CHAPTER 2

The Visual Studio IDE and MEF

This release of Visual Studio sees the IDE overhauled and much of it rewritten using WPF and managed code. The move to WPF allows Microsoft to make some stunning aesthetic additions to the IDE, and also opens up customization possibilities when combined with the new Managed Extensibility Framework (MEF).

Microsoft's use of WPF for a flagship product such as Visual Studio is important, as this demonstrates its commitment to the framework and confidence in its maturity.

In this chapter I will begin by looking at some of the new productivity enhancements in VS2010. I will then create a code snippet and customize the start page. Finally I will introduce MEF and take a look at some of the advanced customizations that it enables.

64-BIT VERSION OF VISUAL STUDIO?

A common question is whether Microsoft will release a 64-bit version of Visual Studio. At the time of writing, Microsoft has said it has no plans to do, for the following reasons:

- Making use of lazy loading techniques would be a more cost-efficient way to improve the IDE's performance, and would benefit 32-bit users as well.
- A 64-bit version could adversely affect performance because data structures will use more memory.
- There are cost issues. Rico Mariani (see the following link) suggests that the cheapest way to provide 64-bit support would be to incrementally convert the IDE to managed code, but this would break many existing extensions.

For a detailed discussion of this issue, please refer to the following link: `http://blogs.msdn.com/ricom/archive/2009/06/10/visual-studio-why-is-there-no-64-bit-version.aspx`.

General Improvements

VS2010 contains some long-awaited changes, including the following:

- There is now support for multiple monitors and the ability to drag windows outside the IDE (see Figure 2-1).
- IntelliSense is now two to five times quicker than previous versions.
- Readability of text is improved.

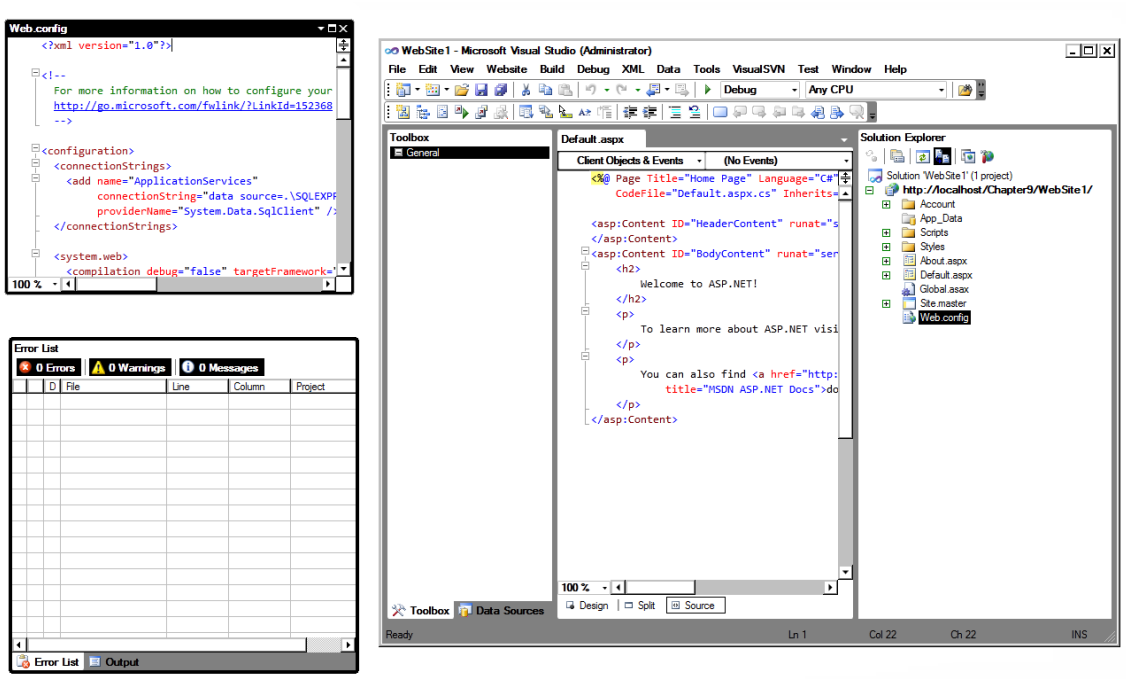

Figure 2-1. *VS2010 allows you to drag windows outside the IDE.*

Improved Multitargeting Support

When a new version of Visual Studio/.NET is released, it can take time to upgrade and test existing applications. This can prevent you from taking advantage of features such as IDE enhancements if you are not ready to upgrade your application yet. VS2010 contains improved support for targeting previous versions of the framework.

As per previous studio releases, the New Project dialog contains a drop-down menu that allows you to select the version of the framework that you are targeting when creating an application (see Figure 2-2). When you make a selection, Visual Studio will filter the project types you can create to those available in that version of the framework. Note that you can also select the framework version you are targeting in the project properties.

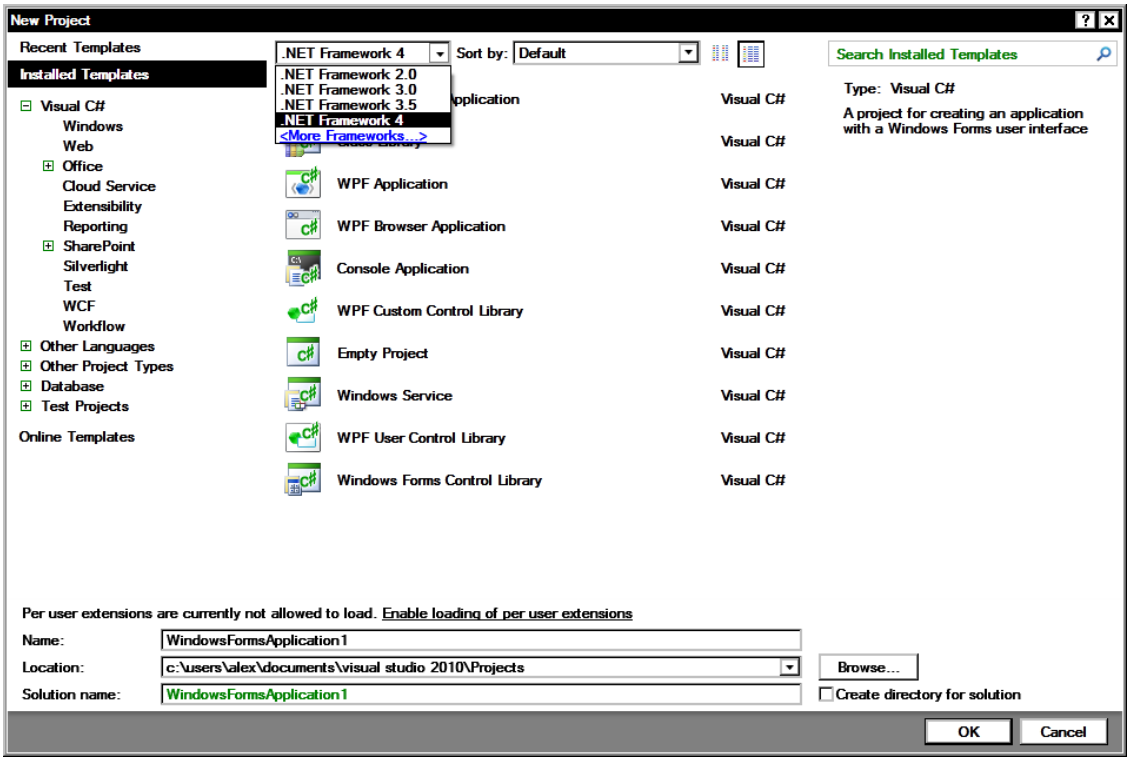

Figure 2-2. *Select the version of the .NET Framework to target from the drop-down menu.*

In VS2010 the Toolbox and Properties windows are filtered to display functionality available in the targeted framework version. Previously, some properties that were not available in the targeted framework would still be exposed. VS2010 will even try to display the correct version of third-party components for the targeting framework version.

VS2010 emulates what is available in previous framework versions through *reference assemblies*. These assemblies contain metadata that describes functionality available in previous versions. VS2010 itself uses the .NET 4.0 Framework, so when adding multitargeting support the team decided against running a previous framework version inside the same process. When your application is compiled to guarantee application compatibility, previous compiler versions are used.

