName("Testing for edge cases")  
public void testEdgeCases() {  
 char[] data = new char[1000];  
 Arrays.*fill*(data, 'a');  
 String str = new String(data);  
 // Testing very long input  
 *assertThrows*(RuntimeException.class, () -> { *contactService*.addContact(str, str, "1234567890", str); });  
 *assertNotEquals*(1, *contactService*.getContactList().size());  
  
 // Testing special characters  
 *contactService*.addContact("!@#$%^&\*()", "!@#$%^&\*()", "1234567890", "!@#$%^&\*()");  
 *assertEquals*(1, *contactService*.getContactList().size());  
  
 // Testing special character for phone number  
 *assertThrows*(RuntimeException.class, () -> { *contactService*.addContact("test", "test", "!@#$%^&\*()", "test"); });  
  
}

Additionally, the ‘ContactTest’ JUnit file has a function that ensures all edge cases are accounted for when adding a phone number, verifying that special characters, letters, and null or empty phone numbers follow the correct error-handling procedure to carefully sanitize input and ensuring conformance to a ten-digit phone number.

@Test  
@DisplayName("Testing Phone Number Requirements - must have exactly 10 characters, not null, not empty")  
public void testPhoneNumberRequirements() {  
 *contact* = new Contact("1", "Bart", "Simpson", "8005557246", "742 Evergreen Terrace");  
  
 // EDGE Case: 11-digit phone number  
 Assertions.*assertThrows*(RuntimeException.class, () -> *contact*.setPhoneNumber("18005557246"));  
 // EDGE Case: 1-digit phone number  
 Assertions.*assertThrows*(RuntimeException.class, () -> *contact*.setPhoneNumber("1"));  
 // EDGE Case: 10 characters with letters instead of numbers  
 Assertions.*assertThrows*(RuntimeException.class, () -> *contact*.setPhoneNumber("800555BOOT"));  
 // Edge Case: 9 characters  
 Assertions.*assertThrows*(RuntimeException.class, () -> *contact*.setPhoneNumber("800555726"));  
 // Edge Case: Only special characters  
 Assertions.*assertThrows*(RuntimeException.class, () -> *contact*.setPhoneNumber("!@#$%^&\*()"));  
 // Test a NULL Phone Number field  
 Assertions.*assertThrows*(RuntimeException.class, () -> *contact* = new Contact("1",  
 "Bart", "Simpson", null, "742 Evergreen Terrace"));  
 // Test an empty Phone Number field  
 Assertions.*assertThrows*(RuntimeException.class, () -> *contact* = new Contact("1",  
 "Bart", "Simpson", "", "742 Evergreen Terrace"));  
}

All files are interrelated with the mobile application. In Java, public classes are kept in separate files to comply with object-oriented programming principles. Likewise, the corresponding tests are kept in separate files to prevent cross-contamination between objects and tests. For example. I would not have the testTaskDescriptionRequirements() test inside of the ‘AppointmentTest’ JUnit file because this would involve instantiating a ‘task’ object within the JUnit test, causing possible confusion and issues with test accuracy, adding additional complexity, dependencies, and interactions that are not a part of testing the appointment functionality.

In writing my unit tests, I limited bias by first only testing the requirements provided by the client. There is no room for bias when one’s code does not satisfy the objective of the entire program. Margheim (2020) states that to limit bias, one should write tests from the perspective of an outsider who does not know how the system works. This sounds similar to black-box testing, where a developer writes a test that only verifies the result without diving into the inner workings of the code. Margheim (2020) imagines a situation where a developer creates a search function that returns a specific image, and the developer creates a solution but only tests to ascertain that the test passes utilizing their solution, therefore only testing the implementation and not the actual result, preventing them from finding a more efficient or elegant solution. In my case, I should have done a better job of testing for collisions when using the SHA-256 algorithm to encrypt appointment dates and generate an ID in the ‘AppointmentService’ class. I was biased because I thought I had created a robust, creative solution but did not realize that multiple appointments on the same date would create collisions and cause errors or corrupt data from having the same unique ID. However, I limited my bias by testing for proper boundaries, specifically when designing the testAppointmentDateNotInPast() function. My test coverage was well over 80%, but it was imperative to prevent the user from being able to create an appointment for a past date by accident.

