Code Generation in Microsoft .NET

KATHLEEN DOLLARD

Apress™

Code Generation in Microsoft .NET
Copyright ©2004 by Kathleen Dollard

ISBN (pbk): 1-59059-137-2

Printed and bound in the United States of America 12345678910

Technical Reviewer: Don Kiely

Editorial Board: Steve Anglin, Dan Appleman, Gary Cornell, James Cox, Tony Davis, John Franklin, Chris Mills, Steven Rycroft, Dominic Shakeshaft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wray, John Zukowski

Assistant Publisher: Grace Wong

Project Manager: Sofia Marchant

Copy Editor: Kim Wimpsett

Production Manager: Kari Brooks

Production Editor: Lori Bring

Proofreader: Linda Seifert

Compositor: Molly Sharp, ContentWorks

Indexer: Kevin Broccoli

Artist: Christine Calderwood

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc., 175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, email orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

Contents

CHAPTER 1

Introducing Code Generation

Principle #1: You have control of the templates that generate your code and can change them as required.

Code generation is code that writes code. By letting you automate the creation of a large portion of your application, code generation offers a radical shift in development. In addition to the obvious benefit of turbo-charging your development, code generation lets you maintain consistent code quality and allows your code to evolve quickly in response to metadata changes (including database changes). Code generation lets you extend reuse to include code with similar patterns, as well as code with identical code segments. The result is increased speed, reusability, agility, and consistency in your application development.

In the end, code generation can be as simple as supplying an Extensible Markup Language (XML) file with a few pieces of information and clicking a button, and—*voilà!*—the repetitive portions of your application (stored procedures, middle tier, and user interface) are complete and ready for you to extend with your own customization. The purpose of this book is to teach you the process that lies behind that click. It'll seem complex at times, but don't lose sight that you rarely deal with the complexity of the underlying tools and only occasionally deal with the complexity of the templates themselves. Code generation is about building these once and reusing them many, many times—within and between applications.

NOTE Both code generation and object-oriented inheritance allow you to reuse common functionality. They differ in that inheritance only allows you to reuse specific code segments that you can then override or modify in the derived class. Code generation allows you to reuse patterns in your code as well as code segments. Code generation and object inheritance are very effective when used together.

Generating code isn't new. What's new is that it actually works in the real world. In the past, I haven't been excited about using code generation supplied with frameworks and templates because they couldn't provide the control and flexibility I demanded. I felt like I was in a whitewater rapid without a paddle, rushing along quickly but quite uncertain which boulder I might hit. I even know one shop that abandoned its multiyear commitment to stored procedures, replacing them with dynamic Structured Query Language (SQL) to accommodate the limitations of a purchased framework.[1]

Sure, we could have done our own code generation three or even ten years ago, but code generation used to be hard work, and we didn't have enough collective knowledge to recognize patterns easily. A number of recent developments, including XML support, inheritance features in .NET, XSL Transformations (XSLT) processing, and wide-scale pattern recognition, make the time ripe for code generation to become part of your development process. I'll show you how to incorporate your own code generation, meaning you stay in control of the architecture you're implementing. I'll provide tools that support whatever techniques and patterns you use, and I'll provide samples you can use to generate applications as soon as you understand the underlying techniques. I'll share those techniques, along with guidelines for making code generation a valuable part of your development strategies. Those guidelines will include steps and principles that keep your code generation effective and high quality.

You can use several mechanisms for code generation. In this chapter, I introduce the three most important approaches for application code generation in .NET, along with the benefits and drawbacks of each. This lets you compare them and decide which fits you best. Chapter 3 has further details of using each mechanism. Continuing to walk through all three would be too cumbersome, and I want to focus on the underlying process common to them all. So, in later parts of the book, I'll focus on one mechanism.

In addition to showing the mechanisms, this chapter presents the five steps you'll use to make code generation a coherent process. Chapters 2–5 cover these steps in more detail. But, before I go any further in describing the five steps to code generation, I want you to see what code generation looks like. Generating a

1. The point here isn't whether stored procedures or dynamic SQL is a better approach but that critical decisions such as this should be made by your own organization, not a vendor.

"Hello World" application in each of the three mechanisms illustrates code generation in its barest form.

Exploring Mechanisms for Code Generation

The mechanisms for code generation in this chapter include an obvious approach, a Microsoft-provided approach, and a novel approach. The obvious approach is simply to write code that outputs code as text to a stream—.NET code directly writing .NET code. I call this *brute-force code generation*. I don't intend that name to be taken negatively. This is the direct approach of simply opening a stream and writing out code.

Microsoft provides the CodeDOM mechanism, which is designed specifically for generating code. The CodeDOM provides a layer of abstraction that lets you output code in C#, Visual Basic .NET (VB .NET), J#, and perhaps additional languages in the future—from the same template code base. Most developers don't have a compelling reason to output code in multiple languages, and you pay for this flexibility with significantly increased complexity and reduced capabilities.

The novel approach uses XSLT templates. You can use XSLT to create any type of text output, including .NET code. Although all three code generation mechanisms have their place, I'll show you why I think XSLT templates are the preferred solution for many code generation scenarios. Later examples in the book use XSLT templates, but the focus remains on the underlying ideas that are valid with any approach to code generation.

You could also create variations of these approaches. For example, I've worked a little on a preprocessor for XSLT. I've also worked with an alternate brute-force approach using token replacement that was simpler than XSLT. The complexity of XSLT makes such a simplified solution appealing. However, as you deal with more complex templates, you'll quickly move beyond token replacement to template logic for loops, conditionals, functions such as string manipulation, and calculations. You wind up rebuilding much of XSLT with a proprietary syntax. If you want to do code generation with any variation of the techniques in this book, you can extend the discussion to your scenario. This works because much of the book isn't about how you actually output code; it's about how you construct the surrounding processes to make code generation highly effective, and it's about what to consider when building any set of application templates.

In each of the three approaches, a pattern is expressed based on the metadata provided. *Metadata* is the information specific to your application that's collected from your database structure, entered manually, or retrieved from other sources. Chapter 2 discusses a variety of metadata sources. The pattern for your code is contained in an XSLT stylesheet for XSLT code generation and a dedicated class for brute-force or CodeDOM generation. In all cases, I use the phrase *template* to refer to whatever holds the pattern, whether it's an XSLT template or a .NET class.

Mechanisms Not Covered

Visual Studio and related technologies present many options for generating code. Because this book sets out to change the way you write *applications* by giving you a full-scale development approach, I'll only cover those mechanisms that work well for full-scale application development in at least some scenarios. But you might be curious about other approaches.

.NET provides the `Reflection.Emit` namespace that allows you to emit Microsoft Intermediate Language (MSIL). All .NET languages compile to MSIL for deployment. The Just-In-Time (JIT) compiler that translates your MSIL into machine code takes MSIL as input. Because MSIL is designed to make the JIT compiler's job easier, MSIL has more similarities to machine language than to high-level human readable languages such as C# and VB .NET. In order to use `Reflection.Emit` for code generation, you have to either write your application templates to output MSIL or create your own precompiler. With the great high-level languages, tools, and compilers available, why would you want to work directly with MSIL for general application development? `Reflection.Emit` might be interesting if you want to generate classes on a small scale at runtime, but features of the CodeDOM also provide runtime compilation.

The Visual Studio environment exposes an extensibility layer that allows you to access your source code through the Visual Studio automation model. The automation model might be useful if you want to integrate your code generation into the Integrated Development Environment (IDE), but I'm not convinced you'd gain much, and integrating into Visual Studio is really a separate issue from the process of generating code. The Visual Studio code automation model can output C# code but is read only for VB .NET. VB .NET trades outputting code through an automation model for its boatload of cool IDE features.

Two other approaches are Enterprise Templates and Visio code generation. I skip them because they don't provide a full cycle of development and regeneration. Without an incredible amount of work, Visio only outputs method stubs, and no amount of work with the current version provides a regeneration cycle. In the future I hope you'll be able to harvest Unified Modeling Language (UML) metadata from Visio, but even this is quite difficult currently. See the "Additional Reading" section of Chapter 2 if you're interested in Visio code generation. Finally, the Enterprise Templates feature may play a role in your development, but it's not a full application approach to code generation. The templates are intended as a starting point and don't support regeneration.

Table 1-1 later in this chapter summarizes these techniques, along with the ones I'll discuss in detail.

Generating a Simple Program

Looking at a "Hello World" program in each of the three primary code generation mechanisms lets you focus on the similarities and differences in the mechanisms themselves. Listing 1-1 shows the target "Hello World" program.

Listing 1-1. A Target "Hello World" Program for a Preliminary Look at Code Generation

```
Option Strict On
Option Explicit On

Imports System

' Class Summary: Hello World target output

Public Class TargetHelloWorld

#Region "Public Methods and Properties"
   Public Shared Sub Main()
      Console.WriteLine("Hello World")
   End Sub
#End Region

End Class
```

In the real world, this isn't a good candidate for code generation because it isn't a pattern applied with predictable variations. It's always the same, so it's easier to write this "Hello World" program without code generation than with it. But the goal here is to strip code generation down to its naked essentials to illustrate the underlying process.

NOTE The code you can download for this book (in C# and VB .NET) will include the best practices I use. In VB .NET I always use `Option Strict On` and `Option Explicit On` for strict typing. In VB .NET, I include `Imports System` and limit other imported namespaces to leave part of the namespace in each class reference. `Imports System` provides access to the system data type names, which I prefer for clarity. I also heavily use regions. Regions make it easier to trace between the output code and the emitting code while you're debugging your templates. I'll rarely use the VB library, which, among other things, makes it easier for C# developers to read my code. I'll avoid setting options or imports at the project level, preferring that each file be explicit and self-contained. These details aren't always evident in the code fragments printed in the book. You can assume that all VB .NET code is strictly typed (`Option Strict` and `Option Explicit` both set to `On`).

TIP Specify code options in generated code, and don't rely on project settings. You can't be sure of what project setting will be in use in the generated application.

Generating "Hello World" via Brute Force

The obvious approach to code generation is to create a stream writer and start outputting code. The big benefit of this method is that it requires only a basic understanding of writers and streams, which you may already have. The vast majority of programmers I've talked to who have experimented with code generation have used this approach. Later you'll see one way to organize your templates to make this option easier. Brute-force code generation doesn't *impose* the same level of organization on you that XSLT code generation does, but with some discipline you can manually supply this same high level of organization, and it'll benefit your code generation process.