■ **TIP** You may be interested in the ability to specify that your application should be run using a specific version of the framework. I discuss this in Chapter 3.

IntelliSense

IntelliSense will now perform partial string matching. For example, if you were to type the word *build*, Visual Studio would display both the `StringBuilder` and `UrlBuilder` options. This can be very useful if you cannot remember the exact property or type name.

IntelliSense also supports lookups based on capitalization. Because all .NET types are Pascal case, you can simply enter just the uppercase letters of the type. For example, entering **SB** would return the type `StringBuilder`, among others with the same Pascal casing (as shown in Figure 2-3). IntelliSense performance has also been improved, particularly for JavaScript libraries.

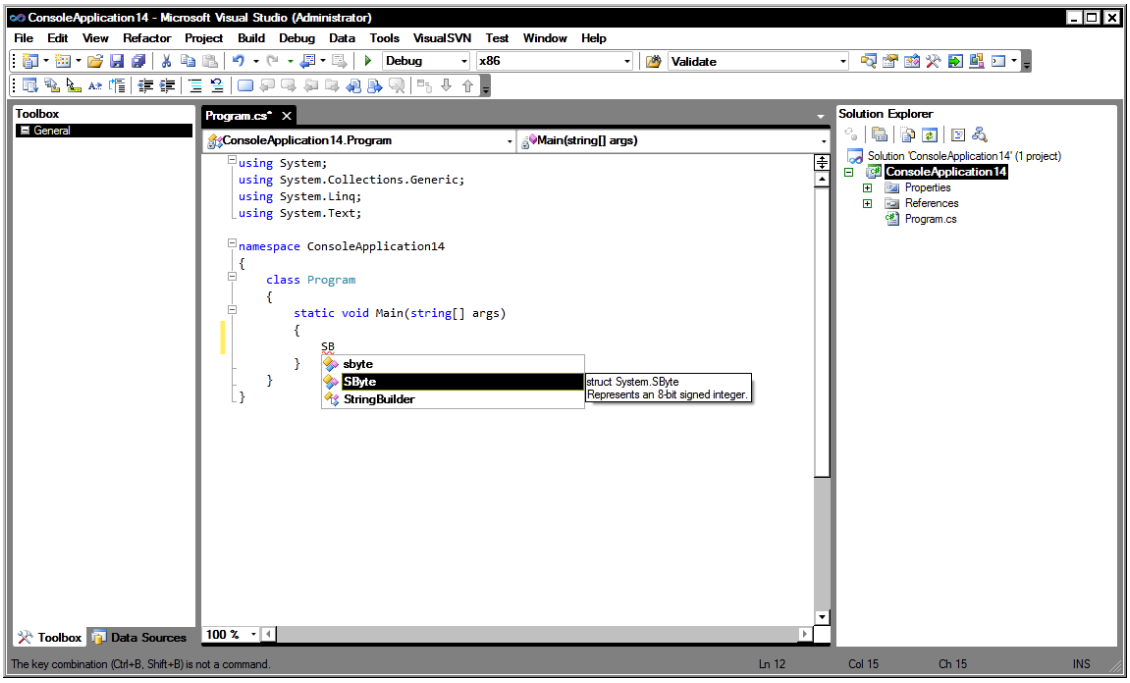

Figure 2-3. *IntelliSense supports partial string matching.*

Add References

Adding a reference to a project was previously very slow. In VS2010 it is now lightening quick. When the Add Reference dialog first displays, the focus is set to the Projects tab while separate threads load up the .NET and COM tabs.

Web Development (Code-Optimized Profile)

VS2010 contains some environment profiles, including the Web Development (code-optimized) profile shown in Figure 2-4. This profile is optimized for code and hides design features. You can select

the code-optimized profile when you first load Visual Studio or by selecting Tools ➤ Import, and then selecting Export Settings.

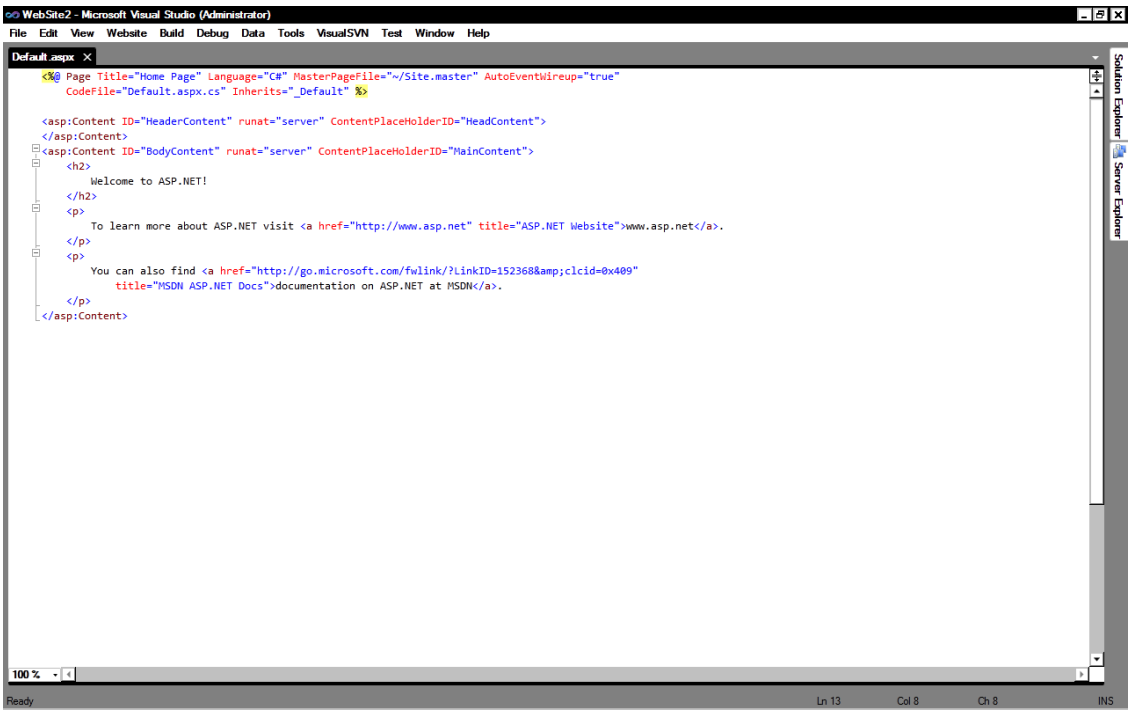

Figure 2-4. *VS2010 contains environmental profiles, such as the Web Development (code-optimized) profile.*

Zoom

As much of the IDE is written in WPF, it was easy for Microsoft to add functionality such as the ability to zoom into the code editor (as shown in Figure 2-5). To zoom into the code editor window, simply press Ctrl and use the mouse wheel to increase and decrease the zoom level. You can utilize this feature in presentations/code reviews or to zoom out to help you navigate a lengthy piece of code.

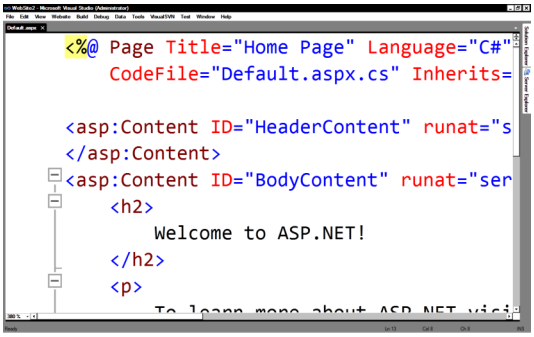

Figure 2-5. *VS2010 includes the ability to zoom into the code editor window.*

Highlight References

Highlight References allows you to quickly navigate through different instances of the same method call within a file. For example, if you want to navigate through all calls to the `ToString()` method, then click once anywhere on the `ToString()` method text (note that you don't have to highlight the text), and you will find that the IDE marks all the other `ToString()` calls in the same file with a light-gray background, as shown in Figure 2-6. You can then navigate to the next `ToString()` method by pressing Ctrl+Shift+Down or Ctrl+Shift+Up to return to the previous instance.

