**Ali ARSLAN**

**CS-320 7-2 Journal Project Two**

**Alignment with Software Requirements**

Our testing approach was closely aligned with the software requirements for the contact service and task service. We ensured that each component (Contact, ContactService, Task, TaskService) was thoroughly tested against the specified requirements. Here's how our tests aligned with key requirements:

1. **Unique Identifiers:** I tested that contact IDs and task IDs were unique and not more than 10 characters long.

***Evidence:*** *testInvalidContactIdSecond() in ContactTest.java*

***Evidence:*** *testTaskCreationIdSecond() in TaskTest.java*

1. **Field Constraints:** I verified that fields like first name, last name, phone, address for contacts, and name and description for tasks adhered to their specified length constraints.

***Evidence:*** *testInvalidNameSecond(), testInvalidLastNameSecond(), testInvalidPhoneSecond(), testInvalidAddressSecond() in ContactTest.java*

***Evidence:*** *testTaskCreationNameSecond(), testTaskCreationDescriptionSecond() in TaskTest.java*

1. **CRUD Operations:** I tested the ability to add, delete, update, and retrieve contacts and tasks.

***Evidence:*** *testAddSingleContact(), testDeleteContact(), testUpdateContact() in ContactServiceTest.java*

***Evidence:*** *testAddSingleContact(), testDeleteTask(), testUpdateTask() in TaskServiceTest.java*

1. **Input Validation:** I ensured that invalid inputs were rejected and appropriate exceptions were thrown.

***Evidence:*** *Various test methods in ContactTest.java and TaskTest.java checking for null, empty, or oversized inputs*

**Quality of JUnit Tests**

When creating test scenarios and writing test code, I designed them in a way that enhances test quality as follows:

* **Comprehensive Coverage:** All public methods and constructors have been tested to cover both valid and invalid scenarios.

***Evidence:*** *The testInvalidContactIdFirst() and testInvalidContactIdThird() methods in the ContactTest.java file.*

* **Edge Cases:** I included tests for edge cases such as null inputs, empty strings, and boundary values.
* **Isolation:** Each test method focuses on a specific behavior, facilitating the identification and resolution of issues.
* **Readability:** Test methods have clear and descriptive names that explicitly indicate what is being tested.

***Evidence:*** *The testAddMultipleContacts() and testUpdateNonExistentContact() methods in the ContactServiceTest.java file.*

* **Setup and Tear Down:** I used the @BeforeEach annotation to create fresh instances for each test, ensuring independence.

***Evidence:*** *The setUp() method in the ContactServiceTest.java and TaskServiceTest.java files.*

The positive impact of the aforementioned features on test quality will be confirmed through two additional metrics:

**Code Coverage:** This metric is not sufficient on its own; even if your tests cover all code, it does not indicate how high the quality of the tests is. However, it can still be a significant metric.

**Mutation Testing:** This involves making changes to the original code to evaluate how well your tests produce quality results. High rates indicate that the applied tests are of high quality.

A screenshot of a graph

Description automatically generatedA close-up of a report

Description automatically generated

In conclusion, as can be seen from the results below, tests conducted with 100% code coverage and 88% mutation coverage can be considered quite successful, and it can be inferred that the features mentioned above contribute to enhancing test quality.

**Technical Soundness**

Our code demonstrates technical soundness through:

1. **Encapsulation:** I've implemented strong encapsulation by using private fields with public getters and setters in both Contact and Task classes. This approach offers several benefits:

* **Data integrity:** It prevents direct, uncontrolled access to the object's state.
* **Flexibility:** It allows us to change the internal implementation without affecting the public interface.
* **Validation:** We can enforce data validation rules within setters.

***Evidence:***

private final String contactId;

private String firstName

public String getContactId() {

return contactId;

}

public void setFirstName(String firstName) {

if (firstName == null || firstName.isBlank()) {

throw new IllegalArgumentException("Invalid first name");

}

this.firstName = firstName;

}

This encapsulation in Contact.java enhances the code's technical quality by ensuring that the object's state remains consistent and valid at all times.

1. **Input Validation:** We've implemented thorough input validation in constructors and setters. This practice significantly improves the code's robustness by:

* **Preventing invalid states:** Objects are guaranteed to be in a valid state from creation.
* **Early error detection:** Issues are caught at the point of data entry, making debugging easier.
* **Improved reliability:** The system is less likely to encounter runtime errors due to invalid data.

