Learning Test Driven Development

Gaurav Aroraa,

Introduction

In these days, Test Driven Development (TDD) is one of the most growing things in the technical world. Most of us are following Agile methodology where we would like to test our code within code.

Why We Need to Learn TDD?

It's a real scenario based questions. Lately, I spoke in an engineering college at Noida, India and I have been asked numerous similar questions. Many read books, blogs and some great articles of great authors but they still in doubt why we should work with Test Driven Development? Some were comparing TDD with their Unit testing all pushed me to answer everything. Here, I am sharing my views, what I had talked with all those guys?

- First of all, do not mix Unit Testing and TDD.
- TDD is just like you are writing and testing your code at the same time.
- It is something like testing a code from within a code.
- TDD makes sure that the written code is tested and refactored.

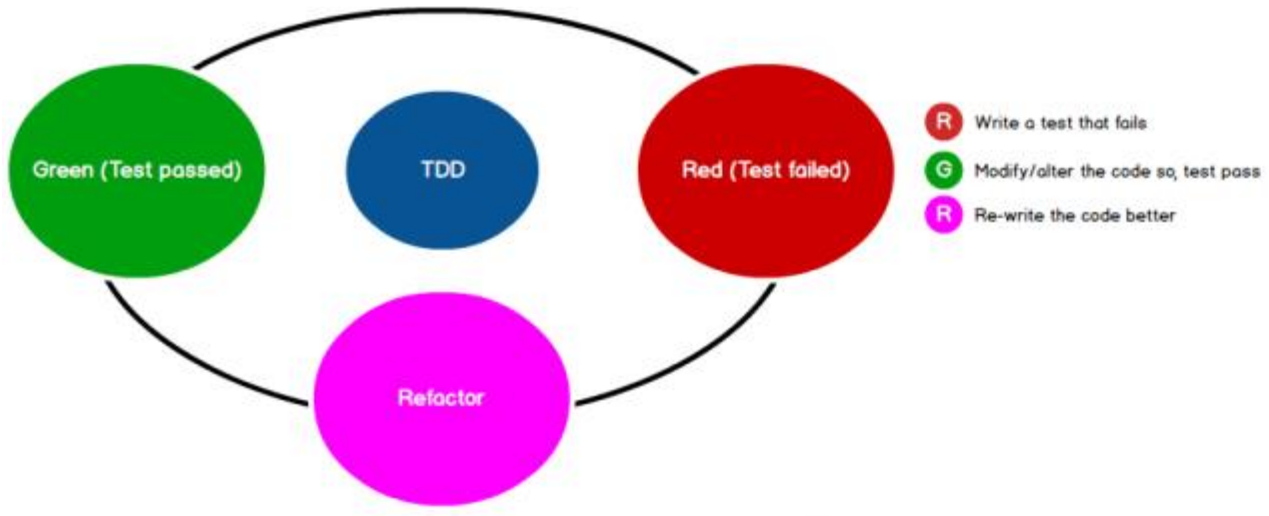
Now, the question arises, why we even need to perform or learn TDD? For this, just think of a scenario where a developer did complete unit testing and the Quality Assurance guys did their job and marked the deliverables ready for production. What happened if the other developer's code/changes break the first developer's code? Now, here we can think about TDD. As it is self-explanatory, Test Driven Development means what we are writing, we are testing. In this way, if someone broke your code, then your test(s) will get failed and you can check the broken area. This is just a one aspect, TDD is a huge one.

So, the answer is simple, if you want to make your code good, follow TDD.

What is TDD?

As per wiki, TDD is defined as:

"Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards"


Green (Test passed)
TDD
Red (Test failed)
Refactor
R Write a test that fails
G Modify/alter the code so, test pass
R Re-write the code better

Test Driven Development (TDD) : Red, Green and Refactoring

TDD has three stages called Red Green Refactor (RGR).

- `Red` – First write test that fails.
- `Green` – Modify/alter code so, test pass.
- `Refactor`- Re-write the code better.

To understand, let’s take an example of FizzBuzz – a very simple code. To start developing code, we need to create a Unit Test project, here I am using NUnit Test Framework, you can use as per your choice:

- Launch Visual Studio
- Create New Project (C# library project) by pressing ctrl + shift + N
- Named it as per your choice, I named it `‘TDD-Katas-project’`
- Add two classes named as `‘FizzBuzz.cs’` and `‘TestFizzBuzz’`
- Add NUnit support to your project: add nugget package of NUnit using either Console Manager or Nuget UI Dialog.

We are ready to get start. Get back to description of our program code, first say ‘Write a program that prints the numbers from `1 to 100`’.

At very first instance, we can write a simple snippet which just prints numbers between `1 - 100`, so, add following snippet to *FizzBuzz.cs*:

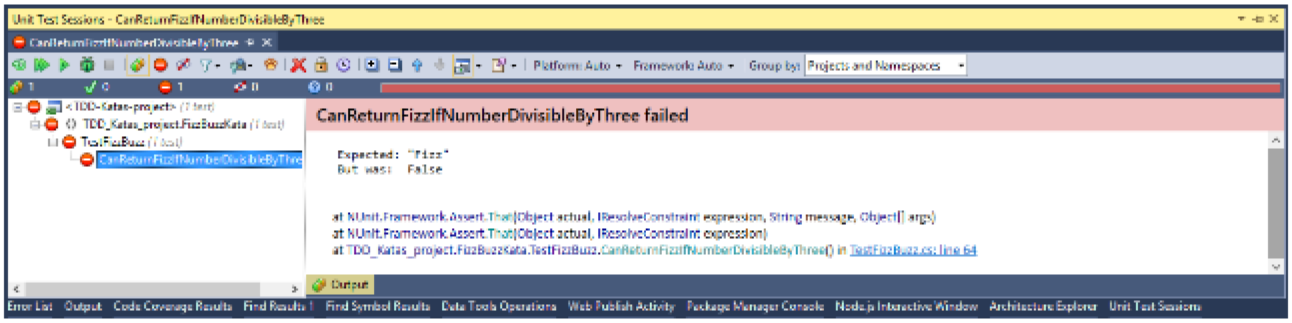
```
public string PrintNumber()
        {
            var number = string.Empty;
            for (var i = 1; i < 100; i++)
                number += " " + i;

            return number.Trim();
        }
```

