10 Tips to Boost Your Productivity with C# and Visual Studio 2008

johnwpowell **23 Mar 2008 7:48 AM** 27

Learn Key Bindings

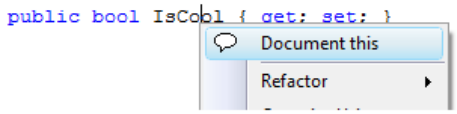
It's an obvious and trivial thing, but the timesaving will add up, especially for those actions you perform tens or hundreds of times a day such as building and debugging. Here are some basic bindings every Visual Studio developer should know:

- Build: **CTRL** + **SHIFT** + **B**
- Word completion: **CTRL** + **SPACE**
- Start with debugging: **F5**
- Start without debugging: **CTRL** + **F5**

Even expert Visual Studio developers can benefit by learning new bindings. Download the Visual C# 2008 Keybinding Reference Poster and hang it in your work area.

Generate XML Comments with GhostDoc

Instead of typing XML comments by hand, let a tool do the work for you. Although macros and snippets are reasonably effective for this, I would recommend Ghost Doc over any other solution. This *free* add-in uses customizable templates to generate consistent, English-readable documentation based on the current context. To use it, right-click (or use CTRL + SHIFT + D) to document the current element. For example:



This generates the following documentation (note GhostDoc split the property name into words and created a sentence from it):

```
/// <summary>
/// Gets or sets a value indicating whether this instance is cool.
/// </summary>
/// <value><c>true</c> if this instance is cool; otherwise, <c>false</c>.</value>
public bool IsCool { get; set; }
```

Auto-Implement Properties

Take advantage of a new feature of C#: auto-implemented properties. Rather than creating a private backing field for your properties, let the compiler do it for you. The following demonstrates the syntax:

```
/// <summary>
/// Gets or sets my property.
/// </summary>
/// <value>My property.</value>
public string MyProperty { get; set; }
```

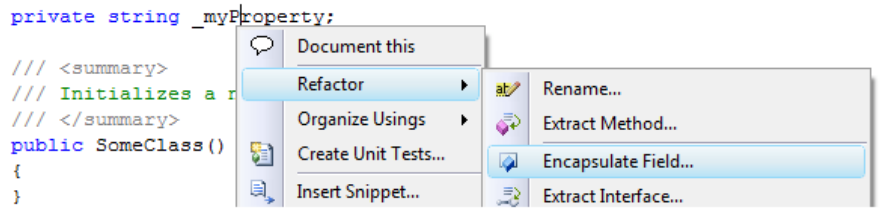
Use the code snippet to make this even faster. Type **prop** (the shortcut for an auto-implemented property) followed by **TAB TAB**. Then fill in the data type and property name:

```
public int MyProperty { get; set; }
```

Refactor

The refactor feature in Visual Studio is indispensable for many tasks, especially renaming, but one productivity feature I particularly like is *Encapsulate Field*. If you are unable to use an auto-implemented property, declare a

private field and let Visual Studio generate the Property for you. To use this feature, right-click on the field and select *Refactor* > *Encapsulate Field...*



The property is created for you:

```
private string _myProperty;

public string MyProperty
{
    get { return _myProperty; }
    set { _myProperty = value; }
}
```

Add Commands to Visual Studio 2008

Install the PowerCommands for Visual Studio 2008 to add several productivity commands such as:

- Close all documents
- Copy and paste a class (automatically renames)
- Remove and sort using statements project-wide
- Copy and paste references (including a project reference)

Install the Team Foundation Server Power Tools to add several TFS productivity commands such as:

- Find in source control
- Open source folder in Windows Explorer
- Work item templates (can be used to set values on multiple work items at once)

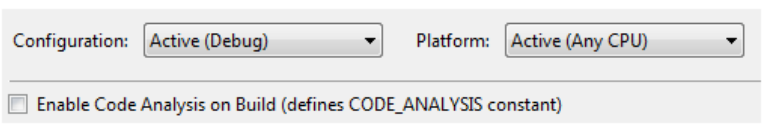
Add your own productivity commands. For example, to add Reflector so it automatically opens on the current project.

- Select Tools > External Tools
- Click Add
- Name it Reflector and browse to the executable
- Enter $(TargetPath) for the Arguments

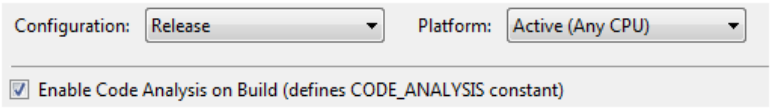
Speed up Compilation with Project Configuration

You may build tens of times during a programming session, so don't enable anything that isn't absolute necessary such as code analysis and XML documentation. Develop in Debug configuration, and switch to Release configuration just before check-in to run code analysis and generate XML documentation. In a large solution I recently worked on, this shaved a minute off compilation time.