Creating the Template

The procedure that creates the sample "Hello World" program has a single shared method named `GenerateOutput`. It creates a `StreamWriter` that implicitly creates a stream. `WriteLine` outputs each line of the target code. The `Finally` block flushes and closes the stream:

```
' Class Summary: Create Hello World program using direct stream output
Public Class HelloWorldViaBruteForce

#Region "Public Methods and Properties"
   Public Shared Sub GenerateOutput(ByVal outputFileName As String)
      Dim writer As IO.StreamWriter

      Try
         writer = New IO.StreamWriter(outputFileName)
         writer.WriteLine("Option Strict On")
         writer.WriteLine("Option Explicit On")
         writer.WriteLine("")
         writer.WriteLine("Imports System")
         writer.WriteLine("")
         writer.WriteLine("' Class Summary: Hello World target output")
         writer.WriteLine("")
         writer.WriteLine("Public Class TargetOutput")
         writer.WriteLine("")
         writer.WriteLine("#Region " & Chr(34) & _
                     "Public Methods and Properties" & Chr(34))
         writer.WriteLine("   Public Shared Sub Main()")
         writer.WriteLine("      Console.WriteLine(" & Chr(34) & _
                    "Hello World" & Chr(34) & ")")
         writer.WriteLine("   End Sub")
         writer.WriteLine("#End Region")
         writer.WriteLine("")
         writer.WriteLine("End Class")
      Finally
         If Not writer Is Nothing Then
            writer.Flush()
            writer.Close()
         End If
      End Try
   End Sub
#End Region

End Class
```

This code resembles what many people who've played a bit with code generation have done. It uses a standard `StreamWriter` and explicitly handles horizontal whitespace via hard-coded spaces.

NOTE Outputting an empty string ("") using `WriteLine` creates a blank line in your output.

I find the layering of .NET code on .NET code in this sample to be rather mind bending. `Option Strict` appears twice, at the top of the code and again as a quoted string to be output. I find it difficult to read the code and mentally separate the instructions that are currently running from those that are part of the template. Imagine extending this to hundreds or thousands of lines of code-generating source!

Other than this problem and some ugliness in the quotes, this code is straightforward. This code works very well and reliably, and you don't have to learn anything new to use it. These are important benefits, but the long-term maintainability of code with long output blocks is poor—unless you're vigilant about your template organization. Otherwise, it's difficult to trace problems in the output back to the generating code. In Chapter 3, I'll show you how organizational tricks such as regions along with separate methods to generate each region make it easier to trace between your templates and output code.

Running the Template

Because the sample manages its own streams, generating the code for this simple brute-force sample just requires calling the shared method of the template class:

```
' Generate Hello World using Brute Force
HelloWorldViaBruteForce.GenerateOutput(IO.Path.Combine(outputDir, _
               "HelloWorldViaBruteForce.vb"))
```

NOTE Chapter 3 shows a more sophisticated way to run brute-force templates that makes it easy to output multiple files, such as running a particular template for every table in your metadata.

Generating "Hello World" via CodeDOM

Remarkable changes have occurred at Microsoft during the past five years. Evidence of Microsoft's increasing openness is the development of internal tools as first-class features, such as the CodeDOM. Visual Studio .NET uses the CodeDOM to create the strongly typed DataSet and in several places to create User Interface (UI) supporting code. The CodeDOM allows you to create an abstract model of your code as a containment hierarchy of objects that represent every namespace, class, method, statement, expression, comparison, variable occurrence, and so on. This abstraction isn't simple. It results in a tree diagram, or CodeDOM graph, of the code. Unless you're a compiler jock, this graph is probably quite different from the sequential manner you generally use to think about the code in your applications.

There's one payback for this effort. Once you create an abstraction of your target code, the code generator can create the code in any supported language.

You can even output code in new languages as soon as someone creates a CodeDOM provider. Unfortunately, this benefit comes at a high price because code generation using the CodeDOM is quite complex, and many language features aren't supported. Chapter 3 goes into these issues in more detail.

Running the Template

Before looking at how you create a CodeDOM graph, you'll learn how you use it to output code. There are two steps to generation—creating a CodeDOM graph and generating the code. The first step creates a `CodeCompileUnit` as a container for the CodeDOM graph as well as some other information using `BuildGraph`. A separate step passes this `CodeCompileUnit` to the `GenerateViaCodeDOM` method that generates the code. This method takes the `CodeCompileUnit` and a `CodeDomProvider` as parameters. The `CodeDomProvider` is language specific and indicates how to translate the CodeDOM graph into actual code. Microsoft supplies CodeDOM providers for C#, VB .NET, and J#. To illustrate the flexibility of the CodeDOM, I use a single `CodeCompileUnit` with both a C# and a VB .NET provider. The result is code output in two languages:

```
Dim compileUnit As CodeDom.CodeCompileUnit
Dim provider As CodeDom.Compiler.CodeDomProvider
compileUnit = HelloWorldViaCodeDOM.BuildGraph()
provider = New Microsoft.VisualBasic.VBCodeProvider
GenerateViaCodeDOM(IO.Path.Combine(outputDir, "HelloWorldViaCodeDOM.vb"), _
               provider, compileUnit)
' Use same compile unit to generate C# Hello World
provider = New Microsoft.CSharp.CSharpCodeProvider
GenerateViaCodeDOM(IO.Path.Combine(outputDir, "HelloWorldViaCodeDOM.cs"), _
               provider, compileUnit)
```

NOTE `BuildGraph` is the template that creates the CodeDOM graph and is shown in the next section "Creating the Template."

The `GenerateViaCodeDOM` method contains generic code to process any compile unit. You can call this method multiple times, with different providers to generate code in different languages. The `GenerateViaCodeDOM` method creates a

.NET CodeGenerator from the CodeDOM provider you pass. Like the CodeDOM provider, the CodeDOM generator is language specific:

```
Private Shared Sub GenerateViaCodeDOM( _
             ByVal outputFileName As String, _
             ByVal provider As CodeDom.Compiler.CodeDomProvider, _
             ByVal compileunit As CodeDom.CodeCompileUnit)
   Dim gen As CodeDom.Compiler.ICodeGenerator = provider.CreateGenerator()
```

The GenerateViaCodeDOM method also creates an IndentedTextWriter to contain the output code. You'll see more about the IndentedTextWriter later in this chapter and in Chapter 3, but it allows easy control of horizontal whitespace in the output. The IndentedTextWriter wraps a stream, and writing to a stream gives you flexibility in how you later use the generated code:

```
Dim tw As CodeDom.Compiler.IndentedTextWriter

Try
   tw = New CodeDom.Compiler.IndentedTextWriter(New IO.StreamWriter( _
               outputFileName, False), "    ")
```

Once you have everything ready, outputting code is a single call to the GenerateCodeFromCompileUnit method of the code generator. This example uses a new CodeGeneratorOptions object with the default values for generator options:

```
gen.GenerateCodeFromCompileUnit(compileunit, tw, _
            New CodeDom.Compiler.CodeGeneratorOptions)
```

The remainder of the method handles cleanup in a Finally block that ensures you flush and close the stream before leaving the method:

```
   Finally
      If Not tw Is Nothing Then
         tw.Flush
         tw.Close()
      End If
   End Try

End Sub
```

Creating the Template

Generating code in multiple languages is cool, no doubt, but you should consider the cost. Just how hard is it to create the CodeDOM graph for something as simple as the "Hello World" program? The `HelloWorldViaCodeDOM.BuildGraph` method creates a CodeDOM graph (a `CodeCompileUnit`) and returns it for further processing:

```
' Class Summary: Hello World via the CodeDOM

Public Class HelloWorldViaCodeDOM

#Region "Public Methods and Properties"
   Public Shared Function BuildGraph() As CodeDom.CodeCompileUnit
      Dim CompileUnit As New CodeDom.CodeCompileUnit
```

A CodeDOM graph always starts with a collection of namespaces. Each namespace contains `Imports` or C#'s `using` statements, comments, and types:

```
Dim nSpace As New CodeDom.CodeNamespace("HelloWorldViaCodeDOM")
CompileUnit.Namespaces.Add(nSpace)
nSpace.Imports.Add(New CodeDom.CodeNamespaceImport("System"))
```

Types can be structures, enums, delegates, classes, and so on. This code creates a type that's a class named `Startup` and adds it to the namespace:

```
Dim clsStartup As New CodeDom.CodeTypeDeclaration("Startup")
nSpace.Types.Add(clsStartup)
```

Types contain members. Members can be fields,[2] methods, properties, and so on. You can specify different details depending on the member category, such as an initial value for fields or a collection of statements for methods. `CodeEntryPointMethod` is a member of the `Startup` class that specifies a method used to start up the application:

```
Dim main As New CodeDom.CodeEntryPointMethod
```

So far, the CodeDOM requires extra work, but the approach is logical and intuitive.

2. Class-level variables are also called *fields*.

Building individual statements is the difficult and probably nonintuitive part of using the CodeDOM. You separately create each piece of the statement—each variable, literal expression, operator, invocation, and so on. Variables are created for the literal "Hello World," for the reference to the console class, and for the invocation of the `WriteLine` method using these two variables. Once you have the statement created, you add it to the statement collection of the method:

```
      Dim exp As New CodeDom.CodePrimitiveExpression("Hello World!")
      Dim refExp As New CodeDom.CodeTypeReferenceExpression("System.Console")
      Dim invoke As New CodeDom.CodeMethodInvokeExpression( _
               refExp, "WriteLine", exp)
      main.Statements.Add(New CodeDom.CodeExpressionStatement(invoke))

      clsStartup.Members.Add(main)

      Return CompileUnit
   End Function
#End Region

End Class
```

Whew! Some of that didn't look too bad, but there are four lines of weird code for the one line of output:

```
Console.WriteLine("Hello World")
```

And, yes, you have to write code like that for almost every line of .NET code in your target file—and, yes, it gets worse with less simplistic code. That's the core of why the CodeDOM is hard. By abstracting to this detailed level, it's hard to read the template to predict the output, and you can't search in it. Even worse, it isn't going to produce the same output. Some things require extra work, sometimes with poorly documented features. For example, vertical whitespace, VB .NET option statements, and regions are all difficult or impossible to produce via the CodeDOM.

It's easy to see why you have to work with this complete abstraction to meet a goal of complete language independence. The generator slams these fragments together quite differently to build code in the different languages. This stuff gets really sticky as the complexity increases. There are 74 classes in the CodeDOM namespace itself as well as many additional classes in the `CodeDOM.Compiler` namespace. The CodeDOM is workable only for programmers who are tenacious, deal well with abstractions, and have plenty of time.