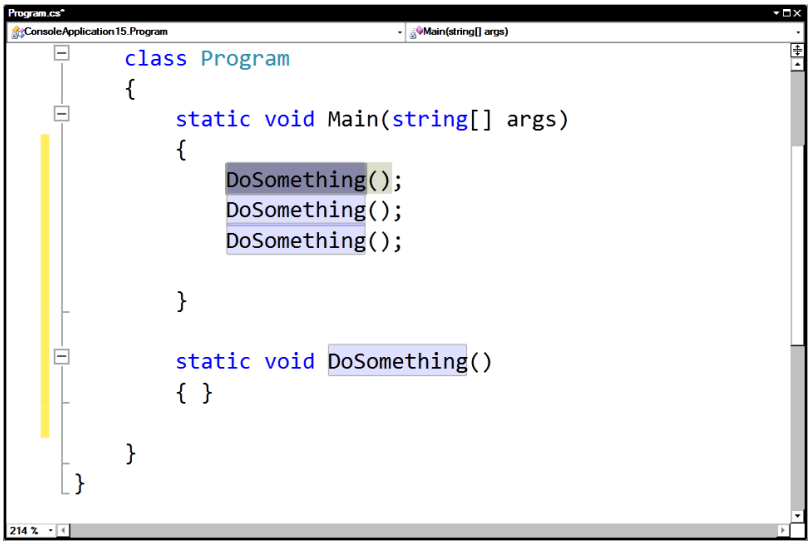

Figure 2-6. *Highlight References allows you to quickly move between calls to the same method.*

Navigate To

Sometimes when you need to find a specific piece of code, it can be much quicker to use the search functionality than to trawl through Solution Explorer. VS2010 improves on the existing search and search-in-files functionality with the Navigate To window.

To bring up the Navigate To window, simply press Ctrl+comma or select Navigate To on the Edit menu. You can then enter a phrase you want to search for in your solution, and Navigate To will immediately filter results as you type, as shown in Figure 2-7. You can then click these results to be taken directly to the results location. Navigate To will perform partial and in-string matches, and also supports Pascal casing searches (e.g., typing **BT** would return a class called `BigTiger`).

Navigate To supports all the commonly used file types, including C#, Visual Basic (VB), and XML, and is much quicker and easier to navigate than previous search methods.

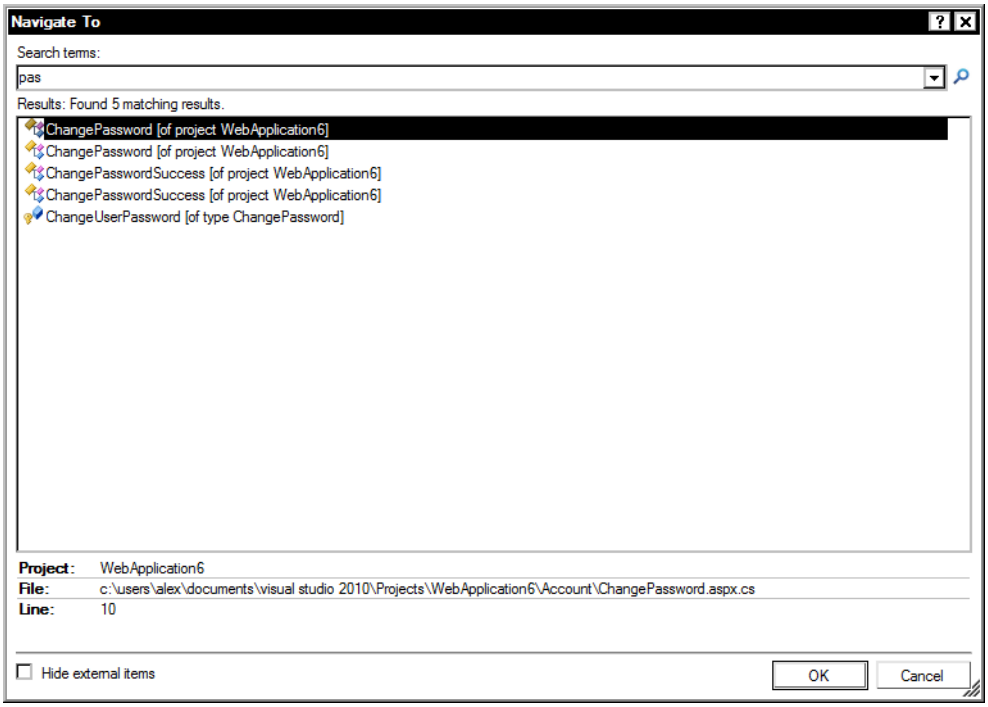

Figure 2-7. *Search files in your project with the Navigate To window.*

Box Selection

Box selection is one of my favorite new features. It allows you to quickly perform the same change on many lines of code. To use box selection, place the cursor where you want to make the change, and then hold down Shift+Alt in combination with the arrow keys to create a "box" where the change will be applied. Finally, enter your change and it will be applied to all the selected lines.

Box selection could, for example, be used to refactor a number of class variables' access levels from `private` to `public` in one edit, as shown in Figure 2-8.

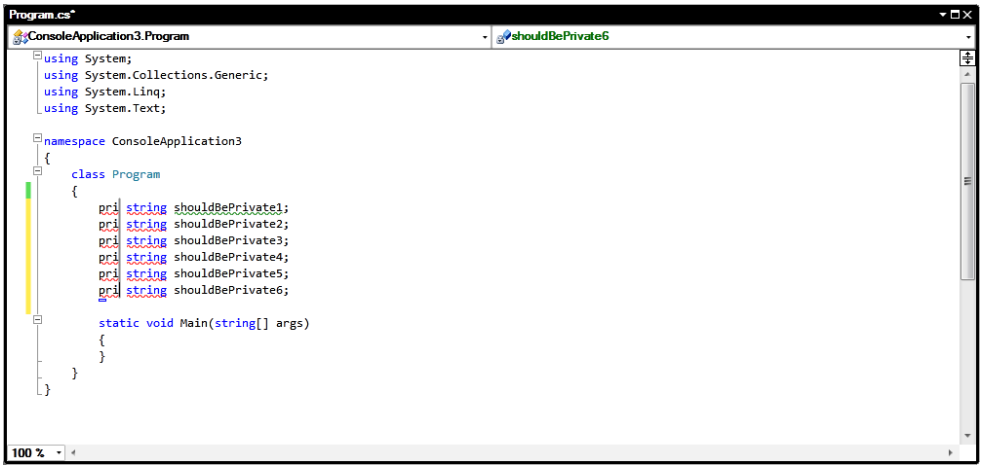

Figure 2-8. *Quickly make changes to multiple lines of code with box selection.*

Call Hierarchy

The Call Hierarchy window allows you to see all calls made to a particular method and all calls from the method. Call hierarchy is recursive. To open the Call Hierarchy window, right-click a method, property, or constructor and select View Call Hierarchy. The Call Hierarchy window will then open, displaying calls to and from the method (see Figure 2-9). Note that you can filter the Call Hierarchy window by solution, project, and document.

