Building Tomorrow's Enterprise

Infosys Global Agile Developer Certification

# Test Driven Development

Study Material

www.infosys.com

## Revision History

| Version | Date | Remarks |
|---|---|---|
| 1.0 | Dec’2014 | Initial Version |

# Table of Contents

# Table of Figures

# 1 Introduction

Agile approach to software development follows the four agile manifesto principles. It aims to deliver quality code to customers at frequent intervals enabling early return on investment (ROI). “Clean coding” as stated by Robert Martin and simple design principles stated by Kent Beck emphasizes on code that passes all the test cases.

*Clean Code:

- Elegant & efficient
- Minimal dependencies
- Brings out the intent of design
- Can be enhanced by other developers apart from the original author

**Rules for Simple design:

- Runs all the tests
- Contains no duplication
- Expresses the intent of the programmer
- Minimizes the number of classes and methods

This implies that in the agile approach to software development, the programmer needs to ensure:

a) Code is thoroughly tested

b) Code meets the needs of the customer

c) Changes can be incorporated easily into the code

d) Code is ‘clean’ (well-structured, self-documenting)

TDD (Test Driven Development) is a frequently heard term in the Agile world today. It is an engineering practice used in projects following Agile approach to software development. It is a very useful way to approach the Agile principle of delivering working software. This technique helps in developing “Just enough code” for a given functionality, results in unit tested code and avoids speculative programming. TDD is one of the engineering practices proposed by Extreme Programming (XP) and can be implemented in projects following Scrum. It is more of a habit and practice that makes TDD an integral part of software development. Test driven development means that the development is progressively carried out by writing the test cases first without any implementation, followed by writing enough implementation code just to make the test cases pass. A novice trying to learn the ropes of TDD would require to know the need for this approach and may ponder over the following questions:

1. What are the limitations of the conventional approach of coding followed by testing?
2. How would the programmer know what to test if the functionality is not yet implemented using code?
3. What would be the role of testers and QA team if the programmer writes the tests?

Let’s try to explore the answers to the above questions.

The conventional approach of coding followed by testing has the biggest limitation of developer being biased by the code s/he has developed. The developer knows what function s/he has developed and what would be the conditions his/her code satisfies. Thus, the tests built around this code are always to make those conditions pass. There is not much exploration of thoughts of failing those conditions. Thus the testing is not so effective and there are greater chances of defects getting leaked to later stages in the development life cycle.

**Source: Clean code, Robert C Martin, Prentice Hall*

***Source: http://xprogramming.com/classics/expemergentdesign/*

There is an obvious question of what to test if nothing has been written yet. This will be elaborated as we proceed in the document and dive deep into test driven development. But it is to be noted that TDD is a Development approach and not a Testing method.  It compels the developer to think on how to design the code, break it into small steps of implementation and ensure that what is aimed, is achieved at the end.

With the advent of TDD many started questioning the role of testers and QA in development cycle. It should be understood that the role and purpose of including testers and QA are different from having developers. The entire application comprises of many developers and each developer is more or less focused on the module that he / she is assigned to build. Test driven development is a development approach that allows the developer to build his/her code more robust and reduce his labor later in the cycle. The tester and the QA have still more elaborate and inclusive view of the application as a whole and their perspective brings more angles of robustness in the software. Their effort includes ensuring that the individual modules work as expected under all conditions and boundary limits, integrate as expected with other modules and the final response is correct, in time and consistent with all the external forces playing their part.

The advantages of using TDD are:

- It adds reliability to the development process
- It inspires programmers to maintain a comprehensive set of repeatable tests which may be run exhaustively with the help of tools
- Developers strive to improve their code without the fear associated with code changes
- It makes programming more fun due to reliability on code
- It removes/ reduces dependency on the debugger and no postponement of debugging

# 2 Concepts

Test Driven Development (TDD) is an approach used in software engineering that involves short development iterations. It consists of writing failing test case(s), first covering a new functionality and then implementing the necessary code to pass the tests, and finally refactor the code without changing external behavior.*

As mentioned in the definition, developers write automated tests before development. This means that development is driven from the tests. This ensures that no untested code goes into the software. Refactoring ensures that the code is 'clean'. In simple terms, TDD suggests developers to write tests for a new functionality or feature and then develop code to pass those tests, thus avoiding speculative coding.

[**Case Study**: The following story may be hard to find in any online reference, but may find traces in newspaper / magazines of 90's. Those days, the internet was not so prevalent!]

*This story goes back to days of Windows 98. Before Windows XP and NT, Windows 98 SE (second edition) was considered the most stable operating system of Microsoft. But very few know that Win98 first edition was a huge failure! There were many crashes and defects when it first came out in the market. It was extremely embarrassing for Microsoft as many design flaws were found out by the community. Microsoft devised a fresh strategy to the second edition. They engaged a parallel team that was equal in numbers to the development team of windows 98 with a single mandate to them – kill Windows 98! The team came up with extreme test strategies, data and scenarios before even the design was implemented! As an outcome, Win 98 SE has been a cherry in the series of operating systems released by Microsoft!*

The key to success of test driven development is not just writing tests but rather thinking from a tester perspective as can be inferred from the case study given. Since the programmer writes the test cases and the corresponding code, blind spots in code can be taken care in a relatively better manner as compared to the traditional approach. This approach also includes refactoring as one of the activities and hence results in robust code. This cycle of writing test cases, writing enough code and refactoring the code is continued till all possible tests are written and the programmer feels the code is robust.

The TDD cycle as explained above is diagrammatically explained in Figure 1:

**Source: Beck, K. Test-Driven Development by Example, Addison Wesley*

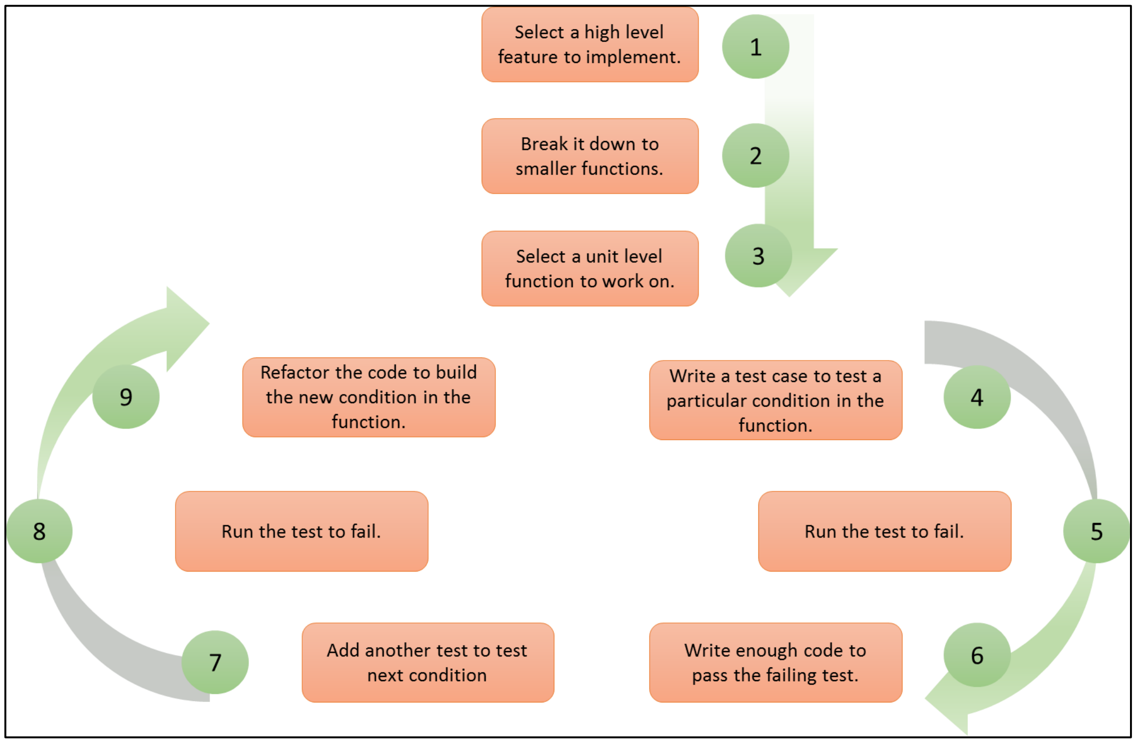

Select a high level feature to implement.
1
Break it down to smaller functions.
2
Select a unit level function to work on.
3
Write a test case to test a particular condition in the function.
4
Run the test to fail.
5
Write enough code to pass the failing test.
6
Add another test to test next condition
7
Run the test to fail.
8
Refactor the code to build the new condition in the function.
9

**Figure 1: TDD cycle**

TDD can be implemented well with the help of tools. The following are the key components:

1. **A Test Framework**: It is a set of class libraries that helps write the test cases in a particular syntax. It hooks into the source code methods for executing the 'implemented' feature and returns with the result as compared to what is expected as an output. The framework defines certain keywords, which needs to be followed while writing the tests. The framework also helps manage the tests (called 'unit tests'), execute them and see the result. The framework either has its own IDE or seamlessly integrates with the IDE that is used for software development i.e. Visual Studio or Eclipse.
2. **An IDE**: An Integrated Development Environment (IDE) that integrates the source code with the test framework and allows easy merging and switching between the two.
3. **An Analytics framework**: This is normally integrated with the IDE and Test framework. It shows the code coverage (how much code has been covered) with tests along with all the diagnostics of the result of the tests – for e.g. the library, file or line of code that is not covered.