Plain-Vanilla Code

Another fundamental issue plagues the CodeDOM. For the CodeDOM to create code valid in all languages, the abstraction can only work with elements found in all languages. You're forced to write code from a least common denominator approach, using only elements that appear in all languages. For example, you can't use VB .NET's project-level namespaces or `WithEvents`, `With`, or `Event`. You have to understand subtle but important differences in constructs such as between VB .NET's `Imports` and C#'s `using`. There's no built-in support for preprocessor directives, including region directives. The CodeDOM doesn't even support all features currently provided by both languages (see Chapter 3 for details). Also, it isn't clear how frequently the CodeDOM will be updated with new language features as .NET evolves because the CodeDOM wasn't updated to reflect new language features between .NET 1.0 and 1.1 (Visual Studio 2002 and 2003).

And to make this worse, I know of no reference you can rely on to list all of the caveats and special considerations (except the ones I discuss in Chapter 3 and Appendix D). The result is that not only do you have to write plain-vanilla code, but you have to think that way. This effort is only justified if your job is specifically to create code in multiple languages and you can afford to ignore some of the best features in each language.

Generating "Hello World" via XSLT Templates

The third approach is relatively novel in .NET. This is code generation using XSLT templates. XSLT is a template language designed to create any type of text, XML, or Hypertext Markup Language (HTML) output from XML input. Because source code is text output, XSLT can create source code.

The XSLT language might be new to you, and it might appear to be a strange paradigm. It presents a new way of thinking because it's explicitly designed from the ground up to apply patterns to information in order to transform information into text. Because metadata is information and source code is text, XSLT does exactly the job presented by code generation.

TIP Appendix A introduces XML, XPath, XML Schema Definition (XSD), and XSLT technologies targeted to what you need for code generation.

Understanding the Benefits of XSLT Code Generation

XSLT code generation presents a couple of compelling benefits. First, the template is closer in appearance to your output code. You can create a functioning sample class, copy code from the sample class into the XSLT template shell, and add tags to indicate the variable content that depends on metadata information. The templates are far more readable, editable, and easily searchable for .NET elements. It's much easier to segment processing to isolate complex sequences and mimic the resulting document sufficiently to support continued maintenance of templates.

When you're working with code templates, searches can be problematic, especially if you're using CodeDOM or brute force because you're overlaying the syntax of the output code with the code that runs the template. For example, regions are your primary tool in correlating output with the part of the template that produces it. If you search for the word *region*, your results will include the regions that are part of the generating code itself, as well as those destined for your target output. String concatenations further complicate your searches, especially when your target contains quotes. XSLT avoids these problems almost entirely because the syntax of XSLT differs markedly from the syntax you're outputting. XSLT is also flexible about the use of single and double quotes, allowing easier handling of nested strings.

An important secondary benefit is that XSLT code generation doesn't allow you to cheat on the isolation of metadata. Later in this chapter and in Chapter 2, you'll see the benefits of isolating metadata and its creation. In a brute-force or CodeDOM approach, you could mix running queries that grab fragments of information into the code outputting process. Mixing these two processes makes debugging and reuse difficult. XSLT forces this separation of metadata collection and code generation. You can also supply this separation manually with brute-force or CodeDOM generation.

In Chapter 3 you'll see generating code as a process that contains many steps, and you'll see how to orchestrate these steps via a script. I call the tool that runs this script the *code generation harness*, or just the *harness*. When you're working with the code generation harness, you can edit XSLT and regenerate without stopping and restarting the harness. Because brute-force and CodeDOM templates are compiled .NET code, the modules containing them are loaded in the harness's process space. These modules aren't unloaded, so you have to stop the harness before you compile changes to the templates. Because no method of code generation avoids syntax errors, you'll be generating code, finding syntax errors, correcting your code templates, and rerunning the generation. The turnaround for fixing common silly mistakes will be significantly faster with XSLT because you don't need to stop and restart the code generating application.

The core of XSLT is XPath, which you have to learn anyway. XPath is the language used to query XML within .NET (and most other technologies). Even if you're doing code generation through one of the other techniques, you'll be learning and using XPath if you're using XML metadata.

Running the Template

To create the output, the example calls a generic `GenerateViaXSLT` method, passing the name of the template:

```
' Generate Hello World via XSLT Template
GenerateViaXSLT(IO.Path.Combine(xsltDir, "HelloWorld.xslt"), Nothing, _
               IO.Path.Combine(outputDir, "HelloWorldViaXSLT.vb"))
```

TIP Shared methods of the `IO.Path` class allow you to do various cool things with file paths. In this case, `IO.Path.Combine` combines paths, letting you ignore whether intervening backslashes are handled correctly and other details.

`GenerateViaXSLT` wraps the powerful XSLT features of the `System.Xml.Xsl` namespace. This method takes the name of the XSLT file, an XML document to transform, and the name of the output file. The method that provides the outputting is generic because the entire template is contained in the XSLT file. You can pass parameters, but the "Hello World" program requires neither an input XML file nor parameters:

```
Private Shared Sub GenerateViaXSLT( _
             ByVal xsltFileName As String, _
             ByVal xmlMetaData As Xml.XmlDocument, _
             ByVal outputFile As String, _
             ByVal ParamArray params() As xsltparam)
```

Objects created from several XML and XSLT classes handle the translation. The `XslTransform` object performs the actual translation. XSLT works most efficiently if the input XML is an `XPathNavigator` because `XPathNavigator` objects have internal features for efficient XPath access. The `StreamWriter` takes the output of the transformation:

```
Dim xslt As New Xml.Xsl.XslTransform
Dim xNav As Xml.XPath.XPathNavigator
Dim streamWriter As IO.StreamWriter
```

This method creates an XML document if you don't pass one because the transformation needs at least an empty XML document for input. It also creates XSLT parameters if you pass any parameter values:

```
Dim args As New Xml.Xsl.XsltArgumentList
Dim param As XSLTParam
```

```
Try
   If xmlMetaData Is Nothing Then
      xmlMetaData = New Xml.XmlDocument
   End If

   For Each param In params
      args.AddParam(param.Name, "", param.Value)
   Next
```

The method then creates the navigator from the XML document and creates the `StreamWriter` connected to the output file:

```
xNav = xmlMetaData.CreateNavigator()
streamWriter = New IO.StreamWriter(outputFile)
```

Once everything is ready, the method just loads the XSLT file and performs the transform. When the call to the `Transform`[3] method is complete, the code is output although you won't see it if you forget to flush the stream:

```
xslt.Load(xsltFileName)
xslt.Transform(xNav, args, streamWriter, Nothing)
```

As with other approaches, flush and close before you leave the method. Placing this in the `Finally` block ensures it can't be bypassed:

```
   Finally
      If Not streamWriter Is Nothing Then
         streamWriter.Flush()
         streamWriter.Close()
      End If
   End Try

End Sub
```

Creating the Template

XSLT templates intersperse the .NET output code with XSLT directives. Not only is this more readable, but I created it by copying the target code into the stylesheet. Because I had already modified my default template so it contained the header I use, creating this template took less than 60 seconds.

3. The `Transform` method underwent considerable revision between .NET 1.0 and .NET 1.1. The changes mostly involved supporting XML resolvers. You probably won't need to be concerned with resolvers when doing code generation and can pass `Nothing` for the resolver.

TIP See Appendix A for information about how you modify your default XSLT template to provide the starting point you want.

Each XSLT stylesheet starts with some standard XSLT "goo," including a legal XML header. Although it isn't strictly required, XSLT is a specialized version of XML, so it's a good idea to include a legal XML header. XML requires a single root element, and for XSLT stylesheets this is the `xsl:stylesheet`[4] element. This element contains attributes defining the namespaces for the transformation. Appendix A discusses namespaces and the rest of the file opening goo in detail as well as shows you how to automate the header creation via your default template so you can fix it once and forget about it:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" encoding="UTF-8" indent="yes"/>
<xsl:preserve-space elements="*" />
```

Stylesheets contain templates, and templates do the real work of XSLT code generation. I copied the sample code into the single template in this stylesheet. XSLT will output verbatim any text that appears in a template that isn't an XSLT directive:

```
<xsl:template match="/">
Option Strict On
Option Explicit On

Imports System

' Class Summary:

Public Class TargetOutput

#Region "Public Methods and Properties"
   Public Shared Sub Main()
      Console.WriteLine("Hello World")
   End Sub
#End Region