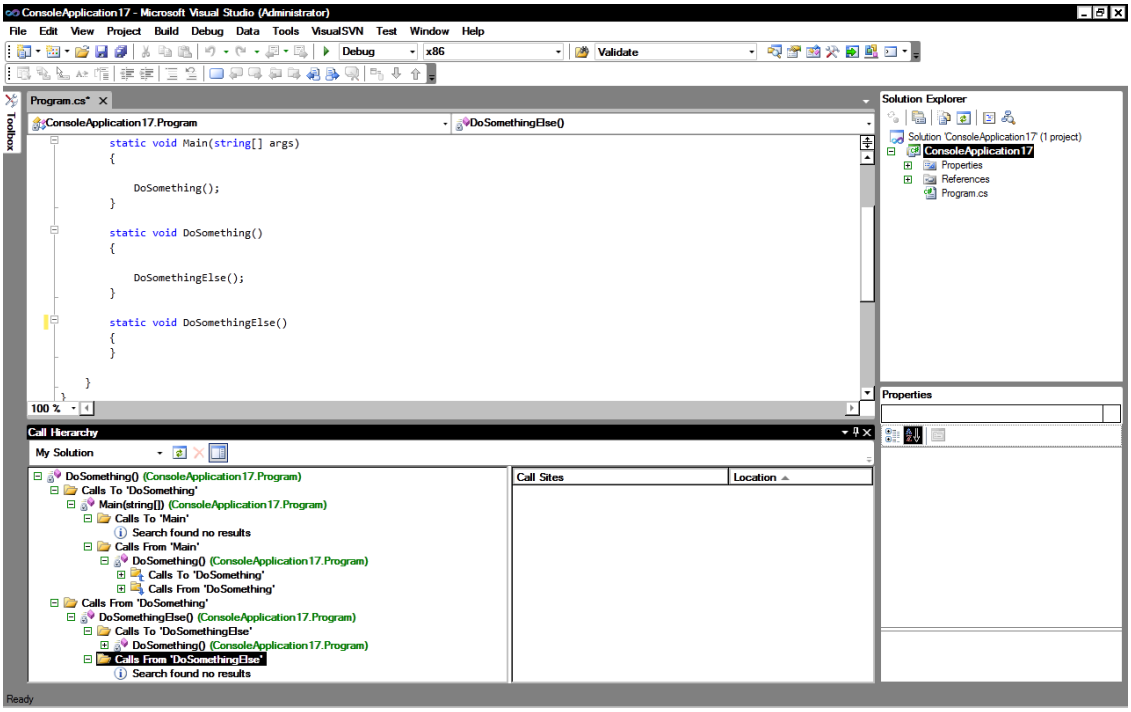

Figure 2-9. *See calls made to and from a particular method with the Call Hierarchy window.*

Code Generation

A great feature in VS2008 is that you can enter a new method name that doesn't exist and have the IDE create a stub of it for you (to do this, enter a method name that doesn't exist, press Ctrl+. and select the "Generate method stub..." option). VS2010 expands on this functionality and allows you to create classes, structs, interfaces, and enums in a similar manner. This is a great feature when you are starting the development of an application, and is particularly suitable for TDD-style development. Let's try this out now.

1. Create a new console application.
2. Enter the following code:

```
Zebra MyZebra = new Zebra();
```

3. Either click the smart tag beneath the Z in zebra or press Ctrl+. (the easier ay) to bring up the menu (as shown in Figure 2-10).

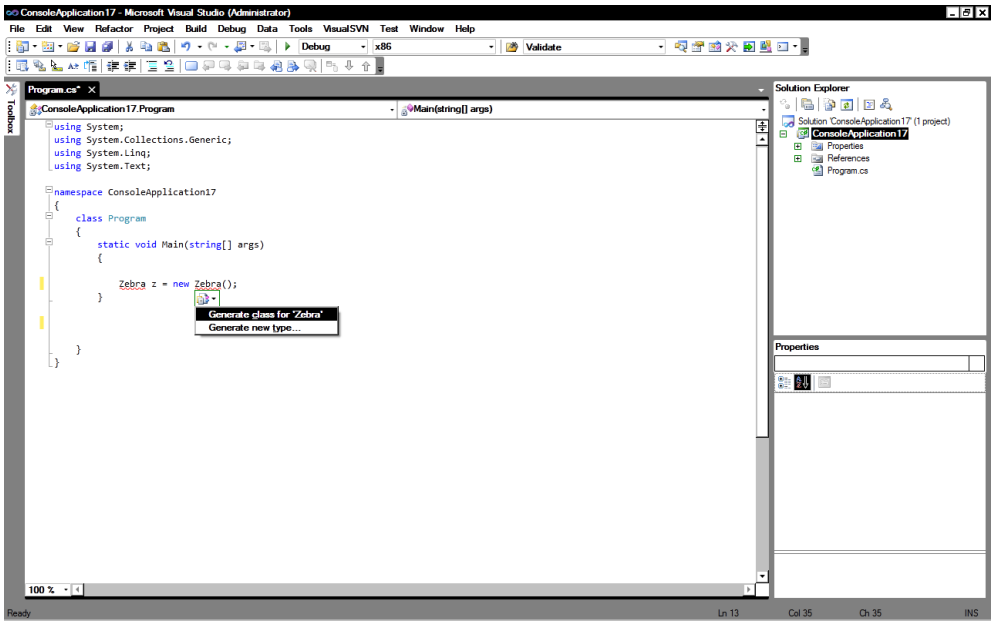

Figure 2-10. *There are new options available in VS2010 for generating classes and method stubs.*

4. You now you have the choice of creating a `Zebra` class in a separate file (`Zebra.cs`) by selecting "Generate class," or you can select "Generate new type" to bring up an options screen that allows greater control of generated type. For this example, please select "Generate new type."

5. The Generate New Type screen (shown in Figure 2-11) will appear, allowing you to specify a number of options, including the access level, file name, and item to create. Select Class on the Kind drop-down menu and change the access level to Internal.

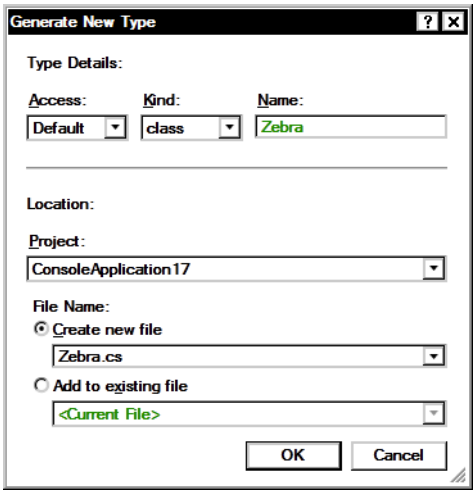

Figure 2-11. *Generate New Type allows you greater control over what is created.*

Visual Studio will then generate a new `Zebra` internal class.

Consume-First Mode

IntelliSense is a great feature, but can sometimes get in your way. For example, imagine an application where you have a class called `TigerCage` and you now want to create a `Tiger` class. If you want to use VS2010's new class generation features and you start typing `Tiger`, then Visual Studio's IntelliSense will jump in and smugly change your code to `TigerCage`.

To resolve this issue, IntelliSense now operates in two modes: *default* and *consume-first*. Consume-first mode prevents IntelliSense from automatically completing a type or member that has not yet been defined. To switch to consume-first mode, press Ctrl+Alt+Space. You can press Ctrl+Alt+Space again to switch back to default mode.

■ **NOTE** IntelliSense is now programmed to switch automatically to consume-first mode in common cases where it is known to be problematic.

Breakpoints

VS2010 allows you to export/import and label breakpoints. You can use this feature to share a collection of breakpoints with a colleague or quickly return to a previous debugging setup. Note that the exported file holds the breakpoint location by line number, so if you modify your code and import breakpoints, they will no longer be positioned correctly.

Individual breakpoints can be exported by right-clicking on them and then selecting the Export option. Or you can export all breakpoints (or those matching a specific search criteria) by opening the

Breakpoints window (Debug ➤ Windows ➤ Breakpoints) and selecting the "Export all breakpoints" option. Breakpoints can be imported in the Breakpoints window.

VS2010 allows you to apply a label to a breakpoint, as shown in Figure 2-12. This may be useful to associate it with a particular issue or with grouping in the Breakpoints window. To label a breakpoint, right-click one and select the "Edit labels" option. VS2010 will then give you the option of entering a new label for the breakpoint or reusing an existing one.

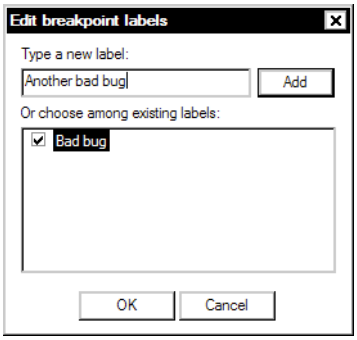

Figure 2-12. *Labelling a breakpoint*

Toolbox

If you start typing a letter, the toolbox will jump to items containing the letter typed. You can also tab through to the next item that matches.

Code Snippets

Previous versions of Visual Studio contained a feature called snippets, which allowed you to save blocks of code for later insertion, preventing you from having to retype (or remember) them. VS2010 contains a number of new snippets (in particular for ASP.NET) and allows you to easily create your own. Although you can create your own snippets in VS2008, it isn't an easy process without the use of third-party applications (e.g., Snippet Editor; `http://msmvps.com/blogs/bill/archive/2007/11/06/snippet-editor-2008-release.aspx`).

