MEKELLE UNIVERSITY

EIT-M

SCHOOL OF COMPUTING

DEPARTMENT OF SOFTWARE ENGNEERING

COURSE TITLE: SOFTWARE TOOLS AND PRACTICES

Team members :		
FULL NAME		**ID**
HAFTOM	ABRHA	EITM/UR158030/11
MULUNEH	G/MEDHIN	EITM/UR157880/11
HAFTAMU	REDAE	EITM/UR158047/11
HALEFOM	HADUSH	EITM/UR129573/10
RAHEL	G/HAWARIA	EITM/UR156145/11

Submitted To : inst. Haftamu

Submission date:17/05/2024

Test-Driven Development (TDD) is a software development methodology that emphasizes writing tests before writing the actual code. The process of TDD involves the following steps:

1. Write a Test: The developer starts by writing a test case that defines the desired behavior of the code they are about to write. This test case typically fails initially, as the code has not yet been implemented.

We have implemented our program to perform LinkedList operations. The test cases will check if the linked list operations are working as expected. The test cases are defined as follow :

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

public class LinkedListTest {

    @Test
    public void testAdd() {
        LinkedList list = new LinkedList();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        list.add(5);
        list.printList();
    }

    @Test
    public void testRemove() {
        LinkedList list = new LinkedList();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        list.add(5);
        list.remove(3);
        System.out.println("List after remove");
        list.printList();
    }

    @Test
    public void testEmptyList() {
        // This checks if the method works as we need
```

```
        LinkedList list = new LinkedList();
        assertTrue(list.isEmpty());
        assertEquals(0, list.size());
    }

    @Test
    public void testContains() {
        // This method tests if the contains() works as desired
        LinkedList list = new LinkedList();
        list.add(1);
        list.add(2);
        list.add(3);
        list.add(4);
        list.add(5);

        assertTrue(list.contains(3));
        assertFalse(list.contains(6));
    }

    // TDD aproach is wirting tescases befor implementing of the algorithms.
    @Test
    public void testSize() {
        LinkedList list = new LinkedList();
        assertEquals(0, list.size()); // Test for an empty list

        list.add(1);
        assertEquals(1, list.size());

        list.add(2);
        list.add(3);

        assertEquals(3, list.size());

        list.remove(2);
        assertEquals(2, list.size());
    }
    // some comments
}
```

2. Write the Code: After writing the test case, the developer writes the minimum amount of code necessary to make the test pass.

The following code implements a basic LinkedList class in Java, complete with methods to add, remove, and search for element, as well as to print the list and check its size:

```
public class LinkedList {
    ListNode head;
    int size;
    public void add(int val) {
        ListNode newNode = new ListNode(val);
        if (head == null) {
            head = newNode;
        } else {
            ListNode current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }
        size++; // Increment size when adding a new element

    }
    public void remove(int val) {
        if (head == null) {
            return;
        }

        if (head.val == val) {
            head = head.next;
            size--; // Decrement size when removing an element
            return;
        }

        ListNode prev = head;
        ListNode current = head.next;
        while (current != null) {
            if (current.val == val) {
                prev.next = current.next;
                size--; // Decrement size when removing an element
                return;
            }
            prev = current;
            current = current.next;
        }
```

```
    }
    public boolean isEmpty() {
        return head == null;
    }
    public boolean contains(int val) {
        ListNode current = head;
        while (current != null) {
            if (current.val == val) {
                return true;
            }
            current = current.next;
        }
        return false;
    }
    public int size() {
        return size;
    }
    public void printList() {
        ListNode current = head;
        while (current != null) {
            System.out.print(current.val + " ");
            current = current.next;
        }
        System.out.println();
    }


}
```

And ListNode Class

```
class ListNode {
    int val;
    ListNode next;

    ListNode(int val) {
        this.val = val;

    }
}
```

3. Refactor the Code: Once the test passes, the developer refactors the code to improve its design, maintainability, and performance, while ensuring that the test still passes.

4. Repeat: The developer then repeats the process, writing a new test case, implementing the code to make the test pass, and refactoring the code as necessary.

The key principles of TDD are:

1. Red-Green-Refactor: The developer writes a test that fails (red), then writes the minimum amount of code to make the test pass (green), and finally refactors the code to improve its quality.

2. Small Increments: TDD encourages developers to write small, incremental changes to the codebase, rather than large, complex changes. First, we have written one test method, the testAdd() method, and then the testRemove() method, and so on, step by step.

3. Automated Testing: TDD relies heavily on automated testing, which helps ensure that the code works as expected and that changes to the codebase do not break existing functionality.

The benefits of TDD include improved code quality, better design, and reduced debugging time, as the tests help catch bugs early in the development process. Additionally, TDD promotes a more modular and testable codebase, which can make it easier to maintain and extend over time.