C# METHODS & TEST DRIVEN DEVELOPMENT OR TDD

PRESENTATION OVERVIEW

- This presentation is about methods and creating unit tests for them to accomplish the basic building block for test driven development (TDD).
- In classic TDD, each new functional requirement began with writing a unit test method.
- Unit Testing is a level of software testing where individual units or components of a program are tested.
- The purpose is to validate that each unit of the software performs as designed: the unit is the smallest testable part of a program like a method.
- A method usually has one or a few inputs with ideally a single output.
- In this presentation, we will create a good method and then the unit test for it.

INTRODUCTION TO METHODS AND TDD

- A method is a block of code in Object Oriented Programming (OOP) that performs a set of C# statements to implement the logic required for a required specific behavior or operation for a class.
- A method was called a function in procedural programming like C or COBOL.
- A programmer defines a method by first identifying the problem or task it must solve.
- The method should have a single purpose to make it easily debugged and unit tested for TDD.
- The programmer should specify the method inputs and outputs.
- The programmer should state any assumptions about the method like expected inputs and outputs.
- Finally, the programmer should consider and handle any errors that could occur in the method.

BUILDING A METHOD

- A method consists of a signature defined as the name, parameters, and return type.
- A method body implements a required functionality in code statements.
- A method return statement defines the type of output required for it.

METHOD SIGNATURE

```
public bool PlaceOrder(Product product, int quantity)
```

- Public is the accessibility modifier, and if not specified, default is private.
- Bool is the return type if required. No return type is specified with a void.
- PlaceOrder is the method name in Pascal Casing (public). I use Camel if private. Method name usually starts with a verb and the object noun processed by it.
- product and quantity are the method parameters with defined types. Product is a business object and quantity is an integer. Empty () specifies no parameters.

ADDING XML COMMENTS TO METHOD

- When you enter "///" in from of the method signature in VS 2015, you will get the summary displayed in intellisense:
- Don't forget to finish the summary.
- Use Sandcastle to create XML

  Help document (See References).

```
/// <summary>
///
/// </summary>
/// <param name="product"></param>
/// <param name="quantity"></param>
/// <returns></returns>
0 references
public bool PlaceOrder(Product product, int quantity)
{

    bool isOkay = false;

    return isOkay;

}
```

XML COMMENTS (CONTINUED)

- Finishing the XML comments:

```
/// <summary>
/// Method to place order for product and quantity
/// </summary>
/// <param name="product">Business object with product properties</param>
/// <param name="quantity">Quantity ordered</param>
/// <returns>bool true if placed and false if not</returns>
0 references
public bool PlaceAnOrder(Product product, int quantity)
{
```

METHOD BODY

```
public bool PlaceOrder(Product product, int quantity)
{
    bool isOkay = false;
    //code to validate an order for a product
    if (product == null ) throw new ArgumentNullException(nameof(Product));
    if (quantity <= 0) throw new ArgumentOutOfRangeException(nameof(quantity));
    var emailService = new EmailService();
    var orderText = "Order from Blah, Inc.: " + Environment.NewLine +
        "Product: " + product.ProductName + Environment.NewLine +
        "Quantity: " + quantity.ToString();
    var confirmation = emailService.SendMessage("New Order", orderText, this.Email);
    if (confirmation.StartsWith("Message sent:"))
    {
        isOkay = true;
    }
    return isOkay;
}
```

