Code Generation in Microsoft .NET

KATHLEEN DOLLARD

EXHEART

Apress™

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

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

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

Printed and bound in the United States of America 12345678910

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

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 information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

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

For Phyllis, who shows us that whether it's in a wheelchair, with a walker, or on our own two feet, we take life one step at a time.

EXHEART

EXHEART

Contents at a Glance

Foreword by Rockford Lhotka vii
Foreword by Don Kiely ix
About the Author xi
About the Technical Reviewer xii
Acknowledgments xiii
Introduction xvi

Part One The Process 1
Chapter 1 Introducing Code Generation 3
Chapter 2 Extracting Metadata 53
Chapter 3 Outputting Code 113
Chapter 4 Writing Handcrafted Code 179
Chapter 5 Tying It All Together 215

Part Two Code Generation in Action 291
Chapter 6 Mapping Database Structure into Business Objects 293
Chapter 7 Generating Stored Procedures 345
Chapter 8 Generating Middle Tier Templates 377
Chapter 9 Generating a WinForms Application 421
Chapter 10 Generating a Web User Interface 471
Chapter 11 Establishing Perspective 509
Appendix A X-Survival 543
Appendix B Dissecting the Metadata Extraction Tools 609
Appendix C Details of the Code Generation Harness 647
Appendix D CodeDOM Issues, Tips, and Tricks 691

Index 707

EXHEART

Contents

Foreword by Rockford Lhotka vii
Foreword by Don Kiely ix
About the Author xi
About the Technical Reviewer xii
Acknowledgments xiii
Introduction xvi

Part One The Process 1

Chapter 1 Introducing Code Generation 3

Exploring Mechanisms for Code Generation 5
Generating a Simple Program 7
Picking the Right Mechanism 20
Breaking Down the Code Generation Process 24
Performing Real-World Code Generation 38
Summary 50
Additional Reading 51

Chapter 2 Extracting Metadata 53

Introducing the Process 56
Understanding XSD's Role in Code Generation 74
Working with SQL-92 Databases (SQL Server) 79
Creating Freeform Metadata 103
Merging Metadata 109
Summary 111
Additional Reading 111

Chapter 3 Outputting Code 113

Understanding the Elements of the Code Generation Harness 114
Examining Code Generation Mechanics 125
Summary 177
Additional Reading 177

Chapter 4 Writing Handcrafted Code 179

Effectively Using Handcrafted Code 180
Exploring Mechanisms for Isolating Handcrafted Code 196
Summary 212

Chapter 5 Tying It All Together 215

Getting Into the Harness Details 215
Organizing Files via the Directory Structure 236
Building Project Files 240
Incorporating Source Control 254
Debugging 265
Managing the Project over the Long Haul 280
Summary 287
Additional Reading 289

Part Two Code Generation in Action 291

Chapter 6 Mapping Database Structure into Business Objects 293

Creating the Object Relation Mapping (ORM) Solution 295
Morphing ORM into Generation-Friendly Metadata 328
Summary 343

Chapter 7 Generating Stored Procedures 345

Exploring Stored Procedure Generation 345
Generating Retrieve Stored Procedures 350
Generating Create Stored Procedures 364
Generating Delete Stored Procedures 368
Generating Update Stored Procedures 370
Generating SetSelect Stored Procedures 372
Summary 374
Additional Reading 375

Chapter 8 Generating Middle Tier Templates 377

Introducing Component-based, Scalable, Logical Architecture (CSLA) 380
Changing the Base Architecture 387

Generating the Middle Tier 399
Summary 418
Additional Reading 419

Chapter 9 Generating a WinForms Application.....421

Reviewing WinForms Architecture 422
Developing Code Generation Strategies for WinForms.....429
Exploring User Interface Templates 453
Summary 470

Chapter 10 Generating a Web User Interface 471

Reviewing ASP.NET 472
Exploring Code Generation Strategies for ASP.NET 482
Walking Through ASP.NET Templates 493
Summary 506
Additional Reading 507

Chapter 11 Establishing Perspective.....509

Selling Code Generation to Management 510
Investigating Upcoming Features 520
Envisioning the Future.....527
Building New Paradigms.....537
Going Forward 541
Summary 541

Appendix A X-Survival.....543

Starting the XML Basic Training 543
Dealing with the XSD Drill Sergeant 559
The Commando's Guide to XSLT.....574
XPath Marks the Spot 593
XSLT and XPath Reference 602
Additional Reading 608

Appendix B Dissecting the Metadata Extraction Tools.....609

Converting XSD into Friendly XML 609
Extracting Metadata from Databases 618

Appendix C Details of the Code Generation Harness **647**

Building a Dynamic Data-Driven Application 649
Executing the Script 676
Looking Back at the Generation Harness 690

Appendix D CodeDOM Issues, Tips, and Tricks **691**

Extending the CodeDOM 691
Looking at Broad Issues with the CodeDOM 693
Exploring Language-Specific Details 700
Testing in All Target Languages 706

Index **707**

FLYHEAD

Foreword by Rockford Lhotka

I WAS IMMENSELY FLATTERED when Kathleen approached me with the idea of using my framework as an example in her book on code generation.

I am a strong proponent of object-oriented design and programming. I'm also very pragmatic. Like you, I make my living programming computers, and all the cool theories in the world don't meet the deadlines we face every day.

If you've tried object-oriented programming, you've undoubtedly discovered its primary weakness—you have to write a lot of fairly repetitive code. Certainly you write more code than if you took the road more traveled and just used a DataSet. Of course, your end result is more maintainable and readable, so the use of object-oriented technologies is almost always warranted. Unfortunately, that doesn't negate the fact that you *did* write a lot of extra code.

As programmers, we're constantly required to make decisions based on trade-offs. Do I use object-oriented programming and write more code but get a more maintainable system? Or do I use a DataSet-based approach and write less code but end up with a less maintainable system? Short-term vs. long-term costs—that's what it is all about.

What if it turned out that maybe you didn't need to write all that extra code after all? In fact, maybe—just maybe—you could spend more time in the design phase, less in the development phase, and still meet that looming deadline.

This thought was the driving force behind my writing *Expert One-on-One Visual Basic .NET Business Objects* (Apress, 2003). I didn't want to write all the low-level plumbing code to deal with object state, remoting, and so forth over and over again. I wanted to reuse it, so I created a framework to handle those details so that I could focus on my business code.

Unfortunately, it doesn't take long to realize that even with all the low-level plumbing wrapped up in a framework, you still do a lot of repetitive coding. In each class you need to declare a variable, write a property, add validation rules, and repeat for the next variable….

You know you need a code generator when you can describe your coding process as an algorithm.

Computers run algorithms; people shouldn't be reduced to that level. We should remain focused on the prize—on the business value we can provide by coming up with that awesome design to meet the user's needs. If code generation can help reduce the amount of repetitive, algorithmic coding we do, then we have more time for the really important things—like getting that raise for a job well done.

Code generators can't do everything, but they reduce the drudge work. By allowing a code generator to do the boring bits, we can focus on the interesting coding problems and the implementation of sticky and complex business rules and processing.

It turns out that writing a robust code generator is no walk in the park. It's easy enough to create a simple, limited generator. It takes a lot more thought and design work to come up with one that's robust and flexible enough to generate the code for even a moderately sized business application.

Fortunately for all of us, Kathleen has done a lot of the thinking and designing already and has been gracious enough to put her thoughts into this book. It should get you started down the path toward using code generation in your software development.

Rockford Lhotka
Magenic Technologies

EXCERPT

Foreword by Don Kiely

Kathleen Dollard (who some may remember as Kathleen Joeris from the early years of her participation on online forums) is one of the smartest people I know, spending much of her time and talent probing Microsoft development technologies to figure out how to put them to solid practical use. Through her writing for various software publications, speaking at conferences and user groups, and many years being active on various forums (for which Microsoft made her an MVP), she has proven herself to be devoted to helping developers make the most of technology.

When Kathleen began fooling around with XSLT and talked about generating code using it, it took me a while to grok why she was so excited about the possibilities and why she so quickly became rabid in her devotion to code generation techniques using .NET. But then I began to see what she had understood about the architecture of .NET applications—how well the infrastructure lends itself to tools that eliminate the need to write boring but important code over and over again in well-structured object-oriented applications. It makes my job as a developer that much more fun that I can write the tedious code once—not just once for a project but once for *all* projects.

With code generation based on a solid framework, developers can spend their time on the interesting part of any application, the custom code that makes each app unique. Her use of Rocky Lhotka's innovative CSLA framework in this book provides two major benefits. First, she didn't have to spend a lot of space in the book explaining the concepts behind frameworks and why they benefit your business applications. Instead, she was able to tightly focus on making code generation work even for complex frameworks such as Rocky's. Second, you have the benefit of the combined thinking of Kathleen Dollard and Rocky Lhotka, two of the most innovative thinkers and developers in our industry. Even if you don't adapt their approaches to development directly, I guarantee they'll stretch your understanding of .NET development in new and exciting directions.

While Kathleen was writing the book, I implemented her ideas and tools in a Windows Forms application I was building for a client. The app is fairly straightforward but built on a very complex database structure with many tricky relationships. I was a bit more of a guinea pig than I would have liked using the code generation tools but was repeatedly impressed by the sheer brilliance of the ideas embodied in her code generation techniques. And it was exciting to watch her ideas grow and mature into the book you are holding and the code generation tools that accompany it.

Few people have thought longer or harder about multitier .NET applications than Kathleen, making her code generation techniques all the more worthy of careful study. They're not easy; you'll need to work hard to understand the ideas

even if you know the underlying technologies well—such as .NET, XML, XSLT, and reflection. You'll also have to work hard to implement them in your first project, particularly if you have your own custom framework you use in your applications. (And that's part of the beauty of her techniques: You aren't roped into a single framework or techniques that may not fit your development style.) But once you get it going, you truly have one-click generation of the repetitive infrastructure of an enterprise application, with clear and manageable delineation between generated and handcrafted, custom code. I can confidentially vouch for the efficiencies of this approach and how much easier it makes adopting changes and maintenance over the life of a project.

Although we live and work thousands of miles apart, through the magic of instant messaging and occasional phone calls, I can vouch for Kathleen's fervor and single-minded devotion to the concepts she writes about in this book. And the sacrifices she made (don't worry, Kathleen…you'll be back in a kayak on a river soon now that the book is done!). It's been a rush watching the process, exciting to see it take shape from early concepts to what you are reading now.

It's not easy material, but true innovation rarely comes easy, even when someone such as Kathleen has worked so hard to break the trail.

Don Kiely
Fairbanks, Alaska
`donkiely@computer.org`

About the Author

Kathleen Dollard has been developing business applications for more than 20 years, programming in Visual Basic for ten years, and working with .NET since the early betas. As an independent consultant, she's had the opportunity to work in a variety of domains, including the finance and justice sectors. She's been active in online communities since the BBS days and is a longtime Microsoft MVP. She speaks numerous times a year at user groups and conferences and is a regular contributor to *Visual Studio Magazine*. She's president of the Northern Colorado .NET SIG (`http://www.northerncolorado.net`) and an active member of the Denver Visual Studio User Group (`http://www.denvervisualstudio.net`). What she shares in all these venues is valuable primarily because it's based on real-world development experience as an independent consultant helping organizations such as yours come up to speed on .NET and application code generation.

Kathleen has a bachelor's of science degree in Chemistry from the University of Alabama in Huntsville and a master's of science degree in Analytical Chemistry from Texas A&M University. She began working with computers shortly after leaving school and rarely looks back. Kathleen lives in Colorado with her two sons. She received help on the book from her English springer spaniel and cat. The cat, who is fascinated by the printer, had ample opportunity to watch it. Before tackling this book, she enjoyed hiking, skiing, biking, and kayaking. You can reach Kathleen at `kathleen@kadgen.com`.

