Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Software Testing and Quality Assurance

Name: Varenya Uchil

SAPID: 60004210121

Branch: Computer Engineering C2-2

Experiment No. 4

Aim: BlackBox Box Testing on Units/Modules of Income Tax Calculator using any suitable tool

Theory:

Black box testing is a software testing method in which the tester does not have access to the internal code structure, implementation details, or logic of the system under test. The focus is solely on the *inputs and outputs* of the software system, and whether it behaves as expected based on the functional and non-functional requirements. It is also known as **behavioral testing**, **data-driven testing**, or **functional testing**.

This strategy is typically employed during the later stages of software development such as system testing, acceptance testing, or during quality assurance (QA) processes. Since testers do not require programming knowledge or understanding of the internal architecture, black box testing can be done by independent testing teams or even end users in some cases.

Objectives of Black Box Testing

- To validate the behavior of the software against the user requirements.
- To identify discrepancies between expected and actual outputs.
- To ensure that the system works correctly with all kinds of valid inputs and handles invalid inputs gracefully.
- To verify that non-functional attributes such as performance, usability, and reliability meet expectations.

Key Characteristics

- **No knowledge of internal code** is required.
- Focus is on **functionality** rather than the implementation.
- Testing is done from the **user's perspective**.
- Based on **specifications, user stories, and requirements documents**.

- Test cases are developed to cover a range of inputs and expected outputs.

Advantages of Black Box Testing

1. **User-Centric**: Reflects how an end-user would interact with the system.
2. **Unbiased Testing**: Since testers don't know the internal structure, their tests are unbiased toward implementation logic.
3. **Efficient for Large Systems**: Allows testing of complex applications without needing to understand all underlying components.
4. **Applicable to All Levels**: Can be applied to unit, integration, system, and acceptance testing.

Disadvantages of Black Box Testing

1. **Limited Coverage**: Not all paths and scenarios may be tested since internal logic is unknown.
2. **Difficult to Identify All Conditions**: Without code access, some branches or edge cases might be missed.
3. **Redundant Testing**: There may be overlaps with other tests or missed areas that require white box testing for thorough coverage.
4. **Poor Debugging Support**: When a defect is found, pinpointing the root cause within the code is difficult without developer involvement.

Common Black Box Testing Techniques

1. **Equivalence Partitioning**
    - Inputs are divided into valid and invalid partitions.
    - Only one input from each partition is tested, assuming all behave similarly.
    - Example: For an age input field accepting ages 18–60, test with 25 (valid), 17 (invalid), and 61 (invalid).
2. **Boundary Value Analysis (BVA)**
    - Focuses on testing the boundaries between partitions.
    - More defects are found at the edges of input ranges.
    - Example: For a field accepting 1–100, test with 0, 1, 100, and 101.
3. **Decision Table Testing**
    - Used for systems with complex business rules and logic.
    - Converts rules into a decision table format (inputs vs actions).
    - Ensures all rule combinations are tested.
4. **State Transition Testing**

- Tests the system's response to different events depending on current state.
- Useful for workflows, navigation, or multi-step processes.
- Example: ATM interface that changes state from "idle" to "processing" after card insertion.

5. **Error Guessing**
   - Based on experience and intuition, testers guess error-prone areas and test accordingly.
   - Not systematic but can uncover hidden bugs.

Types of Black Box Testing

1. **Functional Testing**
   - Tests specific functions or features (e.g., login, checkout, search).
   - Focus on correctness of outputs for valid and invalid inputs.
2. **Non-Functional Testing**
   - Checks system attributes like performance, usability, security, and compatibility.
   - Example: Load testing a website to handle 10,000 concurrent users.
3. **Regression Testing**
   - Ensures that new changes or bug fixes haven't broken existing functionality.
4. **Acceptance Testing**
   - Final level of testing before release.
   - Determines if the system meets business requirements and is ready for use.

Tools and IDE used -

1. **Java Development Kit (JDK):**

   This is the core requirement for developing and running Java applications. JUnit relies on the JDK to compile and execute your test cases.

   JUnit Framework:

   This is the testing framework that provides annotations, assertions, and other functionalities for writing and running unit tests.

2. **Eclipse IDE:**

   Eclipse is a popular open-source IDE well-suited for Java development, including JUnit testing.

