Not quite what you are looking for? You may want to try:

- Test Driven / First Development by Example
- Designing Application Using Test Driven Development

highlights off

Developing Factorial Application Using **Test Driven Development**

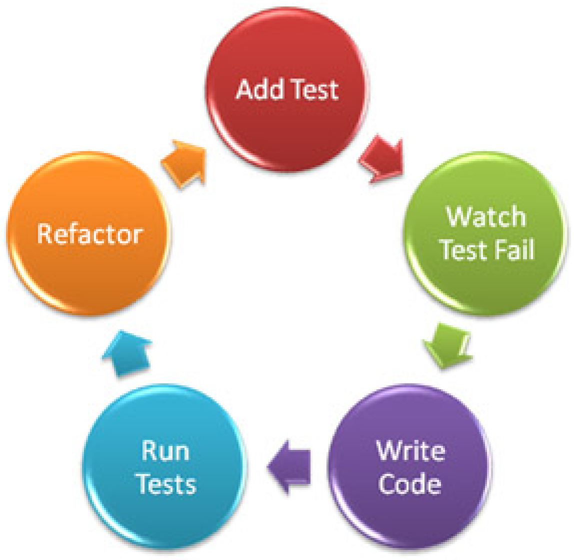
Bayram Üçüncü, 28 Jan 2012 CPOL

4.78 (8 votes)

Rate this:

The aim of this article is developing a Factorial Application using **Test Driven Development**

Download project source code - 58.96 KB



Introduction

In this article, we have a look at **Test Driven Development** with an example. The example is **Factorial** calculator application. This simple application shows us how to develop an application by **test driven development**. I will write the "**Test Driven Development**" as "TDD" in article.

Background

When I first faced the **Test Driven Development**, I thought "it is unnecessary". But later, I read the articles about it and I show that TDD prevents the errors and problems before the delivered projects. It also creates reliable products, shows code coverages. These benefits were enough for me to begin TDD.

In this article, we will use the Visual Studio default Unit **Test** Framework. But if you don't want to use default framework, you have more choices in the market of Unit **Test**s.

Before You Start

Before writing your code, you should determine the requirements of the application. We will be developing a **Factorial** application in this article. So we should determine the **Factorial** requirements. Later, we can determine the **test** methods. If you want, you can directly begin writing **test**s and skip these steps. But making is worth it.

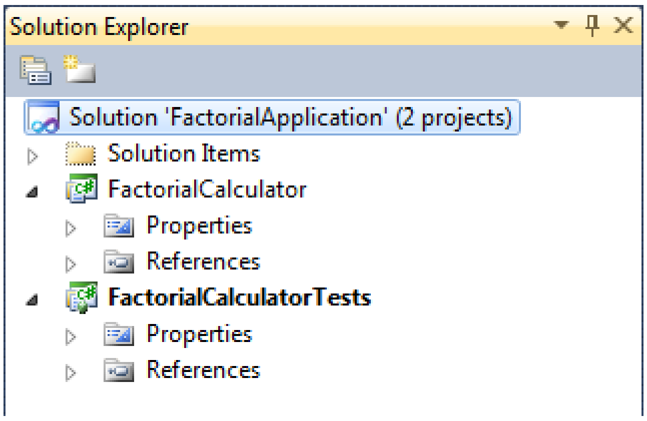
Requirements

1. Factorial calculates the multiplication of the all positive numbers less than or equal Target Number(n).
2. Target Number(n) is a non-negative integer.

Unit **Tests**

1. Zero factorial is One 0! = 1
2. One factorial is One 1! = 1*0! = 1
3. Two factorial is Two 2! = 2*1! = 2
4. Three factorial is Six 3! = 3*2! = 6

First step is creating a solution named `FactorialApplication`. And add a **test** project named `FactorialCalculatorTests` and a class library named `FactorialCalculator` in solution.