**Evidence:**

public void setPhone(String phone) {

if (phone == null || phone.isBlank()) {

throw new IllegalArgumentException("Invalid phone number");

}

else if (phone.length() != 10) {

throw new IllegalArgumentException("Invalid phone number");

}

else {

this.phone = phone;

}

}

This validation in Contact.java ensures that phone numbers are always 10 digits long, enhancing data consistency and preventing potential issues in systems that might use this data.

1. **Immutable Fields:** We've used the final keyword for fields that shouldn't change after initialization, such as IDs. This practice improves code quality by:

* **Ensuring thread safety:** Immutable objects are inherently thread-safe.
* **Preventing accidental modifications:** It's impossible to accidentally change the ID after object creation.
* **Enhancing code clarity:** It clearly communicates which values are meant to be constant.

**Evidence:**

private final String taskId;

In Task.java, this immutability of the taskId field prevents any accidental changes to a task's identity, maintaining data integrity throughout the object's lifecycle.

1. **Consistent Error Handling:** I've used IllegalArgumentException consistently for invalid inputs. This approach:

* **Improves code predictability:** Developers know what to expect when invalid data is provided.
* **Enhances debugging:** It's clear when and why exceptions are thrown.
* **Facilitates proper error handling:** It allows for specific catch blocks in the calling code.

**Evidence:**

Throughout Contact.java and Task.java, we see consistent use of IllegalArgumentException:

if (contactId == null || contactId.isBlank()) {

throw new IllegalArgumentException("Invalid contact ID");

}

1. **Use of Interfaces:** By declaring the contacts map as Map<String, Contact> instead of HashMap<String, Contact>, we've programmed to an interface. This practice:

* **Increases flexibility:** We can easily switch to a different Map implementation if needed.
* **Improves abstraction:** The code focuses on what the map does, not how it does it.

**Evidence:**

private final Map<String, Contact> contacts = new HashMap<>();

By implementing these practices, we've significantly enhanced the technical quality of our code. The result is a system that is more robust, maintainable, and less prone to errors. These design choices make the code easier to understand, modify, and extend in the future, which are all hallmarks of high-quality software engineering.

**Code Efficiency**

My code demonstrates efficiency through:

1. **Use of HashMaps:** For O(1) average-case time complexity in add, delete, and retrieve operations.

* **Evidence:** private final Map<String, Contact> contacts = new HashMap<>(); in ContactService.java

1. **Minimal Object Creation:** Updating existing objects rather than creating new ones for updates.

* **Evidence:**

public boolean updateContact(String contactId, String firstName, String lastName, String phone, String address) {

Contact contact = contacts.get(contactId);

if (contact == null) {

return false;

}

}

1. **Early Returns:** Checking for invalid conditions early to avoid unnecessary processing.

* **Evidence**

if (contacts.containsKey(contact.getContactId())) {

return false;

}

1. **Avoiding Redundant Checks:** The service layer trusts the Contact-Task objects to maintain their own validity, avoiding duplicate checks.
2. **Immutable IDs:** Using final fields for IDs prevents unnecessary object mutations and potential errors.

In conclusion, our testing approach was well-aligned with the software requirements, producing high-quality JUnit tests that thoroughly validate the functionality of the contact and task services. The code demonstrates sound technical principles and efficiency in its design and implementation.

**Testing Techniques Used**

In all three milestones, the necessary testing procedures were provided within the rubric. According to the rubric, we only needed to perform unit testing within the codes. However, based on my experience, I can say that this was not sufficient during the development phase. Unit tests allowed us to check the quality of the code and whether it met the requirements, but I realized that manual review and testing were necessary for the planning phase of the code and how it would perform the operation. This allowed me to write the codes smoothly.

**1. Linting**

During the writing of the codes, the Java Code Analysis warnings in Eclipse automatically applied linting tests, which are a type of static test that checks for syntax errors. It warned me in case of mistakes and allowed me to make corrections simultaneously. According to Saladino (2024), linting is the name given to the automatic detection of style (syntax) and program errors (use of unknown methods, incorrect variable assignment, type incompatibility, etc.) in source codes, and programs that perform this operation are called linters. Almost every advanced IDE comes with a linter automatically installed, which warns the programmer about errors by displaying warning icons and coloring the codes in the code writing area.