EXHEART

About the Technical Reviewer

Don Kiely (MVP, MCSD, MSDE) is a senior technology consultant specializing in developing desktop and Web applications that integrate databases, Microsoft Office, and related technologies using tools including SQL Server, Visual Basic, ASP.NET, and XML. Don has authored and coauthored several programming books, including *Visual Basic 6 Client/Server How-To* (Sams, 1998) and *Visual Basic Programmer's Guide to the Windows Registry* (Mabry Software, 1998). He writes regularly for many industry journals, including *InformationWeek*, IEEE *Computer* magazine, and *Visual Studio Magazine*. Don trains developers and speaks regularly at industry conferences, including TechEd, VSLive!, DevConnections, and others.

Don earned a bachelor's of science degree in Civil Engineering from the University of Notre Dame and a master's of business administration degree from the University of Colorado but fortunately learned the error of his establishment ways. In his spare time he roams the Alaska wilderness by foot, dog sled, skis, and kayak with his dogs Mardy (Irish Shedder) and Izzi (Karelian bear dog) while vehemently opposing the opening of the Arctic National Wildlife Refuge to the devastation of oil exploration and production. Contact him at `donkiely@computer.org`.

EXHEART

Acknowledgments

Writing this book has been an incredible experience. I couldn't have done it without the help and support of many people.

My friend and technical editor, Don Kiely, has watched my code generation from the beginning and failed to talk me out of writing the book. I asked him up front to be sure he didn't go too easy on me with the tech editing—and, well, he didn't. This was a very difficult book to tech edit because there's a lot of code and unfamiliar technical concepts that were difficult to explain. Don helped shape this book in many ways. For example, when I thought I was nearing the end of writing the book, I sent him a major release and explained the file he had to create manually. His response was "What? You've already got that from the metadata extraction!" Over the next two months, the important implementation of Object Relational Mapping (ORM) emerged. He used the tools and templates in this book on a real project while I was still developing it—which was brave, occasionally painful, and a huge benefit in the quality of the book. He found numerous places where handling a few more scenarios made life easier for you, as well as many outright bugs. Dozens of key aspects such as removing prefixes, recognizing lookup tables, and handling checkboxes were a direct result of his feedback. He stuck with me throughout this process, and his friendship and encouragement meant the most.

I'm very grateful to Rocky Lhotka for creating and releasing the CSLA architecture, for supporting the way I'm using it, and for reviewing Chapter 8. CSLA is the latest in a series of contributions Rocky has made to our understanding of architectures in general and the implementation details driven by specific platforms. Rocky gives you real architectures that work on real projects. I admire the passion he has in sharing what he's learned with the community and encouraging the rest of us to pick up the ball and run further with it.

My son, Ben Joeris, helped me on many fronts. He reviewed chapters and was a sounding board for my good and bad ideas. He also did a large portion of the C# conversion available at the Web site. His confidence and skills have been amazing and contributed directly to the CodeDOM sections of the book. Writing a book impacts a family, and Ben and his brother Peter's patience with late dinners and a stressed-out Mom were amazing. I also feel very lucky to have had the support of the other half of their family—Len Joeris and Helen Rajala.

Bill McCarthy contributed several pieces to this book. In our past collaboration, he introduced me to the concept of minimizing state covered in Chapter 10 and to running stored procedure scripts through .NET. He also instigated many discussions on ways application code generation wasn't quite good enough yet and challenged me to make it better. And throughout he has been a good friend and has reminded me that there is more to life.

Nancy Folsom taught me the concepts behind debugging the Scientific Method, and I'm just stubborn enough to think that if Nancy could finish her book, I could too.

Many people at Microsoft have contributed to my understanding of .NET and therefore to this book. I'd particularly like to thank John Rivard for his help on CodeDOM issues and Jeff Geller for pointing out the relationship between requirements and metadata to me.

I thank Deborah Kurata for introducing me to publishing when I tech edited her first book. I followed this with a long relationship with *Visual Studio Magazine* (VSM)—previously *Visual Basic Programmer's Journal* (VBPJ). I particularly appreciate Patrick Meader's support and what I've been able to learn from Patrick and the wonderful editors at VSM, particularly Eileen Cohen.

Technical communities have been important to me for a long time—since the old BBS days. Happily, the technical communities I'm part of now extend into the real world. I'd like to thank the members of the Denver Visual Studio .NET User Group for their long-term support and not snoring through my early speeches. I have a very special place in my heart for the individual members of the Northern Colorado .NET SIG. Writing a book is a process and these folks helped me remember that it did matter—both personally and professionally—and held me accountable to also keep my head on straight, at least sort of straight.

I'd like to thank the individuals that took their time to review parts of this book. In addition to their reviewing efforts: John Bugarin helped shape the early chapters of this book and encouraged me to take application code generation beyond simplistic solutions without making it unreachable. David Cowles offered his UML expertise and broad picture perspectives. Joe Feser is my hero because he's been doing XSLT code generation in the trenches for a long time—and he bothered to post the xslt.xsd file that Microsoft forgot to give us (discussed further in Appendix A). J. Mel Harris has been a friend and offered advice when I needed it. L. J. Johnson and his wonderful family helped me take a much needed vacation partway through the book—yeah, for me going to TechEd is a vacation. J. Doug Kent with his personal commitment to teaching himself what he wants to know also reminds me that one of the great things about this industry is the range of ages of people involved. Chris Wallace not only spearheads the vibrant Denver Visual Studio .NET user group but also encourages individual members to reach out for their professional goals. The unique input each of these people gave in their reviews certainly made the book better.

I'd also like to thank the folks at Apress. Dan Appleman convinced me a book was the right format for conveying what I know about application code generation (and I might one day forgive him). Tracy Brown Collins showed me how to use Tips and Notes. Kim Wimpsett was an awesome copy editor, patiently correcting my redundant mistakes and ensuring that she marked anything where edits might change the meaning. Lori Bring, Christine Calderwood, and Molly Sharp brought the whole thing together into what you now hold. Sofia Marchant attempted to keep us all in line.

And, finally (the hokey part), I'd like to thank Stephen King. Oh, don't worry; what I learned from him wasn't to incorporate either horror or fiction into my writing. Although he never said it in these words, I walked away from his *On Writing: A Memoir of the Craft* (Scribner, 2000) deeply believing that it's all in how you use the language, baby.

EXHEART

EXHEART

Introduction

Application code generation revolutionizes the way you write programs by letting you reuse solutions wherever and whenever you want. Focusing on the big picture, you can build better solutions. Or industry gurus can use templates as an IV to feed their expertise directly into your project. Either way, you'll build a higher-quality application faster and be ready for changes during development and throughout the application life cycle. Code generation provides this level of software agility at the same time it allows you to create massive amounts of code.

Code generation is any way you create code other than writing it line by line in an editor. It's already part of your development strategies because Visual Studio uses it to perform several key tasks. But that type of code generation is free with Visual Studio, so you don't need to read a book about it. Code generation can also become a tool in your own arsenal for solving problems, and this book teaches you how to do your own code generation. Third-party tools don't yet offer the flexibility and control you need.

Code generation is simply code that writes code. At first glance that may seem to offer more complexity than benefit, but a great deal of what you write is applying similar patterns over and over. You're probably already effective in spotting identical code to extract common logic into utility methods and class hierarchies. What if you could be just as effective in pulling out pattern-based code that was similar but not identical?

Code based on patterns is code that differs in predictable ways—including naming and typing. As a quick gut check, it's probably the boring code you hate writing and would delegate to a junior programmer if you could. A good place to start looking for it is in your business objects (or your strongly typed DataSet). This is an example of pattern-based code:

```
Public Property CustomerID() As System.Int32
   Get
      Return m_CustomerID
   End Get
   Set(ByVal Value As System.Int32)
      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
```

Everything that changes between the code for these two properties is marked in bold. That's only two things! `CustomerID` changes to `CompanyName`, and `Int32` changes to `String`. If you could say "Hey, do [this] with [that]," then you could build these two blocks of code very easily. That's the trick of code generation. You separate the pattern—everything not in bold—from the variable information. Throughout this book, I'll call the pattern a *template* and the variable information *metadata* because it's the information about your application. Code generation is this:

```
< template > + < metadata > = < generated code >
```

Code generation without regeneration can be worse than nothing. If you incorrectly create 100,000 lines of code and have to fix it by hand, the job is enormous. Everything in this book assumes you'll regenerate your code at a whim. You can regenerate daily when you know something has changed or just because you feel like it. You regenerate with total confidence that you won't mess anything up. This is fundamental to making code generation a principled process.

Code generation without giving your classes individual personalities to accomplish unique tasks is useless outside trivial applications. You're still very important, and you still have to write code. The goal is to focus your efforts on the code that needs your attention. You integrate this handcrafted code with generated code and components to quickly build an effective, robust application. That's hard to do without a roadmap.

To make code generation a valuable part of your development strategy, you need to take it further and build a coherent process out of the mechanics of generating code, and you need to base this process on sound principles. This principled process is your roadmap. It incorporates creating your architecture, collecting metadata, outputting code, personalizing the resulting classes, and addressing details that make the quality of code generation at least as good as the code you write today.

There are a lot of potholes along the road of code generation. Some of these are like potholes on the Yukon River ice road. If you hit a big one, it could swallow your whole snow machine and derail your project. Code generation is going to revolutionize our industry over the next five years. But along the way, there are going to be people pulled out of ice holes in late stages of hypothermia. This book is your map to keeping your code generation on track to being effective.

How Will Code Generation Help You?

Creating 150,000 or 500,000 lines of code in a couple of minutes is not a dream—I can show you how to do it. It makes no difference whether you have 5,000 lines of code or 5,000,000 lines of code; as long as it follows expected patterns, you can generate the code in roughly the same amount of time.

What's more, you and your team develop those patterns. It's not a framework you adopt from someone else and adjust your strategies to match. You build patterns that match your development and architectural strategies—and you can change these strategies as your architecture evolves. You can make broad changes and see them in code almost immediately. You can move ahead with confidence as soon you've clarified your interfaces because anything else can be adjusted as you fine-tune your templates and test performance.

Because the changes you express in templates affect code throughout your application, the resulting code remains entirely consistent and predictable with no additional effort. Special cases, such as unusual data types, complex validation, and unique processing are in predictable, protected locations—either in metadata or in handcrafted code.

You decide how to incorporate code generation to jump-start your development. It might save you a lot of time, effort, and hassle, or it might let you write your application using a robust architecture that's beyond your current resources. You can use the samples in this book to apply code generation this afternoon. That's an exaggeration because you'll want to finish reading the book—there's a lot to cover—but these are valid real-world techniques that are ready right now for your development. If the samples don't match your architecture, you can learn from the samples and then build whatever architecture suits your needs. In either case, you'll appreciate the code generation tools provided with this book, but your greatest gain will be understanding how the steps of code generation and the corresponding principles work together to make code generation an effective process. Some of the benefits of the application code generation approach presented in this book include the following:

- The obvious benefit of code generation is that you can speed up your development and scale your work to create 3 or 3,000 classes with the same amount of effort.

- Application code generation offers software agility. When you change your metadata, perhaps adding new database columns or validation rules, these changes are available for use within your user interface code at the touch of a button.

- The higher level of software quality that application code generation provides results in lower maintenance costs; inconsistencies don't creep in during development and maintenance. Code generation allows you to make conscious decisions about where and how to implement evolving best practices.

- Code generation offers a level of abstraction that separates information about your application (metadata and handcrafted code) from the technology behind it. This gives you a practical path to separately reuse both your architecture and applications details.

- Even though the architecture your templates express is under your control, you aren't on your own. Templates have huge potential in providing a conduit for the best available experts to contribute directly to your project.

- This book includes templates that implement business objects for Rockford ("Rocky") Lhotka's Component-based, Scalable, Logical Architecture (CSLA). This means the middle tier templates in this book are robust, well documented, and ready for the real world.

Traveling Down this Path

If you're interested in code generation, I assume you're not a beginning programmer or even a beginner to the .NET languages targeted in this book. You're probably frustrated with the difficulty of the development process itself and your inability to consistently turn the high quality of your own work into successful applications.