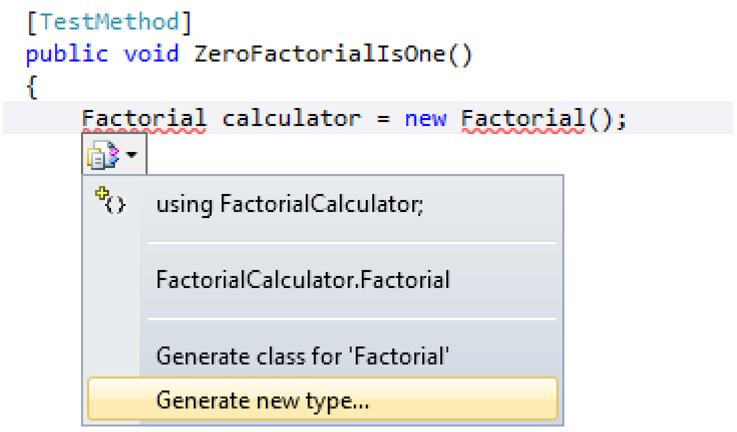
And now create a new class and write the first **test** method.

Test 1: Zero Factorial is One

Hide Copy Code

```
[TestMethod]
public void ZeroFactorialIsOne()
{
    Factorial calculator = new Factorial();
}
```

As you can see above, I created a `calculator` object that is of type `Factorial` but there is no class named `Factorial`. And you will get a lot of red squiggly lines under `Factorial` word.



```
[TestMethod]
public void ZeroFactorialIsOne()
{
    Factorial calculator = new Factorial();
```

using FactorialCalculator;
FactorialCalculator.Factorial
Generate class for 'Factorial'
Generate new type...

Let's create a class using options menu under the `Factorial` Word.

Hide Copy Code

```
[TestMethod]
public void ZeroFactorialIsOne()
{
    Factorial calculator = new Factorial();
    float result = calculator.GetFactorialOf(0);

    Assert.AreEqual(1, result);
}
```

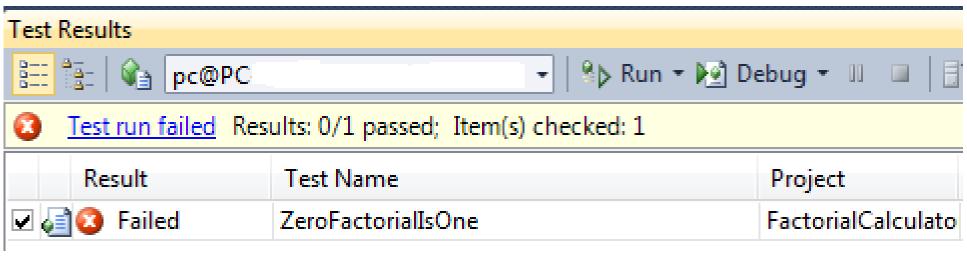
Yes, now I added a new `result` variable and assigned a `calculator.GetFactorialOf(0)` method result in it. But there is no method of `Factorial` in such. And I created a method in `Factorial` class.

Now we can have a look at the `Factorial` class.

Hide Copy Code

```
public class Factorial
{
    public float GetFactorialOf(int p)
    {
        throw new NotImplementedException();
    }
}
```

And now let's run the **test** method and see the result.



Test run failed Results: 0/1 passed; Item(s) checked: 1

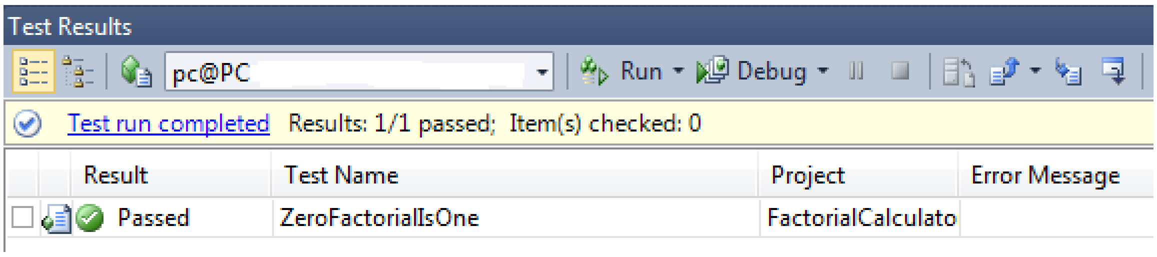
	Result	Test Name	Project
✓	Failed	ZeroFactorialIsOne	FactorialCalculato

The **test** failed because we created method `GetFactorialOf(int p)` but did not implement it as seen above. Now we should do minimum intervention to pass the **test**.