PSEUDO CODE TO SEND AN EMAIL

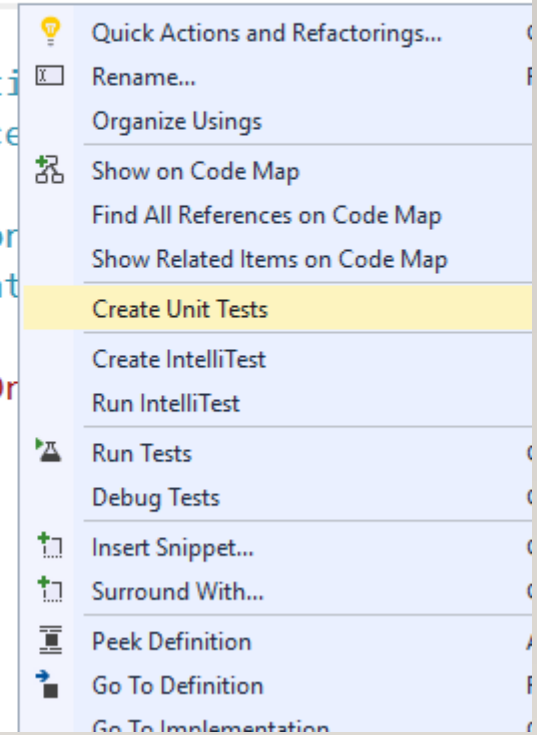
Code:

```
/// <summary>
/// Sends email message
/// </summary>
/// <param name="subject">Subject of the message.</param>
/// <param name="message">Message text</param>
/// <param name="recipient">Email address of the message recipient.</param>
/// <returns></returns>
5 references | 0/1 passing
public string SendMessage(string subject, string message,
                          string recipient)
{
    // Code to send an email

    var confirmation = "Message sent: " + subject;

    return confirmation;
}
```

BUILDING THE UNIT TEST

Right click in the method and click on "Create Unit Tests":

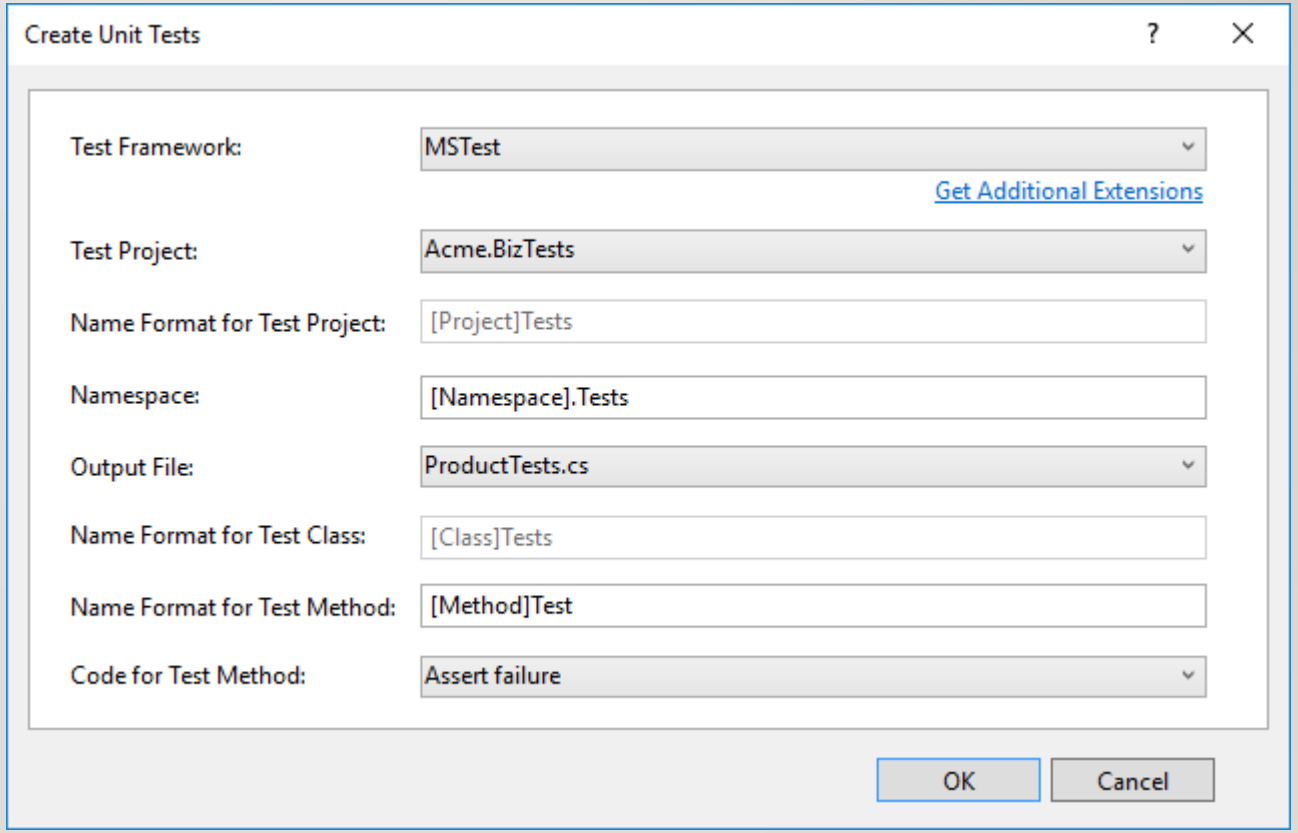
```
public bool PlaceOrder(Product product, int quantity)
{
    bool isOkay = false;
    //code to validate an order for a product
    if (product == null ) throw new ArgumentNullExcepti
    if (quantity <= 0) throw new ArgumentOutOfRangeExce
    var emailService = new EmailService();
    var orderText = "Order from Blah, Inc.: " + Enviror
        "Product: " + product.ProductName + Environment
        "Quantity: " + quantity.ToString();
    var confirmation = emailService.SendMessage("New Or
    if (confirmation.StartsWith("Message sent:"))
    {
        isOkay = true;
    }
    return isOkay;
}
```



