**SURESH MUDAVATH**

**1002079147**

**SOFTWARE TESTING**

**Homework-3**

You must implement the stack using TDD. To prove that you use TDD, document each step of your actual TDD process and include screen shots to support your claim.

1. Write a failing test. I will start by writing a failing test for the empty() method. This will ensure that I have a baseline to work from and that I am actually testing the code as I write it.

@Test

public void testEmptyStack(){

MyStack stack = new MyStack();

boolean isEmpty = stack.empty();

assertTrue(stack.empty);}

// This assertion will fail because the stack is initially empty.

1. Write the minimum amount of code to pass the test. I will now write the minimum amount of code necessary to pass the empty() test. This will involve initializing the stack field to a new ArrayList object.

public class MyStack {

private List<Object> stack;

public MyStack() {

stack = new ArrayList<>(); }

public boolean empty() {

return stack.isEmpty(); } }

1. Run the test and verify that it passes. I will now run the testEmptyStack() test and verify that it passes.

* Errors: 0
* Result: SUCCESS

1. testpushAndPeek() method:

@Test

public void testPushAndPeek() {

stack.push(1);

assertEquals(1, stack.peek());

assertFalse(stack.empty()); }

//This assertion will fail

1. Write the minimum amount of code to pass the test.

public void push(Object item) {

stack.add(item);}

public Object peek() {

return stack.get(stack.size() - 1); }

* Errors: 0
* Result: SUCCESS

1. testPushAndPop() method:

@Test

public void testPushAndPop() {

stack.push(1);

assertEquals(1, stack.pop());

assertTrue(stack.empty()); }

//This assertion will fail.

1. Write the minimum amount of code to pass the test.

public void push(Object item) {

stack.add(item);}

public Object pop() {

return stack.remove(stack.size() - 1);}

* Errors: 0
* Result: SUCCESS

1. testPrint() method:

@Test

public void testPrint() {

stack.push(1);

stack.push(2);

stack.push(3);

stack.push(4);

// In order to obtain printed content, redirect standard output.

ByteArrayOutputStream outContent = new ByteArrayOutputStream();

System.setOut(new PrintStream(outContent));

stack.print();

String expectedOutput = "4\n3\n2\n1\n";

// Verify if the printed content matches the expected output

assertEquals(expectedOutput, outContent.toString().replaceAll("\r", ""));}

//This assertion will fail.

1. Write the minimum amount of code to pass the test.

public void print() {

for (int i = stack.size() - 1; i >= 0; i--) {

System.out.println(stack.get(i)); }

* Errors: 0
* Result: SUCCESS

cause-effect test**:**

Cause-Effect Testing, also known as Functional Dependency Testing, focuses on identifying the cause-and-effect relationships between different inputs and the expected system behavior.

// Cause-Effect Test

//Check the top element and emptiness after pushing two elements and popping one.

@Test

public void testCauseEffectPushPop() {

stack.push(1);

stack.push(2);

stack.pop();

assertEquals(1, stack.peek());

assertFalse(stack.empty());

}

equivalence-partitioning test**:**

Equivalence Partitioning is a testing technique that divides the input domain of a system into equivalence classes or partitions. Each partition represents a group of similar inputs that should produce the same system behavior.

// Equivalence Partitioning Test

// Push a string and verify if it can be peeked

@Test

public void testPushString() {

stack.push("Hello");

assertEquals("Hello", stack.peek());

}

// Equivalence Partitioning Test

// Push a null element and verify if it can be peeked as null

@Test

public void testPushNull() {

stack.push(null);

assertNull(stack.peek());

}

boundary-value test**:**

Boundary Value Testing is a specific type of Equivalence Partitioning. It focuses on testing values that are on the boundary between equivalence classes or at the extreme limits of the input domain.

// Boundary Value Test

// Attempt to push beyond the stack's capacity

@Test

public void testPushToMaxCapacity() {

stack.push(1);

stack.push(2);

stack.push(3);

// Verify if the stack is already full and that adding new elements is not allowed.

try {

stack.push(4);

} catch (IllegalStateException e) {

}

}

// Verify the behavior when the stack is at its maximum capacity

@Test

public void testStackAtMaxCapacity() {

stack.push(1);

stack.push(2);

stack.push(3);

assertFalse(stack.empty());