This changes, however, in VS2010 and is now very easy. There are two types of snippet:

- `Expansion` (the snippet is inserted at the cursor)
- `SurroundsWith` (wraps around existing code)

Let's take a look at this now and create a code file header snippet:

1. Add a new XML file to your project called `verHeader.snippet` (snippets always have the extension `.snippet`).
2. Right-click the editor window and select Insert Snippet ➤ Snippet. VS2010 will then create a basic XML snippet template.

3. In this example we will create an expansion snippet, so we need to remove the tag that reads as follows:

   ```
   <snippetType>SurroundsWith</snippetType>
   ```

4. Modify the `Title` tag to read "Code File Header."

5. Modify the `Author` tag to your name.

6. Modify the `Shortcut` tag (this is the trigger word that activates the snippet) to "codehead."

7. Enter a description for the snippet, such as "Adds a header to a code file."

8. Snippets can be created for different languages (such as VB and XML), but in this example we are creating a C# snippet. Change the `Language` attribute of the `Code` tag so it reads as follows:

   ```
   <Code Language="CSharp">
   ```

9. We now need to alter the `Literal` section. Literals allow you to define editable values that are inserted into your snippet. We want the user to enter his or her own name in the author section, so change the `ID` value to `Author` and enter your name as the `Default` tag.

10. The `Code` section contains what will be added when the snippet is inserted. Modify it to look like the following:

    ```
    <Code Language="CSharp">
          <![CDATA[
          **********************************
          Author: $Author$
          Date:
          Version:
          Purpose:
          **********************************
         ]]>
    </Code>
    ```

Your finished snippet should end up looking like the following:

```
<CodeSnippet Format="1.0.0"
 xmlns="http://schemas.microsoft.com/VisualStudio/2005/CodeSnippet">
  <Header>
    <Title>Code File Header</Title>
    <Author>Alex Mackey</Author>
    <Shortcut>codehead</Shortcut>
    <Description>Adds a header to a code file</Description>
    <SnippetTypes>
      <SnippetType>Expansion</SnippetType>
    </SnippetTypes>
  </Header>
```

```
  <Snippet>
    <Declarations>
      <Literal>
        <ID>Author</ID>
        <Default>Alex Mackey</Default>
      </Literal>
    </Declarations>
    <Code Language="CSharp">
      <![CDATA[
      **********************************
      Author: $Author$
      Date:
      Version:
      Purpose:
      **********************************
     ]]>
    </Code>
  </Snippet>
</CodeSnippet>
```

Loading the Snippet into Visual Studio

Before we can use our snippet, we need to load it into Visual Studio. Because snippets are pretty useful, you will probably want to create more than one. Follow these steps to create a new directory somewhere on your computer called `MySnippets`.

1. Copy the `verHeader.snippet` file in your solution to the snippets directory you just created.
2. On the main menu, go to Tools ➤ Code Snippets Manager, and you should see a screen similar to Figure 2-13.
3. Select Import.
4. Select the snippet you saved earlier, and click OK. Visual Studio will then confirm that you want to place the snippet in the My Code Snippets directory.

That's it; your snippet is ready to use. You can now use this snippet by typing **codehead**.

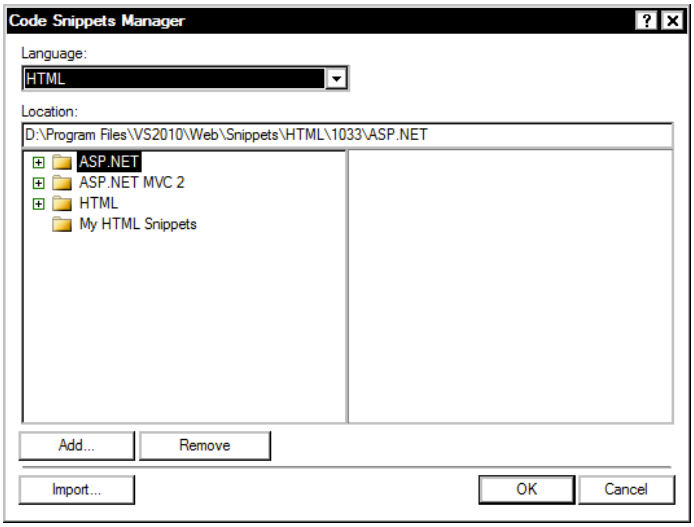

Figure 2-13. *Code Snippets Manager*

■ **TIP** You can avoid the previous installation steps and have Visual Studio automatically pick up the snippet by saving it to VS2010's Code Snippets directory. Its default `location is C:\Users\<Username>\Documents\Visual Studio 10\Code Snippets\`. You will not even have to restart VS.

Using Snippets

There are a number of ways to add snippets to your code. No doubt the quickest way is to use a trigger word (such as *`textbox`* in an ASP.NET application). However, sometimes you may not know the trigger word to use. In that case you can pick the word from the Snippet dialog.

To open the Snippet dialog, right-click the editor and select Insert Snippet. You can then choose from either the ASP.NET or HTML snippets. You can also press Ctrl+K and then Ctrl+X to bring up the Insert Snippet enhancement, which allows you to navigate through the snippets using the keyboard.

Creating Custom Start Pages

VS2010 allows you to customize the start page that is displayed when the IDE first loads. You could use this feature to display items such as current bugs, last night's build status, and so on. On my Windows 7 machine, this directory is held at the following path: `C:\Users\alex\Documents\Visual Studio 10\StartPages`.

1. Open the project `StartPage.csproj`. Note how `StartPage.xaml` is a standard XAML page with some Visual Studio–specific controls to display items such as recently opened projects.
2. Perform a simple modification, such as altering some of the text content.
3. Save the file with a new file name, such as `CustomStartPage.xaml`, in the same directory.

Before you can use your new start page, you have to select it in the Visual Studio options. Go to Tools ➤ Options, and then select the Startup node. Select the new custom start page from the Custom Start Page drop-down menu.

Close Visual Studio and reopen it. Your new start page should now appear the next time Visual Studio is loaded.

Text Template Transformation Toolkit Files

A Text Template Transformation Toolkit (T4) template is a code-generation language that has been around since VS2005. You should be aware of T4 templates, as they are used in areas such as Entity Framework and ASP.NET MVC, and can be useful for your own development. To see T4 templates in action, create a file with the extension `.tt`, add some text content, save the file, and note how Visual Studio will generate a code file from the template. You can apply complex logic using T4 templating language to change the output that is generated depending on various conditions.

T4 templates in VS2010 are compiled when they are saved (preprocessed). This means that they are another type that can be instantiated.

Scott Hanselman has some great and information on this area, so please refer to the following article: `www.hanselman.com/blog/T4TextTemplateTransformationToolkitCodeGenerationBestKept➥VisualStudioSecret.aspx`.

T4 templates don't have IntelliSense, so your best bet is to download the Tangible T4 plug-in: http://visualstudiogallery.msdn.microsoft.com/en-us/60297607-5fd4-4da4-97e1-3715e90c1a23.

For more information, see `http://code.msdn.microsoft.com/DslTools/Wiki/View.aspx?title=What%27s%20new` and `http://karlshifflett.wordpress.com/2009/10/30/t4-preprocessed-text-templates-in-visual-studio-2010/`.

VS2010 Premium and Ultimate

I will only be covering VS2010 Professional edition in this book, but I want to make you aware of a couple of fantastic features available in more expensive versions.

Generate Sequence Diagram

The Generate Sequence Diagram feature creates a diagram of a methods calls. To use this feature, simply right-click a function and select Generate Sequence Diagram.

Historical Debugging (Team System Edition Only)

Visual Studio Team System edition contains a very cool feature called Historical Debugging. Ian Who, a developer on the profiler team, says the following about it:

> *The Historical Debugger plays a role similar to that of a black box in a plane. We keep track of important points in your programs execution and allow you to play back what happened at those points at a later time.*

`http://blogs.msdn.com/ianhu/archive/2009/05/13/historical-debugging-in-visual-studio-team-system-2010.aspx`

Static Analysis of Code Contracts

Code contracts (which I cover in Chapter 3) allow you to express constraints within code that can be analyzed at compile time to check if your code violates them. Although code contracts are present in all versions of Visual Studio, only Premium and Ultimate provide static analysis.