End Class
</xsl:template>
</xsl:stylesheet>
```

4. `xsl:transform` is synonymous with `xsl:stylesheet`, and you can use either as the stylesheet root. I standardize on `xsl:stylesheet`.

Because XSLT is well-formed XML, template and stylesheet closing tags appear at the end of the file. That's it. That's all there is to XSLT code generation. (Okay, there's really more to it when you're generating real-world code with tokens and internal logic, but that's a start at seeing how you can copy code into your templates.)

Picking the Right Mechanism

To integrate any of these three mechanisms into your code generation process, you'll separate your application into code you'll generate and code you'll write by hand the old way, line by line in an editor. I call the manually written code *handcrafted code* because it brings to mind the respect I have for handcrafted items. If I build something in my wood shop, I take joy and pride in the crafts I shape by hand. But if I built furniture without power tools, I'd probably never finish, or the quality of the piece would suffer when I rushed. I simply can't rip[5] an 8-foot oak 2×4 efficiently with a handsaw.

Code is the same way. Some jobs are like ripping that long board. Doing them by hand is tedious, demands careful attention to detail, and requires significant skill to maintain the desired consistency. If you decide not to rip the board by hand and create the programmer's version of a table saw via code generation, you're creating the right tool for the job. That leaves you more time to focus on parts of the job that aren't appropriate for the table saw—or creating the parts of your application that aren't appropriate for code generation.

If you go into a well-stocked building supply store, you'll see three different categories of table saws. There are entry-level models that are quite easy to get started with because they're simple to use and require little investment. This is analogous to brute-force code generation. It's very accessible because the investment in learning new techniques is small, and it's a good fit for you if you only occasionally use the tool. These tools lack features, but with extra attention you can achieve high-quality output.

The store will also have some large, high-end, and extremely heavy floor models with large fixed out-feed tables. These models are cumbersome and expensive. This is analogous to the CodeDOM. The initial investment is quite large, and the ongoing maintenance is nontrivial. If you're a full-time cabinetmaker, this might be a good tool, but for the average woodworker, it's just

5. To *rip* a board means to cut it lengthwise. There's no doubt that some people can do it with only a manual tool. For example, hopeful miners built hundreds of plank boats from felled trees in a few months along the Yukon Trail. But it isn't an easy, quick, or fun job!

too much. Similarly, I wouldn't take on the long-term overhead of CodeDOM unless generating code in multiple languages was my project's core requirement.

The store will also have models that lie in the middle. These are the high-quality tabletop models. These have features such as powerful motors and adjustable blades, and they're built for long-term durability. They're not only the right tools for anyone with moderate needs, they're the right tools for anyone needing capacity, portability to job sites, reliability, and ease of use but not wanting a high initial investment or long-term overhead. Similarly, I prefer XSLT for code generation because I think the medium level of initial investment pays off in the long term.

NOTE Both my shop and my brain tend to be a bit overcrowded. My table saw can also serve as a router table, saving significant space. Similarly, you can reuse skills required for XSLT code generation in working with the XML Document Object Model (DOM) of Visual Studio (which uses XPath). XSLT is also useful for other transformations such as preparing data for HTML display or modifying the format of an XML document.

Each of these table saws performs the same basic tasks. The differences between them would be quite significant as you were selecting a model, but a single book could explain how to build furniture with any of the saws. Similarly, this book covers how to build applications with any of the code generation mechanisms. Unless there are underlying differences between the three, examples in later sections of this book illustrate only XSLT to keep things focused on the core aspects of code generation beyond how you spit out code.

Table 1-1 and Table 1-2 summarize seven potential methods of code generation along with some benefits and drawbacks of each. I've included all seven available approaches to put them in perspective and show why only three are viable for your application development and discussed further in this book. Table 1-1 shows the mechanisms I'll cover in this book, and Table 1-2 shows the mechanisms I won't discuss further (also discussed in the earlier sidebar "Mechanisms Not Covered").

Table 1-1. Code Generation Mechanisms Covered in This Book

	XSLT	BRUTE FORCE	CODEDOM
Covered in this book?	Yes	Yes	Yes
Viable for application code generation?	Yes	Yes	In some cases
Ease of searching	Easy	Difficult	Often impossible
Ease of pasting in sample code	Easy	Difficult	Effectively impossible
Multiple language output from one source?	No	No	Yes
Can generate stored procedures?	Yes	Yes	No
Restart generation process for syntax fixes?	Continue	Restart	Restart
VB option statements?	Yes	Yes	Yes, but undocumented
Changeable indent width?	Difficult	Yes, with `IndentedTextWriter`	Yes
Regions and vertical whitespace?	Yes	Yes	To some extent, but difficult
Other benefits	Relatively easy to read your code while developing code generation	No new language or way of thinking about code required	Code output in multiple languages
Drawbacks	Requires learning new language	Hard to read and maintain	Complex model with tons of classes; very hard to read

Table 1-2. Four Mechanisms for .NET Code Generation Not Covered

	REFLECTION.EMIT	IDE EXTENSIBILITY CODEMODEL	VISIO	ENTERPRISE TEMPLATES
Covered in this book?	No	No	No	No
Viable for application code generation?	No	No	No	No
Ease of searching	Difficult	Difficult	Impossible	Moderate
Ease of pasting in sample code	Impossible (unless you paste in MSIL)	Difficult	N/A	Moderate
Multiple language output from one source?	No, MSIL only	No	No	No
Can generate stored procedures?	No	No	No	No
Restart generation process for syntax fixes?	N/A	N/A	N/A	N/A
VB option statements?	N/A because no language-specific features	N/A	N/A	Yes
Changeable indent width?	N/A because no source code is produced	Yes	No	Yes
Regions and vertical whitespace?	N/A because no source code is produced	Yes	No	Yes
Other benefits	Can be used at runtime	Integrated into Visual Studio IDE	Integrates with UML model	Unified project starting point; allows Information Technology (IT) control over environment
Drawbacks	Code written is MSIL, not a high-level language; no high-level source code available for debugging	Can't produce VB .NET output	Only creates initial stubs; very hard to use with VB .NET because of a Visio bug	Only creates initial stubs

Breaking Down the Code Generation Process

If you separate the jobs you can do effectively with tools from those that deserve handcrafted attention, you'll get the job done faster, and your results will be more precise. Precision means consistency—you're reproducing the same thing. Accuracy means your output is correct. Code generation can make you precise, but it alone can't make you accurate.

For accuracy, you want a process that guides you to effective output. A full-cycle development methodology based on five steps of code generation lets you create better applications significantly faster.

Understanding the Five Steps of Code Generation

The conscious code generation process I use consists of five steps, as shown in Figure 1-1.

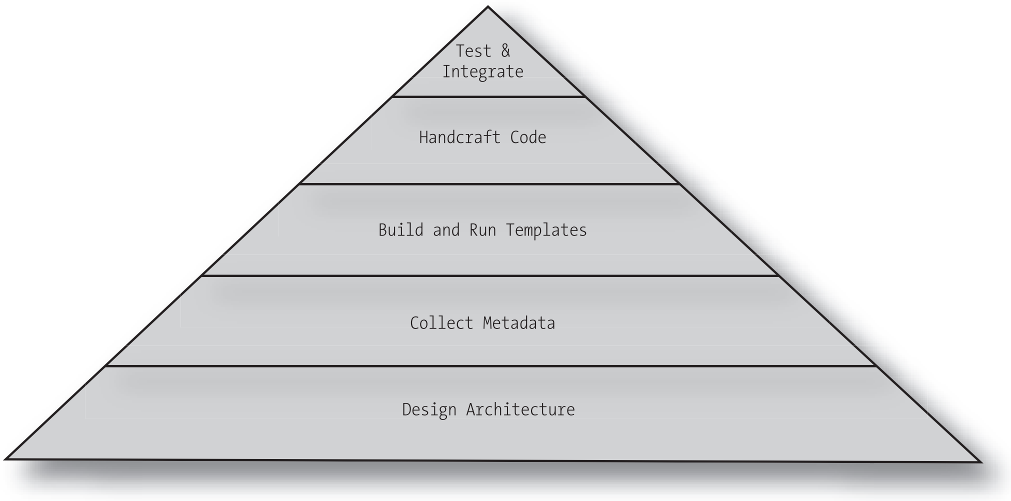

Figure 1-1. Each of the five code generation builds on the previous step.

As you can see from Figure 1-1, the five steps are as follows:

1. Design architecture.

2. Collect metadata.

3. Build and run templates.

4. Handcraft code.

5. Tie it together with integration and testing.

The pyramid in Figure 1-1 illustrates that you'll want a strong architectural basis for metadata collection. Your architecture determines what metadata you need. Strong architecture and valid metadata are the basis for accurate templates to generate code. Handcrafted code relies on generated code. You integrate and test the whole thing. Each step relies on the previous one.

These steps aren't always sequential, and there certainly isn't a waterfall-style progression between them. They generally occur in a sequence because each builds on the output of the previous step. For example, you need to know what you're building before you build it, and you can't tie it all together in implementation until you have a complete system. But the process lends itself to iterative refinement, as shown in Figure 1-2. Each step lays the groundwork for the next step, but it also provides information for refining the previous step.

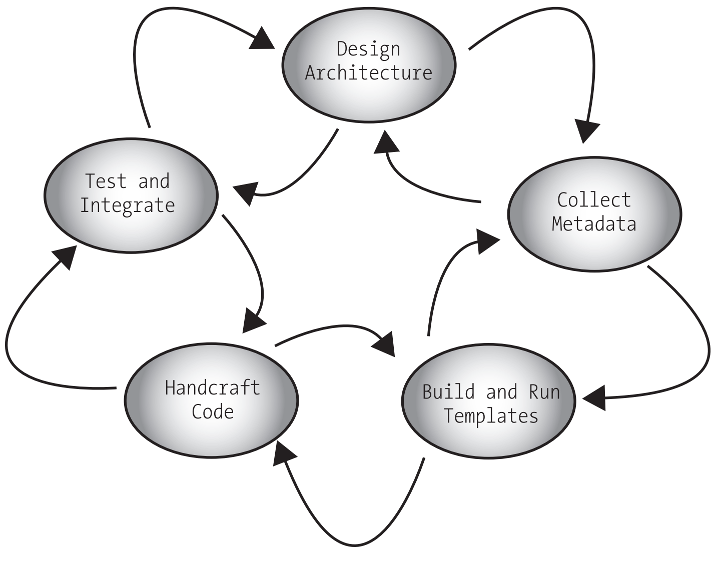

Figure 1-2. The process is cyclical, with each step contributing to the next and also providing more information for improving the previous step.

Designing Architecture

Architecture overarches the other steps. It tells you how to build your templates and what metadata to collect. *You* design the architecture for code generation. It doesn't tie you to any particular architectural pattern.

NOTE Although you can use whatever architecture you'd like, Chapter 8 presents templates for a very robust middle-tier architecture based on Rockford (Rocky) Lhotka's Component-based, Scalable, Logical Architecture (CSLA). Chapter 8's "Additional Reading" section contains a full reference to this book. Chapters 9 and 10 use this middle tier in generated Windows and Web user interfaces.

Many of you are working in an environment where your architectures are already developed and you know what you want to build. Specifically, I assume you probably use a three-tier or n-tier architectural approach—or at least understand that terminology. If that's not the case, there are other books focusing on architecture. The highest payback for code generation is creating middle-tier data containers and data access layers, so a lot of the code in this book focuses on those areas.

NOTE *Design* and *architecture* are vague, fluffy terms. I don't want to challenge your current semantic understanding of them. From a code generation perspective, design and architecture encompasses all the work you do to understand what your code generation output should look like.

The easiest way to move from design/architecture to code templates is to create a sample file and test it. This example source code file illustrates what you're creating, and I'll call it your *sample file* or *target file*. Once you know it's right (you can significantly enhance it later), you can express it in any of the code generation techniques using a varying amount of cut and paste. You can then confirm that your code generation is doing exactly what you want by comparing your generation output with your target file using WinDiff, another comparison tool, or even Word. I'm so reliant on this technique that even when creating the "Hello World" examples, I first created a target source code file and then copied and pasted it into the template.

Transferring code from a target file to the template is one of the places I prefer XSLT code generation. If I'm using brute-force or CodeDOM generation, I have to split the code into individual output statement strings. With XSLT, I copy the entire target file in, then replace variable blocks, delete some code, and organize the result.

TIP Create and test a sample class for each template. As you translate your sample code into templates, you'll discover or confirm your metadata requirements.

Collecting Metadata

Once you know what you're going to build, you need to capture the metadata that drives the code generation process. This metadata defines details of your application, such as database tables, fields, and stored procedure details. The metadata tailors the generated code to your specific application and environment. Metadata literally tells the template patterns how to implement themselves, so you can't create quality code-generated systems without quality metadata. For simplicity, I'll assume you're expressing metadata as XML. This is the most straightforward mechanism and is immediately accessible to all programmers on all platforms. It's also the required metadata format for XSLT code generation.

Metadata for business entities or middle-tier data containers are generally based either on your database or a parallel description of your business objects. This metadata includes the predictable stuff such as column names, types, relations, and nullability. It potentially includes a great deal of additional information, including constraints, captions, and information to create lookup combo boxes. All three output mechanisms directly use this metadata.

