Addis Ababa Institute of Technology
School of Information Technology and
Scientific Computing

Quality Assurance and Software Testing

White Box Testing Techniques
Lab Report

Name: Betselot Kidane
Section: 1
Id: UGR/8473/13

Date:May 23, 2025
Submitted to: Mr. Wondimagegn Desta

White Box Testing Activities Documentation

The source code and test results are compiled in the following GitHub repository:

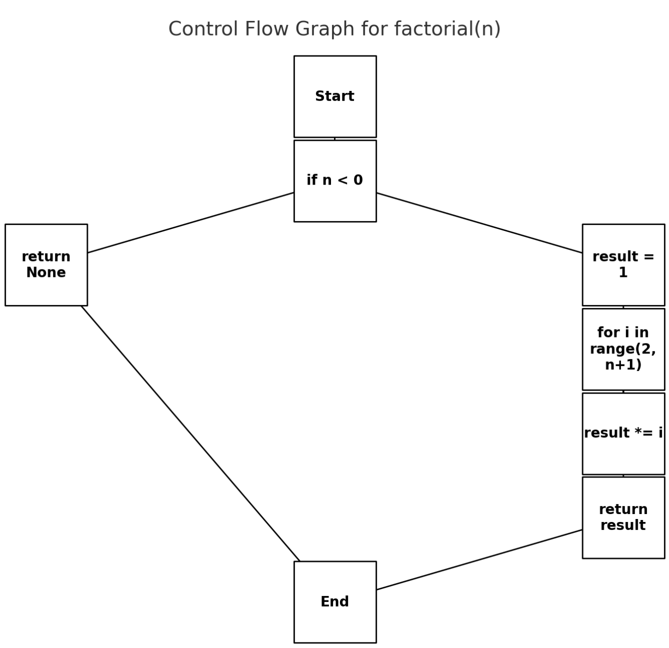
https://github.com/betselot49/White-Box-Testing

Activity 1: Control Flow Graph & Cyclomatic Complexity

For this activity I chose a factorial python function, here is a snippet code

```python
def factorial(n):
    if n < 0:
        return None
    result = 1
    for i in range(2, n + 1):
        result *= i
    return result
```

Control Flow Graph (CFG)



Cyclomatic Complexity Calculation

Using the formula: **C = E − N + 2P**

Description	Value
Number of Nodes (N)	8
Number of Edges (E)	9
Number of Components (P)	1
Cyclomatic Complexity (C)	**3**

Linearly Independent Paths

Path #	Path Description
1	Start → n < 0 → return None → End
2	Start → n >= 0 → initialize result → skip loop → return result → End
3	Start → n >= 0 → loop runs → multiply in loop → return result → End

Test Cases (One per Path)

Test Case	Input	Expected Output	Path Covered	Reason
TC1	-3	None	Path 1	Tests n < 0 condition
TC2	0	1	Path 2	Skips loop, returns base case
TC3	5	120	Path 3	Enters loop and multiplies

Activity 2: Statement, Branch, and Condition Coverage

The assess_risk function evaluates a person's health risk category based on age, smoking habits, and preexisting conditions using nested and compound conditional logic.

```python
def assess_risk(age, smoker, has_preexisting_condition):
    if age < 18:
        return "Underage - Not applicable"

    if smoker and has_preexisting_condition:
        return "High Risk"
    elif smoker or has_preexisting_condition:
        return "Moderate Risk"
    else:
        return "Low Risk"
```

Control Flow & Logical Conditions

Conditions involved:

1. age < 18
2. smoker and has_preexisting_condition
3. smoker or has_preexisting_condition

TC	age	smoker	has_preexisting_condition	age < 18	smoker	has_preexisting_condition	Expected Output
1	16	True	True	True	-	-	Underage - Not applicable

2	30	True	True	False	True	True	High Risk
3	40	True	False	False	True	False	Moderate Risk
4	45	False	True	False	False	True	Moderate Risk
5	50	False	False	False	False	False	Low Risk

Coverage Breakdown

Coverage Type	Covered?	Justification
Statement	Yes	All return statements are executed across test cases
Branch	Yes	Each if, elif, and else path is taken at least once
Condition	Yes	All boolean sub-expressions are evaluated both True and False individually