Hide Copy Code

```
public class Factorial
{
    public float GetFactorialOf(int p)
    {
        return 1;
    }
}
```

The **test** method can pass now. Because assertion has verified.



Test run completed Results: 1/1 passed; Item(s) checked: 0

Result	Test Name	Project	Error Message
Passed	ZeroFactorialIsOne	FactorialCalculato	

Test 2: One Factorial is One

And now, we can create our second method. New assertion is "One Factorial is One". The new **test** method is as follows:

Hide Copy Code

```
[TestMethod]
public void OneFactorialIsOne()
{
    Factorial calculator = new Factorial();
    float result = calculator.GetFactorialOf(1);

    Assert.AreEqual(1, result);
}
```

This method has passed as well because `GetFactorialOf(int p)` method still supports our assertion.

Test 3: Two Factorial is Two

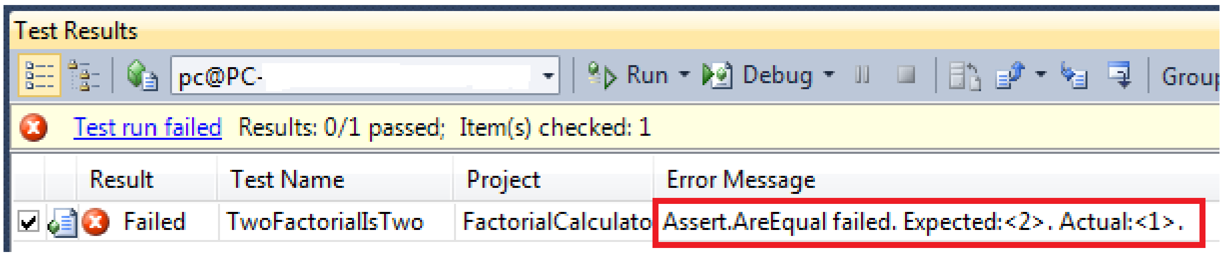
The third assertion is "Two factorial is Two".

Hide Copy Code

```
[TestMethod]
public void TwoFactorialIsTwo()
{
    Factorial calculator = new Factorial();
    float result = calculator.GetFactorialOf(2);

    Assert.AreEqual(2, result);
}
```

This method failed as follows:



Test run failed Results: 0/1 passed; Item(s) checked: 1

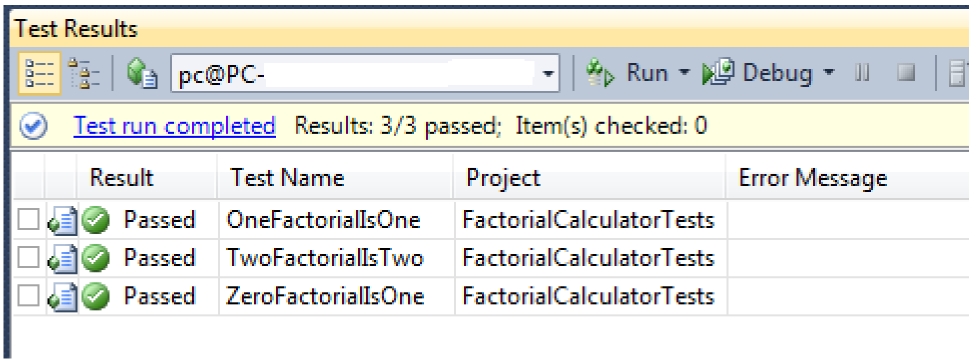
Result	Test Name	Project	Error Message
Failed	TwoFactorialIsTwo	FactorialCalculato	Assert.AreEqual failed. Expected:<2>. Actual:<1>.

Assert exception is shown, expected value is Two but method returned One. Now we should make minimum modification on `GetFactorialOf(int p)` method to pass the method. The modification shouldn't affect the other **test** results.