The pre-requisite of writing a function to 'test' a feature **is same as** writing a function to 'implement' that feature. The programmer needs to be clear with the requirements and design of the feature that is being implemented.

TDD requires the code being added to pass a new test case written to necessarily pass all the other test cases as well. This ensures that regression testing is done for the changes brought in code.

Figure 2 depicts the lifecycle of a feature which is being developed following a TDD approach from a programmer's perspective.

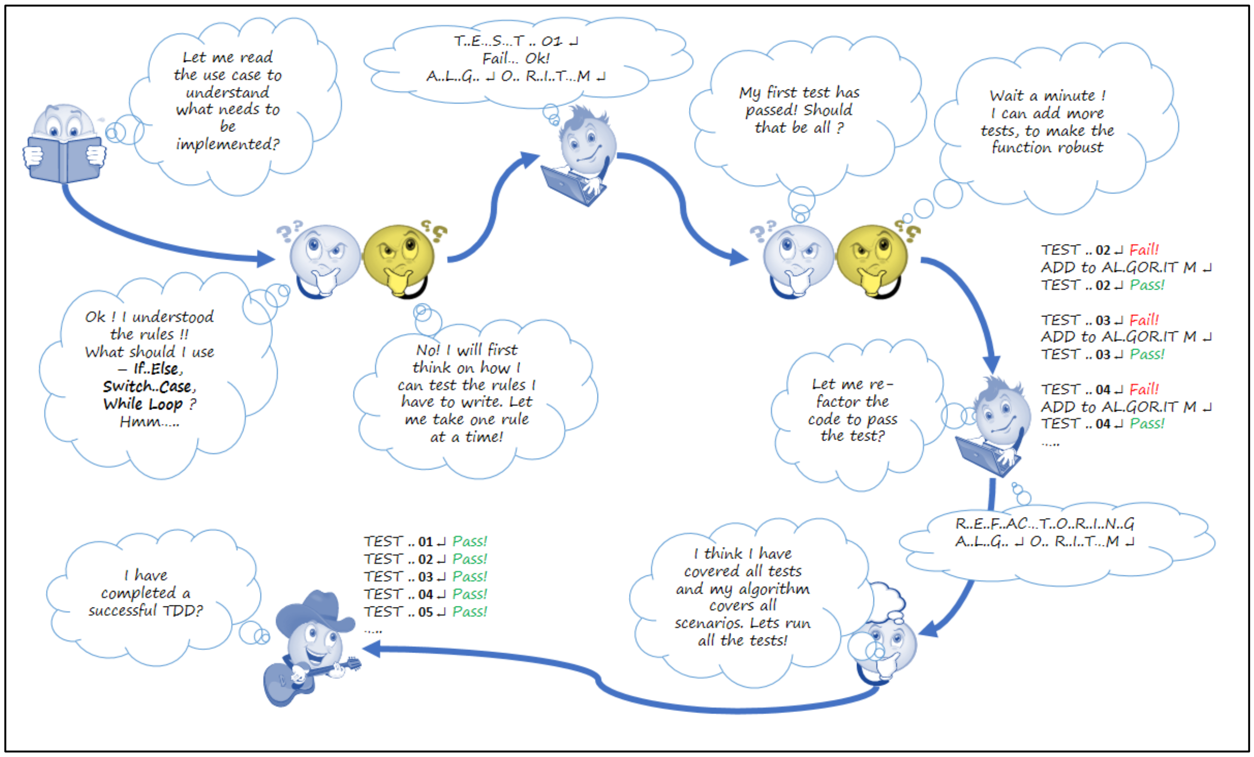

Let me read the use case to understand what needs to be implemented?
T..E...S...T .. 01 ↵
Fail... Ok!
A..L..G.. ↵ O.. R..I..T...M ↵
My first test has passed! Should that be all ?
Wait a minute ! I can add more tests, to make the function robust
Ok ! I understood the rules !! What should I use – If..Else, Switch..Case, While Loop ? Hmm…..
No! I will first think on how I can test the rules I have to write. Let me take one rule at a time!
Let me re-factor the code to pass the test?
TEST .. 02 ↵ Fail!
ADD to AL.GOR.IT M ↵
TEST .. 02 ↵ Pass!
TEST .. 03 ↵ Fail!
ADD to AL.GOR.IT M ↵
TEST .. 03 ↵ Pass!
TEST .. 04 ↵ Fail!
ADD to AL.GOR.IT M ↵
TEST .. 04 ↵ Pass!
…..
R..E..F..AC...T..O..R..I..N..G
A..L..G.. ↵ O.. R..I..T...M ↵
I have completed a successful TDD?
TEST .. 01 ↵ Pass!
TEST .. 02 ↵ Pass!
TEST .. 03 ↵ Pass!
TEST .. 04 ↵ Pass!
TEST .. 05 ↵ Pass!
…..
I think I have covered all tests and my algorithm covers all scenarios. Lets run all the tests!

**Figure 2: Life Cycle of Feature following TDD approach**

# 3 An Example using Pseudo Code

Following is an example of TDD approach. Pseudo code has been used to avoid programming language specificity. However, the same approach can be applied to Java, Flash or DOT NET in their respective environment compatible tool.

**Problem description:** The pseudo code written below takes a year as an input and returns 'Pass' if it is a leap year and 'Fail' if it is not.

**Algorithm**: To mark a year as a leap year, the following rules of divisibility by 4,100 and 400 need to be followed:

1. The year ordinal is divisible by 4 and not by 100: **Leap year**. E.g. 1904, 1980, 1992, 2004, 2012 etc.
2. The year ordinal is divisible by 4 and 100 but not by 400: **Not a Leap year**. E.g. 1700, 1800 1900 etc.
3. The year ordinal is divisible by both 4 and 400: **Leap year.** E.g. 1600, 2000 etc.

*[Note the naming convention followed in the test function. This is a good practice to be indicative in the naming of the test case. When this test is listed among 500 ~ 600 tests in the testing tool, the test name is a good indicator to what is being tested. Also, the methods used for implementing the test strategy are preceded with annotation* **[TEST METHOD]** *to differentiate them from the business logic implementation function which are decorated with annotation* **[IMPLEMENTATION]**.*]*

| | |
|---|---|
| `[TEST METHOD]`<br>`Function THIS_IS_A_LEAPYEAR_1980()`<br>`(`<br>`    If (ISLEAPYEAR(1980) = True`<br>`        Return Pass`<br>`    Else`<br>`        Return Fail`<br>`)` | ◀ ◀<br>*Begin with writing the test-function. Note that the code has not been written. So if the function is executed, the test case will surely fail!*<br>`THIS_IS_A_LEAPYEAR_1980 = Fail` |
| ▶ ▶<br>*Define the implementation functions and write the first implementation. As the basic rule of leap year is that it is divisible by 4.* | `[IMPLEMENTATION]`<br>`Function ISLEAPYEAR(year)`<br>`(`<br>`    If (year ÷ 4 = 0)`<br>`        Return True`<br>`    Else`<br>`        Return False`<br>`)` |
| `[TEST METHOD]`<br>`Function THIS_IS_NOT_A_LEAPYEAR_1900()`<br>`(`<br>`    If (ISLEAPYEAR(1900) = False`<br>`        Return Pass`<br>`    Else`<br>`        Return Fail`<br>`)` | ◀ ◀<br>*Run the test again. 1980 is divisible by 4, it is a leap year!*<br>`THIS_IS_A_LEAPYEAR_1980 = True`<br><br>*But not all year ordinals divisible by 4 are leap years. 1800 or 1900 were not leap years! Thus, next step is to create* ***a new*** *test method with this parameter in mind. Note the change in name of the test method and the change in the return result that is expected. Running the test:*<br>`THIS_IS_NOT_A_LEAPYEAR_1900= Fail` |
| ▶ ▶<br>*The problem with this implementation is that it has not catered the exception condition of exempting years that are both divisible by 4 and 100. Thus include the new condition in.* | `[IMPLEMENTATION]`<br>`Function ISLEAPYEAR(year)`<br>`(`<br>`    If (year ÷ 4 = 0) and (year ÷ 100 ≠ 0)`<br>`        Return True`<br>`    Else`<br>`        Return False`<br>`)` |
| `[TEST METHOD]`<br>`Function THIS_IS_ALSO_A_LEAPYEAR_2000()`<br>`(` | ◀ ◀ |

```
        If (ISLEAPYEAR(2000) = True
             Return Pass
        Else
             Return Fail
)
```

