Green University of Bangladesh

Department of Computer Science and Engineering (CSE)
Semester: (Fall, Year: 2023), B.Sc. in CSE (Day)

Implement unit, module, integration, system, black, and white box testing for a project.

Course Title: Software Testing and Quality Assurance
Course Code: CSE-433
Section: D

Students Details

Name	ID
Jesmin Chakma	193002078
Md.Mehedi Hasan	193002106

Submission Date: 18-06-2023
Course Teacher's Name: Md. Noyan Ali

Contents

0.1 Main Java File 2
0.2 Unit Testing Code 2
0.2.1 Unit Testing Explanation 3
0.3 Unit Testing Output 4
0.4 Integration Testing Code 4
0.4.1 Integration Testing Explanation 5
0.5 Integration Testing Output 5
0.6 Module Testing Code 5
0.6.1 Module Testing Explanation 6
0.7 Module Testing Output 7
0.8 Black Box testing code 7
0.8.1 Black Box Testing Explanation 8
0.9 Black Box Testing Output 9
0.10 White Box Testing Code 9
0.10.1 White Box Testing Explanation 10
0.11 White Box Testing Output 10

0.1 Main Java File

```
package com.mycompany.sqaproject;
public class SQAproject {

    public static void main(String[] args) {

    }
}
```

0.2 Unit Testing Code

```


package com.mycompany.sqaproject;

import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.Before;

/**
 *
 * @author HP
 */


public class UnitTesting {
    private Calculator cal;

    @Before
    public void setup() throws Exception{

        cal = new Calculator(4);
    }
    //Test Case
    @Test
    public void AddTest(){

        cal.Add(5);
        assertEquals(9,cal.GetCurrentValue());
        cal.Add(3);
        assertEquals(12,cal.GetCurrentValue());

    }
    //Test Case
    @Test
    public void SubtracTest(){

        cal.Subtract(5);
```

```
38          assertEquals(-1,cal.GetCurrentValue());
39          cal.Subtract(-10);
40          assertEquals(9,cal.GetCurrentValue());
41      }
42      //Test Case
43      @Test
44      public void Mul(){
45
46          cal.Mul(10);
47          assertEquals(40,cal.GetCurrentValue());
48          cal.Mul(10);
49          assertEquals(400,cal.GetCurrentValue());
50      }
51      //Test Case
52      @Test
53      public void Div(){
54
55          cal.Div(2);
56          assertEquals(2,cal.GetCurrentValue());
57          cal.Div(2);
58          assertEquals(1,cal.GetCurrentValue());
59      }
60      //Test Case
61      @Test
62      public void Modulo(){
63
64          cal.ModuloDivision(2);
65          assertEquals(0,cal.GetCurrentValue());
66          cal.ModuloDivision(2);
67          assertEquals(0,cal.GetCurrentValue());
68      }
69
70
71  }
```

0.2.1 Unit Testing Explanation

Unit testing is a software testing technique that focuses on testing individual units, typically functions or methods, in isolation. The purpose is to verify that each unit of code functions correctly and produces the expected output for a given set of inputs.we create a class called UnitTest to test the Calculator class. We annotate each test method with @Test to indicate that it's a unit test. Within each test method, we use various assertion methods from the Assertions class provided by JUnit to check if the actual result matches the expected result.

0.3 Unit Testing Output

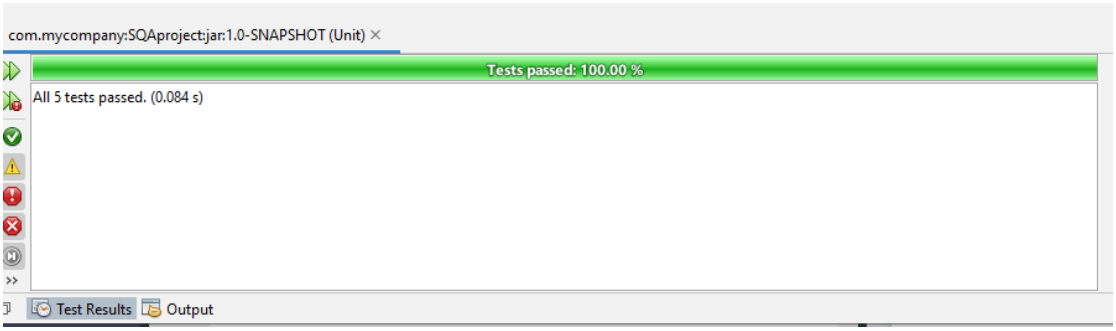

Figure 1: Output

0.4 Integration Testing Code

```
package com.mycompany.sqaproject;
import org.junit.Test;
import static org.junit.Assert.*;
import org.junit.Before;

/**
 *
 * @author HP
 */



public class IntegrationTesting {

    public Calculator cal;

    @Before
    public void setup() throws Exception{
       cal = new Calculator(4);

    }
    //Test Case
    @Test
    public void IntegrationTest(){

        cal.Add(5);
        assertEquals(9,cal.GetCurrentValue());
        cal.Subtract(5);
        assertEquals(4,cal.GetCurrentValue());
        cal.Mul(10);
        assertEquals(40,cal.GetCurrentValue());
```

```
32          cal.Div(2);
33          assertEquals(20,cal.GetCurrentValue());
34          cal.ModuloDivision(2);
35          assertEquals(0,cal.GetCurrentValue());
36
37      }
38
39  }
```

