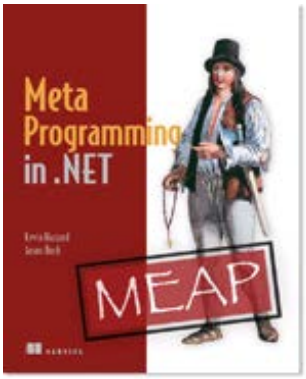
MANNING PUBLICATIONS



Metaprogramming in .NET
Kevin Hazzard and Jason Bock

Roslyn API is a major step forward for developers who want that intimate view of code. In this article based on chapter 12 of Metaprogramming in .NET, authors explain how Roslyn API works through executable code and then go deeper into more of its parts.

Understanding the Basics of Roslyn

.NET developers are talking about Roslyn, a library of analysis APIs you can use to digest and interpret the grammar and syntax of C# and VB code. Roslyn is essential for metaprogramming tasks like building or enhancing your coding environment, creating productivity tools like refactoring engines, and so forth.

Let's start your journey of looking at how Roslyn works by creating executable code fragments at runtime. You'll do it in two ways: you'll use a scripting engine first, and then you'll see how it's done by compiling and executing the code.

Getting Roslyn installed

To play with Roslyn, simply visit the Roslyn site: http://msdn.microsoft.com/en-us/roslyn and download the bits from there. Note that this only works with Visual Studio 2010 with SP1 installed. Also, since this is a CTP, you may not want to run this on your main installation of VS 2010; a safer bet is to create a virtual PC and work with Roslyn there. It's slower, but you won't have to deal with any uninstallation issues that a CTP may have.

Running code snippets with the script engine

The output of creating dynamic code to implement `ToString()` is a double-pipe delimited set of property name and value pairs, which is what you see in figure 2. Let's do the same thing with Roslyn. We won't concern ourselves with caching or other performance improvements and tweaks for now. This is a CTP and trying to gain any insight into performance numbers you gather is suspect at best.

Let's start by defining the extension method that will be used:

```
public static class ToStringViaRoslynExtensions
{
  public sealed class Host<T>
  {
    public Host(T target)
    {
      this.Target = target;
    }

    public T Target { get; private set; }
  }

  public static string Generate<T>(this T @this)
  {
```

You'll see the definition for `Generate()` momentarily, but notice that there's also a class called `Host` that has a read-only property of the same type as `@this`. This is needed for our scripting environment so it can reference the object that you've been given in `@this`.

Now, let's see the code that's generated to create a meaningful object description from `ToString()`:

```
    var code = "new StringBuilder()" +
      string.Join(".Append(\" || \")",
        from property in @this.GetType().GetProperties(
          BindingFlags.Instance | BindingFlags.Public)
        where property.CanRead
        select string.Format(
        ".Append(\"{0}: \").Append(Target.{0})",
          property.Name)) + ".ToString()";
```

You're creating C# code as the output and not something like IL in a dynamic method. What ends up in the `code` variable looks something like this:

```
new StringBuilder().Append("Age: ").Append(Target.Age).Append(" || ")
  .Append("Name: ").Append(Target.Name).ToString()
```

The last piece is bringing Roslyn's scripting engine into play to actually execute the code:

```
      var hostReference =                                    #1
        new AssemblyFileReference(typeof(                    #1
          ToStringViaRoslynExtensions).Assembly.Location); #1
      var engine = new ScriptEngine(                              #2
        references: new[] { hostReference },                      #2
        importedNamespaces: new[] { "System", "System.Text" }); #2
      var host = new Host<T>(@this);      #3
      var session = Session.Create(host); #3
      return engine.Execute<string>(code, session); #4
    }
  }
```

#1 Gets an assembly reference to the host
#2 Creates the scripting engine
#3 Creates the session
#4 Executes the code

You first need to get an `AssemblyFileReference` to the assembly that contains the `Host` type so the scripting engine will know what "Target" means in the code you give it (#1). Next, you create a `ScriptEngine` object, passing it the assembly references it needs to know about as well as any namespaces. You're using `StringBuilder` so that's why "System.Text" is passed into the engine (#2). A `Session` object is also required because you need to pass in a host object that the code can use—namely, an instance of `Host`—and `Session` is what ties the dynamic code to your code that is currently executing (#3). Finally, the last step is to actually execute the code, which is done by calling `Execute()` (#4). The following figure lays out the flow and interaction between these parts so you can see at a higher level how they work together to run your dynamic code.