In Chapter 2, you'll see details of metadata collection. Chapter 6 extends metadata to a simplified Object Relational Mapping (ORM[6]), which allows you to define indirect mappings between your database and middle tier. Chapter 9 then shows the benefit of preprocessing your input metadata to tune it to the requirements of a specific set of templates.

Outputting Code

You need tools to perform code generation. You need to be able to do your code generation at the touch of a button. Your tools need to provide both flexible and reproducible code generation. Flexibility is important because during development you'll probably want to create a single class to initially test your generation (this saves you from generating hundreds of repetitive syntax errors). You also need reproducibility, including being able to perform the same code generation

6. ORM is one of many phrases and acronyms ambiguously used for different concepts. It can also mean Object-Role Modeling, which is an aspect of UML.

two or three years from now to support a change made during your maintenance cycle. Chapter 3 includes a tool to manage code generation via any combination of these three mechanisms.

You may begin to write a large percentage of your application using code generation. The percentage of your application you can generate depends on your specific application and how much of it can be tied to patterns. Just like recognizing common code is a skill necessary for traditional refactoring into generic subroutines, it takes skill to find similar patterns.[7] A side benefit of code generation is that you're forced to decompose your application into these patterns, learning more about how applications are actually structured. Once written, each template can be applied to a near infinite number of individual metadata definitions. The only real limit is the size of the XML metadata file, and the pragmatic limit is the practicality of the resulting runtime.

TIP Although you can use any of the three generation mechanisms described in this book (or variations on them), you'll probably want to settle on one mechanism. You can't really combine them within the generation of one file, and you'll limit reuse if you use different techniques to create different output files. However, the harness in Chapter 3 supports heterogeneous code generation if you choose to use different mechanisms for different files.

Writing Handcrafted Code

Handcrafted code is code you still write line by line in the editor. Handcrafting code isn't a new step in your development; it just places what you're currently doing in a smaller and more focused context. You isolate the code you handcraft from the code built on templates so that regeneration doesn't smash over your important handcrafted code. One of the key characteristics of code generation is respecting handcrafted code, and to do this you need to isolate and protect it.

7. Here I'm talking about very mechanical localized patterns. If you want to explore incorporating formally described patterns, I've included a couple of references in the "Additional Reading" section. It's not essential to describe patterns in this formal sense to do application code generation because you're often working with a pattern such as "that specific five lines of code repeated for every column," which doesn't really seem like working with formalized patterns. Code generation and patterns are closely linked.

There are several types of handcrafted code:

- Startup
- Exception reporting
- Components
- Utility methods
- Base class functionality
- Debug support
- Code with insufficient, inconsistent, or interspersed patterns
- Class-specific code

In addition to the first six categories that are *singleton* in nature (meaning you only need one of them for your application), several characteristics in your target file might lead you to handcraft files that could also be candidates for generation. If the patterns are inconsistent and there are a number of complex variations, the logic of the template will reflect the complexity. If the pattern-based code is tightly interspersed with handcrafted code, it can be difficult to separate the handcrafted code. Finally, if there isn't much of a repeated pattern, then it may be easier to handcraft.

Handcrafted code has two key roles in your application—to provide a framework and to customize your application. The combination of metadata and handcrafted code give your application its unique personality, and the framework and templates provide its structure. Much of this customization resides in the *class-specific code*. Class-specific code includes all the ways one of your pattern-based classes differs from the others. For example, your `Customers` class may have a `FullAlphaName` property that isn't present in any other class. Class-specific code accommodates both predictable variations such as validation rules and unpredictable changes such as additional properties. You'll generally isolate class-specific code by placing it in derived classes.

Application frameworks are also made of handcrafted code. Frameworks—the infrastructure of your application that you only need to write once—are interdependent with generated code. Generated code doesn't implement the architecture you use for your current application by itself but needs handcrafted frameworks. Because they work together, you can move your frameworks and templates into your next application as a unit. Generated code needs base classes, a set of components, utility methods, and so on to complete the job of running your application.

The goal of code generation is to build applications significantly faster and cheaper, so you want to introduce code generation techniques where they'll have the highest payback first—generally middle-tier data containers and data access layers. As your skills and understanding of code generation increase, you'll contemplate what other sections of your application might be good candidates to move from handcrafted code to autogenerated code. The amount of code that has insufficient or interspersed patterns decreases as your understanding increases. Initially your entire UI layer may fit into this category and remain handcrafted, but Chapters 9 and 10 describe strategies to maximize code generation in your UI layer.

TIP When the line between the code you should autogenerate and code you should handcraft seems blurry, lean to the conservative side and handcraft more code. Where templates are helpful, it'll become obvious because you'll notice you're re-creating the same code, and you can add these additional templates later.

Isolating your handcrafted code allows you to protect it across regeneration. Other than class-specific code, handcrafted code is naturally isolated from autogenerated code because it's in different files that aren't generated. It takes more care to isolate and protect class-specific code, as discussed in Chapter 4.

NOTE There are several ways to isolate class-specific handcrafted code. The easiest solutions in .NET rely on inheritance. You derive a class containing your class-specific code from an autogenerated base class. The most important single thing you'll learn from this book may be to become comfortable with design decisions that allow you to combine code generation and class-specific code to provide each class full functionality and personality while retaining your ability to regenerate at will.

Tying It Together: Implementation and Testing

The code you produce is totally "normal" .NET code. Because it's code, you can do all the things you'd do to any other code, including source control. The code is "normal," is compiled to MSIL, and runs with the full speed of .NET at runtime.

Your users pay absolutely no performance penalty because of your use of code generation, and in some cases you can even improve performance.

Everything from here on in your development will follow your current practices, except possibly testing. Development tends to be a zero-sum game, meaning that extra resources required on one task are taken from another. Too often, resources used by development problems or new features are taken from testing. I hope that as code generation speeds your development, you'll have more time for testing. But I'm not optimistic about it.

You can manipulate generated code in familiar ways. It appears in the Class View just like any other code. You'll put generated code under source control, and it can interact with other components of your application. Placing hundreds of similar classes derived from the same template under source control may appear to be a waste of resources. However, this allows you to back up and distribute generated code to your developers in the same manner as any other code. It also lets you to know exactly what edits were made if someone inadvertently edits generated code.

TIP Hold generated code to the same high-quality standards you use for the rest of your code, and treat it with the same care—including source control and testing.

Understanding the Five Principles for Code Generation

To obtain the benefits of code generation, the five steps are closely linked to five principles. These principles are my non-negotiables for code generation:

- **Principle #1**: You have control of the templates that generate your code and can change or replace them as required.

- **Principle #2**: You collect metadata as a separate, distinct step with usable output that can independently evolve.

- **Principle #3**: You, or someone unfamiliar with project, can regenerate your code precisely as a one-click process—now or at any point in the future.

- **Principle #4**: You embrace handcrafted code by isolating and protecting it. Code generation is a supporting player to human programming and doesn't overwrite files unless they were generated and haven't been edited.
- **Principle #5**: The code-generated application is a high-quality application. It allows more effective testing, has equal or better performance, and is more easily maintained than a similar fully handcrafted application.

These may seem like challenging goals, especially because many previous attempts at code generation fell short. But with the tools .NET provides and the current level of understanding of architecture and development processes, you can fulfill all these principles and allow code generation to revolutionize your application development. Throughout the rest of this book, you'll learn how to put these principles into practice.

You Have Control Over the Template

You're in control of what you output. You or your organization remains the boss. Your decision to use my templates, your own, or someone else's is independent of the decision to use code generation. So many nuances determine the "best" way to architect a particular project in .NET, and they change over time, that no one can tell you what architecture is best for your application without being familiar with your unique requirements.

NOTE I hope that we move to a world where there are a number of interchangeable templates available to you, and I talk about that in Chapter 11. But it's important that (even if you're using templates created by someone else) you can easily modify them or replace them if someone builds a better mousetrap. Interchanging templates from different vendors will require a level of standardization that isn't yet available.

Control also means you're in charge of quality and performance. The importance of performance differs phenomenally in different types of applications and even at different points in the deployment life cycle. You can focus your decisions on what's best for the application, including quality and runtime performance where appropriate.

Metadata Is a Distinct Step

The metadata that defines your application details and the templates that define how your application runs are two distinct pieces. There are many advantages to separating them: Different people can work on each. You can separately debug them. You don't have to re-create metadata (generally a slower process) when you're testing templates. You can use dummy metadata or dummy templates to avoid delays early in your development cycle. But most important, you can move the templates (and your template-generating mechanisms) forward as a way to kick-start your next application. A portfolio of templates can evolve within your organization to reflect your current best practices.

Although metadata and templates are highly symbiotic, they evolve in response to separate pressures. The initial source of metadata is a project's requirements, and the initial source of templates is the application's architecture. If you work in a formalized development environment, different people are likely to develop these pieces—experts in understanding domains vs. the underlying technologies such as .NET. Because different pressures lead to changes, metadata and templates evolve on separate timelines. Metadata generally expands in response to new feature requests regarding business logic. Templates evolve when there's an architectural change, generally because of changes in the underlying platform, language, security issues, best practices, or other technology issues. The metadata is unlikely to change just because you're ready to write a new version of your Windows application for the Web, but the templates will certainly change. Conversely, templates won't change when you add a new field, but metadata certainly will.

