SOFTWARE TESTING ASSIGNMENT

Practical Exercises

1) **Unit Testing: Perform unit testing for a simple "Calculator" class that has methods for addition, subtraction, multiplication, and division. Write test cases for each method and ensure they pass successfully. Provide the code for both the test cases and the Calculator class.**

Unit Testing:

Unit testing is a software testing technique that focuses on evaluating individual units or components of a software application in isolation. A "unit" in this context typically refers to the smallest testable part of a software program, such as a function, method, or class. The main purpose of unit testing is to ensure that each unit of the code works as expected and that it performs its specific function correctly. It is done during the coding phase by the developers. To perform unit testing, a developer writes a piece of code (unit tests) to verify the code to be tested (unit) is correct.

Test Cases:

a) **Test Cases on addition:**
   [1] Verify that the addition function correctly adds two positive integers.
   [2] Verify that the addition function correctly adds two negative integers.
   [3] Verify that the addition function handles the addition of a positive and a negative integer correctly.
   [4] Verify that adding zero to a number returns the original number.
   [5] Verify that the addition function handles floating-point numbers correctly.
   [6] Test the addition of two large numbers to ensure that it doesn't result in overflow or precision issues.

b) **Test Cases on subtraction:**
   [1] Verify that the subtraction function correctly subtracts one positive integer.
   [2] Ensure that the subtraction function correctly handles the subtraction of one negative integer from another.
   [3] Verify that the subtraction function correctly handles the subtraction of a positive integer from a negative integer.
   [4] Verify that subtracting zero from a number returns the original number.
   [5] Verify that the subtraction function correctly handles floating-point numbers.

[6] Test the subtraction of two large numbers to ensure it doesn't result in underflow or precision issues.

c) Test Cases on multiplication:

[1] Verify that the multiplication function correctly multiplies two positive integers.
[2] Ensure that the multiplication function correctly multiplies two negative integers.
[3] Verify that the multiplication function handles the multiplication of a positive and a negative integer correctly.
[4] Verify that multiplying any number by zero results in zero.
[5] Verify that the multiplication function correctly handles floating-point numbers.
[6] Test the multiplication of two large numbers to ensure it doesn't result in overflow or precision issues.

d) Test Cases on division:

[1] Verify that the division function correctly divides one positive integer by another.
[2] Ensure that the division function correctly divides one negative integer by another.
[3] Verify that the division function handles the division of a positive integer by a negative integer correctly.
[4] Verify that dividing any number by zero results in an appropriate error or exception.
[5] Verify that the division function correctly handles division involving floating-point numbers.
[6] Test the division of two large numbers to ensure it doesn't result in overflow or precision issues.

Code:

```
import unittest

class Calculator:
    def add(self, x, y):
        return x + y
    def subtract(self, x, y):
        return x - y
    def multiply(self, x, y):
        return x * y
    def divide(self, x, y):
        if y == 0:
            raise ValueError("Division by zero is not allowed")
        return x / y
```

```
class TestCalculator(unittest.TestCase):
    def setUp(self):
        self.calculator = Calculator()

    def test_addition(self):
        self.assertEqual(self.calculator.add(2, 3), 5)  # 2 + 3 = 5
        self.assertEqual(self.calculator.add(-2, -3), -5) # (-2) + (-3) = -5
        self.assertEqual(self.calculator.add(-1, 1), 0)  # -1 + 1 = 0
        self.assertEqual(self.calculator.add(0, 0), 0)  # 0 + 0 = 0
        self.assertEqual(self.calculator.add(2, 0), 2) # 2 + 0 = 2
        self.assertEqual(self.calculator.add(2000000, 3000000), 5000000) # 1000000 + 2000000
= 5000000
        self.assertEqual(self.calculator.add(2.5, 3.5), 6.0) # 2.5 + 3.5 = 6.0

    def test_subtraction(self):
        self.assertEqual(self.calculator.subtract(5, 3), 2)  # 5 - 3 = 2
        self.assertEqual(self.calculator.subtract(-5, -3), -2) # (-5) - (-3) = -2
        self.assertEqual(self.calculator.subtract(-1, 1), -2)  # -1 - 1 = -2
        self.assertEqual(self.calculator.subtract(0, 0), 0)  # 0 - 0 = 0
        self.assertEqual(self.calculator.subtract(3.5, 2.5), 1.0) # 3.5 - 2.5 = 1.0

    def test_multiplication(self):
        self.assertEqual(self.calculator.multiply(2, 3), 6)  # 2 * 3 = 6
        self.assertEqual(self.calculator.multiply(-2, -4), 8) # -2 * -4 = 8
        self.assertEqual(self.calculator.multiply(8, -3), -24) # 8 * -3 = -24
        self.assertEqual(self.calculator.multiply(3.5, 2.5), 8.75)  # 3.5 * 2.5 = 8.75
        self.assertEqual(self.calculator.multiply(0, 5), 0)  # 0 * 5 = 0

    def test_division(self):
        self.assertEqual(self.calculator.divide(6, 2), 3)  # 6 / 2 = 3
        self.assertEqual(self.calculator.divide(-8, -2), 4) # -8 / -2 = 4
```

```
        self.assertEqual(self.calculator.divide(12, -3), -4) # 12 / -3 = -4
        self.assertEqual(self.calculator.divide(5, 2), 2.5)  # 5 / 2 = 2.5
        self.assertEqual(self.calculator.divide(3.5, 2.5), 1.4) # 3.5 / 2.5 = 1.4
        self.assertEqual(self.calculator.divide(0, 5), 0)  # 0 / 5 = 0

        # Test division by zero
        with self.assertRaises(ValueError):
            self.calculator.divide(1, 0)

if __name__ == '__main__':
    unittest.main()
```