Breakdown of Condition Coverage:

- age < 18 → True (TC1), False (TC2–TC5)
- smoker → True (TC2, TC3), False (TC4, TC5)
- has_preexisting_condition → True (TC2, TC4), False (TC3, TC5)

Optional Coverage Report with coverage.py

Testing File (test_assess_risk.py):

Below tests for assess risk and test result is provided

```python
def test_underage():
    assert assess_risk(16, True, True) == "Underage - Not applicable"

def test_high_risk():
    assert assess_risk(30, True, True) == "High Risk"

def test_moderate_risk_smoker():
    assert assess_risk(40, True, False) == "Moderate Risk"

def test_moderate_risk_condition():
    assert assess_risk(45, False, True) == "Moderate Risk"

def test_low_risk():
    assert assess_risk(50, False, False) == "Low Risk"
```

Problems Output Debug Console Terminal Ports

```
Installing collected packages: coverage
Successfully installed coverage-7.8.2

[notice] A new release of pip is available: 25.0.1 -> 25.1.1
[notice] To update, run: python.exe -m pip install --upgrade pip
PS D:\Project\SQA> coverage run -m pytest test_assess_risk.py
No module named 'pytest'
PS D:\Project\SQA> d:\Project\SQA\.venv\Scripts\python.exe -m pip install coverage
Collecting coverage
  Using cached coverage-7.8.2-cp313-cp313-win_amd64.whl.metadata (9.1 kB)
Using cached coverage-7.8.2-cp313-cp313-win_amd64.whl (215 kB)
Installing collected packages: coverage
Successfully installed coverage-7.8.2
PS D:\Project\SQA> d:\Project\SQA\.venv\Scripts\python.exe -m pip install coverage
Requirement already satisfied: coverage in d:\project\sqa\.venv\lib\site-packages (7.8.2)
```

The selected function effectively demonstrates complex condition handling. The test cases ensure complete white-box test coverage across statements, decisions, and individual condition outcomes. This fulfills the requirements of Activity 2 comprehensively.

Activity 3: Data Flow Testing

Function Chosen: count_vowels_and_consonants

This function takes a string and counts the number of vowels and consonants.

1. **Python Code (with annotations)**

```python
def count_vowels_and_consonants(text):
    vowels = 'aeiouAEIOU'  # d1: vowels
    v_count = 0            # d2: v_count
    c_count = 0            # d3: c_count
    for char in text:      # p-use: text
        if char.isalpha(): # p-use: char
            if char in vowels:  # p-use: char, vowels
                v_count += 1    # c-use: v_count | d4: v_count updated
            else:
                c_count += 1    # c-use: c_count | d5: c_count updated
    return v_count, c_count     # c-use: v_count, c_count
```

2. Definition, c-use, p-use Table

Variable	Definition	c-use	p-use
vowels	d1	-	L6: char in vowels
v_count	d2, d4	L6, L10	-
c_count	d3, d5	L8, L10	-
text	-	-	L4: for char in text

char	loop var	L6, L8	L5: char.isalpha()

3. DU Paths Diagram in PlantUML

du_path.puml:

```
@startuml
title DU Paths for count_vowels_and_consonants.py

start
:vowels = 'aeiouAEIOU';\n(v_count = 0; c_count = 0);

partition Loop {
    :for char in text;
    if (char.isalpha()?) then (yes)
        if (char in vowels?) then (yes)
            :v_count += 1;
        else (no)
            :c_count += 1;
        endif
    else (no)
    endif
}

:return v_count, c_count;
stop

' DU Paths

note right: Path 1:\nL2 → L4 → L5 → L6\n(vowel case)
note right: Path 2:\nL3 → L4 → L5 → L8\n(consonant case)
note right: Path 3:\nL2 or L3 → L4 → L5 (fails isalpha)\n→ L10 (non-alpha char)
note right: Path 4:\nL4 → L5 → L6/8 → L10 (used in return)
@enduml
```