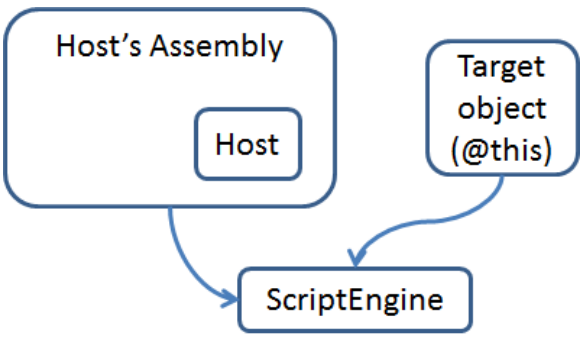


Figure 1 Interaction with the ScriptEngine. Your code passes a reference to the assembly that contains the Host class, along with the target object, to the script engine. The script that is executed will use both parts to run correctly.

To test it out, create a simple class with a couple of properties and `ToString()` overridden to call the extension method:

```
public sealed class Person
```

```
  {
    public Person(string name, uint age)
    {
      this.Name = name;
      this.Age = age;
    }

    public override string ToString()
    {
      return this.Generate();
    }

    public uint Age { get; private set; }
    public string Name { get; private set; }
  }
```

When you run `ToString()` in a console application like so:

```
  static void Main(string[] args)
  {
    Console.Out.WriteLine(
      new Person("Joe Smith", 30).ToString());
  }
```

You should see the output in the following figure:

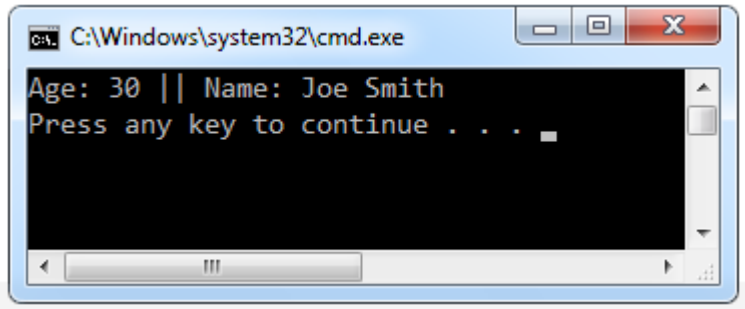


Figure 2 Calling ToString() implemented via Roslyn. By compiling C# at runtime, you can implement anything you want at the abstraction level you always write at.

The end result may look simple, but step back for a moment and think about what you just did. You created code on the fly, but it wasn't IL or an expression tree. It was C# code! The Roslyn engine happily compiled that little piece of code and executed it on the fly. No knowledge of opcodes or expressions necessary; you can write code that literally writes more code and executes it.

Of course, one could argue that this could be mimicked with what's currently available in the C# compiler. That is, just create the code snippet in a file, run the compiler, load the resulting assembly, and execute a method via Reflection. With Roslyn, though, it's all in-memory, and you don't have to mess with things like assembly loading and file creation if you don't want to. To be fair, Roslyn can compile code files and generate assemblies—it has to if it's going to replace csc.exe. But the point is, Roslyn brings code analysis and manipulation to a much higher level than what was ever available before.

Now that you have your feet wet with Roslyn, let's take a deeper dive into its API and see how we can create a simplistic dynamic mock at runtime.

Creating dynamic assemblies with Roslyn

Now you'll see how you can compile a mock C# into an assembly. Before you dive into the Roslyn API again, let's define what a mock is.

What is a mock?

Generally speaking, a mock is an object that can stand in place of another object. The mock is typically used in unit testing scenarios, where a developer will use a mock instead of an object that does a number of long-running or complex calculations. Rather than include these dependencies in the unit test, the developer will inject a mock into

the test such that the test can focus on the code at hand. A mock object can also be used to verify that the code under test used it in an expected manner.

Let's use a simple example to illustrate how mocks are used. Let's say you had a class, `Person`, that used a service to look up address information for that person. The following figure shows the dependency `Person` has on the service:

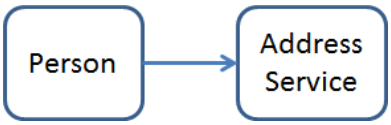


Figure 3 Using a dependency directly. When you use an object that may have complex setup needs or takes a long time to execute, it can make unit testing difficult and time-consuming.