Let's write a test for this we need to write a `Fizz` if a number is divisible by 3. Add the following test in Test *FizzBuzz.cs* file:

```
[Test]
        public void CanReturnFizzIfNumberDivisibleByThree()
        {
            var actualResult = FizzBuzz.PrintNumber();
            Assert.True(actualResult.Contains("Fizz"));
        }
```

Run the above test from within Visual Studio, I am using Resharper, so, I get the results in Unit Test window as:


Unit Test Sessions - CanReturnFizzIfNumberDivisibleByThree
CanReturnFizzIfNumberDivisibleByThree
Platform: Auto
Framework: Auto
Group by: Projects and Namespaces
1
0
1
0
0
<TDD-Katas-project> (1 test)
TDD_Katas_project.FizzBuzzKata (1 test)
TestFizzBuzz (1 test)
CanReturnFizzIfNumberDivisibleByThree
Output
Error List
Output
Code Coverage Results
Find Results 1
Find Symbol Results
Data Tools Operations
Web Publish Activity
Package Manager Console
Node.js Interactive Window
Architecture Explorer
Unit Test Sessions

CanReturnFizzIfNumberDivisibleByThree failed

```
Expected: "Fizz"
But was:  False

at NUnit.Framework.Assert.That(Object actual, IResolveConstraint expression, String message, Object[] args)
at NUnit.Framework.Assert.That(Object actual, IResolveConstraint expression)
at TDD_Katas_project.FizzBuzzKata.TestFizzBuzz.CanReturnFizzIfNumberDivisibleByThree() in TestFizzBuzz.cs: line 64
```

Here, our test get failed and it is a `Red stage`.

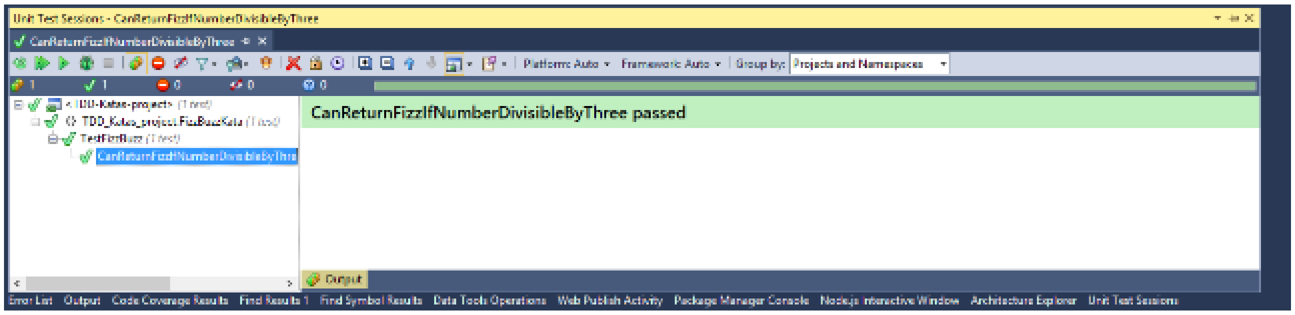
Now, we know we have a failed test and we know the reason why our test gets failed. Correct the code so our test gets passed.

Modify our above snippet and it looks like:

```
public static string PrintNumber()
        {
            var number = string.Empty;
            for (var i = 1; i < 100; i++)
            {
                number += i%3 == 0 ? " " + "Fizz" : " " + i;
            }

            return number.Trim();
        }
```

Now, again run the same test to see whether it will pass or fail.


Unit Test Sessions - CanReturnFizzIfNumberDivisibleByThree
CanReturnFizzIfNumberDivisibleByThree
Platform: Auto
Framework: Auto
Group by: Projects and Namespaces
TDD-Katas-projects (1 test)
TDD_Katas_project.FizzBuzzKata (1 test)
TestFizzBuzz (1 test)
CanReturnFizzIfNumberDivisibleByThree
Output
Error List Output Code Coverage Results Find Results 1 Find Symbol Results Data Tools Operations Web Publish Activity Package Manager Console Node.js Interactive Window Architecture Explorer Unit Test Sessions

CanReturnFizzIfNumberDivisibleByThree passed

Good to see that we wrote code so our test passes.

Here is our test pass, so, it is a `Green stage`.

Now, revisit the code snippet to see if there is any possibility to refactor the code, let's see: In the above, we are checking if number is divisible by 3, then it should be `'Fizz'`, yes, here we can refactor our code:

Create one method which lets us know whether a number is a `Fizz` or not:

```
private static bool IsFizz(int i)
        {
            return i % 3 == 0;
        }
```

And make a call to this method in our responsible method as:

```
public static string PrintNumber()
        {
            var number = string.Empty;

            for (var i = 1; i < 100; i++)
                number += IsFizz(i) ? " " + "Fizz" : " " + i;

            return number.Trim();
        }
```

Here, we refactored our code so, we're at Stage Refactoring. Now, repeat all these stages until you complete all the points of `FizzBuzz`. Remaining points are:

- For the multiples of five print `Buzz`.
- For numbers which are multiples of both three and five, print `FizzBuzz`
- Else print number itself.