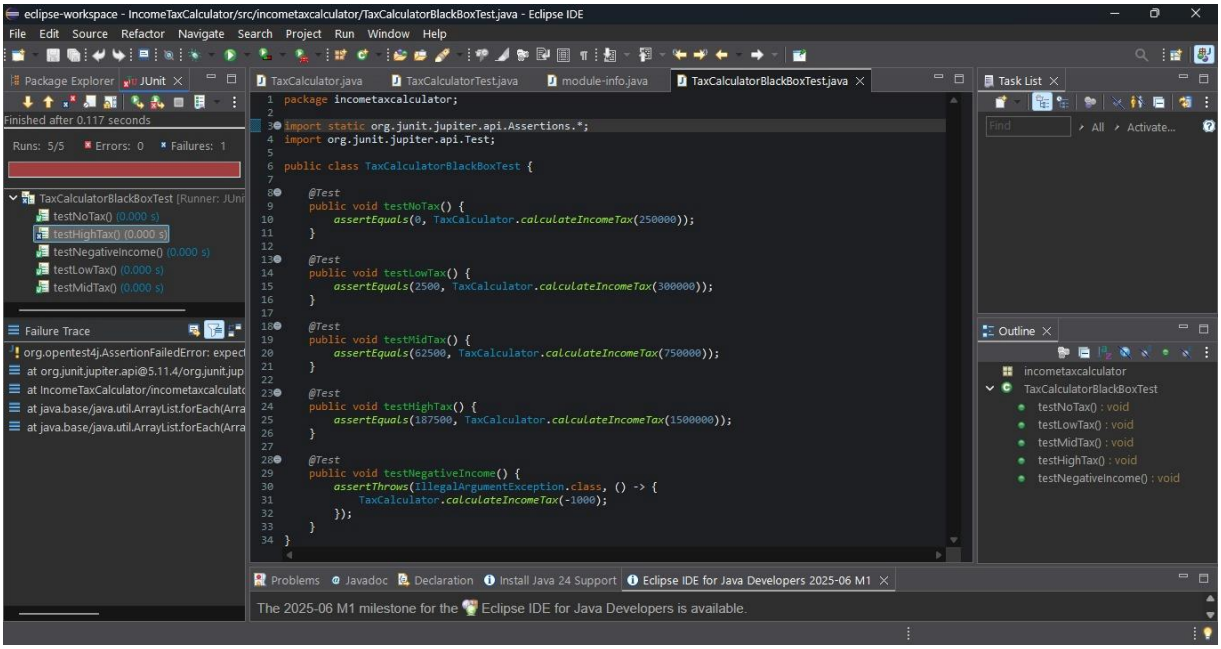
Code to be tested:

```
package incometaxcalculator;
import java.util.Scanner;

public class TaxCalculator {
   public static double calculate(double amount, double percent) {
      return (amount * percent) / 100;
   }
   public static double CalculateIncomeTax(double totalIncome) {
      if (totalIncome <= 250000) {
         return 0;
      } else if (totalIncome <= 500000) {
         return calculate(totalIncome - 250000, 5);
      } else if (totalIncome <= 750000) {
         return calculate(totalIncome - 500000, 10) + 12500;
      } else if (totalIncome <= 1000000) {
         return calculate(totalIncome - 750000, 15) + 37500;
      } else if (totalIncome <= 1250000) {
         return calculate(totalIncome - 1000000, 20) + 75000;
      } else if (totalIncome <= 1500000) {
         return calculate(totalIncome - 1250000, 25) + 125000;
      } else {
         return calculate(totalIncome - 1500000, 30) + 187500;
      }
   }
   public static int calculateIncomeTax(int income) {
         if (income<0) {
                  throw new IllegalArgumentException("Income cannot be neagtive.");
         }
      else if (income <= 250000) return 0;
      else if (income <= 500000) return (income - 250000) * 5 / 100;
      else if (income <= 1000000) return 12500 + (income - 500000) * 20 / 100;
      else return 112500 + (income - 1000000) * 30 / 100;
   }
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);
      System.out.print("Enter your annual income: ");
      double totalIncome = scanner.nextDouble();
      double tax = CalculateIncomeTax(totalIncome);
```

```
        System.out.printf("Total tax applicable at ₹%.2f is ₹%.2f\n", totalIncome, tax);
        scanner.close();
    }
}
```

Testing Code with Output:



Observations:

Black box testing was successfully used to test the functionality of the income tax calculator without knowing the internal code. Different inputs and outputs were validated, edge cases were verified, and error handling was tested. The system responded correctly to valid income ranges and handled incorrect input gracefully.

Conclusion:

Black box testing is an essential part of software quality assurance that ensures the system works correctly from the user's perspective. While it does not uncover internal code flaws, it plays a vital role in identifying missing functionalities, user interface issues, and system-level bugs. By complementing white box testing and integration testing strategies, black box testing contributes significantly to building reliable, user-friendly software systems.

References:

- JUnit 5 Docs: https://junit.org/junit5/docs/current/user-guide/
- Oracle Java Docs: https://docs.oracle.com/javase/8/docs/
- Eclipse IDE: https://www.eclipse.org/downloads/