The "Create Unit Tests"

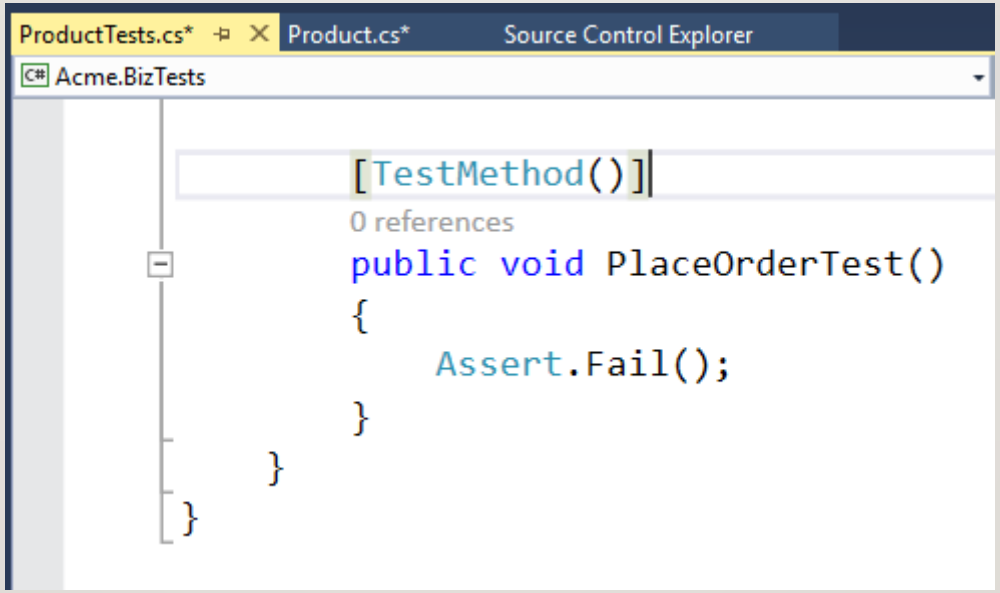
Form is displayed.

Click OK:



UNIT TEST FRAMEWORK CODE CREATED

The test framework outline created:



PARTS OF A UNIT TEST

- Arrange
- Act
- Assert

```
[TestMethod()]
0 references
public void PlaceOrderTest()
{
    //Arrange


    //Act


    //Assert

}
```

SET UP ARRANGE AND ACT IN UNIT TEST

Arrange and Act setup:

(Notice the intellisense hovering over PlaceOrder) from XML comment:

```
[TestMethod()]
0 references
public void PlaceOrderTest()
{
    //Arrange
    var product = new Product(1, "Saw", "Description");
    var expected = true;
    //Act
    var actual = product.PlaceOrder(product, 2);

    //Assert

}
```

```
bool Product.PlaceOrder(Product product, int quantity)
Method to place order for product and quantity
```

