

**Subject:** Software Test Automations & QA

**Project one:** Software engineer for Grand Strand Systems

**Hamza Malik**

**TABLE OF CONTENT**

[Approach to Unit Testing: 3](#_Toc128272306)

[Experience with JUnit: 3](#_Toc128272307)

[Summary: 3](#_Toc128272308)

[Reflection 6](#_Toc128272309)

[Testing Techniques 6](#_Toc128272310)

[Mindset: 9](#_Toc128272311)

During this project, Writing JUnit tests for a Java application was given to me. The primary objective was to make sure the code was dependable and adhered to the specifications. I'll review my approach to unit testing, consider my experiences, and assess the overall caliber of the tests in this report.

# Approach to Unit Testing:

I took a methodical approach to testing to make sure it was thorough. Initially, I went over the specifications and noted the key functions. After that, I created test cases for each functionality that included every conceivable scenario. I checked the code using white-box testing strategies to make sure that each statement and branch had been tested. To find edge situations and bad inputs, I also employed boundary value analysis and equivalence partitioning.

# Experience with JUnit:

I had never written JUnit tests before. JUnit impressed me as a potent tool for testing Java programming. The framework has a good layout, and the annotations make writing and executing tests simple. Understanding the distinction between unit tests and integration tests was one of the difficulties I ran across. However, after doing some study, I was able to distinguish between the two and create useful unit tests.

# Summary:

**Identify the requirements:** Understanding the feature's requirements is the initial step. The inputs, anticipated outcomes, and edge cases can all be identified with the use of this.

**Write test cases:** Write test cases that account for every scenario that might occur based on the requirements. Be sure to test each statement and branch using white-box testing strategies. To find edge situations and invalid inputs, use boundary value analysis and equivalence partitioning.

**Code the feature and test it:** Codify the functionality, then test it using the test cases after the test cases have been written. Verify that the feature satisfies the requirements and that all test cases pass.

**Refactor and test again:** Refactor the code and test then I ensure the code is perfectly fine for particular function or for module.

It is crucial to carefully evaluate the requirements and create test cases that account for all potential circumstances in order to verify alignment with the software requirements. Valid and invalid inputs, edge cases, and error handling should all be included in the test cases as these step I discuss above.

Also, the unit tests must to be created to make sure that the code complies with both the functional and non-functional specifications listed in the software requirements. To make sure that the code complies with the need, the unit tests should, for instance, incorporate performance tests if the software requirements contain one.

Finally, it is important to check the unit tests to make sure they are in line with the program requirements. All functionalities should be covered, all test cases should be complete, and the code should adhere to the specifications.

JUnit tests efficiency depends not just on their coverage percentage but also on their overall quality and thoroughness. An indicator of how much of the code is covered by the tests is the coverage percentage. High coverage does not, however, automatically imply that the tests are thorough or efficient.

The quality of JUnit tests should be assessed based on a number of criteria, in addition to achieving a high coverage percentage, such as the following:

**Comprehensiveness:** All potential outcomes, including edge cases and unreliable inputs, should be tested for.

**Independence:** The tests shouldn't rely on other tests and should run independently of one another.

**Descriptive Names:** The test cases' titles ought to be evocative and make it clear what they are intended to test.

**Simplicity and Maintainability:** The tests should have clean, concise code that is simple to maintain.

These aspects lead me to conclude that my JUnit tests were of a good caliber. I made sure that all functionality, including edge cases and improper inputs, were tested. The test cases were independent of one another, and I gave each test case a descriptive name to make it simple to understand what it was being used for. I also made the tests straightforward and simple to maintain.

I also succeeded in getting a good coverage 100% in my JUnit tests because for now I just create some Junit test for each module when we develop more complex and large project then might be I faced issue but for now it works good for me, in addition to these considerations. I made sure to test each statement and branch using white-box testing strategies. To find edge situations and bad inputs, I also employed boundary value analysis and equivalence partitioning. I think my JUnit tests were of excellent quality and helped to ensure the dependability of the Java programmed by reaching a high coverage percentage and making sure the tests were thorough and efficient.

For contact file I check the constructor code for this I create the test code for it, then check the initial assignment values for each attribute of the contact class for this I assign the some values to each of the attributes according to their data types in Junit test file then test the code of the constructor which is written by the programmer or developer. In a test code we create the void setup method in which I assigned the value to each of the attribute.

# Reflection

## Testing Techniques

Testing individual software application parts or components is known as **unit testing**. Characteristics:

• developer testing at the lowest level which is test by the developer.

• testing isolated tiny, modular pieces of code which is very handy.