Now, whenever a developer needs to test a `Person` object, they also need to ensure that the service is up and running and will return the expected data. This is time-consuming and brittle. A better approach would be to break the direct dependency on `AddressService` as the following figure shows:

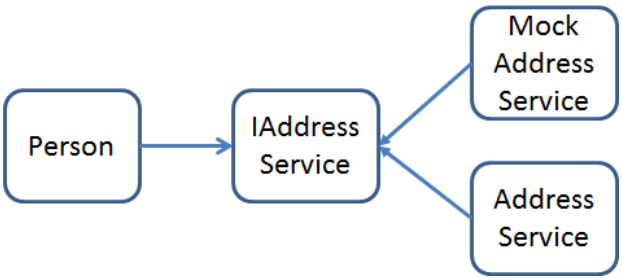


Figure 4 Using an interface in code. Now the Person object doesn't care how IAddressService is implemented and you can use mocks for Person-based unit tests.

In this case, `Person` now has a dependency on the `IAddressService` interface. The code doesn't care how the class that implements `IAddressService` works; it just cares about the contract that the interface specifies. During a test, a `MockAddressService` object is used, and in production, `AddressService` is used.

Learning more about unit testing

There's a lot more to mocks and unit testing than what we've presented in this section. If you're interested in getting a deeper dive into unit testing, please check out "The Art of Unit Testing" (http://manning.com/osherove/) and "Dependency Injection in .NET" (http://manning.com/seemann/).

You can hand-roll the mocks if you want—that is, you can write the code that implements an interface and notifies you when a method has been called. But, you can also use metaprogramming techniques to synthesize a class at runtime that does this for you. Let's see how you can use Roslyn to create a mock at runtime.

Generating the mock code

You'll see how you can create a mock using C# generated code at runtime. You'll compile the code and create an instance of the mock, passing that back to the caller. The caller will use a class with methods that match the signature of the methods in the interface that you want to handle in the test. This will allow you to provide a mocked implementation of the interface method, which is useful in testing scenarios.

Mock frameworks in .NET

The mock structure you're going to create is fairly simple compared with some of the frameworks that currently exist in the .NET world. Some of the ones we recommend are NSubstitute (http://nsubstitute.github.com/), Moq (http://code.google.com/p/moq/), and RhinoMocks (http://hibernatingrhinos.com/open-source/rhino-mocks). Hopefully, once Roslyn is officially released, these frameworks will spend time updating their engines to use the Roslyn API to generate their mocks.

The first thing you want to do is create a string that represents the structure of the class you want to dynamically create at runtime:

```
public sealed class MockCodeGenerator
{
  private const string Template = @"
    [System.Serializable]
    internal sealed class {0}
      : {1}
    {{
      private {2} callback;

      public {0}({2} callback)
      {{
        this.callback = callback;
      }}

      {3}
    }}";
```

Each mock needs a new type name ({0}), the interface it's implementing {1}, a reference to the callback object ({2}), and a list of interface methods with an implementation based on the methods that exist in the callback object ({3}). Let's fill these holes in the code:

```
  public MockCodeGenerator(string mockName,
    Type interfaceType, Type callbackType)
    : base()
  {
    this.MockName = mockName;
    this.InterfaceType = interfaceType;
    this.InterfaceTypeName = InterfaceType.FullName;
    this.CallbackType = callbackType;
    this.Generate();
  }

  private void Generate()
  {
    this.Code = string.Format(MockCodeGenerator.Template,
      this.MockName, this.InterfaceTypeName,
      this.CallbackType.FullName,
      this.GetMethods());
  }
```

Other than generating the methods, everything else is fairly boilerplate. Note that we use the `FullName` property for both the interface and callback types. This makes it a little easier to generate the code as we don't need to include using statements. Now, let's see how you generate the methods that implement the interface.