I'm amazed by the tools we have in front of us and disheartened that as an industry we haven't been able to turn this spectacular development environment into significantly increased productivity. The rate at which our efficiency improves just barely keeps pace with the increasing complexity of the applications we write and the expectations of users.

Over the last couple of years, I have been actively pursuing ways to efficiently express patterns in producing applications. One of the things I've learned is that to leverage code generation you need to consciously incorporate it as a deliberate process within your overall development. The significance of how this process naturally organizes your development is one of the reasons I think code generation will revolutionize our industry. I've written this book to teach you this process along with the details of making it work in the real world. Many of the techniques I'll cover are novel, frequently overlooked, or poorly documented. These include `IndentedTextWriter`, XSL Transformations (XSLT) for code generation, and many aspects of the CodeDOM.

A Note on Language and Tools

In targeting a book to .NET, it was difficult to decide what language to write the code in. The content of the book applies equally to C#, Visual Basic .NET

(VB .NET), C++, J#, and many of the other .NET languages. I decided on VB .NET in the text of the book with the code repeated in C# on the Web site. I think it's easier for C#, C++, and so on programmers to read Visual Basic .NET than the reverse. Although I'll use VB .NET, I generally use framework features and types, avoiding most VB .NET language extensions (the VB .NET functions). The style of VB .NET that I use is strongly typed (`Option Explicit` and `Option Strict` both on) with the `Microsoft.VisualBasic` library removed. This will maximize its readability for C# programmers. C# programmers can also follow along with the code from the Web site that's organized by chapters and contains page references. I'm not planning a code translation into C++ or J#.

NOTE To download the code, go to the Downloads section of the Apress Web site at `http://www.apress.com`. See the ReadMe file in the download file for more information on getting started with the code.

I tested the code in this book against .NET 1.1 (Visual Studio 2003). Other than removing some parameters on the `XSLTransform` method, you should be able to use version 1.0 as well, but I haven't tested this. Nothing in this book strictly requires Visual Studio .NET, but I think you'll have an easier time if you're using it rather than the framework's command line tools.

It's hard to predict where the Mono framework will be in the next six months or a year, but if you're targeting other platforms, you can certainly generate your code for a Mono deployment and can probably do your code generation through Mono as well.

What Does This Book Cover?

The first part of this book is roughly organized around the five steps of code generation as follows:

Chapter 1, "Introducing Code Generation": The first step of code generation is architecture. Because this book uses CSLA, I'm skipping an architecture chapter. Instead, Chapter 1 provides an overview of the five steps and five principles as well as introducing the three basic mechanisms: outputting to streams, XSLT code generation, and the CodeDOM.

Chapter 2, "Extracting Metadata": The second step of code generation is extracting metadata. This chapter looks at extracting metadata from a variety of sources with a focus on SQL Server metadata extraction. It also introduces the intimate merge of metadata that's used to incorporate additional manual metadata information.

Chapter 3, "Outputting Code": The third step of code generation is outputting code, and this chapter introduces the code generation harness and shows details of outputting code in the three mechanisms this book covers.

Chapter 4, "Writing Handcrafted Code": Code generation embraces handcrafted code. Managing it is the fourth step of code generation. The focus of this chapter is how you isolate handcrafted code to protect it across regeneration cycles.

Chapter 5, "Tying It All Together": Taking care of the details critical to generated applications meeting or exceeding your existing quality standards is the fifth step of application code generation. Chapter 5 takes a closer look at the main harness tool, file management, project files, source control, and debugging, as well as how to move your infrastructure forward to start your next project.

The second part of the book looks at templates for different parts of your development. A dedicated chapter covers each of the six key areas you'll focus on in building your application as follows:

Chapter 6, "Mapping Database Structure into Business Objects": The degree of mapping between your database and business objects varies a great deal with your specific scenario, but it's dangerous to commit to maintaining an exact mapping over the project lifetime. This chapter solves this with a relatively simple Object Relational Mapping (ORM) process and shows you some of the gritty details behind defining your mappings in Extensible Markup Language (XML).

Chapter 7, "Generating Stored Procedures": Once you've got ORM output that defines your stored procedures requirements, you're ready to build them. This chapter shows you how to generate stored procedures to create, retrieve, update, delete, and perform a set select.

Chapter 8, "Generating Middle Tier Templates": The middle tier of your application represents your database to your user interface layer. This chapter covers the basic issues you're likely to encounter as you convert an existing architecture to code generation and contains business object templates for Rocky Lhotka's CSLA architecture.

Chapter 9, "Generating a WinForms Application": Once you've got business objects, you're ready to build a user interface. By this point, you've seen a lot of templates, so this chapter focuses on specific WinForms strategies such as an alternate derivation strategy and generating user controls. This chapter also shows how to implement generic container forms and dynamically driven Multiple Document Interface (MDI) parent forms.

Chapter 10, "Generating a Web User Interface": One of the great things about CSLA is that it's easy to create both a Windows and a Web user interface. Unfortunately, Visual Studio throws a couple of kinks into the code generation of code behind classes, and I'll show you how to work around them. I also walk through ASP.NET templates so you can see code generation work in another language.

Code generation is coming into our industry to stay. To get the full picture of code generation, you need to see where it's going as well as where it is today:

Chapter 11, "Establishing Perspective": This final chapter offers support for introducing code generation into your organization, including a checklist of features that support the principles of application code generation. It then looks forward to the near future and what Yukon, Whidbey, and Longhorn mean to code generation. Looking more broadly, I'll look at major paradigm shifts that code generation can usher into the industry to revolutionize it.

I've shoved a lot into this book. As I tried to maintain focus on code generation, there wasn't room for important topics, so I've included a set of appendixes as follows:

Appendix A, "X-Survival": If you aren't up to speed with XML, XSLT, XML Schema Definition (XSD), and XPath, you'll need some review to understand parts of this book. For my own research, I've been grateful for the many thousands of pages written on XML, XSLT, and related technologies, but it makes it all seem too hard. This appendix distills the important information about these technologies.

Appendix B, "Dissecting the Metadata Extraction Tools": There are some scenarios where you might want to understand the metadata extraction tools. You might just want to understand what's happening under the covers, or you might need to extract metadata from an alternate back end. To help you with this, I walk through key parts of the metadata extraction tools.

Appendix C, "Details of the Code Generation Harness": I've included the details of the code generation harness because you might want to extend it and because it uses a cool alternate technology I didn't get to cover in the main body of the book. Dynamic techniques fit a small set of niches, but they're useful in those niches. The generation harness editing tools are entirely dynamic based on an XSD. The script engine is a reflection-based script engine.

Appendix D, "CodeDOM Issues, Tips, and Tricks": Chapter 3 explains how the CodeDOM works but leaves out information that's of interest only to people that elect to work with the CodeDOM. Although this information is probably only interesting to a subset of readers, if you're working with the CodeDOM, I think this appendix will save you a good deal of time.

TIP Read the ReadMe file in the download!

Teaching you to effectively use the five steps that make code generation a coherent process is the core of this book. It has real answers to real-world problems and shows you a route to quick, sane, smooth development that accommodates change. You'll learn the best concrete real-world techniques I've discovered, as well as tricks and best practices. You'll significantly reduce your development costs and increase your software agility. I'll keep this focused on showing you how to quickly become effective using code generation to improve the speed, reusability, agility, and consistency in your own application development.

Part One

The Process

FREEHAND

EXHEART

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.

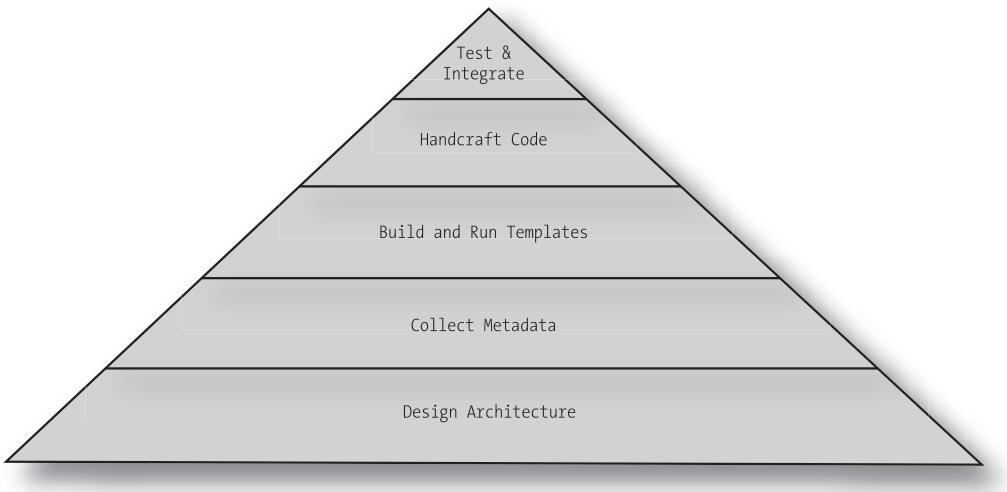


Figure 1-1. Each of the five code generation builds on the previous step.

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.

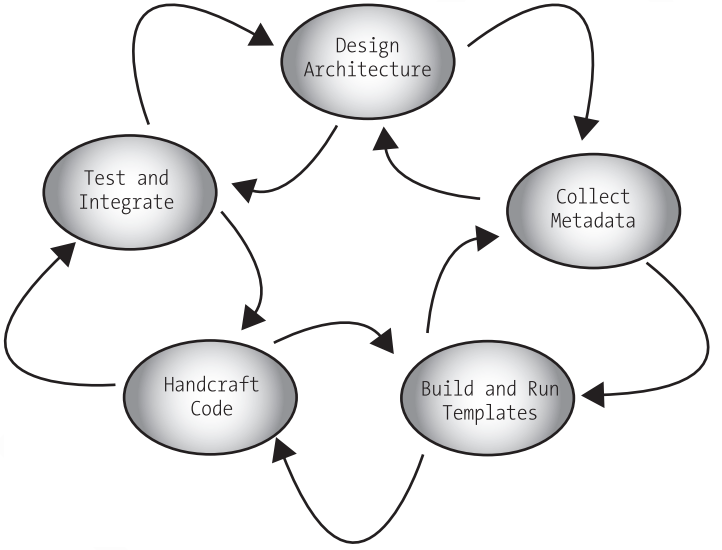


Figure 1-2. The process is cyclical, with each step contributing to the next and also providing more information for improving the previous step.

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).

EXHEART

CHAPTER 2

Extracting Metadata

Principle #2: Collect metadata as a separate, distinct step that can independently evolve.

Extracting metadata means reaching into available sources, dragging out the metadata, and stacking it up neatly so other processes can exploit it. Exploiting the metadata you've stacked up will be the next big jump in programming. It'll look like code generation, but that's just one way to exploit metadata. Metadata is the key that unlocks the door to faster application development.

It's not easy to define metadata concisely. *Metadata* is an Extensible Markup Language (XML) description of your application in terms of databases and/or assemblies, tables and/or objects, columns and/or properties, and so on. The bulk of metadata describes details of the database your application uses, but it can also include details such as user interface information. For instance, if you're building a bookstore application and your database has a table containing all the books you sell, with columns for their prices and availability, a simplified version of your metadata might look like this:

```
<dbs:DataStructures xmlns:dbs="http://kadgen/DatabaseStructure">
  <dbs:DataStructure Name="pubs">
    <dbs:Tables>
     <dbs:Table Name="Titles">
        <dbs:TableColumns>
          <dbs:TableColumn Name="title_id” AllowNulls="False" Type="int" />
          <dbs:TableColumn Name="title" AllowNulls="False" Type="int" />
          <dbs:TableColumn Name="price AllowNulls="False" Type="int" />
          <dbs:TableColumn Name="available" AllowNulls="False" Type="int" />
        </dbs:TableColumn>
     </dbs:Table>
  </dbs:DataStructure>
```

Metadata includes everything you use as input for your templates. Chapter 1 included templates you could apply to any data table by supplying the appropriate XML metadata input file. This chapter is about acquiring XML metadata—automatically extracting it where possible. The easiest format for metadata is XML. This XML file defines the application you're going to build in a way that simplifies code generation. It defines the structure of your application, including its objects, properties, and methods.