Hide Copy Code

```
public class Factorial
{
    public float GetFactorialOf(int p)
    {
        if (p < 2)
            return 1;

        return p;
    }
}
```


Test Results

pc@PC- Run Debug

Test run completed Results: 3/3 passed; Item(s) checked: 0

Result	Test Name	Project	Error Message
Passed	OneFactorialIsOne	FactorialCalculatorTests	
Passed	TwoFactorialIsTwo	FactorialCalculatorTests	
Passed	ZeroFactorialIsOne	FactorialCalculatorTests	

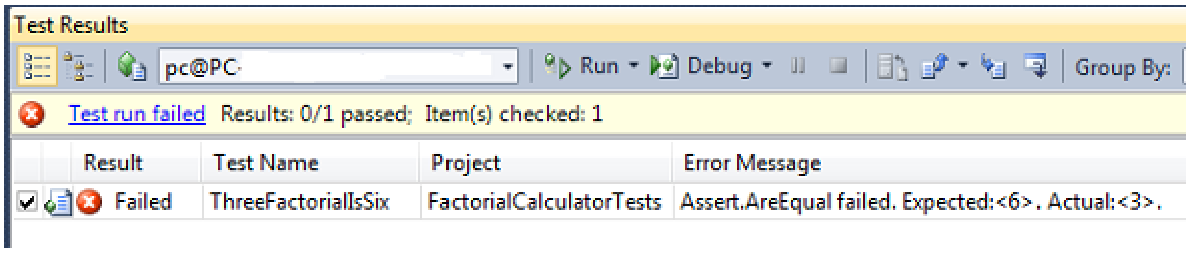
Test 4: Three Factorial is Six

Hide Copy Code

```
[TestMethod]
public void ThreeFactorialIsSix()
{
    Factorial calculator = new Factorial();
    float result = calculator.GetFactorialOf(3);

    Assert.AreEqual(6, result);
}
```

But now this assertion failed.


Test Results

pc@PC- Run Debug Group By:

Test run failed Results: 0/1 passed; Item(s) checked: 1

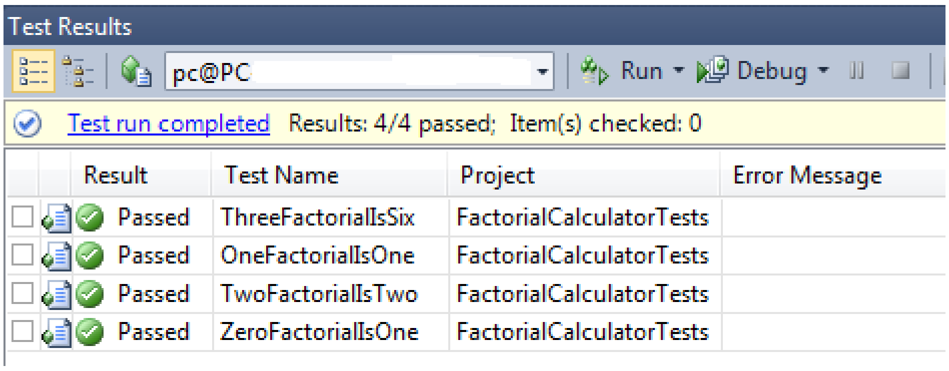
Result	Test Name	Project	Error Message
Failed	ThreeFactorialIsSix	FactorialCalculatorTests	Assert.AreEqual failed. Expected:<6>. Actual:<3>.

New modification is required on `GetFactorialOf(int p)` method.

Hide Copy Code

```
public float GetFactorialOf(int p)
{
    if (p < 2)
        return 1;

    return p * GetFactorialOf(p-1);
}
```