Listing 1 Generating methods for an interface

```
  private string GetMethods()
  {
    var methods = new StringBuilder();
    var callbackMethods = this.CallbackType.GetMethods(
      BindingFlags.Public | BindingFlags.Instance);

    foreach (var interfaceMethod in                #1
      this.InterfaceType.GetMethods())             #1
    {                                                    #1
      methods.Append("public " +                         #1
        MockCodeGenerator.GetMethod(interfaceMethod) + "{"); #1

      var callbackMethod = this.FindMethod(              #2
        callbackMethods, interfaceMethod);               #2
```

```
    if (callbackMethod != null)                          #2
    {                                                    #2
      if (callbackMethod.ReturnType != typeof(void))     #2
      {                                                  #2
        methods.Append("return ");                       #2
      }                                                  #2

      methods.Append("this.callback." +                  #2
        MockCodeGenerator.GetMethod(                     #2
          callbackMethod, false) + ";");                 #2
    }                                                    #2
    else                                                 #3
    {                                                    #3
      if (interfaceMethod.ReturnType != typeof(void))    #3
      {                                                  #3
        methods.Append("return " + (                     #3
          interfaceMethod.ReturnType.IsClass ?           #3
          "null;" : string.Format("default({0});",       #3
            interfaceMethod.ReturnType.FullName)));      #3
      }                                                  #3
    }                                                    #3

    methods.Append("}");
  }

  return methods.ToString();
}
```

#1 Creates an implementation for each method
#2 Calls the method on the callback object if a match exists
#3 Returns the default value for the interface method if the return type isn't void

You need to generate a method for each method on the interface (#1). The key aspect to notice is the way the implementation is done. You look to see if the callback object has a method that matches the signature of the interface method. If so, you call the method on the callback object (#2). Otherwise, you return the default value of the interface method's return type if it's not void (#3).

You call `FindMethod()` to find a match on the callback object.

Listing 2 Finding a method match on the callback object

```
private MethodInfo FindMethod(
  MethodInfo[] callbackMethods, MethodInfo interfaceMethod)
{
  MethodInfo result = null;

  foreach (var callbackMethod in callbackMethods)
  {
    if (callbackMethod.ReturnType == #1
      interfaceMethod.ReturnType)    #1
    {
      var callbackParameters =                       #2
        callbackMethod.GetParameters();              #2
      var interfaceParameters =                      #2
        interfaceMethod.GetParameters();             #2

      if (callbackParameters.Length ==               #2
        interfaceParameters.Length)                  #2
      {                                              #2
        var foundDifference = false;                 #2

        for (var i = 0;                              #2
          i < interfaceParameters.Length; i++)       #2
        {                                            #2
          if (callbackParameters[0].ParameterType != #2
            interfaceParameters[0].ParameterType)    #2
          {                                          #2
            foundDifference = true;                  #2
            break;                                   #2
          }                                          #2
        }                                            #2
```

```
            if (!foundDifference)      #3
            {
              result = callbackMethod; #3
              break;                   #3
            }                          #3
          }
        }
      }

      return result;
    }
```

You iterate through each method on the callback object. First, you check the return types to see if they match (#1). If they do, then you look at each parameter's type in order to see if they match (#2). If there are no differences in the parameter types, you've found a match, and that's what you return (#3).

The `GetMethod()` method returns a stringified version of a `MethodInfo` that can be used in C# code generation:

Listing 3 Generating C# for a method definition

```
    private static string GetMethod(
      MethodInfo method, bool includeTypes = true)
    {
      var result = new StringBuilder();

      if (includeTypes)
      {
        result.Append(method.ReturnType == typeof(void) ? "void " :
          method.ReturnType.FullName + " ");
      }

      result.Append(method.Name + "(");
      result.Append(string.Join(", ",
        from parameter in method.GetParameters()
        select (includeTypes ?
          parameter.ParameterType.FullName + " " + parameter.Name :
          parameter.Name)));
      result.Append(")");
      return result.ToString();
    }

    private Type CallbackType { get; set; }
    public string Code { get; private set; }
    private Type InterfaceType { get; set; }
    private string MockName { get; set; }
    private string InterfaceTypeName { get; set; }
  }
```

Again, note that you're using full type names for the return type (if it's not `void`) and the parameter types.

Let's go through a quick example to see what the generated code looks like. Consider the following interface:

```
public interface ITest
{
  void CallMe(string data);
  int CallMe();
}
```

Now, let's say you defined a callback object like this:

```
public sealed class TestCallback
{
  public int Callback()
  {
    return new Random().Next();
  }
}
```

Note that the `Callback()` method matches the signature of `CallMe()` in `ITest`, but `TestCallback` doesn't implement `ITest`. A generated mock for `ITest` that uses `TestCallback` would look something like this:

```
[System.Serializable]
internal sealed class TestCallbackMock
```

```
    : DynamicMocks.Roslyn.Tests.ITest
{
    private DynamicMocks.Roslyn.Tests.TestCallback callback;

    public TestCallbackMock(
           DynamicMocks.Roslyn.Tests.TestCallback callback)
    {
           this.callback = callback;
    }

    public void CallMe(System.String data){}

    public System.Int32 CallMe()
    {
           return this.callback.Callback();
    }
}
```

The mock defines two methods to implement the `ITest` method, but the `CallMe()` method that takes a `string` doesn't do anything. The `CallMe()` method calls `Callback` on the `TestCallback` object as the signature matches.