@Test  
@DisplayName("IMPORTANT: Testing that Appointment cannot be made for Past Date")  
public void testAppointmentDateNotInPast() {  
 Assertions.*assertThrows*(RuntimeException.class, () -> {  
 Calendar calendar = Calendar.*getInstance*();  
 calendar.set(2024,Calendar.*JULY*,30);  
 Date testDate = calendar.getTime();  
 *appointment* = new Appointment("1", testDate, " Consultation");  
 });  
}

An accurate software tester thinks of ways to “break” the code, even if it hurts the feelings of the software developer (especially if they are the same person).

Finally, discipline is vital when designing comprehensive software test suites. Langr (2015) believes in the “FIRST” mnemonic to remain disciplined when designing tests; they must be fast, isolated, repeatable, self-validating, and timely. Fast tests were not an issue in this exercise because all objects are stored locally and not in a database. The biggest impediment to fast tests is constantly querying a database with every unit test (Langr, 2015). This project teaches one to test each object and function locally to ensure they meet the requirements before storing them in a database. This establishes that the logic and design decisions are correct, preventing tests from bogging down the development system. The goal is to write tests that do not depend on slower code. Langr (2015) believes proper testing discipline involves testing bits and pieces of the code while minimizing the intermingling of tests and respective dependencies should something go wrong (2015). Tests should also be able to run in any order. For example, my ‘Contact’ and ‘TaskService’ test batteries do not need to be run in a specific sequence. Within these files, the test functions stand on their own. To illustrate, the below testDeleteTask() can be run separately because the methods involve adding a task within the test and then deleting it, asserting that the task list hash map is empty.

@Test  
@DisplayName("Should Delete a Task")  
public void testDeleteTask() {  
 taskService.addTask("Open e-mail", "Opens g-mail application");  
 String uniqueIDTest = taskService.getTaskList().keySet().iterator().next();  
 taskService.deleteTask(uniqueIDTest);  
 *assertTrue*(taskService.getTaskList().isEmpty());  
 *assertEquals*(0, taskService.getTaskList().size());  
}

Furthermore, functioning correctly does not require interaction with other unit tests. To test this, I commented out all of the other functions and reran the test. It successfully completed the test and passed. Additionally, all tests are repeatable. If I were to modify the logic in the ‘TaskService’ class while continuing to use the hash map data structure, the test would still be valid. Each test is self-validating in that it asserts a condition that must be met for it to pass.

Moreover, no manual setup is required. After modifying the class to be tested, one builds the test file and runs it automatically to deploy each unit test. My tests are timely because they were constructed throughout the project's development cycle, illustrating the agile principles of constant improvement and frequent testing.

**References**

Ahmad, N. (2024, May 16). *Software testing techniques: types and best practices | LambdaTest*. LambdaTest. https://www.lambdatest.com/blog/testing-techniques/

Bose, S. (2023, May 25). *Software Testing Techniques: Explained with Examples | BrowserStack*. BrowserStack. https://www.browserstack.com/guide/software-testing-techniques

Langr, J. (2015). *Pragmatic Unit Testing in Java 8 with JUnit*. The Pragmatic Programmers LLC.

Margheim, S. (2020, December 17). *How to combat bias in software Testing*. Test IO. Retrieved August 20, 2024, from https://test.io/resources/blog/how-to-combat-bias-in-software-testing

Morgan, P., & Thompson, G. (2019). *Software testing: An ISTQB-BCS Certified Tester Foundation Guide*. BCS, The Chartered Institute for IT.

*Requiremenst based testing.* (n.d.). https://www.tutorialspoint.com/software\_testing\_dictionary/requirements\_based\_testing.htm

Wengrow, J. (2020). *A Common-Sense Guide to Data Structures and Algorithms, second edition: Level Up Your Core Programming Skills*. Pragmatic Bookshelf.