2) **Functional Testing: You have been given a web application with a registration page. Write test cases to validate the registration process. Include test cases for successful registration, registration with missing information, and registration with invalid data. Explain how you would perform these tests.**

Functional Testing:

Functional testing is a type of software testing that verifies the functionality of a software system or application. It focuses on ensuring that the system behaves according to the specified functional requirements and meets the intended business needs.

The goal of functional testing is to validate the system's features, capabilities, and interactions with different components. It involves testing the software's input and output, data manipulation, user interactions, and the system's response to various scenarios and conditions. Functional testing is only concerned with validating if a system works as intended.

Test Cases:

1. Test Case: Successful Registration

Description: Verify that a user can successfully register with valid information.

Test Steps:

[1] Navigate to the registration page.

[2] Fill in all required fields with valid data (e.g., name, email, password).

[3] Click the "Register" or "Sign Up" button.

[4] Check if the user is redirected to a confirmation page.

[5] Verify that a confirmation email is sent to the provided email address.

Expected Outcome: The user is registered successfully, and a confirmation email is sent.

2. Test Case: Registration with Missing Information

Description: Ensure that the registration process handles missing information appropriately.

Test Steps:

[1] Navigate to the registration page.

[2] Submit the registration form with one or more required fields left empty (e.g., name, email, password).

[3] Click the "Register" or "Sign Up" button.

[4] Check for error messages or notifications.

Expected Outcome: The registration process should not proceed, and appropriate error messages indicating the missing information should be displayed.

3. Test Case: Registration with Invalid Email

Description: Verify that the registration process validates email addresses and rejects invalid ones.

Test Steps:

[1] Navigate to the registration page.

[2] Enter an invalid email address (e.g., missing "@" symbol, incorrect format).

[3] Fill in valid information for other required fields.

[4] Click the "Register" or "Sign Up" button.

[5] Check for error messages or notifications.

Expected Outcome: The registration process should not proceed, and an error message indicating the invalid email should be displayed.

4. Test Case: Registration with Duplicate Email

Description: Verify that the registration process prevents users from registering with an email address that is already in use.

Test Steps:

[1] Navigate to the registration page.

[2] Enter an email address that is already registered.

[3] Fill in valid information for other required fields.

[4] Click the "Register" or "Sign Up" button.

[5] Check for error messages or notifications.

Expected Outcome: The registration process should not proceed, and an error message indicating the duplicate email should be displayed.

5. Test Case: Confirmation Email Receipt

Description: Verify that a user who successfully registers receives a confirmation email.

Test Steps:

[1] After a successful registration, check the registered email inbox.

Expected Outcome: The user should receive a confirmation email with relevant registration details.

3) **Exploratory Testing: Imagine you are testing a music streaming application on your smartphone. Perform exploratory testing and list three issues or defects you encounter during this testing. Provide a brief description of each issue.**

Exploratory Testing:

Exploratory testing is an approach to software testing that is often described as simultaneous learning, test design, and execution. It focuses on discovery and relies on the guidance of the individual tester to uncover defects that are not easily covered in the scope of other tests.