*Run the test again. Since 1900 is divisible by 4 and also by 100, it is not a leap year!*

**THIS_IS_NOT_A_LEAPYEAR_1900**= **True**

*Year 1600, and 2000 were leap years in spite of being divisible by 100! Create **a new** test method with this parameter in mind. Note the change in name of the test method and the change in the return result being expected. Running the test:*

**THIS_IS_ALSO_A_LEAPYEAR_2000**= **Fail**

▶ ▶

*Next, it is required to exempt a new condition i.e. years that are both divisible by 4 and 400 are leap years. Thus catering for all year ordinals that are divisible by 4 Or 400 but has to exempt that are also divisible by 100.*

[IMPLEMENTATION]

```
Function ISLEAPYEAR(year)
(
       If (year ÷ 4 = 0) and (year ÷ 100  ≠ 0)
               OR
         (year ÷ 400 = 0 and year ÷ 100  = 0)
            Return True
       Else
            Return False
)
```

◀ ◀

*Since the logic was refactored to take care of all the possible conditions, it should pass all the test.*

**THIS_IS_A_LEAPYEAR_1980** = **True**
**THIS_IS_NOT_A_LEAPYEAR_1900**= **True**
**THIS_IS_ALSO_A_LEAPYEAR_2000**= **True**

*Do you think you have covered all possible scenarios to test for the function? Think hard! Not only positive business parameters, but also the errors generated due to bad parameters need to be handled gracefully. How should the function behave when you pass all the following parameters to the function?*

1. **ISLEAPYEAR (@#&$)**: *passing special characters.*
2. **ISLEAPYEAR (TEXT_YEAR)**: *passing string i.e. 2014 as '2014'.*
3. **ISLEAPYEAR (0000)**: *passing 0000.*
4. **ISLEAPYEAR (990099000009786700987)**: *passing very high numbers to challenge the data type.*
5. **ISLEAPYEAR (-2014)**: *passing negative values.*
6. **ISLEAPYEAR (12-DEC-2000)**: *still a valid year but includes the date and month path as well.*

# 4 Practices

TDD is an iterative and incremental procedure. TDD does not mean that the developer can start working on code only after writing all the tests. A developer can first build the basic and high-level tests and once those are passed, the various other tests may be added. Duplication must be avoided while completing the functionality and development.

## 4.1 Steps for Implementing TDD

### 4.1.1 Add Test

New test gets added for every new User Story or task to be developed *(Refer 'Agile Overview' module for more details on User Stories)*. This test is the basic requisite for that particular user story. This test should be written in an automated way using various tools such as JUnit or NUnit. Developer must thoroughly understand requirements of the story, before writing the test. This test will fail once developer includes this on the automated framework.

### 4.1.2 Run Test

This helps serve two purposes. Firstly, when new code is added, test cases fail. This ensures that the functionality is not already developed as part of any other story. Secondly, a failing test is the first attempt towards TDD efforts in understanding a functionality. Failure in first attempt implies that TDD development is in progress.

### 4.1.3 Write implementation Code

Make changes in the test and add the code in such a way that the test cases pass. Passing test includes the process of writing code and making changes. To pass this test, code written in the test should be isolated only for passing this particular test. No other code or functionality should be added.

### 4.1.4 Run Test

In case test does not pass in the first attempt, the test must be run again. Failing in the test proves that it is not covering the functionality required by story and it requires further changes. Passing of test assures that required functionality is covered.

### 4.1.5 Refactor

After passing test, further changes need to be done to refactor the code. Concept of 'Refactoring' can be described as cleaning of code, applying proper design pattern. This also assures that duplication of code is absolutely avoided. *(Refer 'Refactoring' module for more details on Refactoring)*.

In a Sprint *(Refer "Agile Overview' module for more details on Sprint and Scrum)*, we may have multiple user stories. To keep adding more tests and new features or functionalities, the cycle of tests explained above can be repeated.

## 4.2 Acceptance Test Driven Development (ATDD)

As we build on test driven development, we also need to know about Acceptance Test driven development or ATDD. ATDD is again a practice where the whole team collaboratively brainstorms on the acceptance criteria of the application. Then it is drilled down into set of smaller but concrete acceptance tests before commencing the development.

Acceptance tests are designed from the end user's perspective. They ensure that the system produces correct output, given a value as an input. Acceptance tests are used to verify whether the state of the input changes as required by the business. They can also verify that the interactions with interfaces of other dependent entities, such as databases, external components or web services is working as expected. To elaborate this, following is an example with a User Story:

*"As a registered user, I want to check out (borrow) a book from the library."*

The acceptance criteria of this requirement will include many perspective, including the happy flow that the book is successfully issued to the user in the system.

- What if the book is already checked out by another user?
- What if the book is just listed and not yet available for checkout?
- What if there are multiple copies of the book in the library?
- What if the user has exhausted his/her limits of books that he/she can check out in a given period of time?
- What if the book is currently checked out by another user but is expected to be returned within few hours / days?
- What if the user has defaulted on his / her fees etc.?

While TDD focuses on developing individual function correct and robust, ATDD focuses on getting the external quality of the system correct, with all possible conditions brought into force. This involves ensuring the multiple functions work as expected in tandem.

## 4.3 Mocking and Stubbing

The function to be tested may be interacting with functions or interfaces which has either been already developed or is external in scope of the development in the current context. The behavior of such methods are pre-determined and to use their result in the current test needs these methods to be mocked up. Two methods are often used here – mocking and stubbing.

Mock – Is used to create a simulated object in place of the actual object. The mock method do not return any result and is meant to fit in the compilation of the tests for the current context.

Stubs – This is a class that is hard-coded to return data from its methods and properties. It is often used inside unit tests to test a class or method and derives the expected output for a known input.

# 5 TDD Implementation with Case Study

The following is a Case Study explaining the implementation of Test Driven Development approach using the Leap Year example as mentioned in Section-3. This case study is explained using the .Net programming language through the Screen images taken from Visual Studio 2013 console.

(For a case study with Java programming language, please refer Sec.3.2.3 in 'Test Driven Development' module of T300 Certification content which explains the 'Bowling Game' example at the following link)

http://sparshv2/portals/QualityAndProductivity/Documents/AgileCOE/IGAC/IGAC_Test_Driven_Development.pdf

## Find a Leap Year: DOT NET

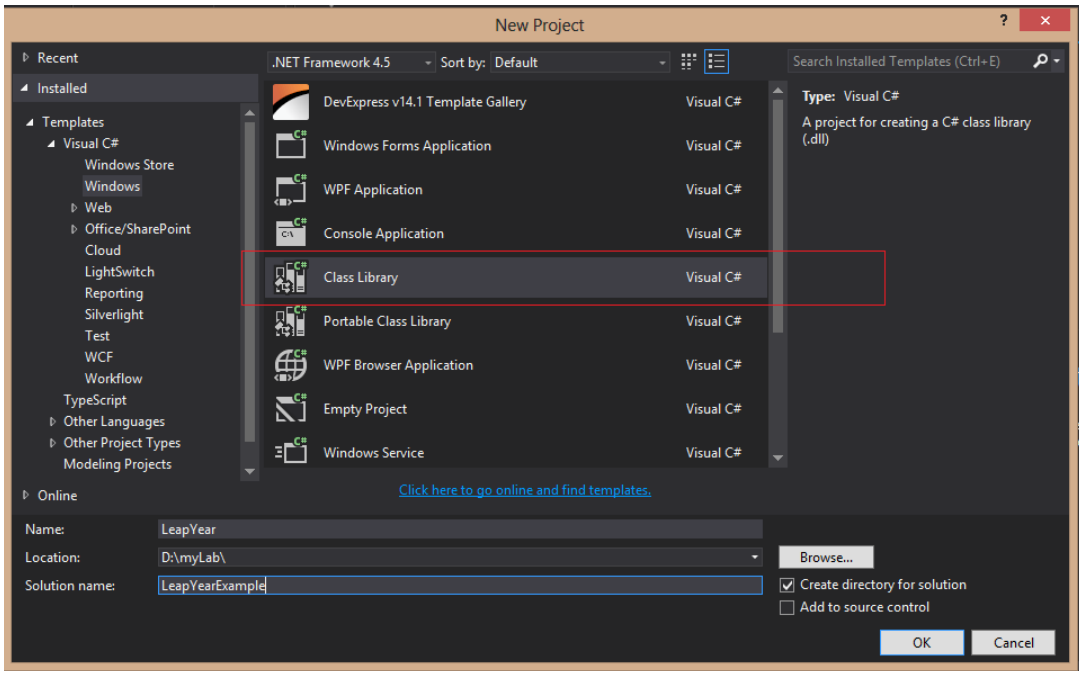
The following example will show the steps to implement the test driven development of the leap year example in DOT NET.