NOTE Metadata won't define all of your classes because your application includes utility and handcrafted classes that you won't generate. Furthermore, it won't define all of your methods because you'll define additional housekeeping methods and properties within your templates.

Metadata fits into your code generation because it's part of this magic incantation:

```
< metadata > + < template mechanism > = < generated code >
```

Metadata comes from one or more sources, often one such as a database that maps directly to your requirements. Metadata *extraction* takes you from some source to metadata usable for code generation. This process reaches into the source and creates an XML document containing the supplied metadata. Figure 2-1 illustrates the role of this extraction in code generation. You may use more than one metadata source in creating your application. Once you've got your metadata, it serves as the input for the templates that generate source code for your application. Throughout this chapter, you'll see mechanics of metadata extraction. Chapter 6 merges this extracted metadata with an Object Relational Mapping (ORM) view of your database. Chapter 3 shows how to exploit metadata to generate code for your application.

NOTE You may have expected that the hard part of code generation would be creating the templates that produce the code. Actually, the hardest part is figuring out your architecture, followed by collecting the metadata. By the time you know what you're going to build and have the metadata ready, the process of building templates is straightforward. Therefore, this chapter covers the toughest stuff in the book. So be patient if this chapter becomes difficult.

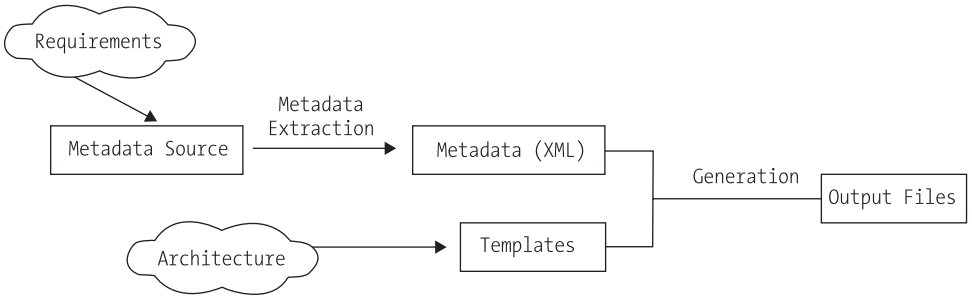


Figure 2-1. Schematic view of the role of metadata extraction

You can handle metadata in a couple of ways. You can just grab metadata whenever you need it, or you can make metadata extraction a distinct process. Grabbing metadata whenever you need it means simultaneously extracting metadata and outputting code, which makes things more difficult than they need to be. Combining these processes makes it difficult to debug and hard to reuse your extraction mechanism. You lose the metadata/template independence that's a key benefit of code generation. It's important to make metadata extraction a distinct step you run and debug it on its own. In other words, instead of going directly to the database when you need to know the columns in a table, retrieve it from an already-created XML file.

Although your database is probably your core metadata source, you also have access to a variety of other metadata sources—anything that has information that could contribute to your application. The problem of extraction is rarely finding the metadata sources; it's getting the information stacked up neatly ready for your templates. To do that, you have to understand how the metadata source exposes information and what format is easy for code templates to consume. Much of this chapter talks about how selected sources expose data. I'll focus on XML Schema Definitions (XSDs) and the SQL-92 standard as primary metadata sources. Understanding these sources will help you work with them and clarify how you retrieve metadata from any source. Extracting data from an XSD illustrates how to extract data from any XML source using XML Transformations (XSLT). Extracting metadata from a SQL-92 database illustrates how to use .NET code to extract metadata from a source with a programmatic interface.

Introducing the Process

The code generation process needs a single monolithic XML metadata input. But a monolithic metadata file is hard to manage. Different people may perform different metadata extraction and code outputting tasks. Chapter 3 shows how to make code generation a granular process. It makes no sense to create a granular process and then have it rely on a monolithic file that everyone is trying to modify. So, I'll show you how to merge individual XML metadata into a transient monolithic file in the "Merging Metadata" section of this chapter.

Figure 2-2 is a high-level view of metadata. On the right, the single monolithic XML metadata file is composed of four segments. You might add another segment or two, such as report definitions, but you'll have a small number of high-level sections in the XML metadata document.

The monolithic XML metadata file is created on individual developers' machines and is transient. You'll output it to disk for debugging, but you don't want to put it under source control. One of the differences between hacking around with code generation and making it a conscious process is intelligently incorporating source control. By keeping only the individual metadata files under source control, you gain important flexibility for your team.

Seeing all these discrete steps might be making you nervous, particularly if you've worked with the tedious process of generating strongly typed DataSets in Visual Studio. I'm committed to one-click code generation smoothly integrating these granular processes. You'll see details of one-click activation of the multi-step code generation process in Chapter 3, so you'll need to trust me on this and postpone testing these tools until then when you'll have the metadata, the templates, and the generation tool.

Understanding Metadata Sources

Metadata is all of the changeable information you need to build source code. This is the bulk of what changes between your applications. Because the easiest metadata to visualize is from databases, I'll often talk about metadata in relation to data structures—database fields and columns. Clearly, not all metadata is related to data structures.

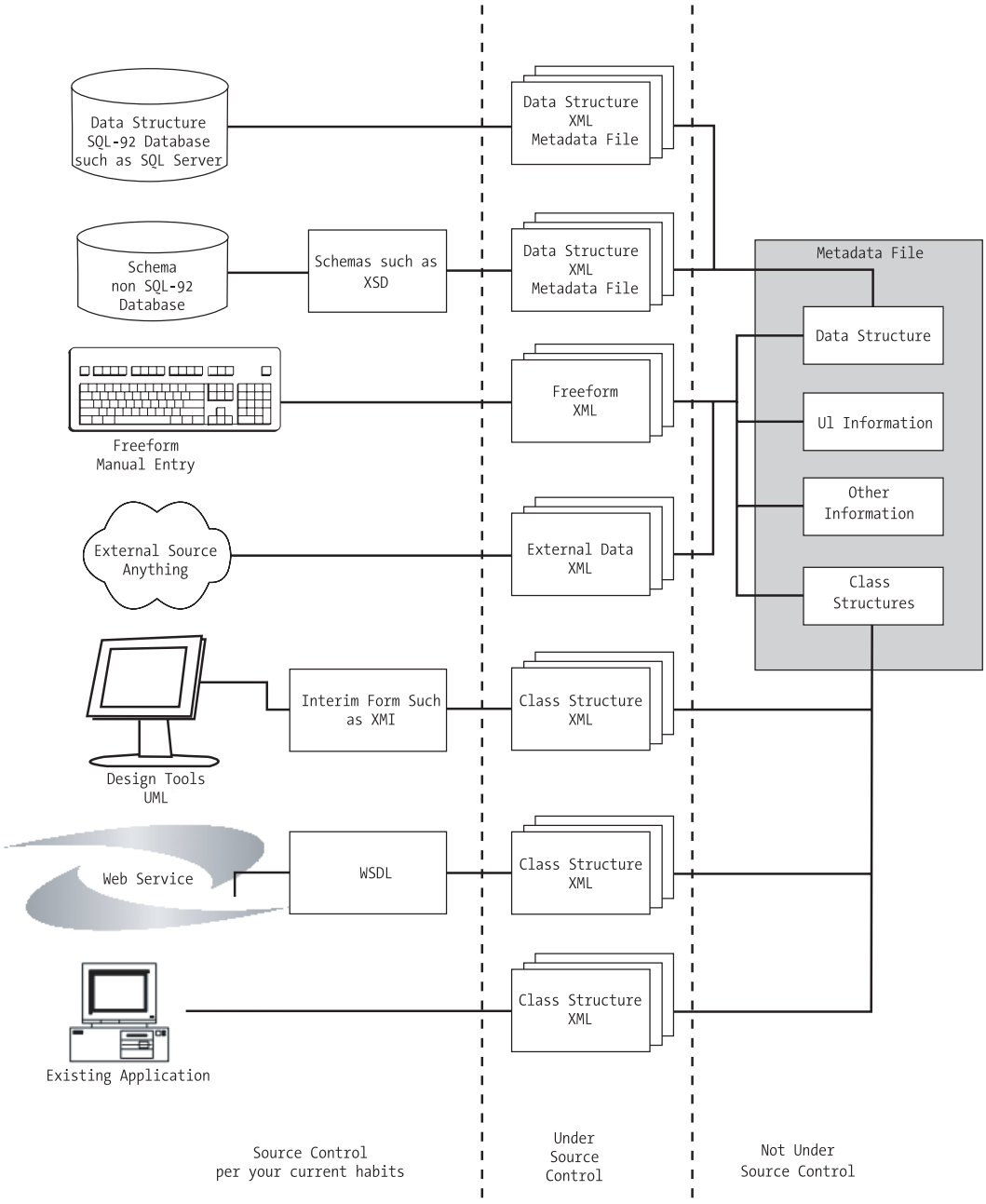


Figure 2-2. Metadata sources

If metadata is literally any data you need to build your application, you can get it from anywhere. It makes no difference to later processing where you get the metadata. There are at least seven kinds of metadata sources as follows:

- Schema definitions such as XSD
- Databases such as SQL Server
- Manually entered freeform metadata
- Web Services via Web Service Description Languages (WSDLs)
- External sources, such as mainframes, Excel, and so on
- Design tools such as Unified Modeling Language (UML)
- Existing applications and source code

You can further divide these types into categories based on how you extract the metadata. You extract using .NET if the source has a programmatic interface such as databases, external sources, and existing applications. You retrieve it using XSLT if it's an XML source such as XSD, WSDL, and XMI output from UML[1] diagrams. You can also manually enter metadata by typing in the XML. As you can see in Figure 2-2, regardless of the process, you create a series of individual files.

You can also categorize metadata sources by the high-level element that contains them in the monolithic metadata file. There are only a few broad categories—such as data structures describing data, class structures describing data and functionality, and User Interface (UI) structures describing UI details.

Because there are so many metadata sources, I could write a very long chapter on metadata extraction, but it would be tedious and repetitive. So, I'll tell you a little about each of these sources and then follow up with a deeper discussion of three metadata sources: XSD schemas, SQL-92 databases (including SQL Server), and freeform metadata. These three sources represent the three key approaches to metadata extraction: XSLT templates, .NET programs, and manual entry. You'll have to develop your own extraction tool for the other five metadata source types (and any others I omittted), but this chapter will give you a significant head start on writing them.

---

1. I win Bingo with five acronyms in one sentence! XML is self-describing data whose structure can be defined using an XSD schema. Appendix A discusses XML and XSD. WSDL is basically a specific subset of XML used to describe Web Services. UML is the Unified Modeling Language and is a diagramming standard intended to improve communication during software design processes. XMI is another subset of XML that's designed for exchanging UML information.

Using Schema Definitions Such As XSD

Schemas explicitly describe the structure of data. Schemas often describe the structure of XML files. Schemas offer a fallback position for finding metadata sources. You can describe almost any data with an XML schema, and you can generally create these schemas via a tool such as XSD.EXE that's part of Visual Studio .NET.[2] This gives you a good route to creating metadata when there isn't a more obvious route, such as the programmatic interface of the SQL-92 standard.

TIP Use schemas such as XSD as a metadata source when your data source is an XML file or a database that's supported by ADO.NET but doesn't support the SQL-92 standard.

Several styles of schemas are available—including three styles for XML.[3] The most important of these is XSD. XSDs are generally created via an inference tool that finds patterns within an XML document and expresses these patterns and rules in XSD. You'll rarely need to create an XSD manually, but you need to know a bit about how XSD files are structured to read and understand them.