4. DU Pair Table

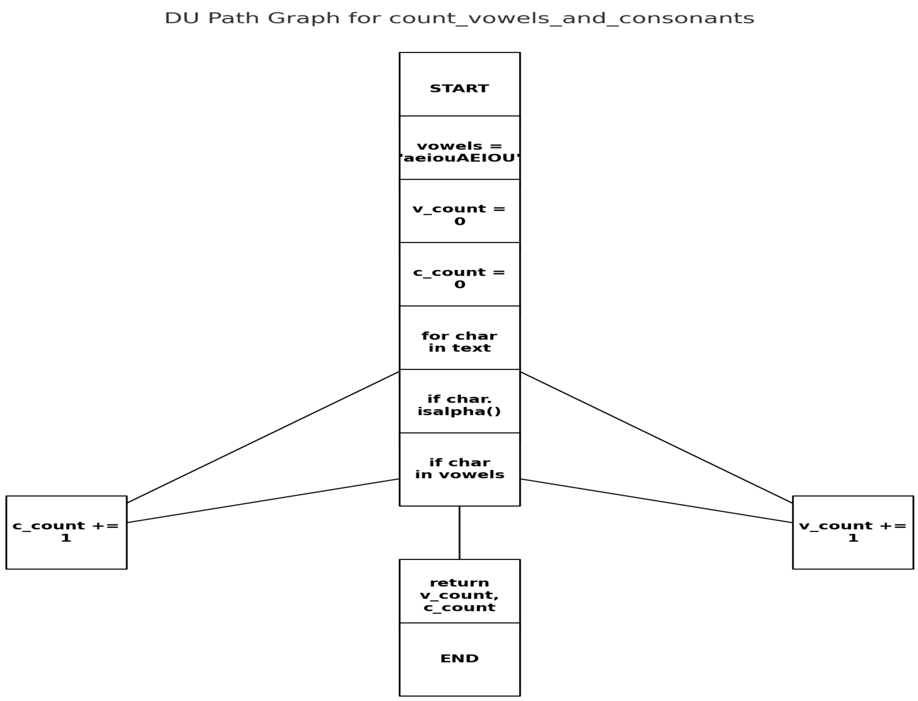
Variable	Definition	Use Location & Type	DU Path	Valid If
vowels	d1	L6 p-use	L1 → L4 → L5 → L6	char is a vowel
v_count	d2	L6 c-use	L2 → L4 → L5 → L6	vowel char updates count

v_count	d4	L10 c-use	L2 → L4 → L5 → L6 → L10	value returned
c_count	d3	L8 c-use	L3 → L4 → L5 → L8	consonant char updates count
c_count	d5	L10 c-use	L3 → L4 → L5 → L8 → L10	value returned
text (arg)	-	L4 p-use	Used in loop	input string used in iteration
char	loop var	L5 p-use, L6/8	Used in if-else	char from text

5. Test Cases (All-defs, DU Pairs, DU Paths)

```python
import unittest
from count_letters import count_vowels_and_consonants

class TestCountVowelsAndConsonants(unittest.TestCase):
    def test_empty_string(self):
        self.assertEqual(count_vowels_and_consonants(""), (0, 0))

    def test_only_vowels(self):
        self.assertEqual(count_vowels_and_consonants("aeiouAEIOU"), (10, 0))

    def test_only_consonants(self):
        self.assertEqual(count_vowels_and_consonants("bcdfgBCDFG"), (0, 10))

    def test_mixed_letters(self):
        self.assertEqual(count_vowels_and_consonants("Hello"), (2, 3))

    def test_mixed_with_non_alpha(self):
        self.assertEqual(count_vowels_and_consonants("He11o!"), (2, 2))  # '1' and '!' ignored
```

DU Path Graph



Test Case	DU Paths Covered	Purpose
""	No use, only return	No defs used
"aeiouAEIOU"	d1 → L6, d2 → L6 → L10	Vowel path only
"bcdfgBCDFG"	d3 → L8 → L10	Consonant path only
"Hello"	All defs and uses triggered	Mixed vowel/consonant

"He11o!"	Includes failed isalpha	Validates predicate guards for robustness

This data flow analysis demonstrates how to methodically identify and test variable lifecycles in a function. Through DU paths, predicate-use checks, and test case mapping, we achieve through white-box coverage for this function.