Now you have the ability to generate a mock in C#. Next, you'll see how you can compile this with Roslyn.

Compiling the mock code

You have the ability to create a C#-based mock. Let's compile that code with Roslyn.

Listing 4 Compiling code at runtime with Roslyn

```
public static class Mock
{
  private static readonly Lazy<ModuleBuilder> builder =
    new Lazy<ModuleBuilder>(() => Mock.CreateBuilder());

  public static T Create<T>(object callback)
    where T : class
  {
    var interfaceType = typeof(T);       #1

    if (!interfaceType.IsInterface)      #1
    {                                    #1
      throw new NotSupportedException(); #1
    }                                    #1

    var callbackType = callback.GetType();         #2
    var mockName = callbackType.Name +             #2
      Guid.NewGuid().ToString("N");                #2

    var template = new MockCodeGenerator(mockName, #2
      interfaceType, callbackType).Code;           #2
    var compilation = Compilation.Create("Mock",                #3
      options: new CompilationOptions(                          #3
        assemblyKind: AssemblyKind.DynamicallyLinkedLibrary),   #3
      syntaxTrees: new[]                                        #3
      {                                                         #3
        SyntaxTree.ParseCompilationUnit(template)               #3
      },                                                        #3
      references: new MetadataReference[]                       #3
      {                                                         #3
        new AssemblyFileReference(                              #3
          typeof(Guid).Assembly.Location),                      #3
        new AssemblyFileReference(                              #3
          interfaceType.Assembly.Location),                     #3
        new AssemblyFileReference(                              #3
          callbackType.Assembly.Location)                       #3
      });                                                       #3

    var result = compilation.Emit(Mock.builder.Value);        #4

    if (!result.Success)                                      #4
```

```
    {                                                          #4
      throw new NotSupportedException(                                  #4
        string.Join(Environment.NewLine,                                #4
          from diagnostic in result.Diagnostics                         #4
          select diagnostic.Info.GetMessage()));                        #4
    }                                                                   #4

    return Activator.CreateInstance(                       #5
      Mock.builder.Value.GetType(mockName), callback) as T; #5
  }

  private static ModuleBuilder CreateBuilder()                      #6
  {                                                                 #6
    var name = new AssemblyName                                     #6
    {                                                               #6
      Name = Guid.NewGuid().ToString("N")                           #6
    };                                                              #6

    var builder = AppDomain.CurrentDomain.DefineDynamicAssembly(    #6
      name, AssemblyBuilderAccess.Run);                             #6
    return builder.DefineDynamicModule(name.Name);                  #6
  }                                                                 #6
}
```

#1 Ensures the generic type is an interface
#2 Creates the mock code
#3 Compiles the mock
#4 Emits the mock into a dynamic assembly and check for success
#5 Returns a new instance of the mock
#6 Creates a new dynamic assembly

The first thing to do is check that `T` is actually an interface (`#1`). Once you've verified that, you use the `MockCodeGenerator` class to create the mock code (`#2`). You pass that generated code to the `Create()` method of the `Compilation` class via a syntax tree. This syntax tree is created by `SyntaxTree.ParseCompilationUnit()`. You also pass `AssemblyFileReference` objects so the compiler knows where the types that are referenced in the mock code (`#3`) are. Once `Create()` is done, you can emit the results into a dynamic module. If `Emit()` wasn't successful, you can check the `Diagnostic` property to find out what isn't correct in your code (`#4`). Finally, a new instance of the mock is created with a reference to the callback object (`#5`). Note that the dynamic assembly is lazily created (`#6`).

With all of this in place, you can now create a mock using the `ITest` and `TestCallback` classes like this:

```
var callback = new TestCallback();
var mock = Mock.Create<ITest>(callback);
var result = mock.CallMe();
```

When this code executes, the result will contain a random integer value.

With Roslyn, you can easily create dynamic code that does complex activities. Rather than resorting to System.Reflection.Emit and IL, you can simply write your C# and compile that instead. The barrier to entry to perform powerful metaprogramming-based implementations is much lower with Roslyn.