1. Buffering Loop Issue:

Description: While using the music streaming app, you notice that a song gets stuck in a buffering loop. The app keeps trying to load the song but never plays it. The buffering indicator remains on the screen indefinitely.

Impact: Frustrating user experience as users cannot listen to the intended song, leading to potential app abandonment.

2. Playlist Shuffle Not Working:

Description: You decide to shuffle your playlist, but the app continues to play songs in sequential order rather than randomly. The shuffle feature does not seem to be functioning correctly.

Impact: Users who expect variety in their music experience will be disappointed as they get the same order of songs each time they use the shuffle feature.

3. UI Lag During Search:

Description: While conducting a search for a specific artist or song, you notice a noticeable lag in the user interface (UI). When you start typing in the search bar, there is a delay before the search results appear.

Impact: Sluggish UI responsiveness can be frustrating for users, affecting the overall usability of the application.

4) **Regression Testing**: **Suppose you are working on a software project, and a new feature has been added to the application. Outline a regression testing strategy to ensure that the new feature does not introduce any new defects. Explain the types of test cases you would execute and any specific areas you would focus on.**

Regression Testing:

Regression testing is a type of software testing conducted after a code update to ensure that the update introduced no new bugs. This is because new code may bring in new logic that conflicts with the existing code, leading to defects. Usually, QA teams have a series of regression test cases for important features that they will re-execute each time these code changes occur to save time and maximize test efficiency.

Here's an outline of a regression testing strategy for this scenario:

1. **Identify Critical Use Cases:** Identify the critical use cases or key functionalities in the application that are most likely to be affected by the new feature. This can include core features and functionalities that many users rely on.
2. **Smoke Testing:** Start with a smoke test to ensure that the application still launches and the basic functionality works. This helps catch any critical issues early.
3. **Test Automation:** Automate relevant test cases, especially those that are part of your regression suite. Automated tests can be quickly executed and are less prone to human error.
4. **Execute a Full Regression Suite:** Run the entire regression test suite to verify that existing features are still working as expected. This suite should cover both the existing features and any new features or changes.
5. **Focus on Boundary and Edge Cases:** Pay special attention to boundary conditions and edge cases to ensure that the new feature doesn't create unexpected behavior in situations that are near the limits of the system's capabilities.
6. **Integration Testing:** Test the interaction between the new feature and existing features, making sure they work harmoniously together.
7. **Compatibility Testing:** Verify that the new feature works as expected across different browsers, operating systems, and devices, especially if the application has a web-based component.
8. **Performance Testing:** Conduct performance testing to ensure that the addition of the new feature doesn't lead to performance degradation or increased resource consumption.

9. **Security Testing:** Verify that the new feature doesn't introduce security vulnerabilities or risks. Pay attention to data privacy, access control, and potential threats.
10. **Usability Testing:** Ensure that the new feature aligns with the application's user interface and user experience design. Check for any usability issues that may arise due to the new feature.
11. **Load Testing:** If the new feature may affect the application's ability to handle concurrent users or high traffic, perform load testing to ensure that the application can still handle the expected loads.
12. **User Acceptance Testing (UAT):** If applicable, involve end-users or stakeholders in user acceptance testing to get their feedback on the new feature and its integration with existing features.
13. **Error Handling and Recovery:** Test how the application handles errors related to the new feature. Verify that error messages are clear and that the system gracefully recovers from issues.
14. **Documentation:** Ensure that documentation is updated to reflect the new feature. This includes user manuals, help guides, and internal developer documentation.
15. **Bug Reporting:** If any defects are found during regression testing, document and report them promptly to the development team for resolution.
16. **Retesting**: After defects are fixed, perform retesting to ensure that the issues have been resolved and that the new feature still works as expected.
17. **Continuous Monitoring:** Implement continuous monitoring and automated testing in the build and deployment pipeline to catch any regressions early in the development process.

Technical Questions

5) Explain the concept of "test coverage" in software testing. Why is it important, and what are the different types of coverage metrics that can be used in testing?