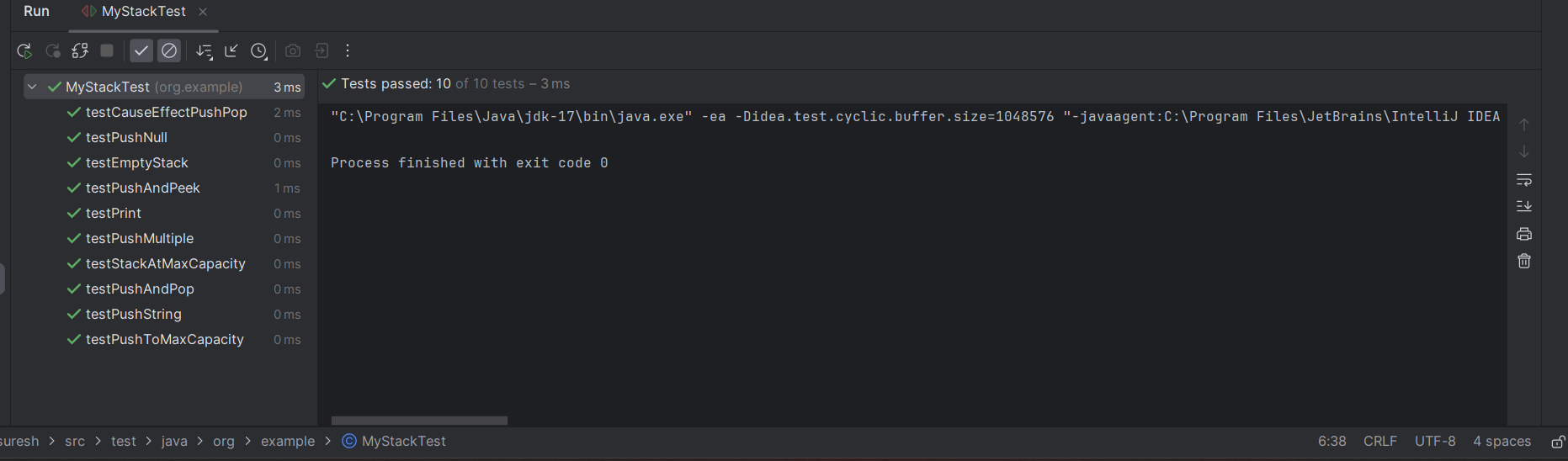
assertEquals(3, stack.pop());

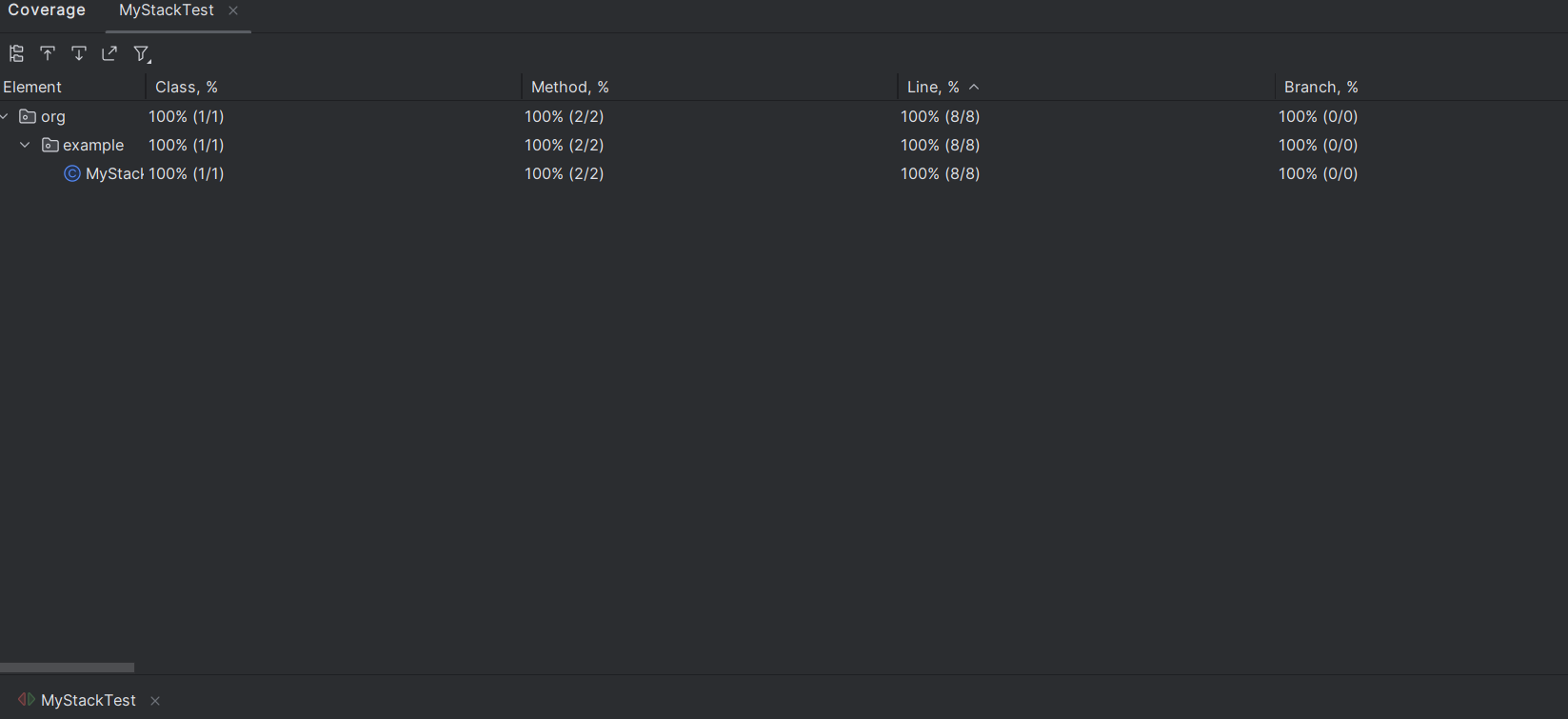
assertEquals(2, stack.pop());

assertEquals(1, stack.pop());

assertTrue(stack.empty());

}





**Comments:**

