Vimal Krishnan

Test driven development (TDD)

# Table of Contents

## Introduction

## Cons and Pros of TDD

## Frameworks

## Sample Code

## Question and answers

# Introduction

Test driven development aka TDD is a process of developing software application with repetition of test cases

In this approach, developer will initially focus on writing the test cases for desired functionality even before starting the actual development.

During each phases of development, this test cases will be constantly used for verifying the functionality

## Cons and Pros of TDD

Let us quickly go through some of cos and pros of TDD

* Pros
  + Because you are writing small tests at a time, it forces your code to be more modular (otherwise they’d be hard to test against). TDD helps you learn, understand, and internalise the key principles of good modular design.
  + TDD also forces good architecture. To make your code unit-testable, it must be properly modularized. Writing the tests first, various architectural problems tend to surface earlier.
  + Documents your code better than documentation (it doesn’t go out of date since you’re running it all the time).
  + Makes code easier to maintain and refactor. TDD helps to provide clarity during the implementation process and provides a safety-net when you want to refactor the code you’ve just written.
  + Makes collaboration easier and more efficient, team members can edit each others code with confidence because the tests will inform them if the changes are making the code behave in unexpected ways.
* Cons
* The test suite itself must be maintained; tests may not be completely deterministic (i.e. reliant on external dependencies).
* The tests may be hard to write, esp. beyond the unit testing level.
* Initially, it slows down development; for rapidly iterative startup environments the implementation code may not be ready for some time due to spending time writing tests first. (But in the long run, it speeds up development)
* Like any programming, there is a big difference between doing it and doing it well.  Writing good unit tests is an art form. This aspect of TDD is often not discussed, many managers tend to focus on metrics like code coverage; those metrics tell you nothing about the *quality* of the unit tests.

## Frameworks

Though there are many frameworks available, we generally use following frameworks for .net

* Nunit
* Xunit
* Fluent Assertions
* Moq

## Sample Code explanation

This sample code is intended to develop application for education institutes. As we know there many domains falls under education such as school, college, professional college and so on.

This piece of code intended to develop a registration for new school admission

Since this involves data base system, we have classified our test cases into 2 for efficient execution

* + L0- This includes those test cases which are filling within the boundary of application. Generally, we write L0 against controller and business services. Considering the sample code, we have tests for controller as well as business classes.
  + L1-This includes those test cases which has external dependency such as DB, Service bus so on. Generally, we write L1 test cases against DB services

Let us take an simple eg for controller tests.

[TestMethod]

public async Task SchoolControllerTestL0\_Register\_Success()

{

// Assemble

\_mockSchoolBusinessService.Setup(m => m.Resgister(It.IsAny<RegistrationRequest>())).Returns(Task.FromResult(true));

\_controller = new SchoolController(\_mockSchoolBusinessService.Object, \_mockLoggingService.Object);

// Act

var res = await \_controller.Register(It.IsAny<Guid>(), new RegistrationModel());

//Assert

res.Should().NotBeNull();

res.Should().BeAssignableTo(typeof(AcceptedResult));

}

Here we are asserting that, whenever we send this particular object to the service, we are expecting successful status code.

## Questions and Answers

Please feel to post your questions if any