Environment used:

1. **IDE**: Visual Studio 2013
2. **Framework**: DOT NET Framework 4.5
3. **Programming Language**: C#

**Step 1:**

Create a Class library as a new project in a solution.


New Project
Recent
Installed
Templates
Visual C#
Windows Store
Windows
Web
Office/SharePoint
Cloud
LightSwitch
Reporting
Silverlight
Test
WCF
Workflow
TypeScript
Other Languages
Other Project Types
Modeling Projects
Online
.NET Framework 4.5
Sort by: Default
Search Installed Templates (Ctrl+E)
DevExpress v14.1 Template Gallery Visual C#
Windows Forms Application Visual C#
WPF Application Visual C#
Console Application Visual C#
Class Library Visual C#
Portable Class Library Visual C#
WPF Browser Application Visual C#
Empty Project Visual C#
Windows Service Visual C#
Click here to go online and find templates.
Browse...
OK
Cancel

Type: Visual C#

A project for creating a C# class library (.dll)

Name: LeapYear

Location: D:\myLab\

Solution name: LeapYearExample

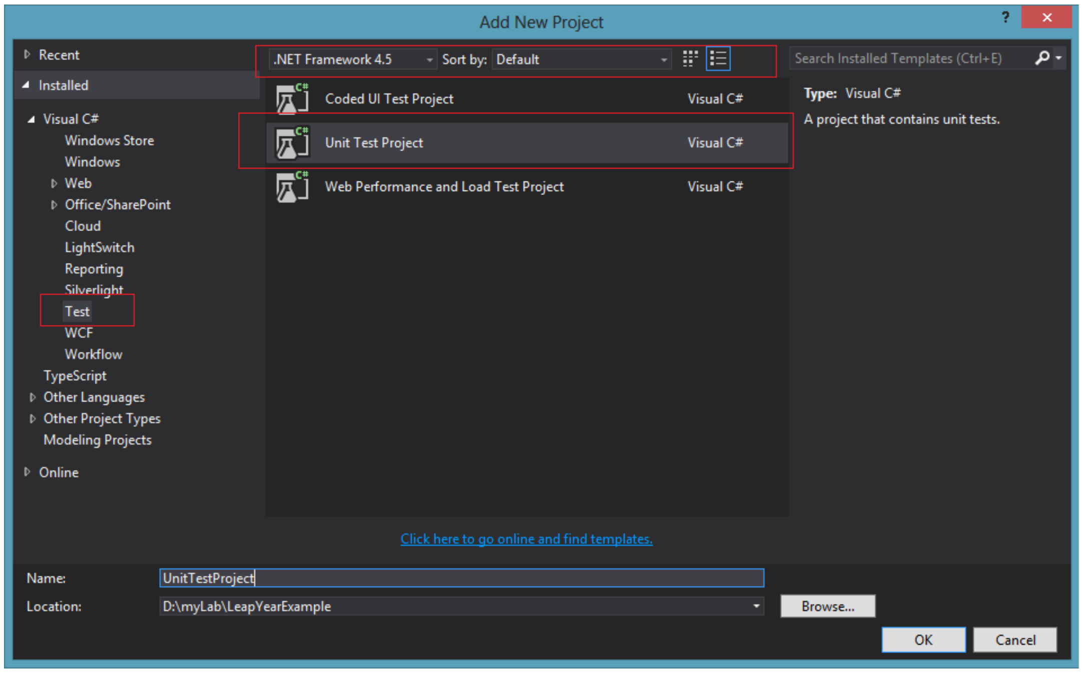
[✓] Create directory for solution

[ ] Add to source control

Rename the class from 'Class1' to 'LeapYearManager'. Now we will add the unit test project to add our tests.

**Step 2:**

Add a new project in the solution. Select the 'Test' type project templates and select the 'Unit Test Project'.



.NET Framework 4.5 Sort by: Default

Type: Visual C#

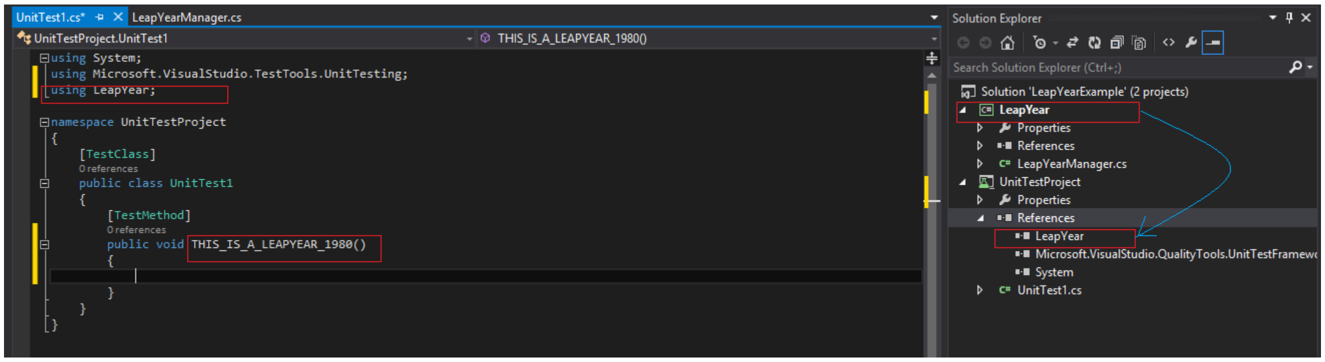
A project that contains unit tests.

Name: UnitTestProject

Location: D:\myLab\LeapYearExample

After adding the test project, add the reference of the class library we added earlier to the Test project. This is required as the test class should have the references of the actual implementation of the leap year logic. Add the namespace in the class and rename the default ‘TestMethod1’ to ‘THIS_IS_A_LEAPYEAR_1980’ as mentioned in the pseudo code above.

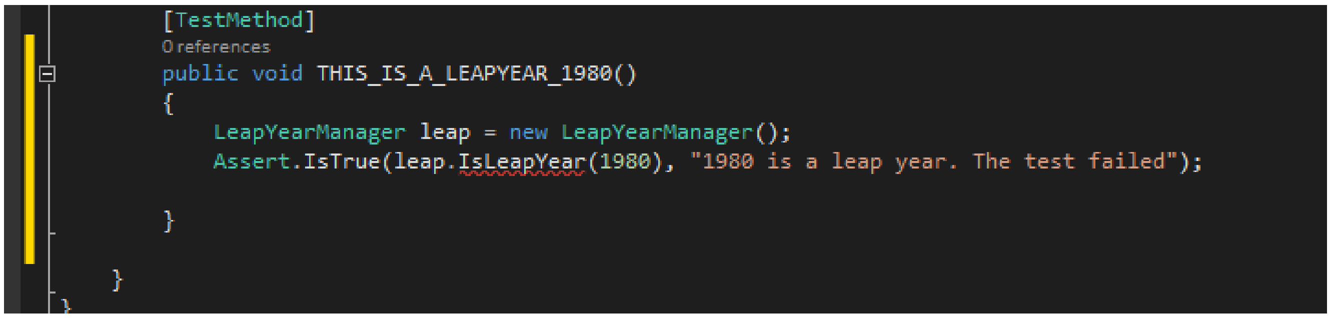
Note that the method as the Attribute ‘[Test Method]’ which distinguishes the method as the test method from the other private methods.


UnitTest1.cs*
LeapYearManager.cs
UnitTestProject.UnitTest1
THIS_IS_A_LEAPYEAR_1980()
Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'LeapYearExample' (2 projects)
LeapYear
Properties
References
LeapYearManager.cs
UnitTestProject
Properties
References
LeapYear
Microsoft.VisualStudio.QualityTools.UnitTestFramew
System
UnitTest1.cs

```
using System;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using LeapYear;

namespace UnitTestProject
{
    [TestClass]
    0 references
    public class UnitTest1
    {
        [TestMethod]
        0 references
        public void THIS_IS_A_LEAPYEAR_1980()
        {

        }
    }
}
```

**Step 3**:

Add the assertion to check whether 1980 is a leap year.