CODE THE ASSERT SECTION

Use Assert.AreEqual test method to compare expected with actual results

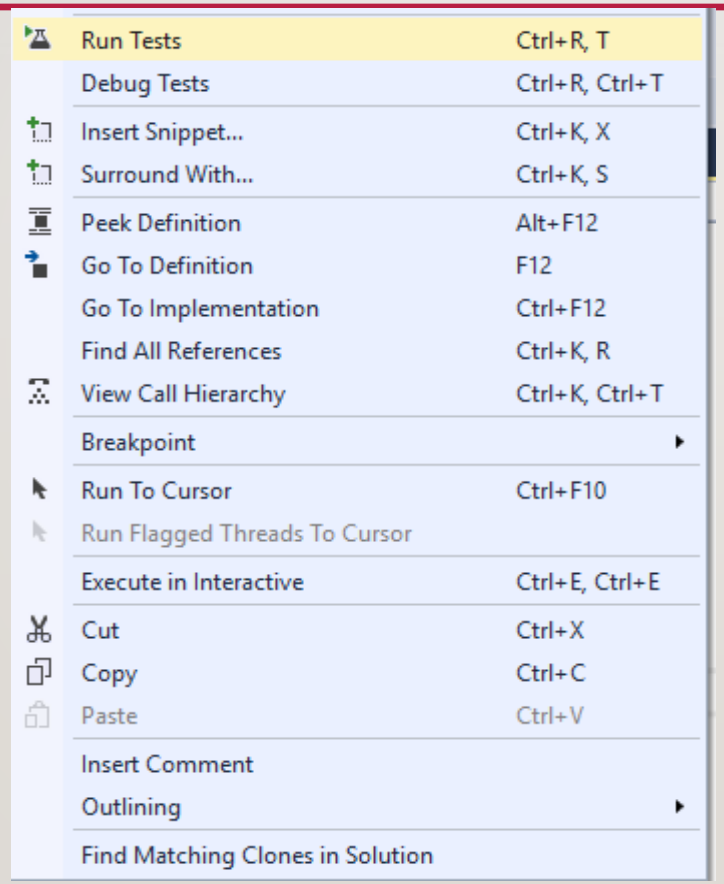
```
[TestMethod()]
    0 references
public void PlaceOrderTest()
{
    //Arrange
    var product = new Product(1, "Saw", "Description");
    var expected = true;
    //Act
    var actual = product.PlaceOrder(product, 2);

    //Assert
    Assert.AreEqual(expected, actual);
}
```