Customization of IDE

VS2010 allows you to create much more advanced customizations than changing the start page or creating snippets. VS2010 has been written from the ground up for extensibility and customization.

- Screens have been rewritten in WPF and managed code.
- The IDE API has been refactored for easier use.
- The IDE API is fully documented.
- New immutable text snapshots make it easier to obtain accurate snapshots of the text editor.

Many areas of the IDE can be overridden by creating a MEF component (I will talk about MEF shortly).

So, what can you customize? VS2010 allows you to customize the following areas, among other things:

- Margins and scrollbars
- Tags
- Adornments (items painted on the editor surface)
- Mouse processors
- Drop handlers
- Options
- IntelliSense and the debugger

Before you can perform any of these customizations, however, you will first need to download and install the Visual Studio SDK (`http://www.microsoft.com/downloads/details.aspx?FamilyID=➥cb82d35c-1632-4370-acfb-83c01c2ece24&displaylang=en`).

Extensions in VS2010 make heavy use of a new technology called MEF. Before you create any customizations, however, you need to understand a bit about MEF.

MEF

Managed Extensibility Framework (MEF) is a new framework for creating customizable applications that can be used by any .NET-compatible language. Glenn Block (PM for the new MEF in .NET 4.0) says the following:

> *Quite simply, MEF makes building extensible apps, libraries, and frameworks easy. It shares some common characteristics of other frameworks out there, but it also addresses a whole new set of problems that arise in building extremely large scalable extensible systems.*

`http://blogs.msdn.com/gblock/archive/2008/09/26/what-is-the-managed-extensibility-framework.aspx`

Let's say you have created a Tetris application and want to allow users to extend it by creating their own shapes of bricks. MEF enables you to do this by defining a brick interface and then dynamically loading and resolving the created extensions.

When creating a MEF application, take the following steps:

1. Define areas of the application that can be extended and decorate them with the `[Import]` attribute.
2. Determine a contract/interface that defines what your extensions must do/be (this could be as simple as stating they must be of type `String`).
3. Create an extension that meets these requirements and decorate it with the `[Export]` attribute.
4. Modify your application to load these extensions.

Why Use MEF?

Using MEF has the following advantages:

- Microsoft hopes that MEF will become the preferred standard method of creating extensions. By utilizing a standard plug-in model, your extensions could be used in many applications.
- MEF provides a number of flexible ways to load your extensions.
- Extensions can contain metadata to provide further information about their capabilities. For example, you may only want to load extensions that can communicate securely.
- MEF is open source and works on VS2008 (`www.codeplex.com/MEF`).

BUT COULDN'T I ACCOMPLISH THIS WITH REFLECTION/DEPENDENCY INJECTION/IOC CONTAINERS/VOODOO?

There is overlap in the functionality provided by the technologies mentioned in this section and MEF. MEF and Inversion of Control (IOC) containers do have some overlap, and many people would classify MEF as an IOC container. MEF's primary purpose is, however, creating extensible applications through discovery and composition, whereas IOC containers are generally more focused on providing an abstraction for testing purposes. It's not a discussion I want to get into, but Oren Eini does, so please refer to `http://ayende.com/Blog/archive/2008/09/25/the-managed-extensibility-framework.aspx`.

Hello MEF

In this sample application, you will create two extensions that print out a message. You will then load them both into an `IEnumerable<string>` variable called `Message` before iterating through them and printing out the messages.

1. Create a new console project and call it `Chapter2.HelloMEF`.
2. Add a reference to `System.ComponentModel.Composition`.
3. Add a new class called `MEFTest`.
4. Add the following `using` statements to the class:

```
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using System.Reflection;
```

5. Modify the `MEFTest` class code to the following (note how we decorate the `Message` property with the `[Import]` attribute):

```
public class MEFTest
{
    [Import]
    public string Message { get; set; }

    public void HelloMEF()
    {
        CompositionContainer container = new CompositionContainer();
        CompositionBatch batch = new CompositionBatch();
        batch.AddPart(new Extension1());
        batch.AddPart(this);
        container.Compose(batch);

        Console.WriteLine(Message);

        Console.ReadKey();
    }
}
```

6. We now need to create the extensions to load, so create a new class called `Extension1`, and add the following `using` statement:

```
using System.ComponentModel.Composition;
```

7. Amend `Extension1.cs` to the following:

```
public class Extension1
{
    [Export]
    public string Message
    {
        get
        {
            return "I am extension 1";
        }
    }
}
```

8. Finally, open `Program.cs` and add the following code:

```
static void Main(string[] args)
{
    MEFTest MEFTest = new MEFTest();
    MEFTest.HelloMEF();
}
```

9. Press F5 to run the application, and you should see that both extensions are loaded and the `Message` is property printed out, as Figure 2-14 shows.

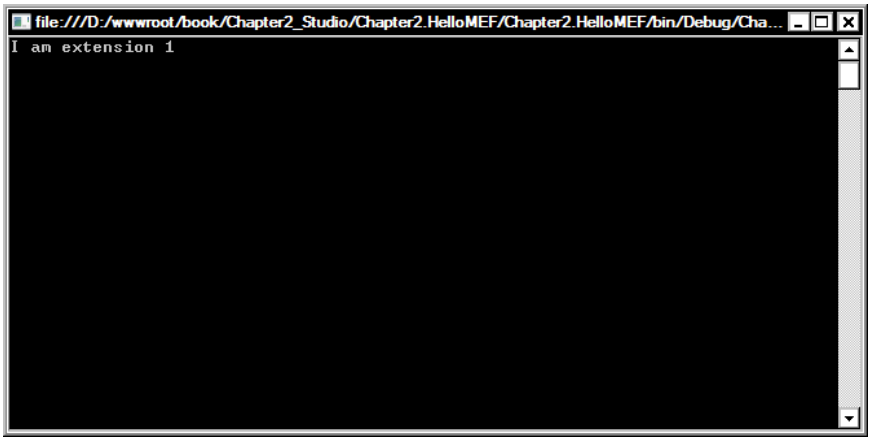

Figure 2-14. *Output from HelloMEF application*

Congratulations, you have created your first MEF application.

How Did This Example Work?

You started off by telling MEF that your `Message` property can be extended by marking it with the `[Import]` attribute. The `[Import]` attribute means "I can be extended" to MEF:

```
[Import]
public string Message { get; set; }
```

You then created an extension class and added the `[Export]` attribute. `[Export]` tells MEF "I am an extension":

```
class extension1
{
    [Export]
    public string Message
    {
        get
        {
            return "I am extension 1";
        }
    }
}
```

You then created a container (containers resolve MEF extensions when they are requested) to hold the extensions and added your extension classes to it using a `CompositionBatch`:

```
CompositionContainer container = new CompositionContainer();
CompositionBatch batch = new CompositionBatch();
batch.AddPart(new extension1());
batch.AddPart(this);
```

The `Compose()` method was then called, which caused MEF to load our extensions into the `Message` property:

```
container.Compose(batch);
```

MEF then loaded extensions into the `Messages` property decorated with the `[Export]` attribute that matched the contract. Finally, you printed out the message to the screen. In this example, you only loaded extensions contained within the project itself, which isn't too useful. Luckily MEF allows you to load extensions declared outside the project.

MEF Catalogs

MEF uses a concept called catalogs to contain extensions. Catalogs come in three different flavors:

- **Assembly**: Extensions are contained in a .NET assembly.
- **Directory**: Extensions are in a physical directory.
- **Aggregate**: This is a catalog type that contains both assembly and directory extensions.

In this example you will use a directory catalog to load an extension defined outside the main project. Directory catalogs scan the target directory for compatible extensions when first created. You can rescan the directory by calling the `Refresh()` method.

