Department of Computer Engineering
Academic Year 2024-2025

NAME:-Shashwat Shah
SAPID:-60004220126
BRANCH:-ComputerEngineering
DIV:-C2 ; BATCH:- C2-1

SOFTWARE TESTING & QUALITY ASSURANCE (STQA)
EXPERIMENT NO.03

AIM:- White Box Testing on Units/Modules of Income Tax Calculator using Junit/Selenium.

THEORY:-

White-box testing, also known as **clear-box testing** or **structural testing**, involves testing the internal workings or code structure of a system. This testing approach requires knowledge of the code itself to create test cases based on the program's internal logic, such as conditions, loops, data structures, and algorithms.

Process of White Box Testing

White box testing include the verify the internal workings of a software application. It checks that every aspect of the code is tested, basically is focusing on the logic, structure, and flow of the software.

Here's a breakdown of how this process works:

Process of White Box Testing

1. Input: The process starts by combining all the related documents, including:

Requirements: These outline what the application is supposed to do and its expected behavior.

Functional Specifications: These describe how the software should perform under specific conditions.

Design Documents: These provide detailed insights into the architecture, components, and flow of the system.

Source Code: This is the actual code written for the application. It is where the logic and functionality are defined and is the primary focus during white box testing.

2. Processing: Once the input is combined, the next step is:

Risk Analysis: This step identifies potential risks in the code. By analyzing the application's functionality and dependencies, testers can identify areas where errors are more likely to occur and prioritize testing those areas. This helps in making the testing process more focused and efficient for the further process.

Test Planning: In this stage, testers design detailed test cases that cover all aspects of the code. The aim is to check all paths, conditions, loops, and functions within the code. Test planning ensures that no part of the application is left untested.

3. Test Execution: Now that the test cases are ready to execute and check if any difficulties which we are facing during the same:

Execute the Tests: The test cases are run to check the behavior of the application. During execution, the application's internal logic is verified during the same. This includes testing individual functions, loops, and conditions to check that they work as expected.

Error Identification and Fixing: If errors or bugs are found, they are reported to the development team. The development team fixes the errors, and the tests are again run to verify the fixes. This cycle continues until the software is free from critical issues.
Results Communication: within the process of testing, the results are documented and communicated to all stakeholders to re-sure everyone is informed of the software's progress.
4. Output: Once testing is completed with all error solving then these is the final step which we are performing:
Final Report: A detailed report is prepared that includes all findings, test case results, error logs, and improvements made in the proper format which is easily understandable. This report documented as a record of the testing process and provides an complete overview of the software's quality. It is typically shared with the development team and other related stakeholders and members.
In white-box testing, the tester must to understand the application's code and write test cases to validate specific parts of it with checking all the function of the software. Then they can execute these tests, identify any issues, and check the software works correctly as expected.

Tools of White box testing

White box testing, also known as clear box or structural testing, involves examining the internal workings of an application to ensure its functionality and security. In 2025, several tools have used for this process. Here are few white box testing tools:
SonarQube
Veracode
OWASP Code Pulse
JaCoCo
PVS-Studio
Checkmarx
Coverity
Klocwork
CodeClimate
Codacy

White-box Testing on Units/Modules of an Income Tax Calculator

An income tax calculator typically involves modules or functions that calculate taxes based on various factors such as income, tax brackets, exemptions, deductions, and applicable rates. When performing white-box testing on such a system, we focus on testing individual units or modules of the program (e.g., tax bracket calculations, exemption application, etc.) by looking at the internal code structure, branches, conditions, and possible paths.
Here's how you can apply white-box testing to different units/modules of an income tax calculator.

1. Module: Income Tax Calculation

Calculator Code:

```
def calculate_tax(income):
    if income <= 250000:
        return 0
    elif 250001 <= income <= 500000:
        return (income - 250000) * 0.05
    elif 500001 <= income <= 1000000:
        return (income - 500000) * 0.1 + 12500  # 12,500 is the tax for the
previous bracket
    else:
        return (income - 1000000) * 0.2 + 12500 + 50000  # 50,000 is the tax
for the previous bracket


test_1 = calculate_tax(200000)
print("Result of testcase 1: ", test_1)

test_2 = calculate_tax(300000)
print("Result of testcase 2: ", test_2)

test_3 = calculate_tax(700000)
print("Result of testcase 3: ", test_3)

test_4 = calculate_tax(1500000)
print("Result of testcase 4: ", test_4)
```

Output:

```
Result of testcase 1:  0
Result of testcase 2:  2500.0
Result of testcase 3:  32500.0
Result of testcase 4:  162500.0
```

Test Coverage and White-box Testing:

- **Control Flow Testing**: Test all possible paths in the function. For the above example, there are four distinct paths:
  - Income <= 250,000 → No tax.
  - Income between 250,001 and 500,000 → Apply 5% tax.
  - Income between 500,001 and 1,000,000 → Apply 10% tax.
  - Income > 1,000,000 → Apply 20% tax.