RUN THE TEST METHOD

Right click inside

the test method

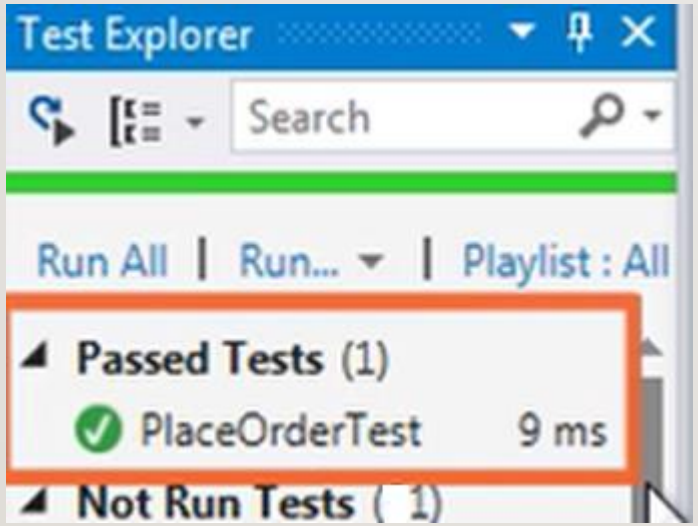
and run the test



THE TEST SHOULD BE SUCCESSFUL

Test Explorer

Successful test



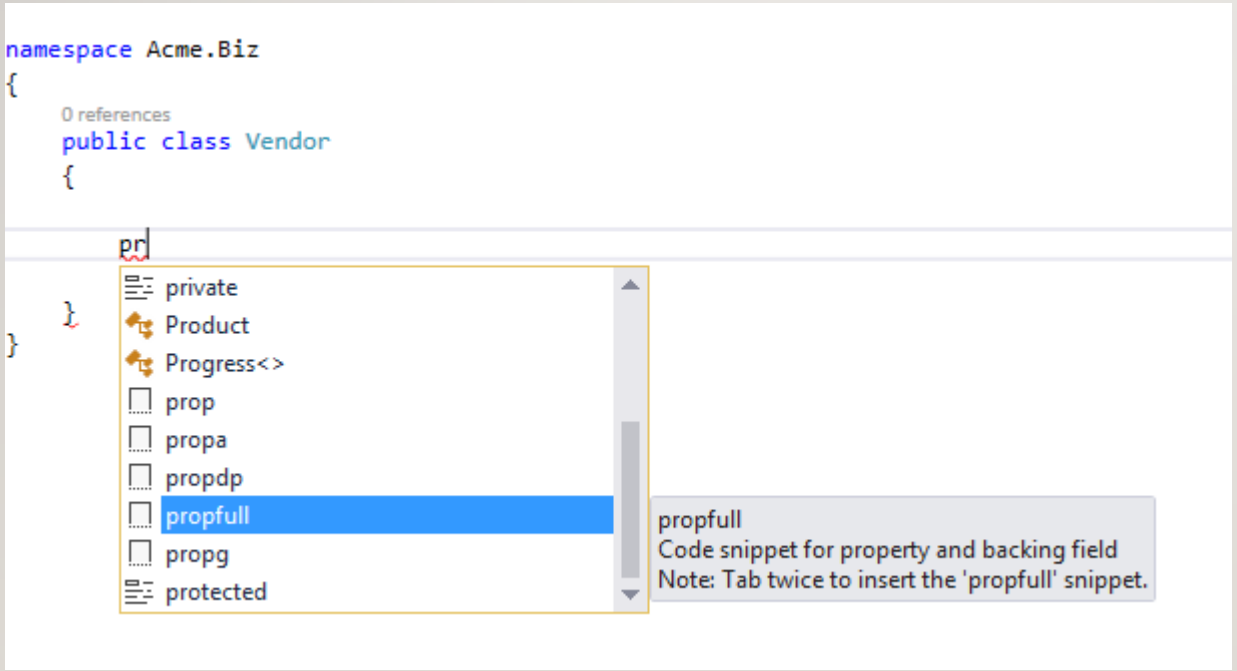
TEST ITSELF SHOWS SUCCESSFUL RUN

- Success marked
- with check mark:

```
[TestMethod()]
0 references
public void PlaceOrderTest()
{
    //Arrange
    var product = new Product(1, "Saw", "Description");
    var expected = true;
    //Act
    var actual = product.PlaceOrder(product, 2);

    //Assert
    Assert.AreEqual(expected, actual);
}
```

BONUS – AUTO CREATION OF FIELDS & PROPERTIES WITH VS 2015 AND C# 6

- Create class, enter prop and tab, tab and select propfull:



OOP BEST PRACTICE PRIVATE FIELD AND PUBLIC PROPERTY CREATED (DATA HIDING ENFORCED)

- Change to suit needs

```
namespace Acme.Biz
{
    0 references
    public class Vendor
    {

        private int myVar;

        0 references
        public int MyProperty
        {
            get { return myVar; }
            set { myVar = value; }
        }


    }
}
```

SUMMARY

- This presentation was about methods and creating unit tests for them to accomplish the basic building block for test driven development (TDD).
- Demonstrated difference between classic TDD and newer C# 6, VS 2015 method creation and test case for it.
- In this presentation, we created a good method and then created the unit test for it.
- Added a bonus field/property OOP best practice.

REFERENCES

- https://app.pluralsight.com/library/courses/csharp-best-practices-improving-basics/table-of-contents
- https://github.com/EWSoftware/SHFB

NEXT 3 INTERVIEW TOPICS

- C# 6 Collections and Generics
- C#6 LINQ
- SOLID (Single responsibility, Open closed, Liskov substitution, Interface segregation, and Dependency inversion) Principles of Class Design in Object Oriented Programming.
- Design Patterns for OOP