Activity 4: Mutation Testing

I picked a function that checks if a number is a prime.

```python
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True
```

Test Suite – test_prime_utils.py

```python
import unittest
from prime_utils import is_prime

class TestIsPrime(unittest.TestCase):
    def test_negative(self):
        self.assertFalse(is_prime(-5))

    def test_zero_and_one(self):
        self.assertFalse(is_prime(0))
        self.assertFalse(is_prime(1))

    def test_small_primes(self):
        self.assertTrue(is_prime(2))
        self.assertTrue(is_prime(3))
        self.assertTrue(is_prime(5))

    def test_small_non_primes(self):
        self.assertFalse(is_prime(4))
        self.assertFalse(is_prime(6))
        self.assertFalse(is_prime(9))

    def test_large_prime(self):
        self.assertTrue(is_prime(29))

    def test_large_non_prime(self):
        self.assertFalse(is_prime(100))
```

Mutants Introduced

Here are the **mutants** with a simple change each:

1. **Mutant 1 – Invert <= Check**

```python
def is_prime(n):
    if n > 1:   # Mutation here
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i == 0:
            return False
    return True
```

2. **Mutant 2 – Flip modulo logic**

```python
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        if n % i != 0:   # Mutation here
            return False
    return True
```

3. **Mutant 3 – Remove return False on divide**

```python
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n ** 0.5) + 1):
        pass   # Mutation: Removed check
    return True
```

4. **Mutant 4 – Always return True**

```python
def is_prime(n):
    return True   # Mutation: skip logic
```

5. Mutant 5 – Return False for all

```python
def is_prime(n):
    return False  # Mutation: skip logic
```

Test Results

Mutant	Description	Killed?	Explanation
1	if n <= 1 → if n > 1	Yes	Tests with 0, 1, and negative numbers fail
2	n % i == 0 → n % i != 0	Yes	Primes wrongly rejected, fails test cases
3	Removed loop body	Yes	All numbers return True
4	Always return True	Yes	Non-primes wrongly considered as primes
5	Always return False	Yes	All prime tests fail

Mutation Score

- **Total Mutants**: 5
- **Killed Mutants**: 5
- **Survived Mutants**: 0
- **Mutation Score** = (5 / 5) × 100 = **100%**

Activity 5: JUnit Unit Testing

This utility class provides three simple but practical string-based methods

```java
public class StringUtils {

    public boolean isPalindrome(String input) {
        if (input == null) return false;
        String cleaned = input.replaceAll("[^a-zA-Z]", "").toLowerCase();
        String reversed = new StringBuilder(cleaned).reverse().toString();
        return cleaned.equals(reversed);
    }

    public String reverse(String input) {
        if (input == null) return null;
        return new StringBuilder(input).reverse().toString();
    }

    public int countVowels(String input) {
        if (input == null) return 0;
        return (int) input.toLowerCase().chars()
                .filter(c -> "aeiou".indexOf(c) >= 0)
                .count();
    }
}
```

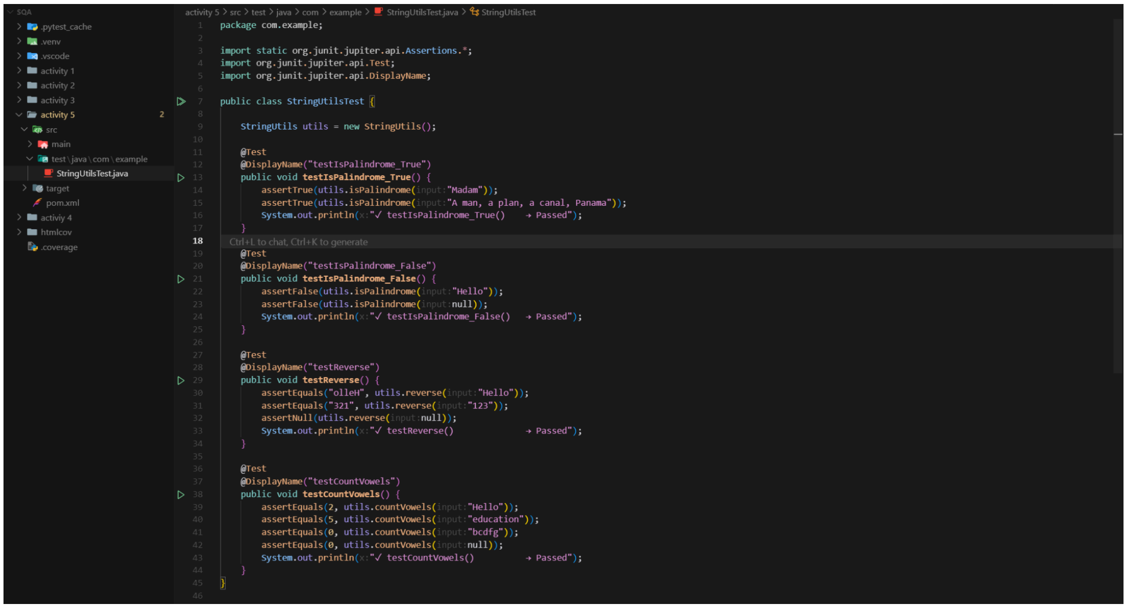
- **isPalindrome(String input)**: Checks if a word or sentence is a palindrome by removing non-letter characters and comparing the original with its reverse.