1. It is a good idea to declare MEF interfaces in a separate project to avoid circular reference issues and facilitate reuse, so open the existing `Chapter2.HelloMEF` project and add a new class library project called `Chapter2.MEFInterfaces`.
2. Inside this project, create an interface called `ILogger`.
3. Replace the existing code in `ILogger.cs` with the following code:

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

namespace Chapter2.MEFInterfaces
{
    public interface ILogger
    {
        string WriteToLog(string Message);
    }

}
```

4. In the `Chapter2.HelloMEF` project, add a reference to the `Chapter2.MEFInterfaces` project.
5. In the `Chapter2.HelloMEF` project, create a class called `MoreUsefulMEF` and enter the following code:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ComponentModel.Composition;
using System.ComponentModel.Composition.Hosting;
using System.Reflection;
using System.IO;
namespace Chapter2.HelloMEF
{
```

```
    class MoreUsefulMEF
    {
        [Import]
        private Chapter2.MEFInterfaces.ILogger Logger;

        public void TestLoggers()
        {
            CompositionContainer container;
            DirectoryCatalog directoryCatalog =
              new DirectoryCatalog(
                (Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location))
              );

            container = new CompositionContainer(directoryCatalog);
            CompositionBatch batch = new CompositionBatch();
            batch.AddPart(this);
            container.Compose(batch);
            Console.Write(Logger.WriteToLog("test"));
            Console.ReadKey();
        }
    }
}
```

6. Open `Program.cs` and amend the `Main()` method to the following:

```
MoreUsefulMEF MoreUsefulMEF = new MoreUsefulMEF();
MoreUsefulMEF.TestLoggers();
```

7. You will now create a logging extension, so add a new class library project to the solution called `Chapter2.EmailLogger`.
8. Add a reference to the `Chapter2.MEFInterfaces` project.
9. Add a reference to `System.ComponentModel.Composition`.
10. Add a new class called `EmailLogger`.
11. Amend the code to the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ComponentModel.Composition;
```

```
namespace Chapter2.EmailLogger
{
    [Export(typeof(Chapter2.MEFInterfaces.ILogger))]
    public class EmailLogger : MEFInterfaces.ILogger
    {
        public string WriteToLog(string Message)
        {
            //Simulate email logging
            return "Email Logger Called";
        }
    }

}
```

12. When you use a directory catalog to load MEF components, you can either compile the `Chapter2.EmailLogger` project and copy the built assembly to `Chapter2.HelloMEF`'s `bin` folder, or add a project reference in `Chapter2.HelloMEF` to the `Chapter2.EmailLogger` project.
13. Once you have done this, press F5 to run the `HelloMEF` project. The Email Logger extension should then be loaded and "Email Logger Called" outputted to the screen.

Metadata

An important feature of MEF is that you can provide additional information about an extension's capabilities with metadata. MEF can then utilize this information to determine the most appropriate extension to load and query its capabilities. For example, in the previous logging example, you might specify whether the logging method is secure, and then in high-security environments only load extensions that communicated securely. Metadata can be defined at a class or method level.

To add metadata to a class, use the `[PartMetaData]` attribute:

```
[PartMetadata("secure", "false")]
[Export(typeof(Chapter2.MEFInterfaces.ILogger))]
public class EmailLogger : MEFInterfaces.ILogger
{..}
```

You can add metadata to an individual method with the `[ExportMetadata]` attribute:

```
[ExportMetadata("timeout", "5000")]
public string WriteToLog(string Message)
{..}
```

Metadata can then be retrieved using a part's `Metadata` property. The following code demonstrates retrieving metadata from a directory catalog:

```
CompositionContainer container;
DirectoryCatalog directoryCatalog =
  New DirectoryCatalog((Path.GetDirectoryName(Assembly.GetExecutingAssembly().Location)));
```

```
foreach (var Part in directoryCatalog.Parts)
{
    Console.WriteLine(Part.Metadata["secure"]);
}
```

Note that querying a method's metadata is slightly different and that you must instead use the `Part.ExportDefinitions` property.

What's This All Got to Do with Visual Studio Extensibility?

Visual Studio utilizes MEF in an almost identical way to the previous examples when it loads Visual Studio extensions. When Visual Studio first loads, it examines the extensions directory and loads available extensions. Let's now look into how these extensions are created.

Visual Studio Extensibility

After you install the Visual Studio customization SDK, a number of new extensibility projects are available for you to create. These projects are templates that demonstrate how to perform various Hello World-type customizations that you can then build on. Figure 2-15 shows these new project types.

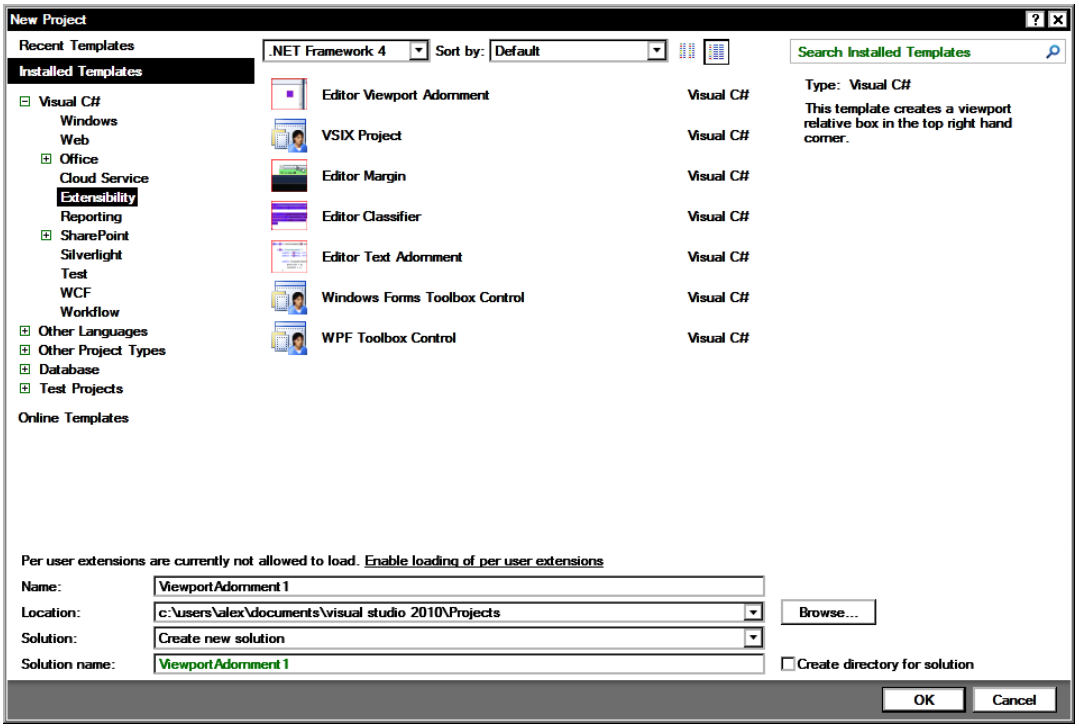

Figure 2-15. *New extensibility projects are available after installing customization SDK.*

The following extensibility projects are available:

- **VSIX Project**: This is an empty extension that contains just the minimum references needed and a manifest file that describes the extension.
- **Editor Margin**: This creates a green box down the bottom of code editor frame.
- **Editor Classifier**: This formats certain types of text a blue color.
- **Editor Text Adornment**: This is a template that highlights all instances of the letter *a*.
- **Editor Viewport Adornment**: This template creates a purple box in the top-right corner of IDE and a Windows Forms toolbox control.

Let's take a look at the Editor Margin extensibility project.

Editor Margin

Open up Visual Studio and create a new Editor Margin project called `Chapter2.EditorMargin`.

1. Open `MarginFactory.cs` and note how it utilizes the MEF `[Export]` attribute (the other attributes contain various bits of metadata utilized by the IDE):

```
[Export(typeof(IWpfTextViewMarginProvider))]
[Name("GreenBar")]
//Ensure that the margin occurs below the horizontal scrollbar
 [Order(After = PredefinedMarginNames.HorizontalScrollBar)]
//Set the container to the bottom of the editor window
 [MarginContainer(MarginContainerAttribute.Bottom)]
//Do this for all content types
 [ContentType("text")]
 [TextViewRole(PredefinedTextViewRoles.Interactive)]
internal sealed class MarginFactory : IWpfTextViewMarginProvider
{
    public IWpfTextViewMargin CreateMargin(IWpfTextViewHost textViewHost,
                                           IWpfTextViewMargin containerMargin)
    {
        return new GreenMargin(textViewHost.TextView);
    }
}
```

2. Let's do something a bit crazy and tell Visual Studio to rotate the text editor 245 degrees. Open `MarginFactory.cs` and add the following `using` statement:

```
using System.Windows.Media;
```

3. Inside the `CreateMargin` constructor, above the line that reads `return new GreenMargin(textViewHost.TextView);`, add the following code:

```
textViewHost.TextView.VisualElement.LayoutTransform = new RotateTransform(245);
```

4. Build and run this project, and the IDE will launch a special test instance containing your extension (this may take a bit of time, so be patient).

5. Once the test instance has loaded, create a new console project. Voila! As you can see, the text editor is rotated and a green "Hello world!" box is created at the base of the editor (Figure 2-16).

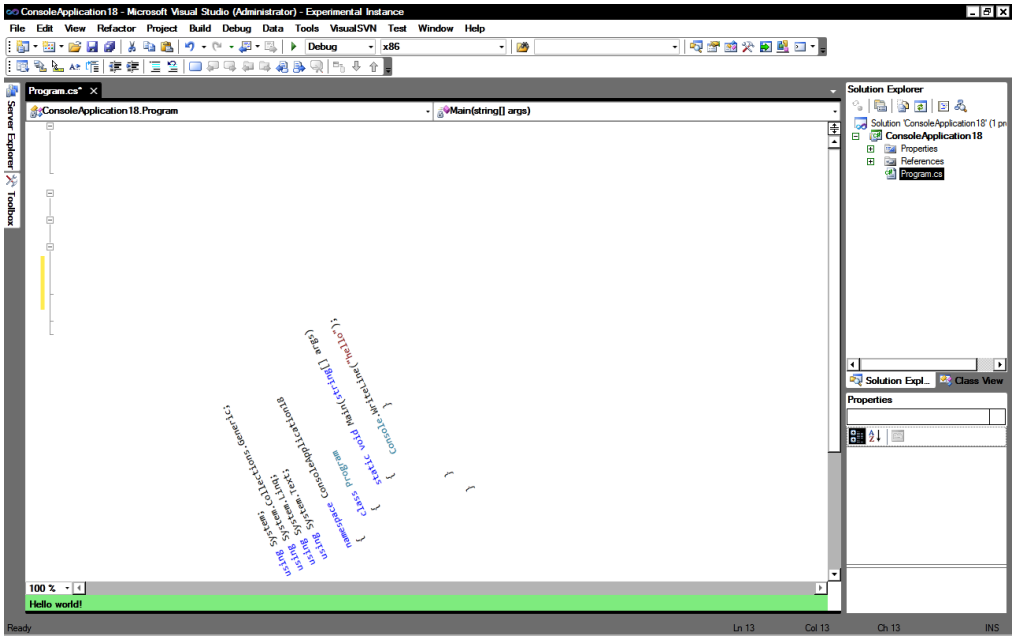

Figure 2-16. *This may not be the most useful of extensions, but it demonstrates the control you now have.*

Note how the text editor still works just as you would expect with syntax checking, IntelliSense, and so on (although the scrollbars behave a little strangely).

Distributing Extensions

Now that you have created a useful extension for rotating a text editor, what if you want to share it with your friends/victims? When extensions are compiled, they are built as `.vsix` files that you can install by double-clicking them or copying them to the extensions directory at `C:\Program Files\Microsoft Visual Studio 10.0\Common7\IDE\Extensions`.

Extension Gallery

The Extension Gallery (see Figure 2-17) allows you to download a number of additions from new project templates to make IDE customizations. A number of extensions for VS2010 are available already, some with source code. To open the Extension Gallery, select Extension Manager on the Tools menu and then select the Online Gallery option.

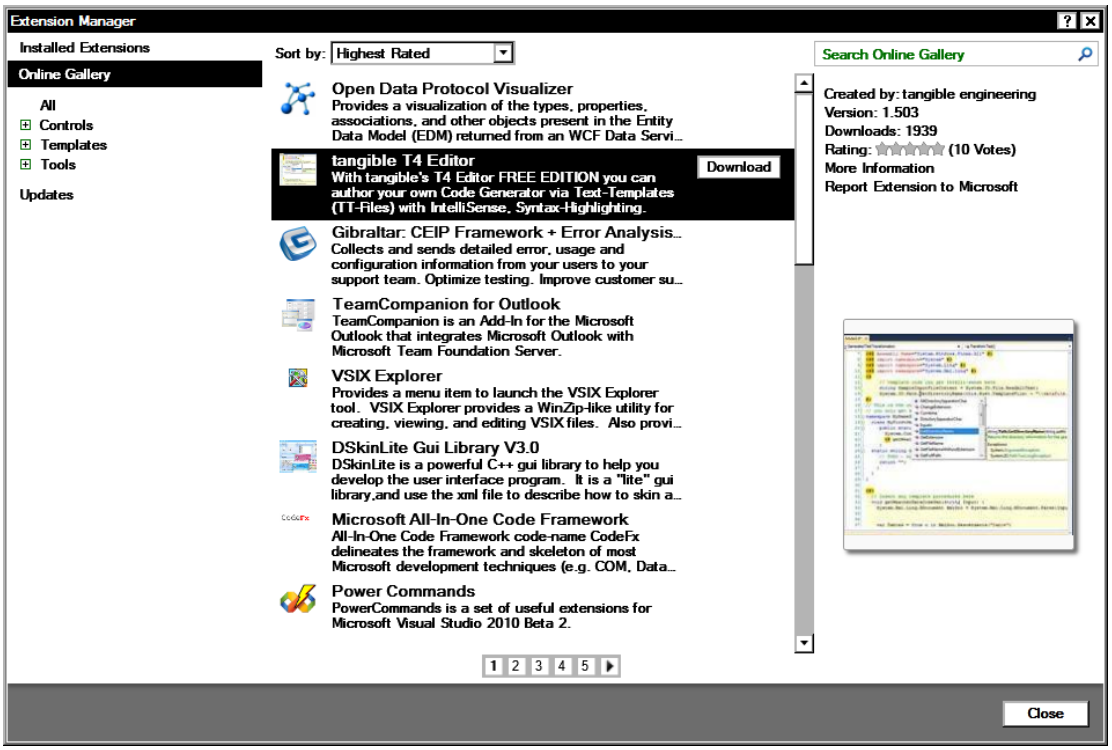

Figure 2-17. *Extension Gallery*

WHAT ABOUT EXISTING EXTENSIONS CREATED WITH THE PREVIOUS API?

Microsoft says that more than 80 percent of existing IDE customization will be supported through the use of shims (code that maps the old API methods to the new). It is important to note, however, that Microsoft plans to remove these shims in the next version of Visual Studio after VS2010.

Visual Studio Shell

It is worth noting that starting with VS2008, Microsoft opened up the ability to make use of the IDE for your own applications. This is called the Visual Studio Shell. A popular project using the Visual Studio Shell is the add-on studio for the online game World of Warcraft (`http://addonstudio.codeplex.com`).

For more information on the Visual Studio Shell, please refer to `http://msdn.microsoft.com/en-us/vsx2008/products/bb933751.aspx`.

Dotfuscator Changes

Dotfuscator is a post-build .NET hardening and instrumentation platform for protecting, measuring, and managing .NET applications. Traditionally, a reduced-functionality version of Dotfuscator has been bundled with Visual Studio, and VS2010 is no exception. However, the new version of Dotfuscator Software Services CE contains runtime intelligence functionality and some great added features, including the following:

- Tamper defense, which detects application modifications
- Application expiration, such after a 30-day trial period
- Session and feature usage tracking, which allows you to track what the user was actually doing within your application
- The ability to send tamper and tracking usage to an endpoint of your choice for later analysis

To access Dotfuscator functionality within Visual Studio on the main menu, go to Tools and select Dotfuscator Software Services. For more information on Dotfuscator, please refer to `www.preemptive.com/dotfuscator.html`, and for more information on runtime intelligence, see `http://en.wikipedia.org/wiki/Runtime_Intelligence`.

Summary

Many developers were concerned at the prospect of Visual Studio's IDE being built using WPF—specifically that it would be slow and clunky. Microsoft has without a doubt demonstrated the flexibility and power of WPF and proved these doubters wrong! VS2010 has some great productivity enhancements in this release, and with the improved multitargeting support, even if you are not ready to move your code base to .NET 4.0, you can make use of many of these features today.