Ensure you have test cases that will execute each path, for example:

- Test case 1: income = 200000 → Expected output: 0
- Test case 2: income = 300000 → Expected output: 2500

- Test case 3: income = 700000 → Expected output: 32500
- Test case 4: income = 1500000 → Expected output: 225000

- **Condition Coverage**: Ensure that all conditions (income <= 250000, 250001 <= income <= 500000, etc.) are tested with both true and false values.
  - Example: Testing the boundary values like income = 250000 and income = 250001.
- **Path Coverage**: Ensure that every possible path in the function is covered. This will require testing edge cases where the income is exactly at the boundaries (e.g., 250,000, 500,000, and 1,000,000).

2. Module: Tax Exemptions

Code Example:

```
def apply_exemptions(income, exemptions):
    if income <= 500000:
        return income
    else:
        return income - exemptions
```

Test Coverage and White-box Testing:

- **Branch Testing**: In this function, there are two possible branches:
  - If income is <= 500,000 → No exemption applied.
  - If income > 500,000 → Apply exemptions.

Test cases should include:

- Test case 1: income = 400000, exemptions = 50000 → Expected output: 400000
- Test case 2: income = 600000, exemptions = 50000 → Expected output: 550000

- **Boundary Testing**: Ensure boundary values are covered. For example:
  - Test case 3: income = 500000, exemptions = 0 → Expected output: 500000
  - Test case 4: income = 500001, exemptions = 10000 → Expected output: 490001

3. Module: Tax Slab Validation

Code Example:

```
def validate_tax_slab(income):
    if income <= 0:
        raise ValueError("Income must be positive")
    elif income > 10000000:
        return "Income exceeds limit"
    else:
        return "Valid"
```

Test Coverage and White-box Testing:

- **Path Testing**: This function has three distinct paths:
  - Invalid income (income <= 0).
  - Income greater than 10,000,000.
  - Valid income.

Ensure you cover all paths:

- Test case 1: income = -500 → Expected output: Raise ValueError("Income must be positive")
- Test case 2: income = 20000000 → Expected output: "Income exceeds limit"
- Test case 3: income = 1000000 → Expected output: "Valid"

- **Boundary Testing**: Ensure boundary values are tested:
  - Test case 4: income = 1 → Expected output: "Valid"
  - Test case 5: income = 10000000 → Expected output: "Valid"
  - Test case 6: income = 10000001 → Expected output: "Income exceeds limit"

4. Module: Deduction Calculation

Code Example:

```
def calculate_deductions(income, deductions):
  if income > 500000:
    return deductions
  return 0
```

Test Coverage and White-box Testing:

- **Decision Coverage**: This module has one decision — whether income > 500000 is true or false. It must be tested with both true and false conditions.
  - Test case 1: income = 600000, deductions = 50000 → Expected output: 50000
  - Test case 2: income = 400000, deductions = 50000 → Expected output: 0
- **Boundary Testing**: Testing around the boundary condition (income = 500000):
  - Test case 3: income = 500000, deductions = 30000 → Expected output: 0
  - Test case 4: income = 500001, deductions = 30000 → Expected output: 30000

5. Module: Final Tax Calculation

Code Example:

```
def final_tax(income, exemptions, deductions):
  taxable_income = income - exemptions - deductions
  if taxable_income <= 0:
    return 0
  return calculate_tax(taxable_income)
```

Test Coverage and White-box Testing:

- **Path Testing**: Test both paths of the if taxable_income <= 0 condition:
  - If taxable income is <= 0, return 0.
  - If taxable income > 0, return tax value calculated from the calculate_tax function.

Test cases:

- Test case 1: income = 400000, exemptions = 50000, deductions = 10000 → Expected output: 0
- Test case 2: income = 600000, exemptions = 50000, deductions = 30000 → Expected output: tax calculated for 520000

- **Code Coverage**: Ensure that the entire code in the function is covered, including the calculation of taxable_income and the call to calculate_tax.

Conclusion

In white-box testing for the income tax calculator:

- **Control Flow Testing** ensures every possible path of execution is tested.
- **Branch Coverage** ensures that each condition in the code (such as tax brackets, exemptions, etc.) is tested with both true and false conditions.
- **Path Coverage** ensures all the paths (sequences of decisions) are tested.
- **Boundary Testing** tests edge cases, ensuring the boundaries of tax brackets, exemptions, and other inputs are correctly handled.

The goal is to systematically verify each unit or module of the tax calculator, ensuring that the program works as intended and handles all possible inputs correctly.