Implement One-Click Regeneration

Code generation isn't a one-time thing. Most of your generated code will need to reflect evolving metadata and evolving templates. It can only reflect changes in these underlying sources if you can regenerate your code at any time.

Although one-click generation is convenient, the reason for committing to one-click generation isn't convenience. When you use code generation, you should commit to it for the life of your project. Someone will need to regenerate your application to add a new field or some other feature. That may happen two or three years down the road, and you may have moved on to other projects. Even if you're still around, will you remember exactly what you did during code generation in three years? The only way to ensure that anyone can exactly reproduce your code generation is to run it as a script initiated by a generation harness.

That's one-click regeneration. And the benefits are a significant decrease in effort required for simple changes and an increase in application stability throughout your project lifetime.

Embrace Handcrafted Code

Handcrafted code is the code you still write line by line in an editor. This is the important stuff representing the unique aspects of your application. It cradles little aspects such as validation and the core algorithms that make up your application. To protect your handcrafted code during regeneration, you need to isolate it from autogenerated code. This isolation will generally occur by placing handcrafted code and autogenerated code in different files and incorporating the handcrafted logic through specific techniques such as inheritance.

Visual Studio doesn't understand that the next regeneration will destroy any edits made in autogenerated files. Those edits are immediately doomed. Whether it's three minutes or three years from now, unless you take special steps the next code generation will overwrite these edits. Don't panic or run to the nearest bar because of images of some junior programmer editing a file three weeks into maintenance and not finding the problem until you regenerate six months later. You could make rules such as, "Thou shall not edit any file marked for generation and teach this to all thy children." I don't know whether that's good enough to keep you sane, but it isn't enough to let me sleep at night. I don't want the long-term viability of projects dependent on whether everyone follows a set of rules. Chapter 3 shows how to insert a hash code into the file's header so you can later check whether the file has been edited. If the file has changed, you can force the programmer to solve the problem before trashing the edits. You can even run this check across your generated code base periodically if you're having nightmares about some fool editing your autogenerated code.

You have control over the implementation of this principle. Due to the intricacies of code generation, different rules are appropriate for different templates, and you'll be able to tune overwriting behavior as part of the one-click script presented in Chapter 3.

Generated Code Has Great Quality, Performance, and Maintainability

I want to have my cake and eat it, too! I want all the benefits of code generation, and I don't want it to affect my users. I want them to have an application with the fastest reasonable performance, highest quality, and long-term maintainability.

Code generation is a design-time feature and will have *no* negative impact on runtime performance. In certain cases, you might do something a little better with a code-generated application. For example, you can easily generate enums for your database column positions or base validation on database information such as providing a maximum length or database constraints. Although there will be variations because of different design decisions, the performance is generally identical between code-generated and handcrafted applications because the runtime code is nearly identical.

Software quality is often an elusive goal. You know that it means your software will do what is intended well without unexpected quirks (bugs). But getting to that goal sometimes seems to require effort that's beyond the available resources. Code generation contributes to software quality, but only by amplifying other good practices. You can break down the process of achieving software quality into four basic areas: planning, implementing, testing/deploying, and maintaining. Code generation aids in the planning stage because it encourages iterative prototype-based requirements gathering. The code generation *prototype* is special because it's capable of smoothly evolving into a robust permanent project element (unless it brings a design problem to light). During implementation, code generation significantly improves consistency of the code.

Code generation affects testing in two ways. First, because code-generated parts of your application are repetitively created, if you're in triage mode, you can limit testing to a few examples of each pattern, rather than testing every class. I'm not suggesting this is a good idea, but triage is always a matter of picking the best of bad choices. The other way code generation affects testing is to automate unit and regression testing. You'll learn in this book how to generate code and run complex scripts. In addition to combining these features for creating application code, you can combine them for testing. You won't be able to do all of your testing, but you can cover the repetitive tests that are the hardest do manually. The key to this is building a specialized set of metadata geared to testing.

Code generation offers automatic benefits in maintainability. Many things that contribute to agility also contribute to maintainability. You can make common types of changes easily and propagate them throughout the application. But maintainability goes beyond this. Maintainable code is well organized and easily read by a human being. It has appropriate scope to help later programmers understand how things should be used. The application code is tracked in source control. Classes have meaningful extensible interfaces. Code generation doesn't get you off the hook on any of these aspects of creating a quality application. It allows you to do it more easily because you can propagate quality, but careless use of code generation propagates sloppy code.

The Strongly Typed DataSet

The ADO.NET strongly typed DataSet is a piece of application-level code generation within the .NET Framework. It's hard know what to say about a construct that moved us so far forward in how we think about data containers and yet presents such a flawed implementation. Visual Studio generates strongly typed DataSets through an external tool named XSD.EXE. Code in this tool accesses your XSD for metadata and uses the CodeDOM to explicitly produce output. There's no available template, and you can't alter the output.

Strongly typed DataSets are a hybrid between the significant power of DataSets and the strong typing benefits generally associated with business objects. Solutions built with the strongly typed DataSet are easily bound[8] and support cascading inserts of primary keys by default. It's "normal" code that you can learn from, rather than some hidden black box. Generation based on the XSD standard gives the strongly typed DataSet broad applicability. That's the good news.

Unfortunately, most projects will run into problems because of one or more of the following problems with the implementation in Visual Studio 2002 and 2003:

- DataTables are inextricably linked to specific DataSets, massively reducing the possibility for reuse.
- Although not sealed, you can't effectively derive from the strongly typed DataTable or strongly typed DataRow because you can't override their instantiation within the DataSet.
- You can't specify the base class.
- Column metadata is `Friend` (`assembly`), so code in a different project can't access it via strong typing.
- Strongly typed DataSets don't incorporate all available metadata from the database or XSD. For example, they don't provide privileges, constraints, or extended properties.
- It's awkward to provide variations in null handling.
- Mapping of columns is difficult and easily lost on regeneration.
- Validation can't be included in the strongly typed DataRow or DataTable.
- Users of your class can circumvent using the strongly typed DataSet, even avoiding strong typing.

If you could modify the way a strongly typed DataSet is generated when you ran into one of these problems, it might be appropriate for your application development. If you want to use the strongly typed DataSet, I include an XSLT template for code generation in both VB .NET and C# in the Downloads section of the Apress Web site (`http://www.apress.com`). It's not complete and doesn't

8. You can bind strongly typed properties at design time. You can't bind an untyped object such as a standard ADO.NET DataSet or DataTable through property dialog boxes at design time.

cover as many details (such as annotations) as the Microsoft-supplied strongly typed DataSet. But for mainstream use, it gives you a fallback position if you run into a problem with the strongly typed DataSet because you can make changes to it. Without a backup plan, I think the strongly typed DataSet is a time bomb in most environments because when you run into a brick wall, it's a very solid brick wall that can derail your development.

You can't solve the problem of programmers being able to circumvent the strong typing in any design that derives directly or indirectly from the DataTable class. That arises because you can't fully hide access to base class members. In your own designs, you can circumvent this issue by wrapping the DataTable, rather than inheriting from it. In designs that do include DataTable derivations, use events such as the `ColumnChanging` event to respond to key actions such as changing the value, rather than properties the user could circumvent.

The strongly typed DataSet *is* useful from a pedagogic, or learning, perspective. The basic approach of code generation is to apply a structure or pattern to the details of your application. One way to get an idea of what structure and patterns mean in your application is to look inside a strongly typed DataSet. To do this, create a strongly typed DataSet and locate its source code:

1. Add a new DataSet to your project through the Project menu of Visual Studio.
2. Drag a couple of tables onto the design surface and set relations between them.
3. Select Generate Dataset from the right-click menu.
4. Click the Show All Files button at the top of Solution Explorer.
5. Expand the XSD entry in your project, and you should find the source code as a .vb or .cs file.

Open the strongly typed DataSet source code, and scroll down to one of the *`tablename`*`DataRow` classes (a few hundred lines). You'll see a typed property for each column in your DataTable. The code varies depending on whether the column is nullable or not nullable. The code generator has applied a pattern to the metadata retrieved about each column in the DataTable. The pattern varies based on the specific input, such as nullability. If you scroll around this file, you'll see many other examples of patterns as well as some pretty big blocks of code that are more or less the same for all instances of the strongly typed DataSet. Your own template patterns will also contain chunks of code that are relatively static and chunks that change for each table, column, or other entity.

The strongly typed DataSet inspired parts of the code generation journey I've been on. It convinced me that .NET offered features that let code generation finally work for my applications. Bashing my head bloody on the strongly typed DataSet left me no doubt that you and I could do a better job and reinforced that we required control over critical pieces of my application, and things just grew from there.

Performing Real-World Code Generation

By now, you know what basic code generation looks like and how three different mechanisms that accomplish code generation work. You may already be leaning toward one of these approaches. You've also seen the five steps making code generation a conscious process and the five non-negotiable principles that back up these steps. This combination builds the environment where you can commit to code generation for your application development, remembering that code-generated applications also have significant amounts of handcrafted code.

To get a deeper understanding of how code generation works and how to implement each of the three mechanisms, you need to see them function in the context of a real-world class. Maybe "real-world" is a stretch for this chapter. This is a simple class, but it demonstrates how you'll deal with data containers with properties and private fields for each column in the table. You'll see this type of class enhanced a bit in Chapter 3 and fully developed in Chapter 8.

Creating a Simple Class

The simplest real-world example is a class with private fields exposed by public properties, as shown in Listing 1-2.

Listing 1-2. A Target Program for Creating a Simple Class