0.4.1 Integration Testing Explanation

Integration testing is a software testing technique that focuses on testing the integration and interaction between multiple components or modules of a system.we create a class called IntegrationTesting to test the integration of the add, subtract, multiply, and divide methods of the Calculator class. The calculator instance is created once for all the tests. Each test method represents a specific integration scenario.

0.5 Integration Testing Output

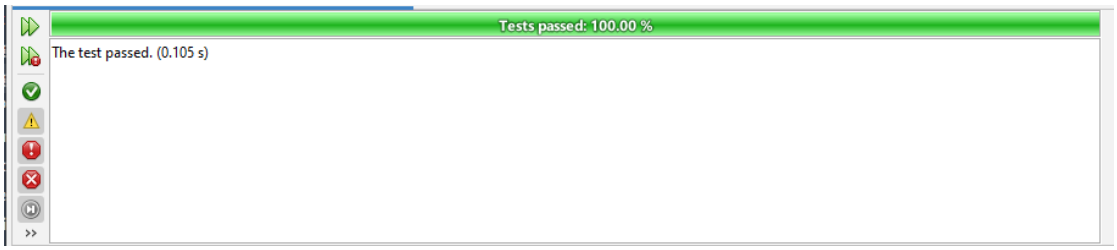

Figure 2: Output

0.6 Module Testing Code

```
1  package com.mycompany.sqaproject;
2  import org.junit.Test;
3  import static org.junit.Assert.*;
4  import org.junit.Before;
5
6
7  /**
8   *
9   * @author HP
10  */
11 public class ModuleTestingTest {
12      public ModuleTesting cal;
13      @Before
```

```
14      public void setup() throws Exception{
15
16          cal = new ModuleTesting();
17      }
18      //Test Case
19      @Test
20      public void ModuleTest(){
21
22          assertEquals(11,cal.Add(5, 6));
23          assertEquals(4,cal.Subtract(10, 6));
24          assertEquals(30,cal.Mul(5, 6));
25          assertEquals(1,cal.Div(6, 6));
26          assertEquals(5,cal.ModuloDivision(5, 6));
27
28      }
29  }
```

0.6.1 Module Testing Explanation

Module testing, also known as component testing or module-level testing, is a software testing technique that focuses on testing individual modules or units of a system in isolation. we create a class called ModuleTestingTest to test the methods of the ModuleTesting class. Each test method represents a specific module or method being tested. Within each test method, we create an instance of the module under test, invoke the relevant method(s), and compare the actual result with the expected result using assertion methods.