Test coverage:

Test coverage in software testing is a metric used to measure the extent to which a test suite covers the code and functionality of a software application. It indicates the percentage or proportion of code, requirements, or features that have been exercised by a set of test cases. Test coverage helps in evaluating the thoroughness and effectiveness of the testing process and provides insights into the areas that have been tested and those that have not.

Test coverage is important for several reasons:

1. **Quality Assurance:** It helps ensure that the software is thoroughly tested, reducing the risk of undetected defects that could lead to software failures or issues in production.

2. **Risk Assessment:** Test coverage provides a clear picture of areas of the code that are not tested. This allows the testing team to focus on high-risk or critical parts of the software.
3. **Requirements Validation:** It helps confirm that the software meets the specified requirements and that the implemented features work as intended.
4. **Test Effectiveness:** It allows testers and developers to identify redundant or ineffective test cases, helping optimize the testing effort.
5. **Compliance:** In safety-critical industries (e.g., aviation, healthcare), test coverage metrics are essential for regulatory compliance and certification.

There are different types of coverage metrics used in software testing:

1. **Code Coverage:**
   - **Statement Coverage:** Measures the percentage of executable statements in the code that have been executed by the test cases. It does not necessarily verify if all branches and conditions have been exercised.
   - **Branch Coverage (Decision Coverage):** Measures the coverage of decision points in the code, ensuring that both true and false branches of conditions are executed.
   - **Path Coverage:** Ensures that all possible execution paths through the code are tested. This is a more detailed and thorough metric but can be impractical for complex software.
2. **Function/Method Coverage:**
   - **Function/Method Coverage:** Measures whether each function or method in the code has been invoked by the test cases.
3. **Statement Block Coverage:**
   - **Block Coverage:** Focuses on code blocks or compound statements, ensuring that all blocks within functions are executed.
4. **Condition Coverage:**
   - Condition Coverage: Measures the coverage of Boolean conditions within code. It ensures that all possible combinations of conditions (true and false) are tested.
5. **Path Coverage:**
   - **Path Coverage:** Ensures that all possible execution paths through the code are tested. This is a more detailed and thorough metric but can be impractical for complex software.
6. **Requirements Coverage:**
   - **Functional Requirements Coverage:** Ensures that all specified functional requirements are tested.
   - **Non-Functional Requirements Coverage:** Ensures that non-functional requirements (e.g., performance, security) are met.
7. **Use Case Coverage:**
   - **Use Case Coverage:** Measures the coverage of use cases in the software, ensuring that all defined use cases are tested.
8. **Interface Coverage:**
   - Interface Coverage: Focuses on the coverage of interfaces and APIs, ensuring that all methods and inputs are tested.

9. **State Transition Coverage:**
    - State Transition Coverage: Used in state-based systems to verify that all possible state transitions have been tested.

6) Describe the differences between black-box testing and white-box testing. Provide examples of when each approach might be more suitable.

Black Box Testing	White Box Testing
It is a way of software testing in which the internal structure or the program is hidden and nothing known about it.	It is a way of testing the software in which the tester has knowledge about the internal structure or the code.
Implementation of code is not need for black box testing.	Code implementation is necessary for white box testing.
It is done by software testers.	It is done by software developers.
No knowledge of implementation is needed.	Knowledge of implementation is required.
It can be referred as external software testing.	It can be referred as internal software testing.
It is a functional test of the software.	It is a structural test of the software.
This testing can be initiated based on the requirement specifications document.	This type of testing of software is started after a detail design document.
No knowledge of programming is required.	It is mandatory to have knowledge of programming.
It is the behaviour testing of the software.	It is the logic testing of the software.
It is also called closed testing.	It is also called glass box testing or transparent testing.
It is least time consuming.	It is most time consuming.
It is not suitable or preferred for algorithm testing.	It is suitable for algorithm testing.
Can be done by trial-and-error ways and methods.	Data domains along with inner or internal boundaries can be better tested.
Example: Search something on google by using keywords	Example: By input to check and verify loops
Black-box test design techniques- • Decision table testing • All-pairs testing • Equivalence partitioning • Error guessing	White-box test design techniques- • Control flow testing • Data flow testing • Branch testing
Types of Black Box Testing: • Functional Testing • Non-functional testing • Regression Testing	Types of White Box Testing: • Path Testing • Loop Testing • Condition testing • Memory testing • Response time testing

It is less exhaustive as compared to white box testing.	It is comparatively more exhaustive than black box testing.