**2. Manual Review and Testing**

At this stage, I performed multiple testing and control operations. First, I made changes to the code based on the warnings from the linter in the previous stage and by examining the codes. Then, I applied compilation tests and runtime tests to check if the application gave any errors during compilation or runtime. For this process, I created an additional class with a main method, imported the classes I wrote into this class, and applied some of the operations I also applied in unit testing as if they were in a real program.

* **Compilation Test**

At this stage, the program was compiled to check if there were any errors during compilation time. The program ran without any errors, ensuring that the code met the requirements during compilation.

* **Run Time Test**

For this process, the compiled code was run, and add, remove, and edit operations were performed with data entered from the console at runtime using codes in the main class created. Errors occurring at this stage were corrected by returning to the source codes, and I made sure that the program did not receive any errors at runtime.

**3. Unit Testing**

According to Kanai (2022), a unit is the smallest testable part of an application, and unit tests are used to understand whether the code behaves as expected. This ensures that the code will give the same result under all conditions. In our case, these units will be classes.

In these 3 projects, unit tests were applied for three classes that form the data structure and 3 operation classes. Characteristics:

* **Code Coverage Analysis:** Ensures not only the execution of code lines but also the testing of logical variations. It should be ensured that the test procedures cover all source codes. It is one of the two factors determining the quality of the unit test, and efforts should be made to achieve the highest possible score. In my projects, all test scenarios were prepared to achieve a 100% ratio.
* **Isolation:** Each class was tested independently of other components.
* **Granularity:** Tests verified the behavior of each method and function separately.
* **Quick Feedback:** Unit tests ran quickly and provided instant feedback.
* **Automation:** JUnit was used. Tests can be run automatically, which supports continuous integration processes.

**4. Mutation Testing**

According to Hamilton (2024), mutation testing tests whether test cases get the same result in every situation by making small changes (mutations) in the source codes. Although it may not be possible to get the same result for some codes with these small changes, the highest possible score should be obtained. In my case, mutation testing was used to evaluate the effectiveness of unit tests. I determined that the test quality was quite high by achieving a high score of 88%.

Characteristics:

* **Test Quality Measurement:** Shows how comprehensive and effective the unit tests are.
* **Error Injection:** Measures the robustness of tests by creating artificial errors in the code.
* **Time-Consuming:** Usually takes longer than unit tests and requires more computational power.
* **Automation:** It was automatically performed with the PIT plugin I installed in Eclipse. Being in automation saved me from spending extra time and also supports continuous integration processes.

**Unused Other Software Testing Techniques**

Although unit tests were successful, in all 3 projects, two classes must work together, and the compatibility of these two classes working together needs to be tested. Therefore, in my opinion, the missing part in these assignments was the lack of integration tests that test whether classes work harmoniously together. In fact, as I noticed in milestone 3, both classes were working perfectly, but I observed that I didn't catch exceptions coming from the data class in one place in the service class, and an error occurred in case of an exception, and I intervened. I think integration tests must be done to avoid such errors.

The other two tests that should be applied are Performance and Security testing. In real projects, there are some requirements according to the platform limitations and types on which the projects run (Hardware performance, security), and these are reported in the analysis and design phase. Performance and security tests are applied to codes written according to these requirements. However, since these requirements were not given in these projects, it was not clear what the security and performance tests would be based on. So it might have been meaningless to apply them. However, if the details of the project are specific, these tests should be done and optimized according to the requirements.

**1. Integration Testing**

According to Elankumaran (2024), integration testing tests how parts (units) that have previously undergone unit testing behave when combined and whether they behave as specified. Thus, it guarantees the accuracy, efficiency, and consistency of this functionality.

Characteristics: According to Elankumaran (2024), we can list the characteristics of this test as follows:

* **Inter-Component Interaction:** Tests how different modules work together.
* **System Integrity:** Verifies the functionality of the system as a whole.
* **Data Flow:** Checks the data flow between components.
* **Time-Consuming:** Can be more complex and time-consuming than unit tests. Therefore, using an automation tool is important to save time.

**2. Performance Testing**

According to Cohen (2024), we can define performance tests as a set of tests that determine how the software performs while running, i.e., its resource usage, speed, response capacity, and limits. There are many sub-test types under these tests, such as Load testing, Stress testing, Scalability testing, and Resource usage analysis.