The last thing you need to look at are the trees that Roslyn produces. This will become important when you start writing code that interacts with code written in Visual Studio.

Understanding trees

As you saw in the last example, you used `SyntaxTree.ParseCompilationUnit()` to create a tree structure that represents the code you pass into the method. As you can imagine, these trees are rich and complex—in fact, when you install Roslyn, you get a couple of visualizers to help you see the tree. The following screenshot is the debug visualizer that is a representation of the mock code as a tree:

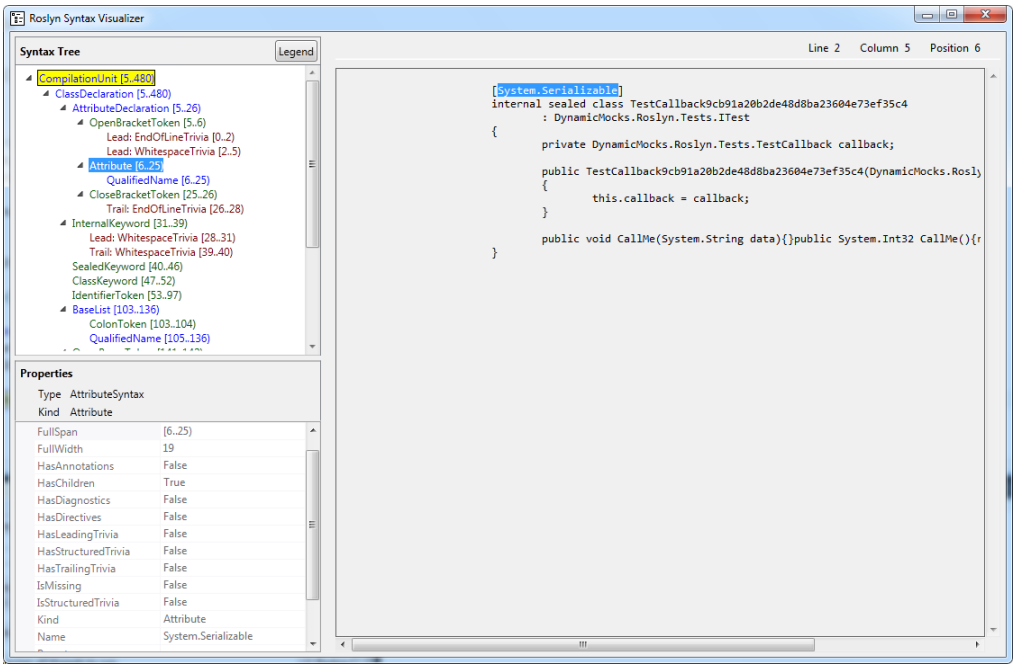


Figure 7 The Roslyn syntax debugging visualizer. If you look at certain objects (like a SyntaxTree) in your Autos window, you can see the tree (and its corresponding code) in the visualizer.

Installing the visualizers

To learn more about using the Roslyn visualizers, please visit this page: http://blogs.msdn.com/b/visualstudio/archive/2011/10/19/roslyn-syntax-visualizers.aspx. Note that there's a bug with the debugging visualize in the CTP. You can fix it by following the steps listed here: http://social.msdn.microsoft.com/Forums/en-US/roslyn/thread/f5adeaf0-49d0-42dc-861b-0f6ffd731825.

The syntax tree is a complete representation of the code you parsed, including whitespace (known as trivia). This is necessary because, if you want to retain code formatting as you change it, you need to know what kind of whitespace exists in the document (and how much of it).

Trees in Roslyn are immutable. That is, you can't change the contents of a tree. Immutable structures have many advantages—for example, they can make concurrent programming much easier to reason about—but, since they're immutable, you can't change them. Thankfully, Roslyn ships with a couple of visitor classes, `SyntaxWalker` and `SyntaxWriter` that you can use to search the contents of a tree and create a new tree based on a given tree.

Summary

You've just seen a preview of the Roslyn API. You saw how Roslyn provides you with a rich view of your code with parsers that provide tokens and trees. You were able to use this functionality to generate and execute C# code at runtime.

Here are some other Manning titles you might be interested in:

Silverlight 5 in Action
Pete Brown

ASP.NET 4.0 in Practice
Daniele Bochicchio, Stefano Mostarda, and Marco De Sanctis

Entity Framework 4 in Action
Stefano Mostarda, Marco De Sanctis, and Daniele Bochicchio

Last updated: August 22, 2012