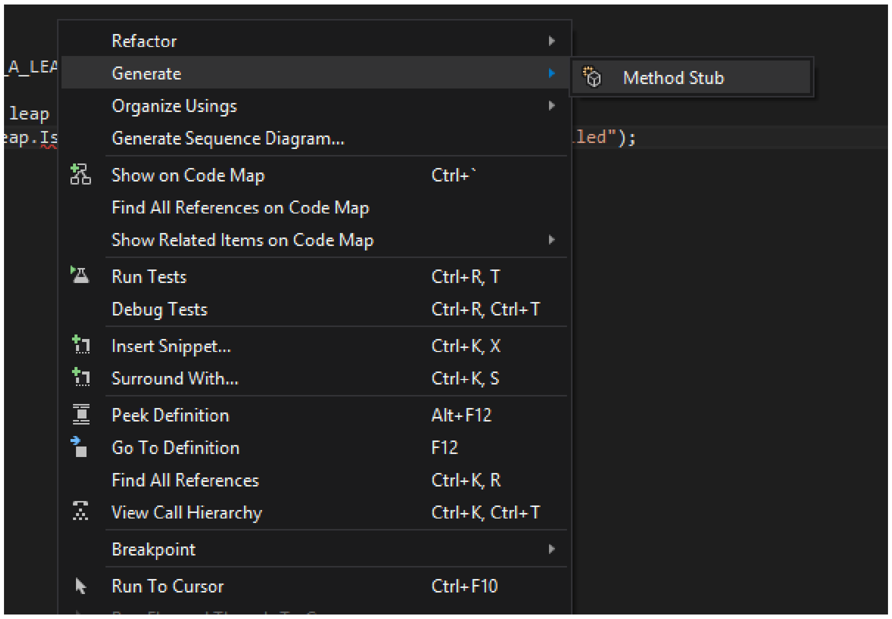


```
        [TestMethod]
        0 references
        public void THIS_IS_A_LEAPYEAR_1980()
        {
            LeapYearManager leap = new LeapYearManager();
            Assert.IsTrue(leap.IsLeapYear(1980), "1980 is a leap year. The test failed");

        }

    }
}
```

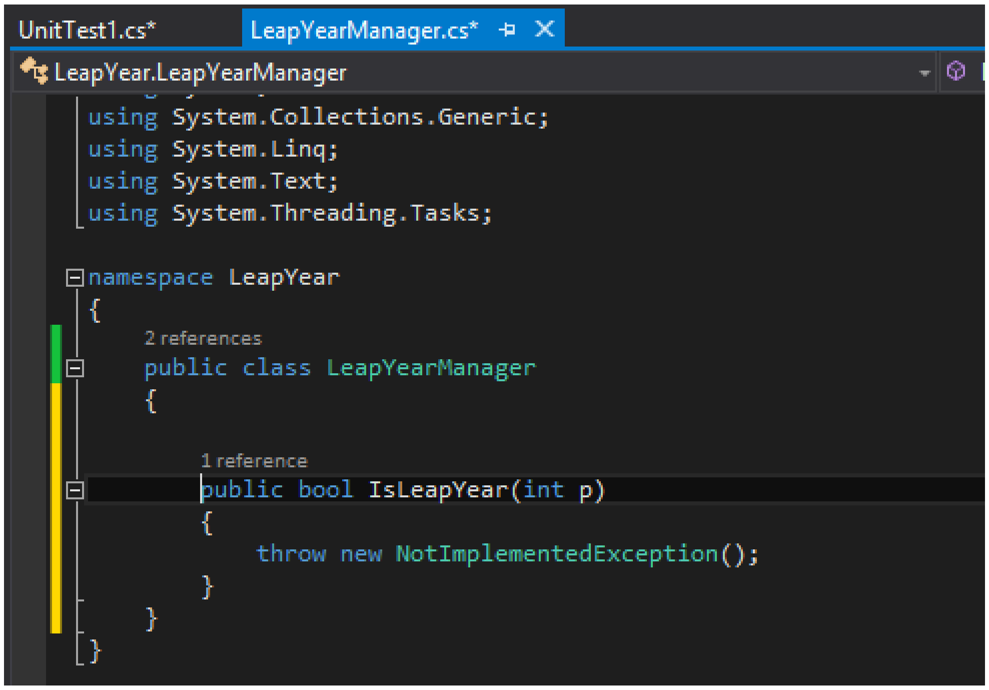
As you can see in the image above, the first code is added to check whether 1980 is a leap year. Create the object instance of the business class 'LeapYearManager'. Note that the function 'IsLeapYear' is not yet added to the class. This is the method we will develop progressively to develop the complete logic of checking a leap year given the year ordinal. That is the reason that the name of the method is underlined with **red** as there is no reference found yet.

**Step 4**:

Add the Method Stub. To correct the above error, right click on the name of the function 'IsLeapYear' and select to generate the Method Stub.


A_LEA
leap
eap.Is
led");
Refactor
Generate
Method Stub
Organize Usings
Generate Sequence Diagram...
Show on Code Map Ctrl+`
Find All References on Code Map
Show Related Items on Code Map
Run Tests Ctrl+R, T
Debug Tests Ctrl+R, Ctrl+T
Insert Snippet... Ctrl+K, X
Surround With... Ctrl+K, S
Peek Definition Alt+F12
Go To Definition F12
Find All References Ctrl+K, R
View Call Hierarchy Ctrl+K, Ctrl+T
Breakpoint
Run To Cursor Ctrl+F10

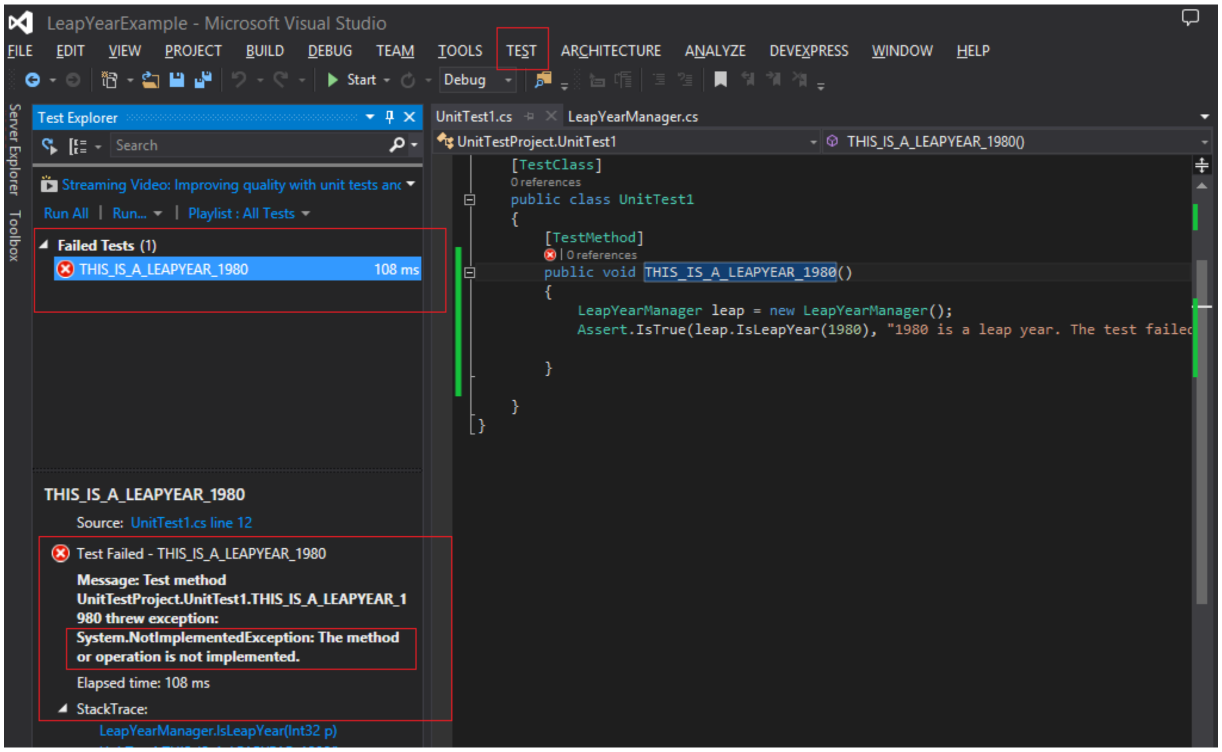
Check the class 'LeapYearManager' and a new default implementation of the function is added. And the '**red'** error line below the method name has disappeared.



```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LeapYear
{
    2 references
    public class LeapYearManager
    {

        1 reference
        public bool IsLeapYear(int p)
        {
            throw new NotImplementedException();
        }
    }
}
```

**Step 5**:

Run your first test. From the menu above, select TEST > Run > All Test. The application code will compile and the result of the first run is presented.