TIP Appendix A's section on XSD gives a detailed breakdown of an XSD file.

Schemas are useful anytime you need information about a data structure. XSDs are important in validating that the structure of an XML file is correct. Visual Studio also uses them for IntelliSense, autocompletion, and ToolTips. I use them for the data-driven XML editor of the code generation harness you'll see in Chapter 3.

NOTE You'll see XSD schemas used two different ways in this book. XSD can itself be a metadata source. It can also be used to describe other XML files, including the individual metadata files and the scripting XML used in Chapter 3. Appendix A covers XSD basics. Later in this chapter you'll see more details about working with XSD as a metadata source.

2. You can access XSD.EXE within Visual Studio by adding a DataSet or XML Schema to your project.

3. The three major schemas for describing XML data are External Data Representation (XDR), Document Type Definition (DTD), and XML Schema Definition (XSD).

Because all ADO.NET data can be expressed as XML, all ADO.NET data can be described by an XSD. This isn't particularly profound because the XSD specification is sufficiently flexible to describe just about anything. Anything you perceive as data is likely to fit in XML that you can describe with XSD. XSD is itself XML, allowing it to be manipulated with any tool that supports XML.

Directly using XSD as metadata turns out to be really ugly. XSD describes tables and columns in terms of complex types and simple types. The result is that the XPath to access the tables in an XSD looks like this:

```
<xsl:apply-templates select=
      "xs:schema/xs:element/xs:complexType/xs:choice/xs:element"/>
```

NOTE This is XSLT usage, but the same XPath expression (the quoted part of the previous XSLT fragment) would also be used in a SelectNodes expression in brute-force or CodeDOM generation.

I don't know about you, but I'm more likely to look at that line and go "Huh?" than think, "Okay, looping through tables here." Translating this to a friendly XML metadata format first allows you to write and maintain code templates that use simpler XPath expressions, which are immediately comprehensible, such as the following:

```
<xsl:apply-templates select="Tables/Table"/>
```

Because the conversion translates the XML of the XSD schema to the XML of the friendly format, XSLT is a good tool for the conversion. You can find an XSDToFriendlyXML.xslt file in the Downloads section of the Apress Web site (`http://www.apress.com`); it's discussed in Appendix B. This chapter's "Understanding XSD's Roles in Code Generation" section further discusses creating friendly metadata.

TIP Use metadata that's friendly to code generation with as much preprocessing, such as data type translations, as possible.

Extracting Metadata from Databases Such As SQL Server

Databases are a great source of metadata; they're arguably the most important metadata source you'll use. But if you can create an XSD for any ADO.NET

source, why not just build an XSD and translate it? After all, Visual Studio .NET uses the XSD to describe database structures to build the strongly typed DataSet. Well, the flexibility of XSD is great, but it doesn't have the richness of a modern database such as SQL Server, particularly if you generate the XSD through the .NET inference engine. It's hard to include details such as defaults, constraints, and extended properties when you're working with an inferred schema. You get more information if you programmatically extract it from the database.

TIP Use SQL-92 techniques when your metadata source is a SQL-92 compliant database such as Microsoft SQL Server or Oracle.

NOTE To extract data from a database, you have to first include the data in the database. Okay, that may sound pretty obvious, but many people don't maintain default values, constraints, and sometimes even relations within their database. If you currently avoid storing this information in your database server, reconsider. SQL Server and most other backends are good at managing data for you and allow a good deal of metadata in a single location. But, if you don't want to store the metadata in your database, you can store it in separate, manually created freeform XML files.

SQL Server offers three different ways to access information on the database structure as follows:

- You can access the system tables directly. But, that's really uncool because Microsoft might change them and invalidate your extraction tool.
- You could use the `sp_help` procedures that are so handy in Query Analyzer. But they're designed for people to read and have a lot of quirks and inconsistencies. Besides, they apply only to SQL Server.
- The best way to extract database structures is the information schema views defined as part of the SQL-92 standard. The "Working with SQL-92 Databases (SQL Server)" section has more details about extracting from these views.

SQL Server, Oracle, and other key databases conform to SQL-92, but others such as Access don't. If you need to extract data from a non-SQL-92 source, you can fall back on the XSD approach, use tools such as .NET's `GetOLEDBSchemaTable` method, or internal queries to create DataSets that contain the system information.

Entering Metadata Manually

It probably doesn't surprise you that some of the metadata isn't in your database or anywhere else outside the brains of your group. This metadata might be tightly associated with database columns or objects, and in that case you can include additional information as extended properties of your SQL Server database. But if there isn't such an association, you're working with a database that doesn't support extended properties, you think extended properties in SQL Server are awkward to use, or your Database Administrator (DBA) resembles an underfed grizzly bear, then the solution is to manually enter the information as freeform metadata.

TIP Use manually entered freeform metadata when the information isn't already contained in a translatable source. This includes information extracted from your requirements and information about UI functionality.

A logical use of freeform metadata is adding captions, ToolTips, and validation to the metadata for your database (assuming you aren't using extended properties to hold this information). Another place you can use freeform metadata is defining UI items. You don't want to redo things the Visual Studio's designers do well, but there are a ton of things you do in your user interface that don't need visual tools. For example, you could create a manual freeform metadata file that defines role-authorized menu structures.

In the "Creating Freeform Metadata" section, you'll learn more about designing freeform metadata and merging it into your monolithic metadata file.

Extracting Metadata from Web Services via WSDL

Is it possible to write a book today without talking about Web Services? For better or worse, its coverage in this book is limited to this section. That's because I think you can extend what you learn from other metadata sources and apply it to Web Services. Web Services are basically a universal messaging system on major steroids. The messaging interface of the Web Service is described using WSDL. WSDLs are metadata. That's their purpose. They describe the service, the data structures that go back and forth, the methods that are called, and the bigger picture stuff about how the service behaves. WSDLs are XML, making them great metadata.

TIP Use WSDL as a metadata source when you're retrieving data via Web Services and the Web Services aren't part of the application you're also generating.

Why would you want to do code generation from WSDLs? If this was a presentation, I could see a number of people waving their hands frantically and the rest with blank stares. There are at least three reasons to do this as follows:

- Creating client-side proxies
- Creating Web Services from a standard or industry-wide WSDL
- Creating code to interact with Web Services

.NET creates client-side proxies for you (either via Add Web Reference or via WSDL.EXE), so you'll only need to do this yourself if the .NET tool comes up short or you have an unusual reason such as wanting to generate a standard set of client proxies for different platforms for testing.

It isn't clear how the vision of industry-wide WSDLs will mature. There will probably be industry-wide WSDLs that you'll support, but most of your work may be extending these standard WSDLs to add value to your products. Implementing these standardized WSDLs will generally involve building entry points into your Web Service with handcrafted code backing them up. In some cases, you might be able to generate your entire Web Service.

If you know the structure of a Web Service, you know the structure of its proxy, and you can write code that interacts with it. Examples of how you might use this are a default Windows application that lets you interact with a Web Service, or doing scatter/gather to a custom user interface. You also might create WSDLs or create Web Service wrappers for middle tier or other data.

Retrieving Metadata from External Sources

The sources you'll retrieve metadata from may be quite varied and include sources such as mainframes and Excel. Think broadly about metadata. What are you working with? What metadata information do you need, and how can you get it? Are you working with laboratory information, process control, or home automation with specific types of sensors and controls you can describe with metadata? Do you need to mimic the structure of mainframe data, either for

synchronization or for an initial import of your data? Can you grab the structure and build `CREATE` stored procedures as well as import and synchronization processes? How can you capture this information?

TIP Keep your mind open to incorporating other metadata sources as you run across them. You can use any predictably formatted information as a metadata source.

Remember that metadata doesn't include information that might change at runtime because you won't regenerate to incorporate changes at runtime.[4] In general, metadata will not include specific network layout information, client configuration information, and so on. If these are part of your application, your metadata might include the structure that holds such details, but not the details themselves. This distinction may be easier to understand if you consider a simple home automation example. Information about individual types of sensors and controls are valid metadata. Each type of sensor or control has a logical set of data, such as a thermometer having a property that can contain its location and another property that can contain the current temperature. The temperature isn't metadata, and in most cases the specific location isn't metadata. The fact that a thermometer is a type of sensor with a current temperature and location is metadata.

Once you understand what metadata you've got and where it currently resides, you can design a freeform XML block. Based on that schema, you can write an XSLT to organize the metadata if it's already in an XML format or write a program to extract it. How hard that program might be depends on how messy the source's interface is.

Extracting Metadata from Design Tools Such As UML

I really wanted to be able to give you good tools for building metadata directly from UML. After much time and grief, I commiserated with one of my UML guru friends (who is a bit gnarly around the edges), and he said, "What? I'd fire anyone who took their UML to so much detail that you could directly build business objects from them. UML isn't an eye chart!" That's not an excuse. Okay, that's an excuse, but it's also a different perspective. I'll admit the real reason I'm not giving you a good UML solution is that it's a complex metadata source to work with, and databases are often a more appropriate metadata source.

4. Strictly speaking, you can, but building and compiling source code at runtime is rarely justified.

TIP UML should make an excellent metadata source. Unfortunately, technical hurdles make it a difficult metadata source to use.

Using Visio

Visio provides a UML modeling tool, but it isn't very good.[5] One of its many problems is that it quite successfully buries the UML metadata where it's extremely hard to access. There are a couple of contorted approaches you can use to extract metadata from Visio UML. You can try to break up the XML save format. But that's a storage format for a drawing program and has no understanding of UML. You can create XML Metadata Interchange (XMI) output if you download a special Dynamic Link Library (DLL) from MSDN[6] (search the MSDN library for *XMI* and *UML*). The XMI standard is an XML layout (defined in a schema) that allows for interchange of information between UML tools. To work with this file, you'll also have to download a Document Type Definition (DTD) schema from the Object Management Group (OMG) Web site, which the MSDN article explains. A simple UML diagram with two classes, three attributes, and three operations was a 313-kilobyte (KB) XMI file. So, for goodness sake, translate it into friendly XML and don't do code generation directly against this file!

Visio also offers a rudimentary code generation feature. From the perspective of real code generation, it's useless. It's buggy, extremely tedious to use, and from a practical perspective just builds shells. However, there's a third approach to extracting UML metadata from Visio lurking here. You can create your UML, generate a dummy C# project, then use reflection to extract the UML structure from these classes, and finally massage problems that occur. Wow, that's so ugly, forget I even suggested it.

Using Other UML Tools

If you're using Rational or another UML tool, you can probably create XSD schemas for the classes you define. The tool may not carry all of the information from your UML design, but it's a start. These products generally also provide XMI support, allowing you to extract a more complete structure—if you're willing to attack XMI's complexities. You also may be able to export the structure to the SQL Server repository and extract it from there—again, a rather roundabout way to accomplish a simple task.

---

5. Visio currently stinks for UML. I remain optimistic that Microsoft will provide a good UML product.

6. You can access MSDN libraries at `http://www.msdn.microsoft.com` even if you aren't an MSDN subscriber.

Extracting Metadata from Existing Applications and Source Code

An existing application can be a metadata source. Wow, that's a weird concept. But it can be a metadata source as source code, a Component Object Model (COM) type library, or via reflection from a .NET application. It turns out that source code is an extremely difficult metadata source to use, and I actually suggest you avoid it unless there's compelling information to extract from it. Reflection, on the other hand, is an easy metadata source to use.

TIP Using existing applications as a metadata source when interfacing with them makes up a significant aspect of your application.

Using Reflection

Reflection is a set of tools in .NET that allows you to query the assembly header (often called *metadata*, but I think that's confusing in this context) and retrieve information about the types within the assembly. *Type* is the .NET word for classes, enums, structures, and so on. In addition to basic type information, the header describes the methods, properties, constants, fields,[7] and nested types within the class or structure. Retrieving this information allows you to build a metadata file describing the type, which can be useful in generating code that accesses the types.