0.7 Module Testing Output

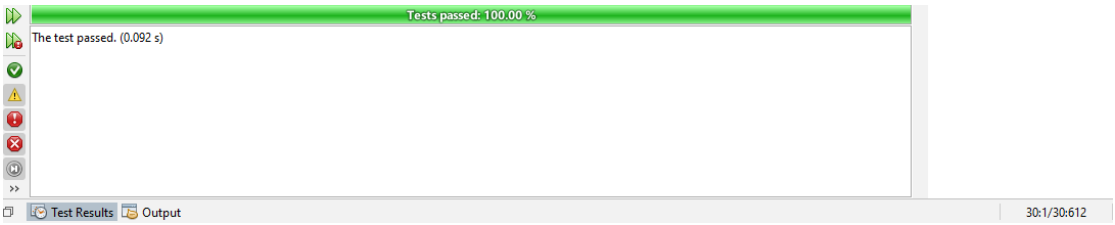

Figure 3: Output

0.8 Black Box testing code

```
1
2  package com.mycompany.sqaproject;
3
4  import org.junit.Test;
5  import static org.junit.Assert.*;
6  import org.junit.Before;
7
8  /**
9   *
10  * @author HP
11  */
12 public class BlackboxTestingTest {
13     public ModuleTesting cal;
14     @Before
15     public void setup() throws Exception{
16
17         cal = new ModuleTesting();
18     }
19     //Test Case
20      @Test
21     public void AddTest(){
22
23         int result = cal.Add(6, 6);
24         assertTrue(result >= 1 && result <= 12);
25
26
27     }
28     //Test Case
29     @Test
30     public void SubtracTest(){
31
32         int result = cal.Subtract(6, 4);
33         assertTrue(result >= 1 && result <= 2);
34
```

```
35      }
36      //Test Case
37      @Test
38      public void Mul(){
39
40          int result = cal.Mul(6, 6);
41          assertTrue(result >= 1 && result <= 36);
42
43      }
44      //Test Case
45      @Test
46      public void Div(){
47
48           int result = cal.Div(36, 6);
49           assertTrue(result >= 1 && result <= 6);
50      }
51      //Test Case
52      @Test
53      public void Modulo(){
54
55           int result = cal.ModuloDivision(5, 6);
56           assertTrue(result >= 1 && result <= 5);
57
58      }
59  }
```

0.8.1 Black Box Testing Explanation

Black-box testing is a software testing technique where the internal workings or implementation details of the system under test are not known or considered. the tests are based on the expected behavior of the Blackboxtesting class. We don't have any knowledge of the internal implementation, but we test the calculator's functionality by providing different inputs and checking the output against the expected results using the assertion methods.

0.9 Black Box Testing Output

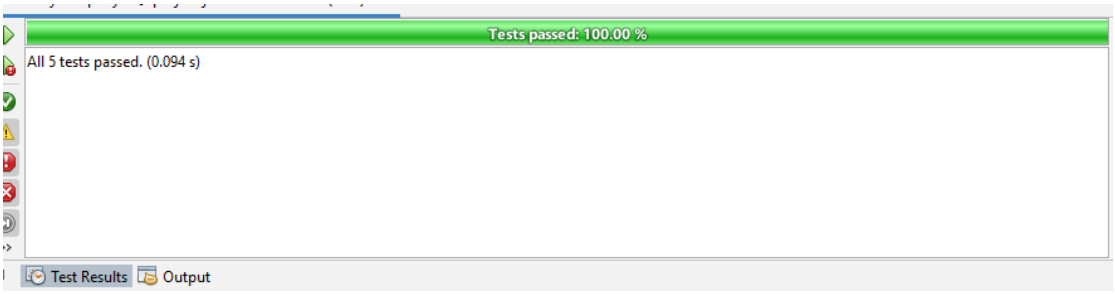

Figure 4: Output

0.10 White Box Testing Code

```
1
2  package com.mycompany.sqaproject;
3
4  import org.junit.Test;
5  import static org.junit.Assert.*;
6  import org.junit.Before;
7
8  /**
9   *
10  * @author HP
11  */
12 public class WhiteBoxTestingTest {
13       public ModuleTesting cal;
14      @Before
15      public void setup() throws Exception{
16
17          cal = new ModuleTesting();
18      }
19      //Test Case
20       @Test
21      public void AddTest(){
22
23          assertEquals(10,cal.Add(5, 5));
24
25      }
26      //Test Case
27      @Test
28      public void SubtracTest(){
29
30         assertEquals(0,cal.Subtract(5, 5));
31
32      }
```

```
    //Test Case
    @Test
    public void Mul(){

        assertEquals(25,cal.Mul(5, 5));

    }
    //Test Case
    @Test
    public void Div(){

         assertEquals(1,cal.Div(5, 5));
    }
    //Test Case
    @Test
    public void Modulo(){

        assertEquals(0,cal.ModuloDivision(5, 5));

    }


}
```

0.10.1 White Box Testing Explanation

White-box testing, also known as structural testing or glass-box testing, is a software testing technique that focuses on examining the internal structure, logic, and code implementation of the system under test.we have access to the internal details of the WhiteBoxTesting class, and we use this knowledge to design test cases that cover different code paths and conditions. We test various operations such as addition, subtraction, multiplication, and division, as well as the modulo division.

0.11 White Box Testing Output

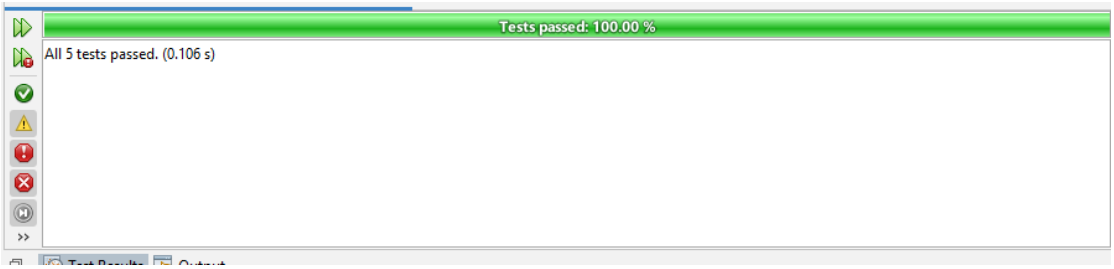

Figure 5: Output