LeapYearExample - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM TOOLS TEST ARCHITECTURE ANALYZE DEVEXPRESS WINDOW HELP
Start Debug
Test Explorer
Search
Streaming Video: Improving quality with unit tests and
Run All | Run... | Playlist : All Tests
UnitTest1.cs LeapYearManager.cs
UnitTestProject.UnitTest1 THIS_IS_A_LEAPYEAR_1980()

Failed Tests (1)
THIS_IS_A_LEAPYEAR_1980 108 ms

```
[TestClass]
0 references
public class UnitTest1
{
    [TestMethod]
    0 references
    public void THIS_IS_A_LEAPYEAR_1980()
    {
        LeapYearManager leap = new LeapYearManager();
        Assert.IsTrue(leap.IsLeapYear(1980), "1980 is a leap year. The test failed

    }

}
}
```

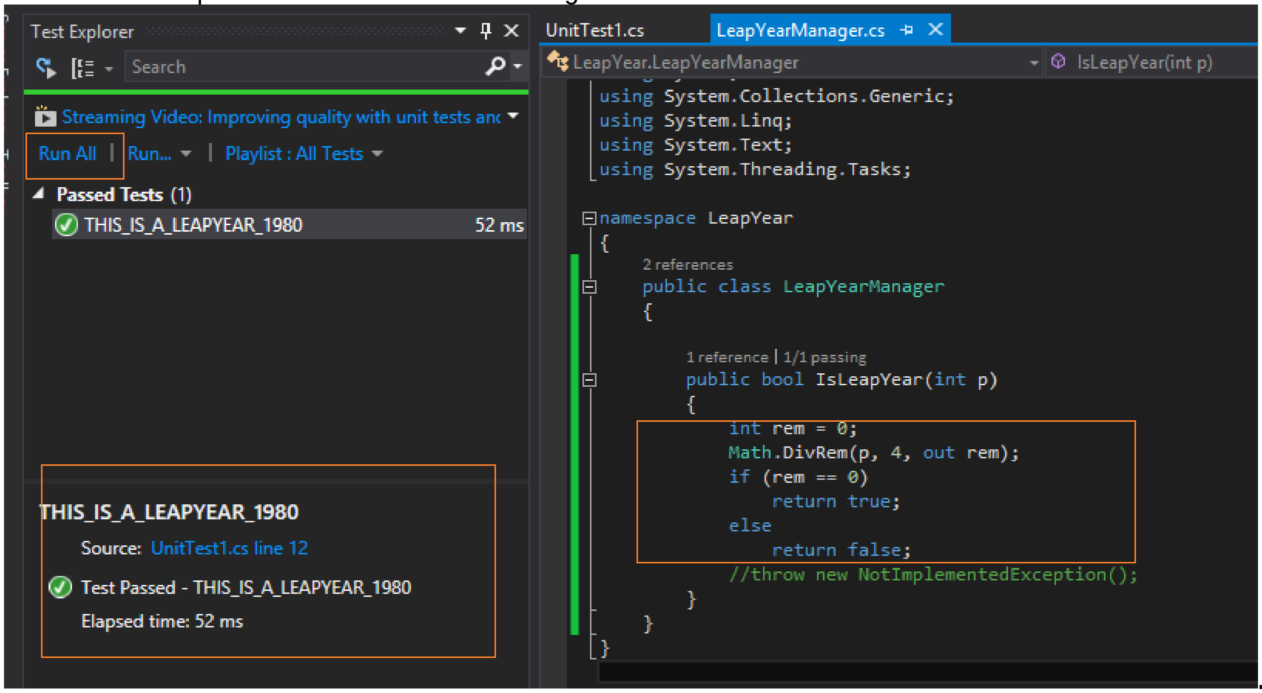
THIS_IS_A_LEAPYEAR_1980
Source: UnitTest1.cs line 12

Test Failed - THIS_IS_A_LEAPYEAR_1980
Message: Test method UnitTestProject.UnitTest1.THIS_IS_A_LEAPYEAR_1980 threw exception:
System.NotImplementedException: The method or operation is not implemented.
Elapsed time: 108 ms
StackTrace:
LeapYearManager.IsLeapYear(Int32 p)

The reason of the failure shows 'NotImplementedException' which is exactly the error that was thrown in the function IsLeapYear(). Next, the implementation should be done for making the failed test to pass.

**Step 6**:

Write the first implementation and run the test again


Test Explorer
Search
Streaming Video: Improving quality with unit tests and
Run All | Run... | Playlist : All Tests
UnitTest1.cs LeapYearManager.cs
LeapYear.LeapYearManager IsLeapYear(int p)

Passed Tests (1)
THIS_IS_A_LEAPYEAR_1980 52 ms

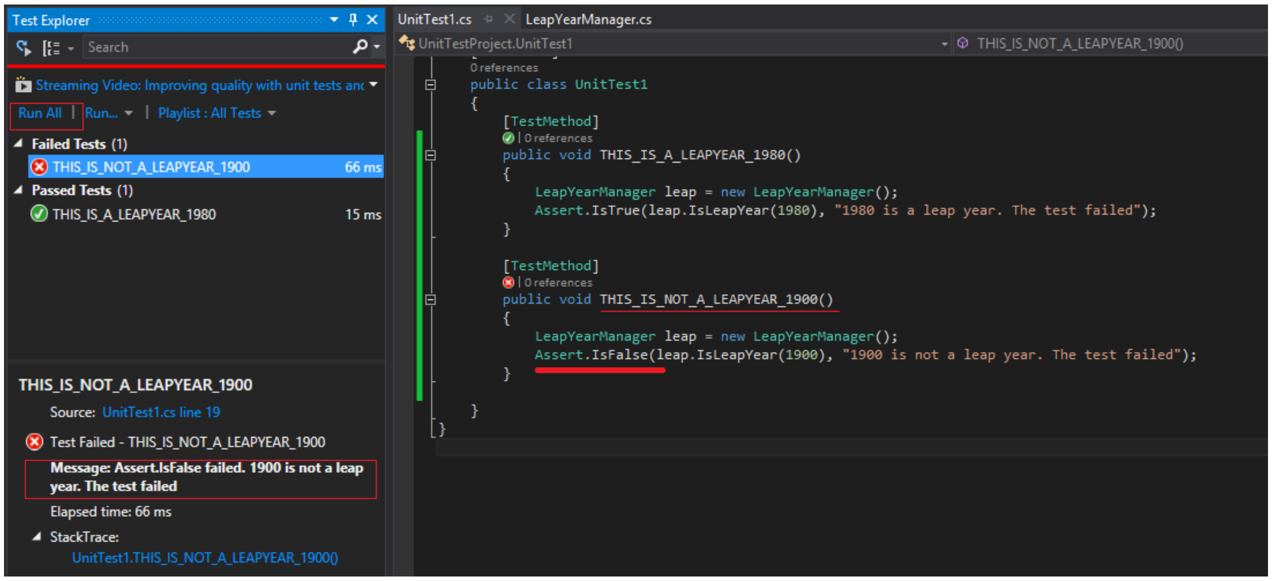
```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LeapYear
{
    2 references
    public class LeapYearManager
    {
        1 reference | 1/1 passing
        public bool IsLeapYear(int p)
        {
            int rem = 0;
            Math.DivRem(p, 4, out rem);
            if (rem == 0)
                return true;
            else
                return false;
            //throw new NotImplementedException();
        }
    }
}
```

THIS_IS_A_LEAPYEAR_1980
Source: UnitTest1.cs line 12
Test Passed - THIS_IS_A_LEAPYEAR_1980
Elapsed time: 52 ms

Implement the first logic of checking 1980 is a leap year. It should be divisible by 4. Once the implementation is done, run the test again and the result turns into ‘Passed’.