For example, knowing the controls on a form allows you to automate binding or scatter/gather processing, particularly if you use *careful naming* for the controls. Careful naming is a specific technique discussed in the "Introducing Careful Naming" section. Careful naming allows names themselves to carry usable information. Attributes can also help you sort out what each method and property actually does. You can use reflection to generate an XML representation of any .NET EXE or DLL, whether you have the source code. This allows you to automate the interface to a component even if you don't have access to its source code.

TIP To use reflection, you'll have to compile your application. Approaches that result in half-built classes you're planning to finish out with code generation often don't compile and thus aren't usable with reflection.

7. *Fields* are variables declared at the top of your class outside the scope of any method or property.

Using TypeLibs

You may want to extract metadata from an existing COM application that was written in Visual Basic 6, C++, or another language. Extracting the defined interface into metadata allows you to generate code that interacts with these applications. You could also use code generation to create wrapper objects on either the COM or .NET side to reduce the chattiness of interop conversations.

NOTE Chapter 4 explains why it's hard to protect handcrafted code in Visual Basic (VB) 6 and thus why it's a poor language for application-wide code generation. However, if you want a static wrapper class without handcrafted code, generating VB 6 code works just fine.

You can use TlbInf32.dll (which you can download from Microsoft) to provide information about most COM type libraries that's similar to the information provided by .NET's reflection features. If you search MSDN for this file, you'll find how to download it, as well as several articles about using it.

Using Source Code

It'd be great to get metadata directly from your source code. This would allow you to extract comments, track internal conditions, work with the internals of files in other languages such as Visual Basic 6, and incorporate regions. Unfortunately, it's a really hard job. The fundamental reason it's hard is that source code is poorly structured. The job of taking poorly structured information and placing it into consistent, highly structured XML considering all possible variations makes parsing very complex. To do it right, you have to incorporate at least variable file openings, nested types, complex class and method headers (XML comments), multiple namespaces and classes, and nested regions; and you'll want to handle attributes at every level. Whew!

NOTE One reason you might want source code parsing is to capture header comments in VB .NET, similar to C# XML comments. Because XML comments are planned for the Whidbey version of Visual Basic .NET,[8] you might just want to be patient. If you want XML comments in VB today in version 1.0 or 1.1, search Google for VB .NET XML comments. At the time of this writing, commercial products from at least two vendors (Fesersoft's VB.NET XML Comments Creator and VBXC—VB.NET XML), an entry on GotDotNet (`http://www.gotdotnet.com`), and an open-source project at SourceForge (`http://www.sourceforge.net`) are available.

8. Whidbey is the code name for the next version of the Visual Studio family after the Visual Studio 2003 version.

If you have other reasons you want to parse .NET source code into metadata, you can try the automation model of Visual Studio. The `EnvDTE` namespace contains the automation model and parses out the source code structure, which is the hard part. It isn't going to hand you the comments and it requires the file be open in the editor, but it may make some jobs easier.

Why Extract Metadata?

Metadata is a key that unlocks many doors. Not only does it play a critical role in CodeDOM, brute-force, and XSLT code generation, but it also plays a similar critical role in many dynamic data/reflection-driven architectures. Even if you code things the old way, line by line, metadata plays a role although you probably don't think in those terms. When you look at things such as data diagrams and hold that information in your head until you manually incorporate it into your application, your brain extracts metadata. Although it doesn't stack the metadata up neatly the way XML can, it performs the same kind of translation with metadata you'll ask templates to perform.

What you create with metadata can be as varied as the metadata sources. You can create any .NET source code, HTML, XML, RTF,[9] stored procedure, or any other text-based file. This means you can use generation from metadata to perform tasks as varied as creating your database and documentation in addition to producing code. This book focuses on creating .NET source code and stored procedures to select and update data because that's the common ground where we all work. Many of you will take generation further in a direction specific to your application or environment.

Metadata is also a key step in your development strategy. It's a bridge between your templates and your data. It's information that's ultimately derived from requirements ready to marry templates expressing your architecture. Getting your metadata ready for template development is a significant milestone. Later template development and metadata extraction will be like hand and glove. The template determines what metadata you need; the metadata defines what the template can do. Where you get the data may vary and the contents of the templates will change, but the process of extracting metadata and applying templates remains the same regardless of what you're building.

---

9. Hypertext Markup Language (HTML) is for Web page development. Extensible Markup Language (XML) encapsulates data with metadata defining the contents. You use Rich Text Format (RTF) in a generic word processing format.

Designing Metadata

What does the target metadata look like? It looks like XML, of course, but that doesn't tell you anything because you can organize XML just about any way you want. The goal is to have the metadata friendly to code generation. This means the following:

- Pushing all possible processing into metadata extraction to maximize reuse
- Having a predictable location for any piece of information independent of how it's extracted
- Simplifying access even when it means providing redundancy
- Not allowing metadata to become language specific

You want to push as much processing as possible into the extraction step because it supplies an automatic level of reuse. For example, you can do type conversions, such as SQL type names to .NET type names, as part of your metadata extraction. Each time you extract a particular column type, you may use it ten or more times in templates. Because you're going to automate things, repeating stuff is okay.

NOTE Within the XML metadata file, you need a single, predictable location for specific types of data so you can find the data you need when you're doing code generation. Few of us have organized desks. Most of us can find what we're looking for without much difficulty despite the chaos around us. But try to explain to a coworker where a particular document is when you're stuck at home with the flu. Metadata is more like the office supply cabinet than your personal desk. You want to get what you need, and get back to work. So does everyone else working on your project now and in the future. It's not your desk. The organization schemas I use are a good starting point. Evolve them intelligently.

Simplifying access sounds like a noble goal, but how do you accomplish it? Imagine doing code generation, and imagine what you might need at a given point. For example, consider writing a class that represents a database table. You need the table name, the column information, the primary key information, child relations, and parent relations. For each relation, you need the specific

parent and child keys of that relation. In some organization schemes, including XSD, this is scattered all over the place, which is bad. Use the extra bits and repeat parent and child relations within data elements. This makes it easy to pick up a single node and work with it.

In creating some aspects of metadata, you'll have to choose between using a .NET Framework or a language-specific element. This will happen with data types and check constraints. For data types, you can use int, System.Int32, or Integer. They all mean the same thing, but only Int32 works with any .NET code. Even if you don't anticipate language changes, you hope your metadata will have a long life, and who knows how languages will evolve. If you really can't imagine having code that looks like this:

```
Imports System
...
Dim j as Int32
```

or this:

```
using System
...
Int32 j;
```

then include both the language-specific and framework name in your metadata.

Establishing Your Own XML Design Guidelines

How do you design XML? The same data can be expressed in XML in a multitude of ways—how do you settle on one? It helps to articulate a set of rules you can use to ensure consistency. My rules are as follows:

- Use attributes for data that can logically occur just once and is a reasonable length.
- Pay careful attention to elements' names—ensuring they describe the breadth of what they can contain now and in the future.
- Use child elements for items that can possibly occur multiple times.
- Separate child elements by category and generally use grouping elements such as *Tables*.
- Use plurals only in the grouping elements, and use correct grammar and spelling for plurals.

- Unless it's unavailable or illogical, all elements have an attribute named `Name` that contains the logical name for the data item. This name is unique at least within the parent element.

- Use Pascal casing for element and attribute names.[10]

NOTE This is a potential trouble spot, particularly for VB developers because XML tools are case sensitive.

- Minimize abbreviations in element and attribute names.

- Use `"true"` and `"false"` for Boolean values.

- In general, preserve raw data within the XML document when massaging is done. For example, when changing a database type to a .NET type, preserve the original database type name, as well. You can break this rule when you specifically remove information to simplify the XML document, such as you'll see in Chapter 6.

- Use a consistent style for namespaces (`urn`, `uri`, and so on).

- Expect to infer XSD schemas, so try to incorporate its rules, particularly not repeating element names in different contexts, and test XSD inference early and often.

- Use prefixes for all namespaces—avoid using empty namespaces or default namespaces.

NOTE There's nothing magic about this set of rules. The magic is having any specific set of rules that you can turn to when you're wondering "Should this be an attribute or element?" or "How should I name this element?". I hope your organization can agree on a set of rules and allow it to evolve. I'll warn you that I'm imperfect and slip up sometimes even on my own rules.

10. Pascal case has the first character capitalized as well as the start of any new logical word. An example of Pascal casing is DataTable. This is sometimes confused with camel casing in which the first letter isn't capitalized. For example, dataTable is camel casing. It helps me remember the difference that a camel has humps only in the middle. Pascal casing isn't standard for XML and XSLT files, but I think it makes the files more readable.

