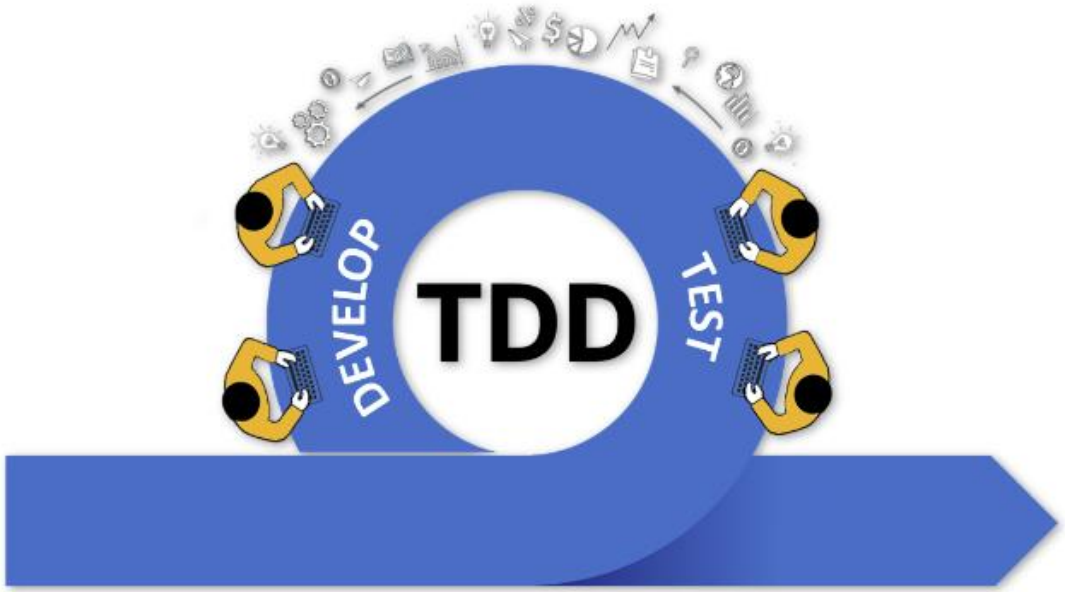
The Power of Test-Driven Development

…Test First, Code Later



Matt Webb – 23/03/2023

What is Test Driven Development (TDD)

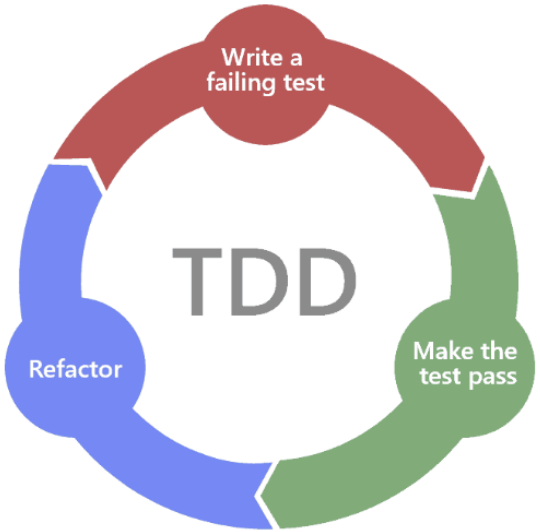
1. Its a discipline based on a set of 3 rules
2. Based on writing automated tests before writing code
3. Helps to reduce bugs
4. Testing behaviour not implementation
5. Reduces the fear in changing/cleaning code

Test Driven Development - The 3 Rules

1. You are not allowed to write any production code until you have first written a failing test

2. You're not allowed to write more of a unit test than is sufficient to fail (and not compiling is failing)

3. You're not allowed to write more production code than is sufficient to pass the currently failing test



In addition to these 3 rules, complete the process by refactoring your code to keep things clean, simple, easy to read and to prevent code smells.

Write A Failing Test - Example

i. Make the test small – it should only test 1 thing!

ii. Use Arrange, Act and Assert blocks as the layout

iii. Write the test in reverse order with the Assert first, then the Action on the system under test and then arrange any dependencies

iv. Name the test so that the class name and the method read as a sentence

v. Ensure the test is failing for the right reason

The test shown would fail as ‘Calculator’ doesn’t exist in the production code yet.

Test Code

```
0 references
public class CalculatorShould
{
    [Test]
    | 0 references
    public void Add_Two_Numbers_Together()
    {
        // Arrange
        var calculator = new Calculator();

        // Act
        var result = calculator.Add(1, 2);

        //Assert
        Assert.AreEqual(3, result);
    }
}
```

Making The Test Pass - Example

Production Code

- Fake It

Quickest way to make test pass, return expected result

```
2 references
public class Calculator
{
    0 references
    public int Add(int firstNumber, int secondNumber)
    {
        return 3;
    }
}
```

- Obvious Implementation

Remove the fake and put the smallest logical code to make tests pass

```
2 references
public class Calculator
{
    0 references
    public int Add(int firstNumber, int secondNumber)
    {
        return firstNumber + secondNumber;
    }
}
```

Triangulation

- **Is the practice of writing a new test which is more specific to force the code to be more generic and include more functionality**

```
0 references
public class CalculatorShould
{
    [Test]
    0 references
    public void Add_Two_Numbers_Together()
    {
        // Arrange
        var calculator = new Calculator();

        // Act
        var result = calculator.Add(1, 2);

        //Assert
        Assert.AreEqual(3, result);
    }
}
```

```
0 references
public class CalculatorShould
{
    [Test]
    0 references
    public void Adds_An_Array_Of_Numbers_Together()
    {
        // Arrange
        var calculator = new Calculator();
        var numbers = new int[] {1, 3, 5};

        // Act
        var result = calculator.Add(numbers);

        //Assert
        Assert.AreEqual(9, result);
    }
}
```

…Don't be afraid to Delete redundant tests!

```
1 reference
public class Calculator
{
    1 reference | 1/1 passing
    public double Add(int[] numbers)
    {
        return numbers.Sum();
    }
}
```

The Benefits

1. Everything passed in the last few minutes
2. Increased confidence in the codebase
3. Better documentation
4. Makes Production Code Testable (Decoupled)
5. Debugging is removed from the process (well almost)

TDD != Totally Dependent on Debugging

…which was how I used to roll!

And Finally……My Thoughts on TDD

1. Resistance To Change
2. Looking too far ahead – Small iterations coax out implementations
3. Patterns start emerging – Promotes refactoring
4. Transformation Priority Premise & Object Calisthenics

Questions?