**Step 7**:

Write the second test and run the tests again



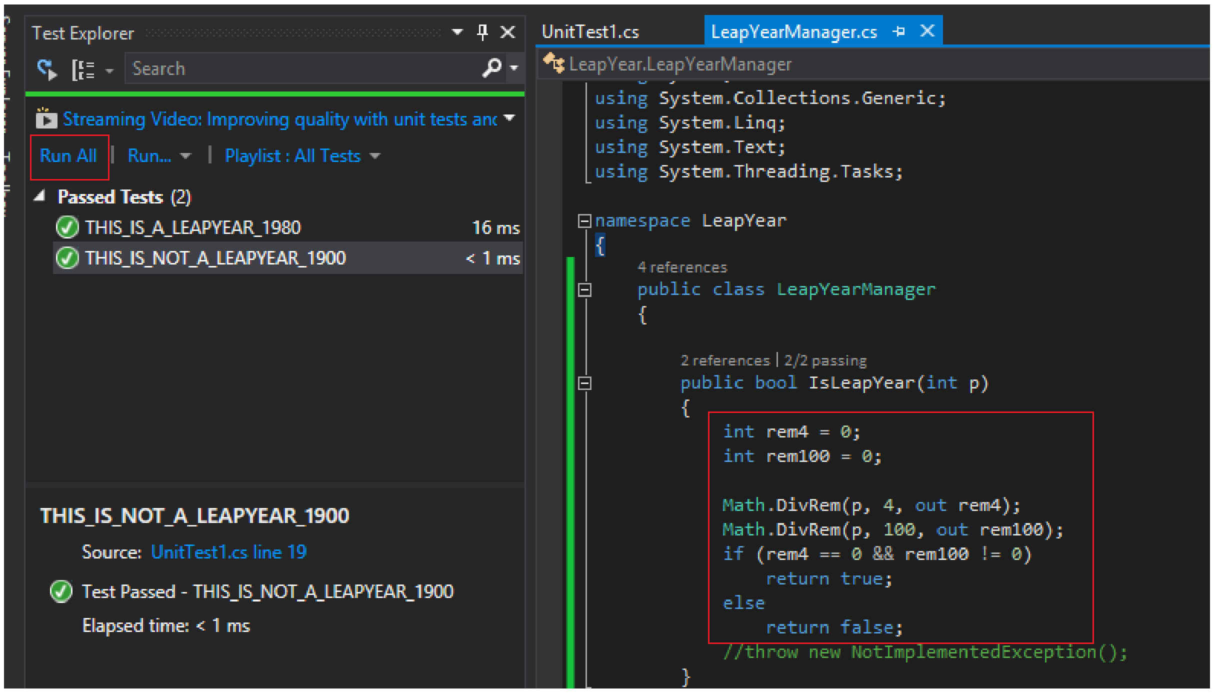
```
Failed Tests (1)
  THIS_IS_NOT_A_LEAPYEAR_1900    66 ms
Passed Tests (1)
  THIS_IS_A_LEAPYEAR_1980        15 ms

THIS_IS_NOT_A_LEAPYEAR_1900
  Source: UnitTest1.cs line 19
  Test Failed - THIS_IS_NOT_A_LEAPYEAR_1900
  Message: Assert.IsFalse failed. 1900 is not a leap year. The test failed
  Elapsed time: 66 ms
  StackTrace:
    UnitTest1.THIS_IS_NOT_A_LEAPYEAR_1900()
```

```csharp
    public class UnitTest1
    {
        [TestMethod]
        public void THIS_IS_A_LEAPYEAR_1980()
        {
            LeapYearManager leap = new LeapYearManager();
            Assert.IsTrue(leap.IsLeapYear(1980), "1980 is a leap year. The test failed");
        }

        [TestMethod]
        public void THIS_IS_NOT_A_LEAPYEAR_1900()
        {
            LeapYearManager leap = new LeapYearManager();
            Assert.IsFalse(leap.IsLeapYear(1900), "1900 is not a leap year. The test failed");
        }
    }
}
```

Write the second test about checking 1900 as a leap year. Note that checking 1900 should return false and thus the assertion should be implemented accordingly. When the tests are run again, there is a new test in the list of tests and one of them failed as shown in the above image.

**Step 8**:

Write the second implementation and run the tests again



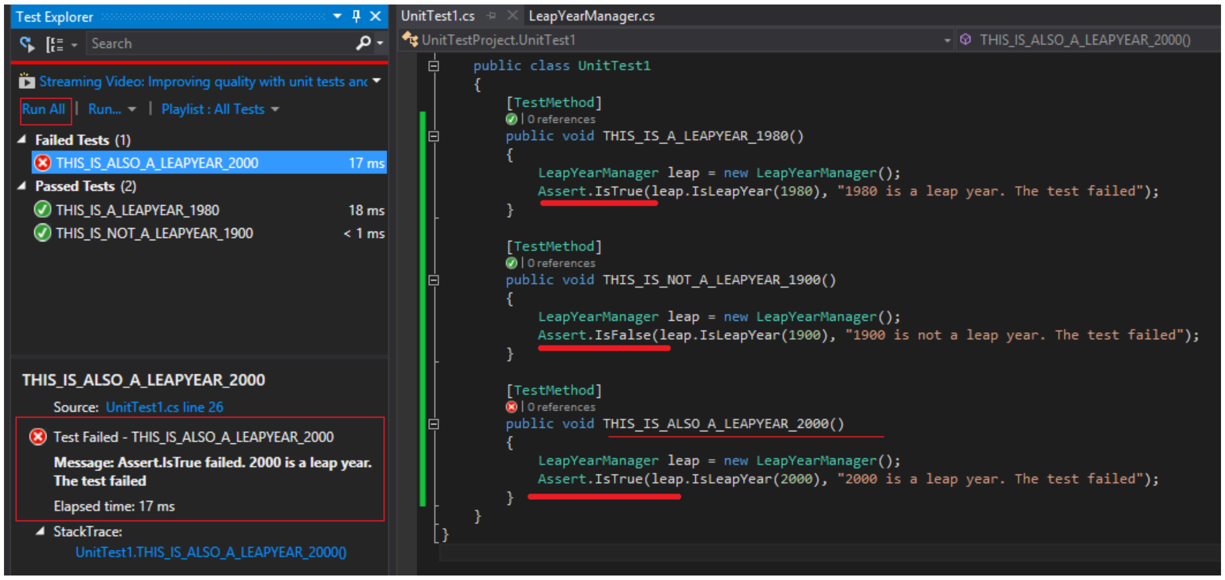
```
Passed Tests (2)
  THIS_IS_A_LEAPYEAR_1980        16 ms
  THIS_IS_NOT_A_LEAPYEAR_1900    < 1 ms

THIS_IS_NOT_A_LEAPYEAR_1900
  Source: UnitTest1.cs line 19
  Test Passed - THIS_IS_NOT_A_LEAPYEAR_1900
  Elapsed time: < 1 ms
```

```csharp
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LeapYear
{
    public class LeapYearManager
    {
        public bool IsLeapYear(int p)
        {
            int rem4 = 0;
            int rem100 = 0;

            Math.DivRem(p, 4, out rem4);
            Math.DivRem(p, 100, out rem100);
            if (rem4 == 0 && rem100 != 0)
                return true;
            else
                return false;
            //throw new NotImplementedException();
        }
```

Add the condition of division by 100 and run the tests again. All should turn **green**.

**Step 9**:

Write the test for checking the year 2000 and run the tests again.



Test Explorer

Run All | Run... | Playlist : All Tests

Failed Tests (1)
- THIS_IS_ALSO_A_LEAPYEAR_2000 17 ms

Passed Tests (2)
- THIS_IS_A_LEAPYEAR_1980 18 ms
- THIS_IS_NOT_A_LEAPYEAR_1900 < 1 ms