Characteristics: Again, according to Elankumaran (2024), we can list the characteristics of this test as follows:

* **Load Testing:** Measures the performance of the system under a specific load.
* **Stress Testing:** Examines how the system behaves when its limits are pushed.
* **Scalability:** Evaluates how the system copes with increasing user or data load.
* **Resource Usage:** Analyzes the use of system resources such as CPU, memory, disk I/O.

**3. Security Testing**

As an EC Council certified CEH, if I need to explain this part based on my own knowledge, these are types of tests that check how resistant the system is to different attack and intrusion attempts from outside and its possible weak points (vulnerabilities). There are many sub-test types.

**Characteristics:**

* **Vulnerability Scanning:** Detects known security vulnerabilities. It can be applied as passive and active testing. In passive testing application, additional components such as libraries, frameworks used, and programming language versions are checked for the presence of a known security vulnerability. Active, on the other hand, focuses more on the parts of the written code that interact, receive data inputs, XSS, SQL injection, Session and Cookie control and distribution, and checks whether security vulnerabilities occur in these areas. This part is generally accepted within the scope of secure code development and should be implemented by developers.
* **Penetration Testing:** Attacks are carried out to infiltrate or disable the system from outside, thus testing the system's resistance to external attacks.
* **Authentication, Authorization, and Domain Control:** Although the definitions of Authentication, Authorization, and Domain control are similar, they are two very different concepts. Basically, both express whether a user or operation is performed within certain limits and preventing unauthorized access, while Authentication and Authorization are used for operations and data within the software itself. Domain Control, on the other hand, provides control of access to lower-level tiers in the OSI architecture (operating system, memory areas, disk areas, network and processor operation modules) and certain specific areas on them.
* **Data Encryption:** Checks whether sensitive data is adequately protected.

**Practical Uses and Effects**

**Unit Testing**

* Usage: Can be used in all types of software development projects, from small-scale projects to large enterprise applications.
* Impact: Enables early detection of errors, improves code quality, and reduces maintenance costs.
* Example Case: It can be used to ensure that the basket calculation function works correctly in an e-commerce application.

**Mutation Testing**

* Usage: Particularly useful in critical systems or projects requiring high reliability.
* Impact: Improves the quality of the test suite and reveals potential error scenarios.
* Example Case: It can be used to ensure that money transfer functions work correctly in all possible scenarios in a financial transaction application.

**Integration Testing**

* Usage: Critical in projects involving microservice architectures or complex system integrations.
* Impact: Ensures compatible operation of system components and prevents integration errors.
* Example Case: It can be used to verify that different modules of a CRM system (customer management, sales, marketing) work correctly together.

**Performance Testing**

* Usage: Important in high-traffic websites, large data processing systems, or real-time applications.
* Impact: Optimizes system behavior under load and improves user experience.
* Example Case: It can be used to evaluate how a social media platform will perform under high user density.

**Security Testing**

* Usage: Vital for financial applications, health information systems, or any application processing personal data.
* Impact: Prevents data leaks and security breaches, increases user trust.
* Example Case: It can be used to test the firewalls and data encryption mechanisms of an online banking application.

Each of these techniques plays important roles in different projects and situations. The size of the project, its complexity, risk level, and industry requirements are effective factors in determining which test techniques to use.

Software testing techniques play a critical role in ensuring the quality, reliability, and performance of applications. The Unit Testing and Mutation Testing I used in this assignment were quite useful in evaluating the accuracy of the code and the effectiveness of the test suite. However, in real-world applications, applying other techniques such as Integration Testing, Performance Testing, and Security Testing will provide a more comprehensive and reliable software development process.

In future projects, I aim to develop more robust and reliable applications by applying these additional test techniques as well. In particular, the application of Integration Testing will allow us to detect potential errors arising from the collaboration of different components at an early stage.

In conclusion, determining and applying appropriate test strategies according to the unique requirements and risk profile of each project is of great importance in improving software quality and detecting potential problems early on.

