

Test Driven Development

Ruvani Jayaweera
(PG dip) (University of Moratuwa) ,
((BSc) hons (University of Moratuwa)



Learning Objectives

Be able to understand about the Test Driven Development.



Outline

- What is Test Driven Development?
- Steps in TDD



What is Test Driven Development?

- Test Driven Development (TDD) is a software development approach where tests are created before writing the actual code.
- The TDD process involves three key steps, commonly referred to as "Red-Green-Refactor":



Steps in TDD

1. **Write a Test:** Before writing the code, write a test that defines a function or improves the new feature. At this point, the test will fail because the feature is not yet implemented.
2. **Run the Test:** Run the test to see it fail. This ensures that the test is working correctly and that it fails for the right reason.
3. **Write the Code:** Write the minimum amount of code necessary to make the test pass.



4. Run the Test Again: Run the test again to see it pass.

If the test passes, it means the code meets the requirement of the test.

5. Refactor the Code: Refactor the code to improve its structure, readability, and efficiency. Run the test again to ensure it still passes after refactoring.

6. Repeat: Repeat the cycle for the next feature or function.



Write a Test

Let's take an example to implement a class for a calculator.

Step 1: First, we write tests for our calculator methods before we implement them. We'll use JUnit as our testing framework.



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

```
public class CalculatorTest {
```

```
    @Test
```

```
    public void testAddition() {
        assertEquals(5, Calculator.add(2, 3));
        assertEquals(-1, Calculator.add(-2, 1));
    }
```

```
    @Test
```

```
    public void testSubtraction() {
        assertEquals(1, Calculator.subtract(3, 2));
        assertEquals(-3, Calculator.subtract(-2, 1));
    }
```

```
    @Test
```

```
    public void testMultiplication() {
        assertEquals(6, Calculator.multiply(2, 3));
        assertEquals(-2, Calculator.multiply(-1, 2));
    }
```

```
    @Test(expected = ArithmeticException.class)
```

```
    public void testDivision() {
        assertEquals(2, Calculator.divide(6, 3));
        Calculator.divide(1, 0); // This should throw ArithmeticException
    }
```

```
}
```


Step 2: Run the Test and See It Fail

Finished after 0.014 seconds

Runs: 4/4 Errors: 4 Failures: 0

✓ CalculatorTest [Runner: JUnit 4] (0.001 s)

✗ testMultiplication (0.000 s)

✗ testAddition (0.000 s)

✗ testDivision (0.000 s)

✗ testSubtraction (0.001 s)


```
3 import static org.junit.Assert.assertEquals;
4 import org.junit.Test;
5
6
7 public class CalculatorTest {
8
9     @Test
10     public void testAddition() {
11         assertEquals(5, Calculator.add(2, 3));
12         assertEquals(-1, Calculator.add(-2, 1));
13     }
14
15     @Test
16     public void testSubtraction() {
17         assertEquals(1, Calculator.subtract(3, 2));
18         assertEquals(-3, Calculator.subtract(-2, 1));
19     }
20
21     @Test
22     public void testMultiplication() {
23         assertEquals(6, Calculator.multiply(2, 3));
24         assertEquals(-2, Calculator.multiply(-1, 2));
25     }
26
27     @Test(expected = ArithmeticException.class)
28     public void testDivision() {
29         assertEquals(2, Calculator.divide(6, 3));
30         Calculator.divide(1, 0); // This should throw ArithmeticException
31     }
32 }
33
34
```



Since we haven't implemented the Calculator methods yet, running this test will result in failures.

Step 3: Implement the Functions

Now, we'll implement the Calculator methods.



```
1
2 public class Calculator {
3
4     public static int add(int a, int b) {
5         return a + b;
6     }
7
8     public static int subtract(int a, int b) {
9         return a - b;
10    }
11
12    public static int multiply(int a, int b) {
13        return a * b;
14    }
15
16    public static int divide(int a, int b) {
17        if (b == 0) {
18            throw new ArithmeticException("Division by zero");
19        }
20        return a / b;
21    }
22 }
```



Step 4: Run the Test Again

After implementing the methods, we run the test again to see if it passes.



Finished after 0.01 seconds

Runs: 4/4  Errors: 0  Failures: 0



✓  CalculatorTest [Runner: JUnit 4] (0.000 s)

✓  testMultiplication (0.000 s)

✓  testAddition (0.000 s)

✓  testDivision (0.000 s)

✓  testSubtraction (0.000 s)



Step 5: Refactor

Now that the tests pass, we can refactor our code if necessary. In this simple example, there might not be much to refactor, but in more complex scenarios, you might find opportunities to improve your implementation.