- **reverse(String input)**: Simply returns the reverse of a given string.

- **countVowels(String input)**: Counts how many vowels (a, e, i, o, u) are present.

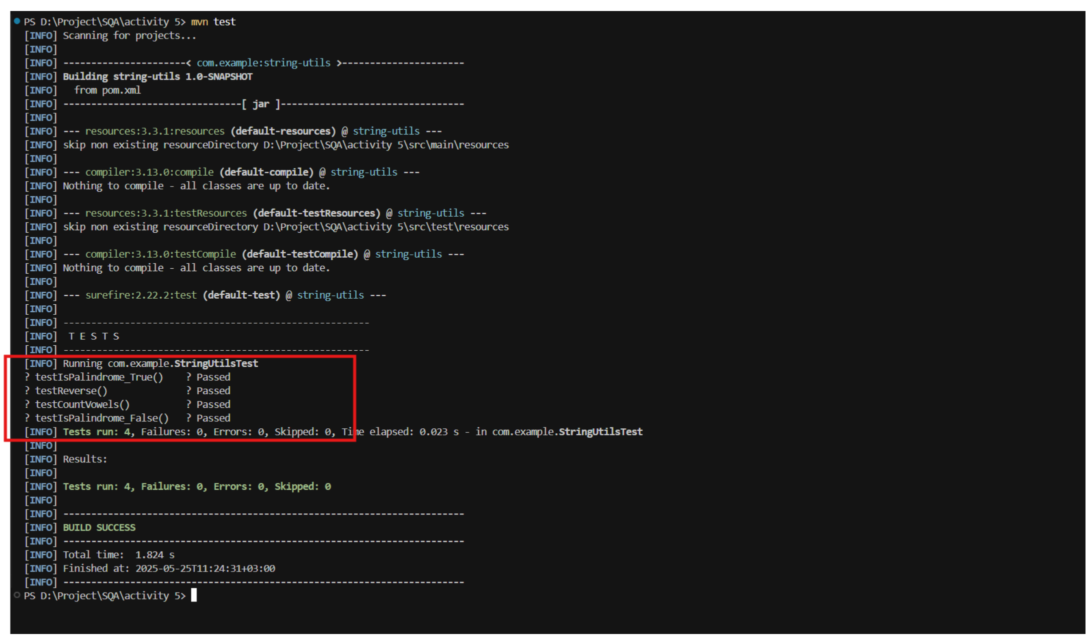
JUnit Test Class

We write several test methods using JUnit 5 to validate our utility class:



- testIsPalindrome_True: Validates known palindromes.
- testIsPalindrome_False: Confirms detection of non-palindromes and null input.
- testReverse: Checks reversal logic, including numeric strings and null case.
- testCountVowels: Tests a variety of strings for vowel counting accuracy.

Test Execution Output



Textual Output Example from Console

[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] --
[INFO] BUILD SUCCESS
[INFO] --

The StringUtils class implements core string utilities with proper null-handling and logic.

The JUnit test class validates each method with both typical and edge-case inputs.

A mix of assertion methods (assertTrue, assertFalse, assertEquals, assertNull) showcases best practices.

All test cases passed, confirms the correctness and robustness of the code