• standardized and automated

• this is the Quick and inexpensive way of testing.

**Integration testing** examines how the various components of the software program function as a whole. Characteristics:

• after unit testing is finished, testing is conducted

• focuses on examining how integrated components interact with one another

• either manually or automatically

• if errors occur in the early detection of integration issues during the development cycle.

Or **acceptance testing** while developing the unit testing, we check the how many tests we pass then we accept our code because it produces the desire results for that.

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

**Functional Testing** involves different techniques and characteristics:

• tests the functional requirements of the software program which ensure that our software functional requirements working perfectly or not.

• consists of gray-box, white-box, and black-box testing

• In ensuring that the software program is operationally working well or not.

**Performance testing** entails putting a piece of software through its paces in order to gauge its speed, scalability, and stability. Characteristics:

• examines how the software performs under different stress conditions

• consisting of volume, load, and stress testing

• enables the software application to handle predicted levels of demand and enables early performance degradation detection.

**Unit test**

**Practical applications:**

• It aids in the early detection and isolation of bugs in the development process.

• It makes it simpler to maintain and refactor code.

• It boosts faith in the code and the creation process.

**Implications:**

• Setting up and maintaining unit testing takes time and effort.

• It works best when done as a part of a larger testing plan that also involves other kinds of testing.

• Unit tests might create a false sense of security if they are not up to date.

**Integration testing:**

**Practical applications:**

• It helps in the detection of problems with the integration of various software components.

• It guarantees that the software program functions properly as a whole.

• Early on in the development process, it assists in identifying and resolving performance concerns.

**Implications:**

• Integration testing can be difficult and time-consuming, especially if there are numerous components that need to be merged.

• To guarantee that all pertinent elements and interactions are evaluated, a well-designed testing strategy is necessary.

• Coordination with other development teams or stakeholders might be necessary.

**Acceptance testing:**

**Practical uses:**

• It includes ensuring that the software program satisfies the needs and expectations of the customer.

• It aids in finding usability and other user-experience issues.

• It offers a last-minute check of the software program before it is made public.

**Implications:**

• End users or consumers must play a considerable role in acceptance the testing.

• Coordination with other development teams or stakeholders might be necessary for cross over.

• It could need to be tested in a real-world setting.

# Mindset:

I). When testing software, a software tester should take a cautious attitude. This requires meticulousness, caution, and thoroughness in the creation and execution of test cases. All functionalities, edge cases, and the effectiveness of the tests in locating flaws must be guaranteed.

In order to detect probable regions of the code that may contain errors, it is crucial to understand the complexity and interrelationships of the code that is being tested. Testers can create test cases that cover all potential scenarios and edge cases by comprehending the structure and interdependencies of the code.

For instance, it's crucial to test a piece of code's behavior under various data and service situations if it depends on external data or services. Testers can find potential flaws that might be introduced by changes in the external data or service by running the code under various situations, but for this I didn’t need some external sources to do that.

II).

**Follow a standardized review process:** Using a consistent review procedure is one technique to reduce bias while evaluating code. Specific requirements for the code should be included in the process documentation. Reviewers can avoid being swayed by prejudices and personal opinions by adhering to a regular process.

**Use automated tools:** Potential problems in the code, such as grammatical errors, code smells, and security vulnerabilities, can be found by automated methods. Reviewers can avoid being swayed by their own prejudices and concentrate on more complicated topics by employing automated methods.

**Collaborate with other reviewers:** By giving a variety of viewpoints on the code being evaluated, collaboration with other reviewers can help prevent bias. Any biases can be found and corrected by talking about the code with other reviewers.

Bias might be an issue if a software developer is in charge of testing their own code. The developer can be too accustomed to their own code and fail to notice potential problems. Without thoroughly testing all potential circumstances, a developer can, for instance, presume that a piece of code works as intended because they developed it.

III). Professionals in software engineering must be rigorous in their dedication to quality. It can be time-consuming and difficult to write and test code, and it might be tempting to use shortcuts to meet deadlines or save time. Cutting corners, on the other hand, might result in technical debt, which represents the expense of correcting errors that could have been prevented during the software development process. Increased expenditures, postponed projects, and worse customer satisfaction are all effects of technical debt.

When writing or testing code, it's crucial not to skimp because mistakes might have serious repercussions. Defects can cause a variety of issues, including output that is erroneous, system failures, and security vulnerabilities. Early defect detection and correction during the software development process is more cost-effective since the cost of addressing flaws rises exponentially the longer, they go unnoticed, but for the short program or for the short project or module it would work or we can remove the bias from this easily as compared to when we have more complex application or code.