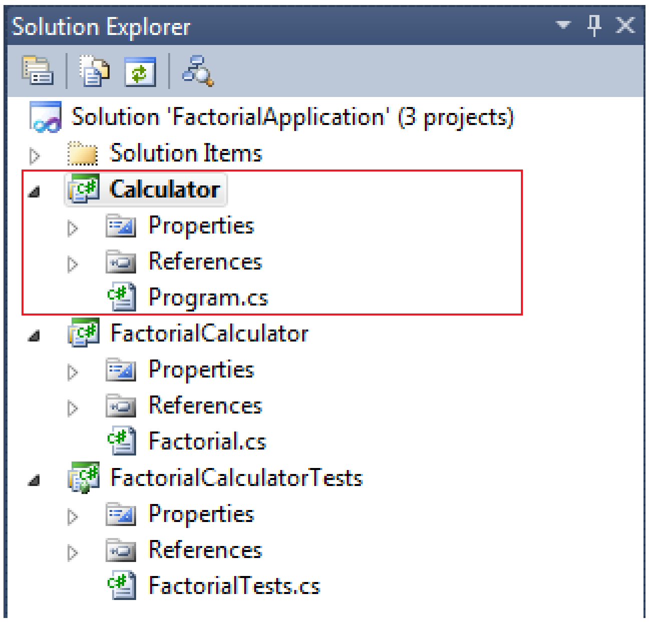

Test Results

pc@PC Run Debug

Test run completed Results: 4/4 passed; Item(s) checked: 0

Result	Test Name	Project	Error Message
Passed	ThreeFactorialIsSix	FactorialCalculatorTests	
Passed	OneFactorialIsOne	FactorialCalculatorTests	
Passed	TwoFactorialIsTwo	FactorialCalculatorTests	
Passed	ZeroFactorialIsOne	FactorialCalculatorTests	

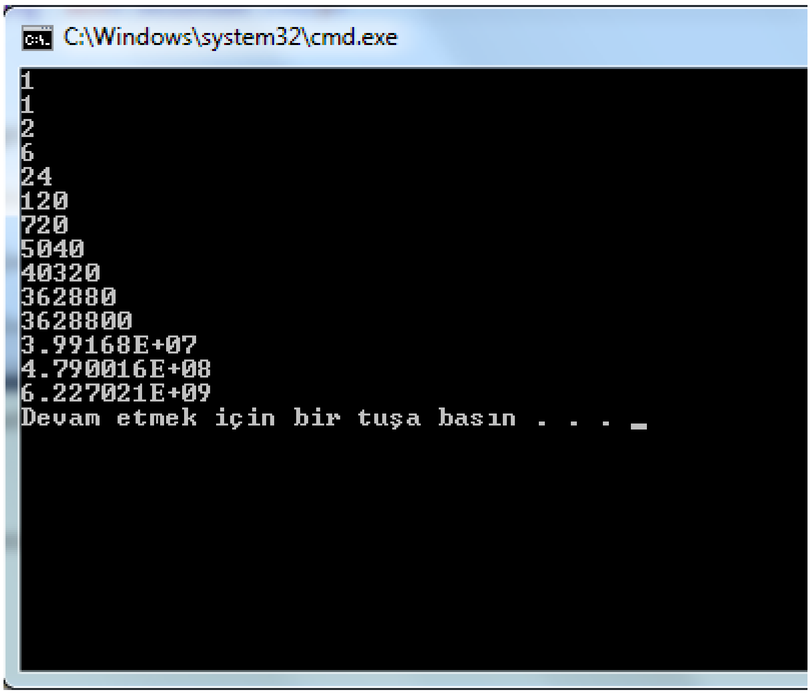
And the solution is as follows. Now we can use this `FactorialCalculator` library anywhere we want. I used in a class library named `Calculator`.



Hide Copy Code

```
class Program
{
    static void Main(string[] args)
    {
        Factorial calculator = new Factorial();

        for (int i = 0; i < 14; i++)
        {
            Console.WriteLine(calculator.GetFactorialOf(i));
        }
    }
}
```



```
1
1
2
6
24
120
720
5040
40320
362880
3628800
3.99168E+07
4.790016E+08
6.227021E+09
Devam etmek için bir tuşa basın . . .
```

Points of Interest

The most impressive aspect of TDD for me is that initially, there is no code and while **test** methods developing the required codes are created automatically. There is no need for planning the classes, no need to plan the variables and so on. Unnecessary operations are not implemented. We implement only requirements specification as needed. Everything is determined during the **test** period.