The following shows code analysis *disabled* in Debug configuration:

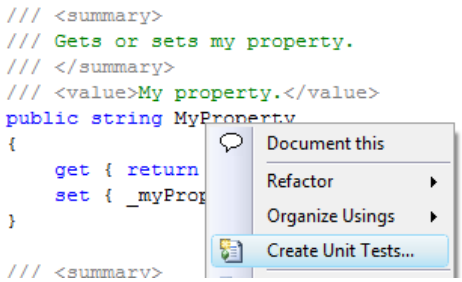


The following shows code analysis *enabled* in Release configuration:



Let Visual Studio Generate Unit Test Code

Although it can't fully automate unit testing yet (check out Pex), Visual Studio does a good job of generating positive unit test code to give you a jump start. To use this feature, right-click on an element you would like to test and select *Create Unit Tests...*



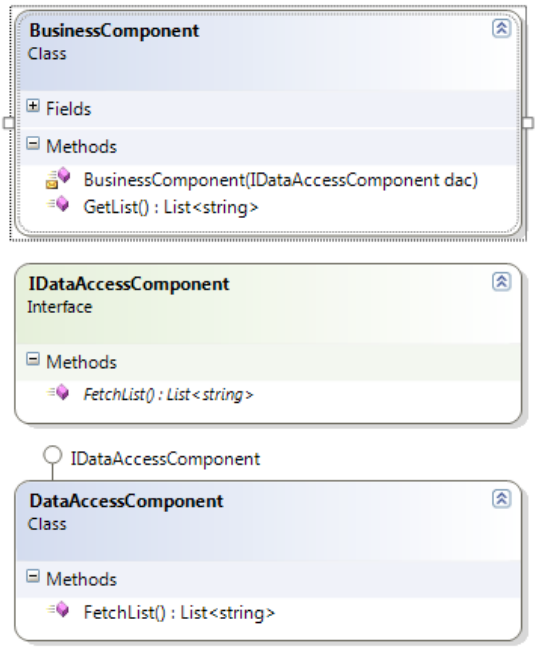
Visual Studio generates the following test method:

```
/// <summary>
///A test for MyProperty
///</summary>
[TestMethod()]
public void MyPropertyTest()
{
    SomeClass target = new SomeClass(); // TODO: Initialize to an appropriate value
    string expected = string.Empty; // TODO: Initialize to an appropriate value
    string actual;
    target.MyProperty = expected;
    actual = target.MyProperty;
    Assert.AreEqual(expected, actual);
    Assert.Inconclusive("Verify the correctness of this test method.");
}
```

Use Interface-Driven-Design

You probably never thought of an interface as a productivity feature, but it can be if your development process is driven by contracts instead of implementation. Let me illustrate with a simplified example. One developer owns the business layer and another developer owns the data access layer, and they need to agree on how to communicate to implement a new feature. In some business object designs, business components will instantiate data components (or call a static method). This is a problem from a design standpoint because the two become tightly coupled. It is also a problem from the productivity standpoint because the business layer developer becomes dependent on the data access layer developer's implementation. *Interface-driven design (IDD) solves this issue*.

Rather than the business component developer waiting on the data component developer, they meet to design and implement the interface. Both developers are then free to go their separate ways and implement the components in parallel. IDD also enables the business developer to mock the data access component, thereby removing any scheduling dependencies. The following illustrates the design:



Make a Mockery of Dependencies

Different developers may own layers or features that your component is dependent on, but don't let that slow you down. Suppose you are responsible for the business layer which depends on the data access layer which in turn depends on database tables and stored procedures. Rather than waiting for dependent layers to be completed, mock the data access layer so you can implement and unit test the business layer. By the way, you should be using mocks anyway; otherwise your unit tests are more than likely integration tests. I recommend Rhino Mocks.

Here is a sample unit test with mocks:

```
/// <summary>
///A test for GetList
///</summary>
[TestMethod()]
public void GetListTest()
{
    // Create the mock repository
    MockRepository mockery = new MockRepository();

    // Create the list that will be returned from the data access component
    List<string> expected = new List<string>() { "customer1", "customer2"};

    // Mock the data access component
    IDataAccessComponent dac = mockery.CreateMock<IDataAccessComponent>();

    // Set expectations and return value of the mocked data access method
    Expect.Call(dac.FetchList()).Return(expected);

    // Move from Record state to Replay state
    mockery.ReplayAll();

    // Create the business component passing in the mocked data access component
    BusinessComponent target = new BusinessComponent(dac);

    // Execute the text on the business component
    List<string> actual = target.GetList();

    // Verify all expected calls on the data access component were met
    mockery.VerifyAll();

    // Verify test results
    Assert.AreEqual(expected, actual);

}
```