**Assessing Mindset in Software Testing**

While testing the code I developed for the project, I understood that testing processes require careful analysis and validation, and I adopted a meticulous and attentive mindset. Attention is extremely important due to the reality that even small mistakes can lead to significant problems later. For example, while testing a method, I analyzed potential erroneous inputs carefully, aware that a faulty operation could lead to issues that would completely affect the program's functionality. I conducted unit tests to ensure that each component worked correctly in isolation and performed mutation tests to evaluate the quality of these tests. However, since integration tests were not required in the project specifications, I did not conduct them, but I was extra careful to ensure that errors that could occur in one part did not cause problems in other parts of the code. This diligence highlighted the importance of the mutual relationships between different code segments. When I encountered an error, tracing its source often led me to fundamental assumptions made in other sections; this inevitably required me to manually conduct some of the integration tests. The coverage levels mentioned above demonstrate how careful I was—performing every test until I reached 100%. This indicates how meticulously the tests were carried out.

Additionally, while writing the code, I considered every possible scenario, especially for the classes that provide the data structure (Contact, Appointment, Task). It can be seen how detailed and error-free they were prepared.

**Biases in Testing**

In my code review, I did not experience any difficulty in overcoming bias; I tend to be harsher on myself when self-criticizing than when I critique others. I strive to do my best and push myself to make each piece of work better than the last. Therefore, I never started a task with a biased approach and have no specific method to cope with it; I simply use some automation tools like JUnit and Pit to see my errors and prepare a ground for myself to identify my mistakes. However, when working with a group, I generally prefer to have someone else review the scenarios to check for gaps in my work; this way, I can identify my deficiencies rather than focusing on biases. All these traits alleviate the concern of bias for me when testing my own code. Therefore, in the workplaces I have been in, I was usually assigned to test scenarios and check the work of others.

**Quality In Testing and Project Management**

In software development, commitment to quality is crucial, but we must also understand what quality means in project management. According to Baratta, there are three independent variables in project management: time, cost, and scope. These are independent variables of the quality function; however, this function can only be established with two of the three independent variables simultaneously. What does this mean? To achieve quality, we can only satisfy two out of these three variables at the same time, which means we have to sacrifice one of them. This is why it is called the "impossible triangle."

If we apply this to software testing, quality does not always mean a comprehensive examination of the code and thorough testing of each stage. Sometimes, project owners may request us to narrow the scope in order to reduce costs and save time. In this case, total quality is still maintained as long as we provide two out of the three priorities. Therefore, in general, quality does not merely mean conducting the most comprehensive and detailed tests but rather satisfying two of these three priorities according to the project's needs.

For instance, if you are asked to change the version of a system that is currently in operation, it should be done quickly and cost-effectively due to the technology no longer being supported or due to an open issue. In this case, it may not be necessary to test every minute detail, but rather just the parts currently used by the program. Thus, the mutation testing phase may be skipped. For example, if I was instructed to change some database libraries in a project written in PHP due to the lack of support, I would need to do this quickly, and it was said that only two people would work on it. In this case, the organization wanted to reduce costs and quickly eliminate potential issues. Instead of these methods, I developed my own methods to overdrive each point, adapting four methods to use more modern structures. This meant that I only needed to perform unit tests to ensure that the old and no longer supported library worked the same way. Since it operated the same way as the old methods, integration tests were skipped, thus saving time and costs.

In conclusion, while cutting corners may be an insignificant behavior that should not be done in perfect scenarios, it may be demanded in real projects. In this case, we can save time from the testing phase by making changes in the code.

**References**

Cohen, N. (2024). Performance Testing vs. Load Testing vs. Stress Testing. Blazemeter.

<https://www.blazemeter.com/blog/performance-testing-vs-load-testing-vs-stress-testing>

Elankumaran, Y. (2024). What is Integration Testing? A Comprehensive Guide. ACCELQ.

<https://www.accelq.com/blog/integration-testing/>

Hamilton, T. (2024). What is Mutation Testing? (Example). Guru99.

<https://www.guru99.com/mutation-testing.html>

Kanai, S. (2022). What is Unit Testing? - A Comprehensive Guide. HeadSpin.

<https://www.headspin.io/blog/unit-testing-guide>

Saladino, G. (2024). What Is Linting + When to Use Lint Tools. Perforce.

<https://www.perforce.com/blog/qac/what-is-linting>

Baratta, A (2006). The triple constraint. Project Management Institute (PMI),

<https://www.pmi.org/learning/library/triple-constraint-erroneous-useless-value-8024>