The other point is any **test** should not affect any other. And method implementations should not affect **test** results. For example, if one method is working correctly for one **test**, but working wrong other methods, then this is an undesired result.

Any changes of product methods require the running the **test**s. We did this during the application.

History

This article shows us how to create unit **test**s and steps of the unit **test**ing simply. And we developed an Factorial application. After the **test** methods, the mainclass generated and it is ready to use. If you can see the shortcomings of the application, you can modify the methods. But any change of application requires the running **test**s to be on.

License

This article, along with any associated source code and files, is licensed under The Code Project Open License (CPOL)

Share

@ EMAIL

About the Author

Bayram Üçüncü http://www.bayramucuncu.com
Software Developer
Turkey

You may also be interested in...

Test Driven / First Development by Example

Microsoft's Guide to Modern Dev/Test

Designing Application Using Test Driven Development

SAPrefs - Netscape-like Preferences Dialog

Learning Test Driven Development with TDD Katas

Generate and add keyword variations using AdWords API

Comments and Discussions

You must **Sign In** to use this message board.

Search Comments Go

First Prev Next

My vote of 4
pradiprenushe 27-Aug-12 1:42

good

Sign In · View Thread · Permalink

Harika!
Erol Esen 7-Feb-12 11:19

Şahane bir makale olmuş. Hem Visual Studio'da unit test nasıl yazılır, hem de unit test ile algoritma'nın kendisini evrimsel bir şekilde yazımını gösteriyorsunuz. Büyük bir zevkle okudum. Tebrikler, ve teşekkürler.

Erol

modified 7-Feb-12 17:34pm.

Sign In · View Thread · Permalink 5.00/5 (1 vote)

Re: Harika!
Bayram Üçüncü 7-Feb-12 19:11

Teşekkürler Erol Bey

Sign In · View Thread · Permalink

Re: Harika!
Bayram Üçüncü 24-Jul-12 21:22

Teşekkürler Erol Bey Faydalı olabildiysem ne mutlu.

Sign In · View Thread · Permalink

Re: Harika!
Erol Esen 1-Aug-12 3:54

Test driven yazılım hem yazılan programı doğrulamaya çalışıyor, hem de gereklileri. Verdiğiniz örnekte özyinelemeli mantığı TDD'nin ne kadar tabii bir şekilde yazıldığını da gösteriyor. Tekrar teşekkür ederim.

Erol

modified 1-Aug-12 16:05pm.

Sign In · View Thread · Permalink

A few comments

PeteBarber 29-Jan-12 22:43

1. The first test should really be "can I create an instance of Factorial?" rather than assuming you can.
2. In the final program use prefix increment. Only ever use postfix increment when you need too, i.e. when the consumer of the return value needs the pre-incremented value.

Sign In · View Thread · Permalink

Re: A few comments

Bayram Üçüncü 30-Jan-12 20:13

Thanks PereBarber.
Yes the fitst test could be "Can I create an instence of factorial". But I think the consumer program can be any name, or it can be a part of a Math class or so on....

Sign In · View Thread · Permalink

My vote of 5

Shahin Khorshidnia 29-Jan-12 9:22

Noteworthy solution.

Sign In · View Thread · Permalink

Re: My vote of 5

Bayram Üçüncü 30-Jan-12 20:09

Thanks Shahin

Sign In · View Thread · Permalink 5.00/5 (1 vote)

Re: My vote of 5

Shahin Khorshidnia 30-Jan-12 21:40

sağ ol 😄

Do not criticise, if you don't have a better idea.

Sign In · View Thread · Permalink 1.00/5 (1 vote)

real tips

moiurbd 27-Jan-12 18:52

this really useful and informative tips , for more details information . you can check here .

small business technology

Sign In · View Thread · Permalink 1.00/5 (7 votes)

Refresh 1

General News Suggestion Question Bug Answer Joke Praise Rant Admin

Permalink | Advertise | Privacy | Terms of Use | Mobile
Web02 | 2.8.160621.1 | Last Updated 28 Jan 2012

Layout: fixed | fluid

Article Copyright 2012 by Bayram Üçüncü
Everything else Copyright © CodeProject, 1999-2016