Data Drive Unit Tests

If you have a unit test that multiple inputs to fully test, you *could* write a test method for every possible

combination, but data-driven unit tests are more efficient. When the unit test is run, it loads data from a table and calls the unit test for each row. You can access the data in the current row using the TestContext.DataRow property.

```
/// <summary>
///A test for GetList
///</summary>
[TestMethod()]
[DeploymentItem("TestProject1\\unit_test.mdf")]
[DataSource(
    "System.Data.SqlClient",
    "Data Source=.\\sqlexpress;AttachDbFilename=|DataDirectory|\\unit_test.mdf;Integrated Security=True;User Instance=True",
    "Names", DataAccessMethod.Sequential)]
public void DataDrivenGetListTest()
{
    BusinessComponent target = new BusinessComponent();
    target.GetList(testContextInstance.DataRow["LAST_NAME"] as string);
}
```

Once the test completes, you can view the results:

Data Driven Test Results: 4 of 4 passed

Result	Duration	Data Row	Error Message
Passed	00:00:00.0165435	0	
Passed	00:00:00.0000625	1	
Passed	00:00:00.0000220	2	
Passed	00:00:00.0000205	3	

Comments

Frank W Holt Jr 24 Mar 2008 9:57 AM

Great blog! I especially like the 'Data Driven Testing' piece. I'm getting into this area and have been running a script for each test to prepare the data. I'll give this a shot. Thanks.

Hosam Kamel 24 Mar 2008 11:27 AM

Some tips posted by John W Powell about how to increase your development productivity using Visual Studio

Uwe Keim 24 Mar 2008 12:33 PM

More tips:

11. Use ReSharper (www.jetbrains.com)

12. Use SonicFileFinder (sonicfilefinder.jens-schaller.de - from the co-worker of the GhostDoc author)

John Powell 24 Mar 2008 9:26 PM

Thanks for the feedback!

Unfortunately ReSharper isn't available for Visual Studio 2008 (yet)

Morgan Cheng 24 Mar 2008 10:00 PM

This post is really helpful!!!

Karl Agius 26 Mar 2008 12:02 PM

Excellent advice and references, thanks!!!

Derik Whittaker 26 Mar 2008 3:30 PM

Let Visual Studio Generate Unit Test Code -- Makes my skin crawl.

Why, because if you allow test to be generated you may not fully understand the intent of the test. Making it almost useless.

Don't get me wrong, having generated tests is better than NO tests at all, but this is the lazy mans version of testing if you ask me.

http://devlicio.us/blogs/derik_whittaker/archive/2008/03/26/39701.aspx

ScottGu's Blog 28 Mar 2008 4:08 AM

Here is the latest in my link-listing series . Also check out my ASP.NET Tips, Tricks and Tutorials

BusinessRx Reading List 28 Mar 2008 4:16 AM

Here is the latest in my link-listing series . Also check out my ASP.NET Tips, Tricks and Tutorials

Mirrored Blogs 28 Mar 2008 4:52 AM

Here is the latest in my link-listing series . Also check out my ASP.NET Tips, Tricks and Tutorials

Hüseyin Tüfekçilerli 28 Mar 2008 5:32 AM

To generate XML Comments for fields, methods, properties, etc. you can tap forward slash 3 times while your cursor was just before the declaration line. Like this (| represents your cursor):

|

public bool IsCool { get; set; }

->

///|

public bool IsCool { get; set; }

->

/// <summary>

/// |

/// </summary>

public bool IsCool { get; set; }

This is built-in to Visual Studio.

Laurent Duveau 28 Mar 2008 10:26 AM

Very interesting.

Regarding the prop snippet I reported a problem on Connect:

http://weblogs.asp.net/lduveau/archive/2008/03/22/code-snippets-not-versioned-according-to-vs2008-multi-targeting.aspx

Ricardo 28 Mar 2008 1:35 PM

The prop snippet in VS 2008 is not very efficient, at least not as good as the one in 2005, which would actually generate the get and set bodies. I ended up copying the one from 2005 to 2008 and that's the one I use!

Joycode@Ab110.com 29 Mar 2008 1:33 AM

【原文地址】March 28th Links: ASP.NET, ASP.NET AJAX, ASP.NET MVC, Visual Studio, Silverlight, .NET【原文发表日期

Kyralessa 29 Mar 2008 12:13 PM

Most of those tips look handy (though most I already knew), but that GhostDoc thing looks awful. Auto-generated "documentation" that documents IsCool with "Gets or sets a value indicating whether this instance is cool" is worse than useless. It's worse because it's a time-waster: You consult the documentation for help, and all it does is waste your time telling you something you already knew from the name.

Better to have *no* documentation than to have reams of pointless "documentation" like that.