THIS_IS_ALSO_A_LEAPYEAR_2000

Source: UnitTest1.cs line 26

Test Failed - THIS_IS_ALSO_A_LEAPYEAR_2000

Message: Assert.IsTrue failed. 2000 is a leap year. The test failed

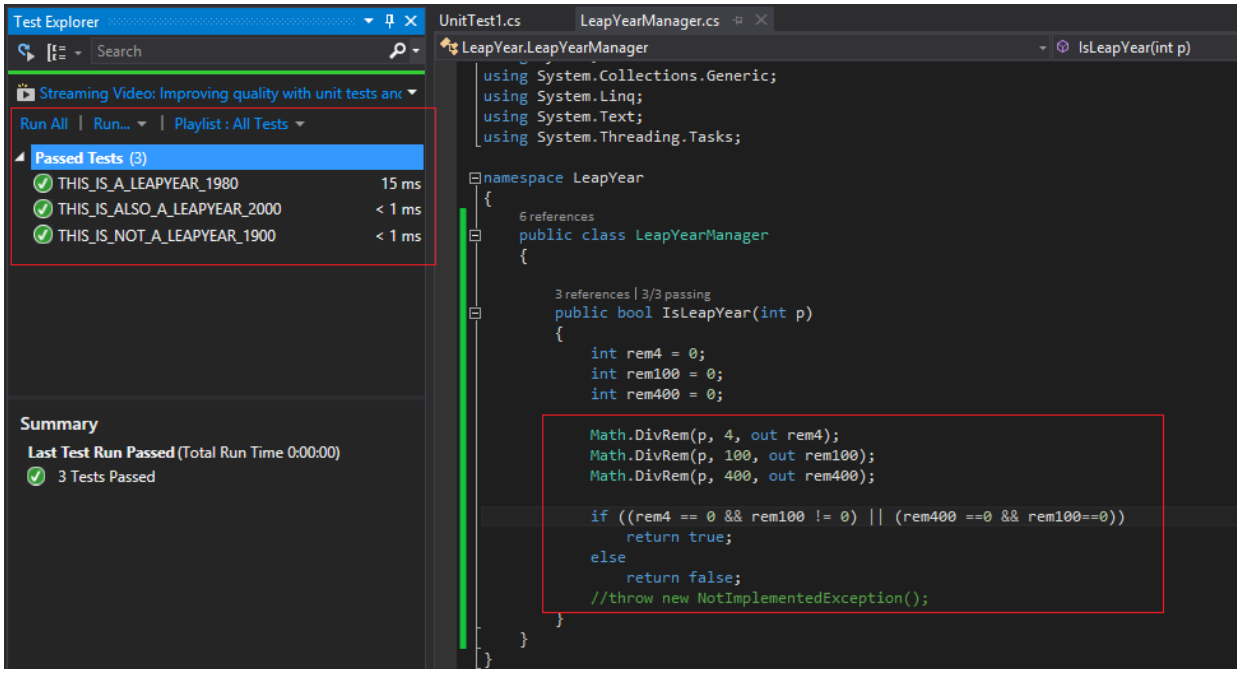
Elapsed time: 17 ms

StackTrace:
UnitTest1.THIS_IS_ALSO_A_LEAPYEAR_2000()

```
public class UnitTest1
{
    [TestMethod]
    0 references
    public void THIS_IS_A_LEAPYEAR_1980()
    {
        LeapYearManager leap = new LeapYearManager();
        Assert.IsTrue(leap.IsLeapYear(1980), "1980 is a leap year. The test failed");
    }

    [TestMethod]
    0 references
    public void THIS_IS_NOT_A_LEAPYEAR_1900()
    {
        LeapYearManager leap = new LeapYearManager();
        Assert.IsFalse(leap.IsLeapYear(1900), "1900 is not a leap year. The test failed");
    }

    [TestMethod]
    0 references
    public void THIS_IS_ALSO_A_LEAPYEAR_2000()
    {
        LeapYearManager leap = new LeapYearManager();
        Assert.IsTrue(leap.IsLeapYear(2000), "2000 is a leap year. The test failed");
    }
}
}
```

**Step 10**:

Put the implementation to check the year 2000.



Test Explorer

Run All | Run... | Playlist : All Tests

Passed Tests (3)
- THIS_IS_A_LEAPYEAR_1980 15 ms
- THIS_IS_ALSO_A_LEAPYEAR_2000 < 1 ms
- THIS_IS_NOT_A_LEAPYEAR_1900 < 1 ms

Summary

Last Test Run Passed (Total Run Time 0:00:00)

3 Tests Passed

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace LeapYear
{
    6 references
    public class LeapYearManager
    {
        3 references | 3/3 passing
        public bool IsLeapYear(int p)
        {
            int rem4 = 0;
            int rem100 = 0;
            int rem400 = 0;

            Math.DivRem(p, 4, out rem4);
            Math.DivRem(p, 100, out rem100);
            Math.DivRem(p, 400, out rem400);

            if ((rem4 == 0 && rem100 != 0) || (rem400 ==0 && rem100==0))
                return true;
            else
                return false;
            //throw new NotImplementedException();
        }
    }
}
```

Add the implementation of checking the 2000 and the logic to check a leap year is complete. Thus,

A year is a leap year if:

1. It should be divisible by 4
2. If the year is also divisible by 100 but not by 400, then the year is not a leap year.
3. If it is also divisible by 400, it is again a leap year.

Run all the tests again and everything should be **green**.

# 6 TDD – Benefits and Challenges

One of the key advantages of following TDD is that the programmer can completely restructure the code. As long as every test passes, the code is good. Also if some test fails, the programmer knows where to look for and fix it, given the programmer friendly features available in most of the tools. For example, assume that there are 10,000 lines of code and 90% of the code has tests. If some major refactoring / redesign is done, the programmer would know what tests have failed and where to fix it rather than having to run the program to figure that out.

**Benefits – Programmer Perspective**

1. Acts as a safety net when a developer performs refactoring. The programmer can restructure the code and be assured of the function as long as the tests are passing. Even if they are failing, developer would know exactly where to target.
2. Challenges the design at every step thus ensuring the design grows more stable and the faults are detected early in the cycle.
3. Design principles compliance: If the tests are becoming increasingly complex, the programmer might be missing out on following SOLID principles. Thus the code maintainability improves if the programmer follows the SOLID principles.
   (Please refer Sec.3.3.2 in 'Design Approach for Agile Projects' module of T300 Certification content at the following link:
   http://sparshv2/portals/QualityAndProductivity/Documents/AgileCOE/IGAC/IGAC_Design_Approach_For_Agile_Projects.pdf )
4. Brings in simplicity for delivering the work
   a) Delivers the work in a simple way to avoid complexity. TDD can be considered as both design and programming methodology which is based on a simple rule of writing production code to fix a failing test.
   b) It might take a while for programmers to get into groove, but once done, it helps in bringing simplicity in development of code.
5. Helps deliver high-quality software through the continuous design and test process and frequent refactoring. Code becomes simple, less defective, better designed and of high quality. The developer gets confidence of writing code and becomes more productive.
6. Reduce the amount of production code it takes to solve a business problem. It is observed often that the ratio of number of lines of code written for unit tests to that of the production implementation is 2:1 in a well-managed test driven developed application. This is not a bottleneck. On contrary, properly followed TDD promotes to reduce the number of lines of code in functional implementation of the business problem. TDD gives the team confidence to adapt reusability without the fear of breaking already running code, thus give-up copy-paste and duplication of code.

**Benefits – Project Perspective**

1. Helps deliver high quality code
2. Faster results and avoids speculative coding/design:
   a) Provides an immediate feedback to reduce reversion errors.
   b) Developer writes the code that is required (with no extra code) resulting in faster results with clear, concise code
   c) Better and rapid response to changing requirements
3. Simplicity: Maintainable code
4. Improved Quality: Test Automation can be achieved with the help of tools
5. Flexibility: Quick commits, short cycles and automation

**Challenges**

- Firstly, it can be little frustrating when you first start trying to use TDD in your coding style. It is required that you don't get discouraged and quit, you will need to give it some time. It is a major paradigm shift in how we think about solving a problem in code.
- Implementing test driven development initially increases the time-lines of the project. It has to be appropriately factored in and communicated to all the stakeholders during the project planning.
- There is a popular perception that test driven development slows the development team down. While it may be true in the initial days of development; as the code base grows, it accelerates the affairs in the long term.
- External system dependencies (e.g. FTP server or services) need to be mocked-up. And mocking an object is not easy.
- It is not enough to create the test code but it has to be updated as functionality evolves and create new ones. Often the discipline to be followed in this respect is as demanding as writing and maintaining the code towards production implementation.
- Another challenge is observed in projects with legacy code. If the code is already implemented and the customer is willing to bring in test driven development during the enhancements of the application, it poses enormous challenge to ensure that the existing code is also factored in, while putting the new tests.
- The ROI (Return on Investment) on TDD is achieved in the long run only if the tests developed are maintained for relevance and run regularly over the lifetime of the application.

# 7 Practice Questions

**Question 1**

Test Driven Development practice is originated from which of the following Agile methods?

a) SCRUM
b) XP (eXtreme Programming)
c) DSDM (Dynamic System Development Methodology)
d) KANBAN

**Question 2**

Which of the following is a true statement for TDD?

i. It is done by testing team personals.
ii. It is done after the design is complete.
iii. It is done by the developer in parallel to writing the code.
iv. The tests once written are good for always. They need not be changed in future.

a) Only ‘i’
b) Only ‘ii’ and ‘iii’
c) Only ‘iii’
d) Only ‘iii’ and ‘iv’

**Question 3**

What is TDD?

a) It is a Development method
b) It is a Testing method
c) It is an subset of feature driven development approach
d) All of the above

**Question 4**

What does ATDD stands for?

a) Agile Test Driven Development
b) Application Test Driven Development
c) Adaptive Test Driven Development
d) Acceptance Test Driven Development

**Question 5**

Which of the following step is NOT applicable while implementing TDD?

a) Requirements prioritization
b) Refactoring
c) Running Test
d) Adding Test

# 8 References

- Kent Beck, *'Test-Driven Development by Example'*, Addison Wesley
- Robert C Martin, *'Clean Code'*, Prentice Hall
- Websites
  - Acceptance Test Driven Development
    http://en.wikipedia.org/wiki/Acceptance_test-driven_development
  - Details on TDD
    http://en.wikipedia.org/wiki/Test-driven_development
    *http://xprogramming.com/classics/expemergentdesign/*
  - DOT NET FRAMEWORK, Microsoft Article on '*Walkthrough: Test-First Support with the Generate from Usage Feature'* http://msdn.microsoft.com/en-us/library/dd998313(VS.100).aspx
  - Good read on Test-Driven Development in .NET
    http://www.codeproject.com/Articles/3781/Test-Driven-Development-in-NET
  - Mocking and Stubbing
    http://martinfowler.com/articles/mocksArentStubs.html
    http://en.wikipedia.org/wiki/Mock_object
  - Some popular unit testing frameworks are listed here:
    http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks
  - SOLID principle guidelines and definitions from http://www.codemag.com/article/1001061

Infosys® | Building Tomorrow's Enterprise

**www.infosys.com**