Test Driven Development – Outline for Lesson Plan

The "test" is part of "spec"ing (Establishing the specification)

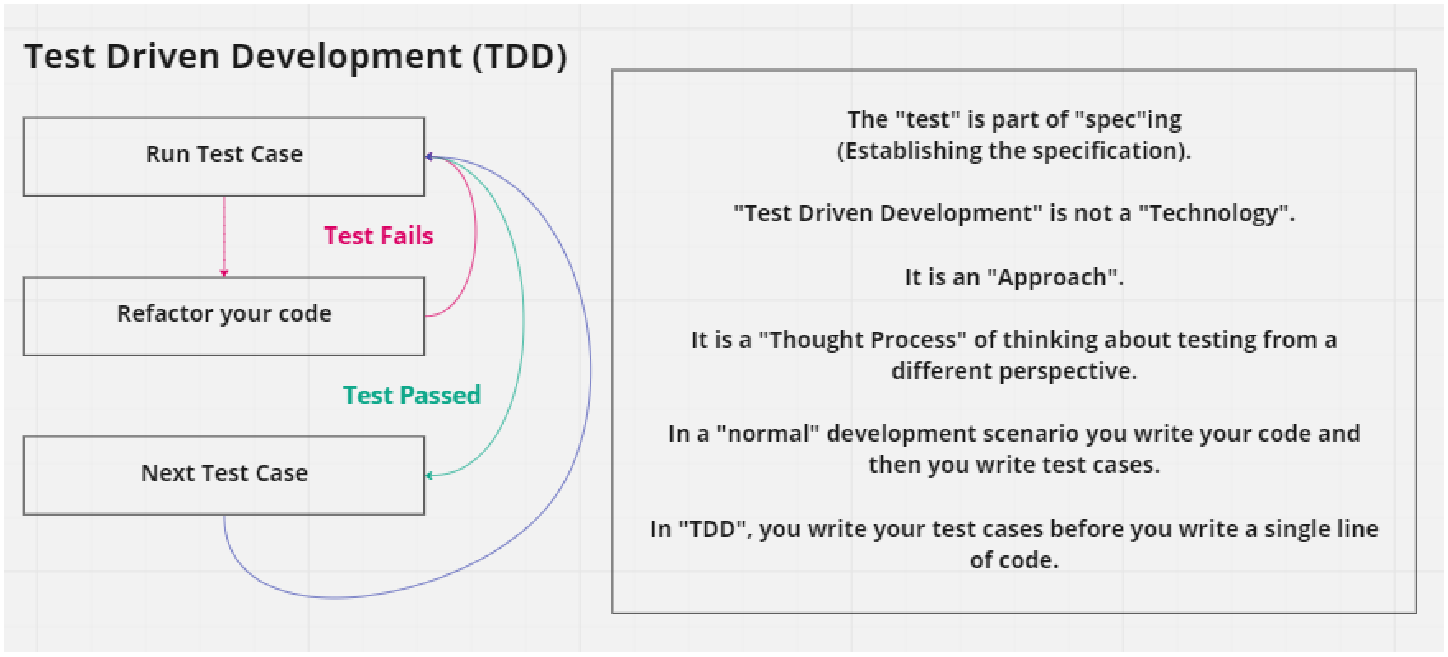
"Test Driven Development" is not a "Technology"

It is an "Approach"

It is a "Thought Process" of thinking about testing from a different perspective

In a "normal" development scenario you write your code and then you write test cases

In "TDD", you write your test cases before you write a single line of app code



The "test" is part of "spec"ing (Establishing the specification).

"Test Driven Development" is not a "Technology".

It is an "Approach".

It is a "Thought Process" of thinking about testing from a different perspective.

In a "normal" development scenario you write your code and then you write test cases.

In "TDD", you write your test cases before you write a single line of code.

Libraries to use in C# Unit Tests

Xunit

Xunit.runner.visualstudio

Libraries to use in React Unit Tests

React Testing Library

React Testing Library builds on top of DOM Testing Library by adding APIs for working with React components.

Projects created with Create React App have out of the box support for **React Testing Library**. If that is not the case, you can add it via npm like so:

```
npm install --save-dev @testing-library/react
```

and you can also find it at:

React Testing Library on GitHub

Mock Service Worker
(https://mswjs.io)

API mocking of the next generation

Mock by intercepting requests on the network level. Seamlessly reuse the same mock definition for testing, development, and debugging.

Scenarios

Arrange

- Setup anything needed to run the test.
  (i.e. "Mocking")

Act

- Execute code on the system under test project and within Visual Studio's "Test Explorer", as well as Visual Studio Code.

Assert

- Making sure that it behaves the way we expect it to behave
  (Usually only ONE Assert, but may have more)

Example

```
namespace TestProject1
{
    using ConsoleApp1;

    public class UnitTest1
    {
        [Fact] // "Decorator"
        public void Should_Add_Two_Numbers()
        {
            // Arrange

            int addNum1 = 5;
            int addNum2 = 10;
            var sum = new Calculator();

            // Act

            int result = sum.Add(addNum1, addNum2);

            // Assert

            Assert.Equal(15, result);

        }

        [Fact] // "Decorator"
        public void Should_Divide_Two_Numbers()
        {
            // Arrange

            double divNum1 = 5;
            double divNum2 = 10;
            var division = new Calculator();

            // Act

            double result = division.Divide(divNum1, divNum2);

            // Assert

            Assert.Equal(.5, result);

        }
    }
}
```

```
namespace ConsoleApp1
{
   public class Calculator
   {
      public int Add(int num1, int num2)
      {
         return num1 + num2;
      }

      public double Divide(double num1, double num2)
      {
         return num1 / num2;
      }
   }
}
```

More Examples/Exercises and Interaction with the Class

[To be developed for (and perhaps in conjunction with) the class]

The following is from the website: https://www.agilealliance.org/

Definition

“Test-driven development” refers to a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit tests) and design (in the form of refactoring).
It can be succinctly described by the following set of rules:

- write a “single” unit test describing an aspect of the program
- run the test, which should fail because the program lacks that feature
- write “just enough” code, the simplest possible, to make the test pass
- “refactor” the code until it conforms to the simplicity criteria
- repeat, “accumulating” unit tests over time

Expected Benefits

- many teams report significant reductions in defect rates, at the cost of a moderate increase in initial development effort
- the same teams tend to report that these overheads are more than offset by a reduction in effort in projects’ final phases
- although empirical research has so far failed to confirm this, veteran practitioners report that TDD leads to improved design qualities in the code, and more generally a higher degree of “internal” or technical quality, for instance improving the metrics of cohesion and coupling

Common Pitfalls

Typical individual mistakes include:

- forgetting to run tests frequently
- writing too many tests at once
- writing tests that are too large or coarse-grained
- writing overly trivial tests, for instance omitting assertions
- writing tests for trivial code, for instance accessors

Typical team pitfalls include:

- partial adoption – only a few developers on the team use TDD
- poor maintenance of the test suite – most commonly leading to a test suite with a prohibitively long running time
- abandoned test suite (i.e. seldom or never run) – sometimes as a result of poor maintenance, sometimes as a result of team turnover