```
Option Strict On
Option Explicit On
Imports System

Public Class Customers

#Region "Class level declarations"
   Private m_CustomerID As System.String
   Private m_CompanyName As System.String
   Private m_ContactName As System.String
   Private m_ContactTitle As System.String
   Private m_Address As System.String
   Private m_City As System.String
   Private m_Region As System.String
   Private m_PostalCode As System.String
   Private m_Country As System.String
   Private m_Phone As System.String
   Private m_Fax As System.String
#End Region
```

```
#Region "Public Methods and Properties"
   Public Property CustomerID() As System.String
      Get
         Return m_CustomerID
      End Get
      Set(ByVal Value As System.String)
         m_CustomerID = Value
      End Set
   End Property

   Public Property CompanyName() As System.String
      Get
         Return m_CompanyName
      End Get
      Set(ByVal Value As System.String)
         m_CompanyName = Value
      End Set
   End Property
```

I'll skip the rest of the properties because they follow a similar pattern.

To determine the metadata needed to create this target output, you want to identify all the variable information. The variable information in Listing 1-2 is marked in bold. That's the class name and the name and type of each field and property. You also want to identify repeating patterns. There are just two repeating patterns in this code: one for the private fields and one for the properties.

TIP Print your sample class, and mark the changeable information with a highlighter. Then use your highlighter to bracket repeating patterns within the class. This will serve as a reference for creating your templates.

Collecting the XML Metadata

Before looking at the template, it's useful to know what the metadata looks like.[9] For the example in Listing 1-3, I just typed the metadata into the .NET XML editor. In addition to the metadata about the Customers table, it also includes metadata about the Orders table. This will let you see how a metadata file containing a full set of data can create a single class file using a subset of the metadata. You can generate similar classes for anything else for which you supply metadata. Listing 1-3 shows the metadata for the Customers and Orders tables.

9. You'll discover in Chapter 2 that this is quite an understatement.

TIP The second time you create code from a template (using a second set of metadata) often exposes changeable information you overlooked.

Listing 1-3. XML Metadata Input for Generating the Simple Class

```
<?xml version="1.0" encoding="utf-8" ?>
<DataSet Name="Northwind">
   <Table Name="Customers">
      <Column Name="CustomerID" Type="String" />
      <Column Name="CompanyName" Type="String" />
      <Column Name="ContactName" Type="String" />
      <Column Name="ContactTitle" Type="String" />
      <Column Name="Address" Type="String" />
      <Column Name="City" Type="String" />
      <Column Name="Region" Type="String" />
      <Column Name="PostalCode" Type="String" />
      <Column Name="Country" Type="String" />
      <Column Name="Phone" Type="String" />
      <Column Name="Fax" Type="String" />
   </Table>
   <Table Name="Orders">
      <Column Name="OrderID" Type="Int32"   />
      <Column Name="CustomerID" Type="String" />
      <Column Name="EmployeeID" Type="Int32" />
      <Column Name="OrderDate" Type="DateTime" />
      <Column Name="RequiredDate" Type="DateTime" />
      <Column Name="ShippedDate" Type="DateTime" />
      <Column Name="ShipVia" Type="Int32" />
      <Column Name="Freight" Type="Decimal" />
      <Column Name="ShipName" Type="String" />
      <Column Name="ShipAddress" Type="String" />
      <Column Name="ShipCity" Type="String" />
      <Column Name="ShipRegion" Type="String" />
      <Column Name="ShipPostalCode" Type="String" />
      <Column Name="ShipCountry" Type="String" />
   </Table>
</DataSet>
```

TIP Using the `System` version of each data type allows generation of both C# and VB .NET code from the same XML metadata file.

All XML must contain a single root element. The name of the root element in this XML document is *DataSet*. The root element contains two `Table` elements, and each `Table` element contains a series of `Column` elements that contain the name and type of each column.

Generating a Simple Class via Brute Force

Initiating code generation via brute force is easy. You open the XML metadata file and pass this to the `GenerateOutput` method of the `ClassViaBruteForce` class. To make it more interesting, the example creates both the `Customers` and `Orders` class files from the same `GenerateOutput` method:

```
Private Shared Sub GenerateClassViaBruteForce(ByVal outputDir As String)
    ' Open Metadata file
    Dim xmlMetaData As New Xml.XmlDocument
    Dim tOrdersableName As String = "Customers"

    ClassViaBruteForce.GenerateOutput( _
             IO.Path.Combine(outputDir, "ClassCustomersViaBruteForce.vb"), _
             xmlMetaData, "Customers")
    ClassViaBruteForce.GenerateOutput( _
             IO.Path.Combine(outputDir, "ClassOrdersViaBruteForce.vb"), _
             xmlMetaData, "Orders")

End Sub
```

The `ClassViaBruteForce.GenerateOutput` method is similar to the earlier "Hello World" brute-force sample. The differences are that it uses `IndentedTextWriter` and that it uses a nonempty XML document for input. The `IndentedTextWriter` makes for a few more lines of code but is a less of a pain to keep the code lined up.

NOTE The CodeDOM namespace contains the `IndentedTextWriter` that simplifies whitespace management. I'm not actually doing CodeDOM generation in this sample, just borrowing a class from its namespace.

This brute-force template generates classes like the one shown in Listing 1-2:

```
' Class Summary: Generates a simple class based on XML metadata
Public Class ClassViaBruteForce
```

```
#Region "Public Methods and Properties"
   Public Shared Sub GenerateOutput( _
               ByVal outputFile As String, _
               ByVal xmlMetaData As Xml.XmlDocument, _
               ByVal tableName As String)
      Dim writer As New CodeDom.Compiler.IndentedTextWriter( _
               New IO.StreamWriter(outputFile))
      Dim node As Xml.XmlNode
      Dim nodeList As Xml.XmlNodeList
```

An `XMLNodeList` is a collection of `XMLNodes`. The `SelectNodes` method of the `XMLDocument` creates the `nodeList`, which contains the `Column` elements. `SelectNodes` takes an XPath expression specifying the `tableName` (Appendix A has more about XPath):

```
nodeList = xmlMetaData.SelectNodes( _
         "/DataSet/Table[@Name='" & tableName & "']/Column")
```

The XPath expression in this line says, "Get a collection of nodes that are `Column` elements under a parent `Table` element having a `Name` attribute that matches `tableName`, with the `Table` element also a child of the root `DataSet` element." The `nodeList` is used later in the method.

`WriteLine` statements output code to the `IndentedTextWriter` in the same way they output to other streams. The first output is nearly static, with only the `tablename` changing:

```
writer.WriteLine("Option Strict On")
writer.WriteLine("Option Explicit On")
writer.WriteLine("")
writer.WriteLine("Imports System")
writer.WriteLine("")
writer.WriteLine("' Class Summary: Simple output class")
writer.WriteLine("")
writer.WriteLine("Public Class " & tableName)
writer.WriteLine("")
writer.WriteLine("#Region " & Chr(34) & "Class level declarations" & Chr(34))
```

Incrementing the indent level by one increases the indent for further output by four spaces (the default indent string is four spaces). The first loop outputs the private fields corresponding to each column in the node list:

```
writer.Indent += 1
For Each node In nodeLIst
   writer.WriteLine("Private m_" & node.Attributes("Name").Value & _
         " As " & node.Attributes("Type").Value)
```

```
Next
writer.Indent -= 1
writer.WriteLine("#End Region")
writer.WriteLine("")
```

Decrementing the `IndentedTextWriter` removes one level of indent from further output. The remainder of the template outputs the property procedure in another `xsl:for-each` loop:

```
      writer.WriteLine("#Region " & Chr(34) & "Public Methods and Properties" & _
               Chr(34))
      writer.Indent += 1
      For Each node In nodeLIst
         writer.WriteLine("Public Property " & node.Attributes("Name").Value & _
               "() As " & node.Attributes("Type").Value)
         writer.Indent += 1
         writer.WriteLine("Get")
         writer.Indent += 1
         writer.WriteLine("Return m_" & node.Attributes("Name").Value)
         writer.Indent -= 1
         writer.WriteLine("End Get")
         writer.WriteLine("Set(ByVal Value As " & node.Attributes("Type").Value & _
               ")")
         writer.Indent += 1
         writer.WriteLine("m_" & node.Attributes("Name").Value & " = Value")
         writer.Indent -= 1
         writer.WriteLine("End Set")
         writer.Indent -= 1
         writer.WriteLine("End Property")
         writer.WriteLine("")
      Next
      writer.Indent -= 1
      writer.WriteLine("#End Region")
      writer.WriteLine("")
      writer.WriteLine("End Class")

      writer.Flush()
      writer.Close()

   End Sub
#End Region

End Class
```

That's it. Maybe it's a tad ugly, but it's marvelously effective in generating code output. It outputs the class shown in Listing 1-2. You could create a similar class for any table in any database or any business object, if you supply the metadata. The next chapter shows how to automate the metadata creation step.

Generating a Simple Class via CodeDOM

Initiating code generation via the CodeDOM requires that you open the XML metadata document for input and call the `ClassViaCodeDOM` to create the `CodeCompileUnit` that contains the CodeDOM graph for the target code. The `GenerateClassViaCodeDOM` method outputs code for the classes. This example only creates VB .NET code, but you could easily change it to output C# or even J# code:[10]

NOTE Don't be concerned if you feel lost or just don't understand the CodeDOM at this point. You might never use it in code generation if one of the other approaches better suits your needs. If you're considering using the CodeDOM, there's an in-depth section about the CodeDOM in Chapter 3 and Appendix D.

```
Private Shared Sub GenerateClassViaCodeDOM(ByVal outputDir As String)
    Dim compileUnit As CodeDom.CodeCompileUnit
    Dim provider As CodeDom.Compiler.CodeDomProvider

    ' Open Metadata file
    Dim xmlMetaData As New Xml.XmlDocument
    xmlMetaData.Load(IO.Path.Combine(xmlDir, "Metadata.xml"))

    ' Generate simple class for Cusotmers in Visual Basic using the CodeDOM
    compileUnit = ClassViaCodeDOM.BuildGraph(xmlMetaData, "Customers")
    provider = New Microsoft.VisualBasic.VBCodeProvider
    GenerateViaCodeDOM(IO.Path.Combine(outputDir, "ClassCustomersViaCodeDOM.vb"), _
             provider, compileUnit)

    ' Generate simple class for Orders in Visual Basic using the CodeDOM
    compileUnit = ClassViaCodeDOM.BuildGraph(xmlMetaData, "Orders")
    provider = New Microsoft.VisualBasic.VBCodeProvider
    GenerateViaCodeDOM(IO.Path.Combine(outputDir, "ClassOrdersViaCodeDOM.vb"), _
               provider, compileUnit)
 End Sub
```

10. To modify the output language, use the alternate provider as shown in the "Generating 'Hello World' via the CodeDOM" section.

A separate method, ClassViaCodeDOM.GenerateGraph, builds the CodeCompileUnit:

```
' Class Summary: Generates a simple class based on XML metadata
Public Class ClassViaCodeDOM

#Region "Public Methods and Properties"
   Public Shared Function BuildGraph( _
               ByVal xmlMetaData As Xml.XmlDocument, _
               ByVal tableName As String) _
               As CodeDom.CodeCompileUnit
```

Creating the compile unit and namespace is similar to the "Hello World" CodeDOM example. The CodeTypeDeclaration creates a class with the name of the underlying table. Creating the node and nodeList variables is similar to the brute-force code generation in the section "Generating a Simple Class via Brute Force." Filling the node list uses the same XPath expression as the brute-force generation:

```
Dim node As Xml.XmlNode
Dim nodeList As Xml.XmlNodeList
Dim compileUnit As New CodeDom.CodeCompileUnit
Dim nSpace As CodeDom.CodeNamespace
Dim clsTable As CodeDom.CodeTypeDeclaration
nodeList = xmlMetaData.SelectNodes( _
         "/DataSet/Table[@Name='" & tableName & "']/Column")
nSpace = New CodeDom.CodeNamespace("ClassViaCodeDOM")
compileUnit.Namespaces.Add(nSpace)

nSpace.Imports.Add(New CodeDom.CodeNamespaceImport("System"))

clsTable = New CodeDom.CodeTypeDeclaration(tableName)
nSpace.Types.Add(clsTable)
```

The field variable is a CodeMemberField object that represents the private field for each column. Each loop creates a new CodeMemberField object and assigns it to the field variable. The Name and Type attributes of the XML node are also assigned to properties of the field variable. The Type property of the CodeMemberField is set by creating a new CodeTypeReference object that utilizes the Type attribute of the XML. The CodeMemberField object contains an Attributes property that, among other things, holds the scope of the field:

```
Dim field As CodeDom.CodeMemberField
For Each node In nodeList
   field = New CodeDom.CodeMemberField
   field.Name = "m_" & node.Attributes("Name").Value
   field.Attributes = CodeDom.MemberAttributes.Private
   field.Type = New CodeDom.CodeTypeReference(node.Attributes("Type").Value)
   clsTable.Members.Add(field)
Next
```

You may have to read this twice because the linguistics of explaining the second loop are pretty twisted. For each column, the generating code creates a `CodeMemberProperty` object and sets several properties on the new `CodeMemberProperty` object. As in the previous example, the `CodeMemberProperty` type is set using a new `CodeTypeReference` object:

```
Dim prop As CodeDom.CodeMemberProperty
Dim statement As CodeDom.CodePropertySetValueReferenceExpression
Dim Name As String
For Each node In nodeList
   prop = New CodeDom.CodeMemberProperty
   Name = node.Attributes("Name").Value
   prop.Name = Name
   prop.Attributes = CodeDom.MemberAttributes.Public
   prop.Type = New CodeDom.CodeTypeReference(node.Attributes("Type").Value)
```

The `GetStatements` collection of the `CodeMemberProperty` contains the code statements for the `Get` method. For this property, it's just a single return statement with a reference expression for the corresponding private field of the current object:

```
prop.GetStatements.Add(New CodeDom.CodeMethodReturnStatement( _
      New CodeDom.CodeFieldReferenceExpression( _
          New CodeDom.CodeThisReferenceExpression, "m_" & Name)))
```

The single statement in the `SetStatement` collection assigns the reference expression of the corresponding private field onto the passed value, accessed by the `CodePropertySetValueReferenceExpression`. This also illustrates a rare instance where really, really long method names aid clarity:

```
         prop.SetStatements.Add(New CodeDom.CodeAssignStatement( _
               New CodeDom.CodeFieldReferenceExpression( _
                   New CodeDom.CodeThisReferenceExpression, "m_" & Name), _
               New CodeDom.CodePropertySetValueReferenceExpression))
         clsTable.Members.Add(prop)
      Next
      Return compileUnit
   End Function

#End Region

End Class
```

That's a lot of code to output a simple class. Not only is it a lot of code, but this code is a lot harder to understand than the brute-force generation of the same class. As in the "Hello World" example using the CodeDOM, this version doesn't accomplish as much as the brute-force or XSLT generation examples. It avoids the option statements, regions, and vertical whitespace. But it's effective in doing the job it sets out to do. Given the metadata, you can use this to create a similar class representing *any* table in *any* database and in *any* supported .NET language.

Generating a Simple Class via XSLT Templates

Now compare these approaches with code generation via XSLT. Similar to the brute-force approach, the XSLT approach opens the XML metadata file. Instead of passing the XML metadata file to a specific class method to generate the code, the XML metadata directly drives code generation through an XSLT template:

```
Private Shared Sub GenerateClassViaXSLT(ByVal outputDir As String)
   ' Open Metadata file
   Dim xmlMetaData As New Xml.XmlDocument
   xmlMetaData.Load(IO.Path.Combine(xmlDir, "Metadata.xml"))

   ' Generate Hello World via XSLT Template
   GenerateViaXSLT(IO.Path.Combine(xsltDir, "Class.xslt"), xmlMetaData, _
                IO.Path.Combine(outputDir, "ClassCustomersViaXSLT.vb"), _
            New XSLTParam("TableName", "Customers"))
   GenerateViaXSLT(IO.Path.Combine(xsltDir, "Class.xslt"), xmlMetaData, _
                IO.Path.Combine(outputDir, "ClassOrdersViaXSLT.vb"), _
            New XSLTParam("TableName", "Orders"))
End Sub
```

The XSLT template named `Class.xslt` holds the pattern definition, and its name is passed as the first argument to the `GenerateViaXSLT` method. The `XMLDocument` containing the metadata is the second argument to the generic transformation routine. The third argument is the output file path. The final argument is an XSLT parameter containing the name of the table for the created class.

Like all XSLT stylesheets, the stylesheet starts with some standard XSLT goo:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<xsl:output method="text" encoding="UTF-8" indent="yes"/>
<xsl:preserve-space elements="*" />
```

The `TableName` parameter allows the creation of the target class for a subset of the XML metadata:

```
<xsl:param name="TableName"/>
```

The XSLT instructions that create output are contained in two XSLT templates. The first template matches the root element. Within this template, a second matching template is processed only for tables whose name matches the `TableName` parameter (there's only one). Don't worry if this syntax seems strange because Chapter 3 has XSLT examples, and you can read Appendix A when you want to learn more about XSLT details. At this point, it's only important that you have a general idea that this template calls the following template for only the table with which you're working:

```
<xsl:template match="/DataSet">
   <xsl:apply-templates select="Table[@Name=$TableName]"/>
</xsl:template>
```

This second template contains the code to output, including the normal option and imports statements. The class declaration uses the first of several XSLT constructs that are important to code generation. The `xsl:value-of` construct provides a mechanism for token replacement. It outputs data from the XML input, XSLT variables, template parameters, and XPath functions. In this case, the inserted value is the contents of the `TableName` parameter. In XSLT, you prefix variables with the dollar sign ($):

```
<xsl:template match="Table">
Option Strict On
Option Explicit On

Imports System

'! Class Summary: Simple class example

Public Class <xsl:value-of select="$TableName"/>
```

Creating the field variables uses another important XSLT construct. The `xsl:for-each` is one of two mechanisms for looping through code. The other is the `xsl:apply-templates` directive. Each of these directives selects a set of nodes and sets the context to each active node while looping through the node-set. (Appendix A clarifies when to use each construct.) XSLT is highly reliant on the concept of context. The simplest way to think of context is to imagine you have a printed copy of your XML input document and point your finger at a particular element. Each time the `xsl:for-each` loop is processed, the context moves to the next node in the selected list of nodes—in this case, child elements named `Column`:

```
#Region "Class level declarations"
<xsl:for-each select="Column">
   Private m_<xsl:value-of select="@Name"/> As <xsl:value-of select="@Type"/>
</xsl:for-each>
#End Region
```

Code in the `xsl:for-each` loop outputs the field variable for each column. The `xsl:value-of` directive outputs the `Name` and `Type` attributes from the current node in the XML input document at the appropriate locations. You prefix attributes with the at (@) sign. In the context of the `xsl:for-each` loop, the `Name` attribute is the current column's name, and the `Type` attribute is the column's type.

The second `xsl:for-each` loop creates the output code for the property. Here the `xsl:value-of` directive again accesses the attributes of the column. The XSLT logic is similar to the previous loop, but the extra text makes the output look quite different:

```
#Region "Public Methods and Properties"
<xsl:for-each select="Column">
 Public Property <xsl:value-of select="@Name"/>() As <xsl:value-of select="@Type"/>
    Get
       Return m_<xsl:value-of select="@Name"/>
    End Get
    Set(ByVal Value As <xsl:value-of select="@Type"/>)
       m_<xsl:value-of select="@Name"/> = Value
    End Set
 End Property
</xsl:for-each>

#End Region

End Class
</xsl:template>

</xsl:stylesheet>
```

If you compare this with the earlier XSLT template, you'll see that the template for a much longer and more complex target file is only a few lines longer than the "Hello World" example. Ignoring the standard XSLT goo copied from the template, the CodeDOM and brute-force examples that create the same file are significantly longer. Each line is also significantly simpler, and the distinct languages (XSLT and VB .NET) differentiate the varying elements and template logic from target output code. The XSLT directives are also visually distinct from the output code.

If you're new to XSLT, the template may look a little strange. It may even be scary or intimidating because XSLT encourages you to think in a different way. But, by the time you're comfortable with XSLT, you'll be very comfortable reading the code in these templates, and you'll see that this different way of thinking can help you filter the target output from the template structure, each of which will be important to you at different times.

NOTE You can find all the code in this book in the Downloads section of the Apress Web site (`http://www.apress.com`).

Summary

.NET gives you three viable options for application code generation depending on what you're trying to accomplish:

- Brute-force code generation (spitting out code to file streams)
- CodeDOM generation
- XSLT code generation

For all three mechanisms, the power of code generation is that you can apply the pattern to any XML metadata that matches the anticipated format. Once created, you can use any of these patterns repeatedly to create the target code for *any* metadata in the expected format.

Just spitting out code might help your project, but it won't give you the full benefit of code generation. For that, you need to make a conscious process. The five distinct steps of application code generation are as follows:

- Architecting and designing (figuring out what to build)
- Extracting metadata (organizing the application-specific input)

- Creating and running code generation (creating reusable templates and generating code)
- Handcrafting code (customizing your application and building components)
- Implementing and testing (delivering your application sooner)

The five principles of code generation parallel these five steps. These are the non-negotiables:

- You're in control.
- Metadata extraction is a separate step.
- Regeneration is a precise repeatable one-click process.
- Code generation embraces handcrafted code and protects it across regeneration cycles.
- Code-generated applications meet or exceed traditional applications in quality, performance, and maintainability.

Additional Reading

To find more information about the topics covered, try these resources:

- For more information about the CodeDOM, go to `www.msdn.microsoft.com` and search for *.NET CodeDOM*.
- A classic book on formally described design patterns is *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley, 1995).
- You can also refer to *Professional Design Patterns in VB .NET: Building Adaptable Applications* by Tom Fischer, Chaur Wu, and Pete Stromquist (Apress, 2003).
- Finally, a helpful book is *.NET Patterns: Architecture, Design, and Process* by Christian Thilmany (Addison-Wesley, 2003).