Using these rules, as well as rules for legal XML, the design of the metadata begins to emerge. You need a root element and want its name to imply the full breadth of data that might be contained in the file. It's helpful to include the word *Root*, such as *MetadataRoot*. The root contains grouping elements. For example, *DataStructures* contains the information about individual data schemas. You might think about naming the data schema root *Databases*, but that's too restrictive because the data may be contained in many forms such as XML, config files, transfer from external programs, and so on. The first cut at the monolithic metadata file is as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<MetadataRoot >
   <dbs:DataStructures FreeForm="true" xmlns:dbs="http://kadgen/DatabaseStructure">
      <dbs:Tables></dbs:Tables>
      <dbs:UserDefinedTypes></dbs:UserDefinedTypes>
      <dbs:StoredProcedures></dbs:StoredProcedures>
      <dbs:Functions></dbs:Functions>
      <dbs:Views></dbs:Views>
   </dbs:DataStructures>
   <ffu:UIStucture xmlns:ffu="http://kadgen/FreeFormForUI.xsd" />
   <OtherStructures></OtherStructures>
   <ClassStructures></ClassStructures>
</MetadataRoot>
```

Introducing the Tools for Metadata Extraction

Three tools for metadata extraction are available at the Apress Web site in the Downloads area as follows:

- **XSDToFriendlyXML.xslt**: An XSLT stylesheet to convert XSD schemas to XML that's friendly to code generation. This effectively extracts usable metadata from an XSD.

- **Database Extraction:** A .NET tool to coerce a rich set of metadata from SQL Server. You can modify this tool to extract metadata from any SQL-92 compliant database, such as Oracle.

- **XML Merge**: A .NET tool that performs an intimate merge to incorporate freeform, external, and reflection-derived (and so on) metadata into a monolithic metadata file.

It's a bit awkward for you to run these tools until you get the code generation harness in the next chapter. You can use a console application if you want to experiment with them now, or you can wait for the generation harness.

NOTE Appendix B offers walkthroughs of key parts of the code for the first two tools, and I explain the third later in this chapter. You can use the appendix for a deeper understanding either because you're most comfortable driving when you know what's under the hood or because you need to tune the engine—making a change to the tools so they work better in your environment.

Isolating portions of your application and development processes that are likely to remain static from those where change is likely improves maintainability. Many tasks of metadata, such as finding parent or child keys or translating a data type, will be done many times within your templates, but the algorithms for creating them will remain unchanged until there are fundamental changes in how databases work. By pushing this work into your metadata extraction step, you can do it once and provide the results wherever it's needed.

TIP Do as much work as you can upfront for creating metadata friendly to your templates. This maximizes reuse and improves maintainability. Extra time developing your metadata extraction tool makes sense because you'll use it not only as this project evolves, but also when you move forward to new applications. Most of the issues you'll encounter are solved in the metadata extraction tool you can download from the Apress Web site.

You can attack metadata extraction at a variety of levels. Imagine a bell curve of all the available metadata with the information you're most likely to use in the middle. For this book, I stretched out beyond the central section and worked to make this tool applicable to most scenarios. But, I've certainly left a lot of blank space at the edges and avoided trying to solve all problems for all people. I wanted to keep these tools and my explanation of them from being excessively complex.

Understanding XSD's Role in Code Generation

XSD plays several roles in code generation. It's the base schema description language used in XML editing, it's used by the code generation harness, and it's a metadata source. XSD is a poor metadata source because it doesn't catch the richness of other sources such as SQL-92 databases and because it's awkward to use. To solve the second problem, you can use a two-phase extraction—first building the XSD and then translating it into a friendly format. When you don't have a rich underlying source, XSD provides effective metadata source.

What's the difference between the structure of an XSD and the friendly XML I'm suggesting for code generation? The basic differences are that the friendly metadata contains familiar terms such as *Table* and *Column* in the element names and contains all information about a table within its hierarchy (descendant elements and attributes). You can compare these formats as you see both XSDs and the friendly XML target (which is the same target as SQL-92 databases) in the next few sections. XSD also provides only a subset of that metadata provided by the databases. The XSDToFriendlyXML.xslt template available on the Web site translates standard XSDs to this friendly XML format. Appendix B discusses this XSLT template.

You usually retrieve the XSD by a process called *inference*. Inference guesses at schema rules based on a sample XML file. XSD is an important but difficult to grasp technology; there's more about XSD in Appendix A.

Inferring XSD Schemas

XSD is itself XML, making it usable with a variety of tools, including XML editors and XSLT. That's cool, but XSD schemas are nonintuitive, ugly, and really hard to build. That's the bad news. The good news is that you may never have to create one from scratch because there are a variety of tools to infer the schema.

The most accessible inference to you is probably the one supplied with .NET—one of several operations performed by XSD.EXE.[11] Table 2-1 shows the different input options for XSD.EXE when you run it from the command line. If you have Visual Studio .NET installed, the easiest way to find XSD.EXE is to open a Visual Studio command prompt. This sets the path to include [Program Files]\Microsoft Visual Studio .NET 2003\SDK\v1.1\Bin, which contains XSD.EXE.

---

11. There's another inference engine downloadable from Microsoft (`http://www.gotdotnet.com/team/xmltools`). If you don't find it, try searching for *XSD Inference* on the gotdotnet site. This engine can build XSDs from XML files that choke XSD.EXE. Another benefit of this alternate inference engine is that you can call it multiple times with different XML documents to evolve a schema. I have used XSD.EXE in this book so you don't have to download an extra tool.

Table 2-1. XSD.EXE Modes

INPUT	OUTPUT	IMPORTANT SWITCHES	SWITCH DESCRIPTION
.xdr	XSD schema	`/o:<directoryName>`	Specifies output directory
.xml	XSD schema	`/o:<directoryName>`	Specifies output directory
.xsd	Strongly typed DataSet or class source code	`/o:<directoryName>`	Specifies output directory
		`/d`	Produces strongly typed DataSet
		`/c`	Produces strongly typed class
		`/l:<languageName>`	Specifies language as CS (C#), VB (Visual Basic .NET), JS (JSharp)
		`/e:<elementName>`	Specifies specific elements to generate (defaults to all)
.dll or .exe	XSD schema for specified types	`/o:<directoryName>`	Specifies output directory
		`/t:<typeName>`	Specifies the specific types to generate

The XSD.EXE application's behavior when you pass a file with the extension .xdr, .xml, .dll, or .exe is important in the context of metadata. With all of these file types, the output is an XSD file.[12] Because you can also express DataSets as XML, you can infer an XSD schema for anything you can stuff into a DataSet. As a result, you can use XSD.EXE to create a schema for most things that you would classify as data. Visual Studio .NET uses the same inference engine to create XSDs if you right-click an XML document and select Create Schema.

NOTE You can also directly create XSDs with Visual Studio .NET. The XSD design surface is displayed when you add a DataSet or XSD to your project. These use the same design surface because a DataSet is defined within Visual Studio .NET as an XSD. Although you may only be familiar with the graphical design tool of the XSD/DataSet surface, you can see the XSD itself if you select the XML tab. You can also programmatically infer a schema for an ADO.NET DataSet using the `GetXMLSchema` method.

12. A separate key role of XSD.EXE is creating the ADO.NET strongly typed DataSet.

Exploring the Structure of an XSD

An XSD schema defines elements. The elements may contain other elements. To show how these elements work to describe the data structure, I'll walk through the XSD schema created by XSD.EXE from the Customers and Orders table of the Northwind database. The root element represents the DataSet and child elements representing the Customers and Orders tables of Northwind.

The opening of the XSD shows that it's just an XML file with a specific set of namespaces. It contains the XML header, a single root element, and a series of defined namespaces. In this case, the name of the DataSet is "DataSet1":

```
<?xml version="1.0" encoding="utf-8" ?>
<xs:schema id="DataSet1"
            targetNamespace="http://tempuri.org/DataSe1t.xsd"
            elementFormDefault="qualified"
            attributeFormDefault="qualified"
            xmlns="http://tempuri.org/DataSe1t.xsd"
            xmlns:mstns="http://tempuri.org/DataSet1.xsd"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
```

NOTE Namespaces are one of the most difficult things to work with correctly in XML and .NET processing of XML. Appendix A covers some of the nuances of namespaces.

The `xmlns:xs="http://www.w3.org/2001/XMLSchema"` defines this file as an XSD. Among the other things this namespace provides, you'll find that if you edit the XSD within Visual Studio .NET, IntelliSense works because Visual Studio maps this namespace to a schema file it can use for IntelliSense.

Complex types contain child elements. Elements representing the DataSet and DataTables are complex types:

```
<xs:element name="DataSet1" msdata:IsDataSet="true">
   <xs:complexType>
      <xs:choice maxOccurs="unbounded">
         <xs:element name="Customers">
            <xs:complexType>
               <xs:sequence>
```

The table elements contain child elements representing columns. Columns are *simple types* that have attributes describing their name, type, and other information. Simple types don't contain child elements:

```
<xs:element name="CustomerID" type="xs:string" />
<xs:element name="CompanyName" type="xs:string" />
<xs:element name="ContactName" type="xs:string"
```

When you combine several elements in a complex type, such as tables within a DataSet or columns within a table, you can restrict them to a specific order or let them appear in any order. If the order of a set of elements must match a defined order, then it's an `xs:sequence`. Only one element of a set of child elements defined using `xs:choice` can normally appear in a document. However, by adding the `maxOccurs="unbounded"` to the element, it may occur as many times as needed. Thus, tables within a DataSet can appear in any order, and can appear only when needed. The tables of a DataSet are defined using `xs:choice` because they can occur in any order. The columns within a table are defined using `xs:sequence` because if a column occurs, it must be in a specific sequence:

```
            </xs:sequence>
         </xs:complexType>
      </xs:element>
      <xs:element name="Orders">
         <xs:complexType>
            <xs:sequence>
               <xs:element name="OrderID" msdata:ReadOnly="true"
                           msdata:AutoIncrement="true" type="xs:int" />
               <xs:element name="CustomerID" type="xs:string"
                           minOccurs="0" />
               <xs:element name="EmployeeID" type="xs:int" minOccurs="0" />
               <xs:element name="OrderDate" type="xs:dateTime"
                           minOccurs="0" />
            </xs:sequence>
         </xs:complexType>
      </xs:element>
   </xs:choice>
</xs:complexType>
```

Nullable columns have the `minOccurs="0"` attribute. This specifically means that having no occurrences of the element is legal. XML handles nulls by leaving out the entry. This is a little surprising if you're expecting all of the elements within a logical XML record to appear. It can also cause issues during inference.

This XSD contains three constraints. Two are primary key constraints, one on each table in the DataSet. These primary key constraints just define the table and the primary key columns:

```
<xs:unique name="DataSet1Key1" msdata:PrimaryKey="true">
   <xs:selector xpath=".//mstns:Customers" />
   <xs:field xpath="mstns:CustomerID" />
</xs:unique>
<xs:unique name="DataSet1Key2" msdata:PrimaryKey="true">
   <xs:selector xpath=".//mstns:Orders" />
   <xs:field xpath="mstns:OrderID" />
</xs:unique>
```

A referential constraint defines the relation between two tables. XSD describes this relationship by referencing the existing primary key constraint for the parent table, such as `DataSet1Key1`. This is the constraint for the parent key. The child table of the relation is defined by its table and column, both stated in terms of an XPath statement:

```
      <xs:keyref name="CustomersOrders" refer=" DataSet1Key1">
         <xs:selector xpath=".//mstns:Orders" />
         <xs:field xpath="mstns:CustomerID" />
      </xs:keyref>
   </xs:element>
</xs:schema>
```

Although there are some additional details, that's the core of understanding XSDs. XSD is an ugly, messy format that's both verbose and effective. It's designed for maximum flexibility across a lot of different kind of data. As the complexity of the data structure increases, obviously the complexity of the XSD increases, and it becomes even more difficult to read.

Working with SQL-92 Databases (SQL Server)

Databases might be the only metadata source you ever use. Everything else may be optional. Most applications are data centric, and most modern relational databases hold a phenomenal amount of information in their schemas. Many of the backends, including SQL Server and Oracle, provide rich information using features corresponding to the SQL-92 standard.

Selecting a Standard

SQL-92 is the most mature in a long series of standards that have been used to define relational databases. It's supported by SQL Server, Oracle, and several other databases.

Most databases go beyond the standards to provide their own extensions. Auto-incremented columns aren't SQL-92 compliant—they're extensions in any backend that supports them. Starting with SQL Server 2000, Microsoft provides extended properties that allow you to put name-value pairs on any object including tables, columns, stored procedures, stored procedure parameters, and so on. This allows you to extend metadata within the database. You can use this to add captions, descriptions for ToolTips, lookup information, and anything else you can imagine. You can add extended properties via Transact SQL (TSQL) statements, by right-clicking the object in Query Analyzer, or by creating your own tool to edit them. Extended properties aren't part of the SQL-92 specification.

NOTE There's another standard you might be interested in if you're doing data warehousing. If you know you need information from the Open Information Model (OIM) standard, then you can write code similar to the SQL-92 extraction tool to extract that metadata. If you don't know about the OIM standard, the chances are excellent that the simpler approach of SQL-92 with database-specific extensions will provide all the metadata you need.

Working with the Metadata Extraction Tool

The metadata extraction tool that you can download from the Web site extracts a ton of information about your database. This section discusses a little theory behind the tool and what it accomplishes. Appendix B walks through important parts of this tool.

TIP The metadata extraction tool at the Web site accesses only one database at a time. To work with multiple databases, you can extract data separately and combine it using the merge techniques discussed later in this chapter.

Understanding the Target Structure

Before looking at how the extraction tool works, it'll help to know what the target XML file looks like. This is the target for both SQL-92 extraction and the XSDToFriendlyXML transformation. The opening of Northwind reveals familiar element names such as DataStructure and Table, rather than the obtuse element names of XSD. Column elements are called *TableColumn* because there are both table and views that contain columns, and that needs to be differentiated:

```
<?xml version="1.0" encoding="UTF-8"?>
<dbs:DataStructures xmlns:dbs="http://kadgen/DatabaseStructure">
  <dbs:DataStructure Name="Northwind">
    <dbs:Tables>
     <dbs:Table Name="Orders"
                OriginalName="Orders"
                SingularName="Order"
                PluralName="Orders">
        <dbs:TableColumns>
          <dbs:TableColumn Name="OrderID"
                      OriginalName="OrderID"
                      Ordinal="1" Default=""
                      AllowNulls="False"
                      SQLType="int"
                      NETType="System.Int32"
                      MaxLength=""
                      IsAutoIncrement="true">
<additional tables clipped for brevity>
```

To help sort out naming, the extraction process provides four names for each table. Table 2-2 explains the source and use of each name. The `Name` and `OriginalName` attributes are the same, unless there are spaces in the original name. For reasons that seem mysterious outside Microsoft, names of database elements can contain spaces. I recommend you avoid names containing spaces because they can cause problems all over the place. Metadata extraction replaces any spaces in the original name with underscores to create the Name attribute. Many table names based on English language words can be automatically converted to singular and plural forms. It makes no difference to this process whether the original table names are singular or plural, but it's rather maddening for humans to work with databases intermixing singulars and plural table names, and the ORM process in Chapter 6 is a little easier if you use singular table names.

CAUTION The logic used for making singular and plural forms of tables names won't work for all table names and won't work with names not based on the English language.

Table 2-2. The Use of Each of the Name Attributes Provided for Tables and Columns

ATTRIBUTE	DESCRIPTION	USAGE	PROVIDED FOR COLUMNS?	PROVIDED FOR TABLES?
`Name`	The name of the element as used in the database, with spaces replaced with underscores and keywords prefaced with an underscore. Plurality isn't changed.	The primary identifier for the element.	Yes	Yes
`OriginalName`	The name of the element used in the database.	Provided for reference and used when generating stored procedures.	Yes	Yes
`Singular`	The Name attribute modified to make it singular if it isn't already singular,	Name of row classes; prefix for collection classes that end with *Collection*.	Yes	No
`Plural`	The Name attribute modified to make it plural if it isn't already plural.	Name of variables pointing to collection classes.	Yes	No

NOTE This and other aspects of the samples and example architectures throughout this book assume you don't have both singular and plural forms as separate database tables (for example, `Customer` and `Customers` tables in the same database). If you do, you'll have to adjust. You can either modify the extraction tool or supply modified singular and plural names via freeform metadata (discussed later in this chapter).

The `TableColumn` element contains information retrieved from SQL Server. The NETType attribute contains a string translated from the SQL type.

Each column has privileges defined in the database. If you include these privileges in your metadata, you can fine-tune your user interface behavior via your templates. Each column may also have check constraints, but the specific column schema shown here doesn't have a check constraint:

```
    <dbs:TableColumnPrivileges>
      <dbs:TableColumnPrivilege Grantor="dbo" Grantee="public"
              Type="REFERENCES" />
      <dbs:TableColumnPrivilege Grantor="dbo" Grantee="public"
              Type="SELECT" />
      <dbs:TableColumnPrivilege Grantor="dbo" Grantee="public"
              Type="UPDATE" />
    </dbs:TableColumnPrivileges>
    <dbs:CheckConstraints />
  </dbs:TableColumn>
</dbs: TableColumns >
```

Like columns, tables have privileges defined in the database. This information is included in the XML as follows:

```
<dbs:TablePrivileges>
  <dbs:TablePrivilege Grantor="dbo" Grantee="public" Type="REFERENCES" />
  <dbs:TablePrivilege Grantor="dbo" Grantee="public" Type="SELECT" />
  <dbs:TablePrivilege Grantor="dbo" Grantee="public" Type="INSERT" />
  <dbs:TablePrivilege Grantor="dbo" Grantee="public" Type="DELETE" />
  <dbs:TablePrivilege Grantor="dbo" Grantee="public" Type="UPDATE" />
</dbs:TablePrivileges>
```

One of the most important things in the XML metadata is information about table constraints. This includes primary keys and table relations. The primary key information contains a list of primary keys. Generally each table has a single primary key, but it may have a compound key made up of several columns so it's a child element, rather than an attribute:

```
<dbs:TableConstraints>
  <dbs:PrimaryKey>
    <dbs:PKField Name="OrderID" Ordinal="1" />
  </dbs:PrimaryKey>
```

Table elements in the metadata file contain both parent and child relations. This means that each relation is described both from the perspective of the parent and from the child. This allows templates using the metadata to access relation information easily and helps avoid convoluted XPath statements. The `Orders` table has one child and three parent tables, each of which uses one key field. The child table element doesn't need to list the parent key fields because it's immediately available as the primary key of the current table. Multiple key fields would result in multiple `ParentKeyField` and `ChildField` elements:

```
    <dbs:TableRelations>
      <dbs:ChildTable Name="Order_Details">
        <dbs:ChildKeyFields Name="OrderID" Ordinal="1" />
      </dbs:ChildTable>
      <dbs:ParentRelations>
        <dbs:ParentRelation ParentTable="Customers">
          <dbs:ParentKeyField Name="CustomerID" Ordinal="1" />
          <dbs:ChildField Name="CustomerID" Ordinal="1" />
        </dbs:ParentRelation>
        <dbs:ParentRelation ParentTable="Employees">
          <dbs:ParentKeyField Name="EmployeeID" Ordinal="1" />
          <dbs:ChildField Name="EmployeeID" Ordinal="1" />
        </dbs:ParentRelation>
        <dbs:ParentRelation ParentTable="Shippers">
          <dbs:ParentKeyField Name="ShipperID" Ordinal="1" />
          <dbs:ChildField Name="ShipVia" Ordinal="1" />
        </dbs:ParentRelation>
      </dbs:ParentRelations>
    </dbs:TableRelations>
  </dbs:TableConstraints>
  <dbs:ExtendedProperties />
</dbs:Table>
```

The remainder of the table elements repeat this sequence. The XML document contains similar sections for views, stored procedures, and so on. By the time the metadata encompasses all of the information in the database, the metadata files can become fairly large. The metadata file for Northwind, which isn't a particularly complex database, is about 200KB.

Understanding the Tool Architecture

The SQL-92 standard features provide most of the metadata you'll want and work with a number of database backends. Each backend provides additional metadata opportunities. That makes a base/derived class pattern a good choice for the metadata extraction tool. A base class named `ExtractMetadata` gives access to the SQL-92 standard features. The `SQLExtractMetadata` class derives from this and provides Microsoft SQL Server–specific features. Figure 2-3 illustrates this relationship. The Oracle box is gray because I haven't supplied an Oracle class.

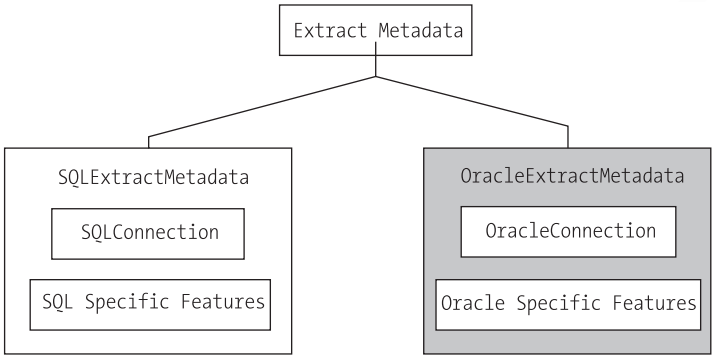


Figure 2-3. Inheritance pattern for metadata extraction classes

The derived class provides the housekeeping duties of accessing the backend as well as backend provider-specific extensions. Housekeeping duties include connection management and accessing the data through a DataAdapter. Because these tasks are handled by the derived classes, the base class can't stand on its own so is defined as an abstract (`MustInherit`) class. You'll need to build a derived class to at least perform this housekeeping functionality for any other backend database you use. You may want to add other features specific to your backend, similar to the way extended properties are added in the SQL Server derived class.

Because I don't know what you're going to do with derived classes you may create, I maximize the flexibility by making everything in the base class `Overridable` (virtual). I don't suggest you do this to most classes. In general, you want `Overridable` to indicate the ways you anticipate derived classes using your base class. In this case, I cheat and make everything overridable.

NOTE You do incur a small performance penalty every time you access an `Overridable` member. This performance hit is inconsequential when you need to allow the member to be overridden. However, doing this for members that you don't expect to be overridden just wastes processing cycles.

TIP It doesn't matter if you waste processing cycles during code generation.

Understanding Security

When you're creating metadata, you need permission to read from the system tables. The easiest way to do this is to create one or more logins for code generation that are different from the logins you'll use at runtime. From the programmer's perspective, the simplest approach is Windows authentication. The downside of this approach is that programmers may wind up testing the application with different rights than end users will have. One solution is to have multiple logins and use the Windows feature Run As when running code generation. Another solution is to provide a login to the code generation and use username/password authentication.

The tools on the Web site use Windows authentication because that's the most appropriate approach for code I'm asking you to run against your server. You can change that in the connection string in the `SQLExtractMetadata` class. This connection string is in the class, rather than App.config, because it's concatenated at runtime with the server and database provided by the harness. To run the samples, you'll need to have access to Northwind via a Windows-authenticated account or make changes to the connection string and provide a user login screen.

TIP Differentiate between permissions for code generation and application testing, especially if your code generation writes to your database. Use separate accounts for code generation and application testing.

Code generation may also need to make changes to your databases. You might do this either because you're creating stored procedures or because you're creating and running scripts to build tables and/or import data. In these cases, you'll need additional database rights. You'll need to fine-tune your code generation login to have the required permissions.

TIP You can mess up working with code generation in basically the same ways you can mess up in Enterprise Manager or Query Analyzer. You can run any script using the harness introduced in Chapter 3, including ones that overwrite stored procedures, add tables, drop tables, and so on. Just as you can "toast" your database with manual tools, you can toast it (faster and more efficiently) with code generation, so you'll want to use backups, test databases, scripts, caution, and other good habits.

Working with Information Schema Views

SQL-92 databases, such as SQL Server and Oracle, provide information schema views as a standard way to provide information about database schemas. These views are normalized, meaning that there are many of them. Information schema views are the best way to access structural information about your database. Accessing system tables directly isn't a good idea because Microsoft might change the layout. The system stored procedure `sp_help` provides most of the information, but it's more difficult to work with and it's specific to SQL Server. So, the best approach is to use the SQL-92 information schema views.

Understanding SQL-92 Terminology

You'll find several features in information schemas named in ways quite different from common SQL Server terminology. Table 2-3 shows the information schema and the corresponding SQL Server terms.

Table 2-3. Translating Terminology Between Information Schema and SQL Server Terminology

INFORMATION_SCHEMA	SQL SERVER
Catalog	Database
Schema	Owner
Routine	Stored procedure or function
Domain	User-defined type
RowVersion	Timestamp
Timestamp	DateTime data type

I'll stick with SQL Server terminology because it sounds more familiar. The tool that extracts metadata from SQL-92 databases uses the underlying SQL-92 based table and column names to access the data, so you'll need to be able to translate between the naming if you work with the internals of this tool.

Object names for tables, views, user-defined types, stored procedures, and so on are all defined via a three-part name in information schema views. This name consists of the catalog or database name, schema or owner name, and the object name itself. Most of you are likely to target one database and the dbo in your applications, so I'll stick with simple object names. If you're targeting multiple databases or multiple owners, you may need to make changes to the metadata extraction classes to avoid potentially ambiguous naming across different scopes.

Introducing the Specific Schemas

Information schemas are views, not stored procedures. This means you select from them rather than call them. Also, unlike most things you encounter in SQL Server, the information schema views have to be fully qualified, such as this:

```
SELECT * FROM INFORMATION_SCHEMA.SCHEMATA
```

SCHEMATA is one of the 20 or so information schema views provided in the SQL-92 standard. The SCHEMATA view returns the databases available to the user within the server instance. Table 2-4 summarizes what information each view provides. For the full information on these views and what each returns, refer to the "INFORMATION_SCHEMA" section in the TSQL help you can access though Query Analyzer (or MSDN).

Table 2-4. Individual Information Schema View Contents

VIEW	DESCRIPTION	COMMENTS
SCHEMATA	Contains a row for each database in which the user has permissions	
TABLES	Contains a row for each table	
TABLE_PRIVILEGES	Contains a row for each table privilege	
TABLE_CONSTRAINTS	Contains a row for each table constraint	See the section "Using Constraints" for a discussion of how constraints work.

Table 2-4. Individual Information Schema View Contents, continued

VIEW	DESCRIPTION	COMMENTS
CONSTRAINT_TABLE_USAGE	Similar to TABLE_CONSTRAINTS with less information	Not used in tool.
COLUMNS	Contains a row for each column	
COLUMN_PRIVILEGES	Contains a row for each column privilege	
CONSTRAINT_COLUMN_USAGE	Contains a row for each column constraint	
COLUMN_DOMAIN_USAGE	Contains a row for each column that's defined as a user-defined type. This information is also contained in COLUMNS	Not used in this metadata extraction.
ROUTINES	Contains a row for each stored procedure or function	
PARAMETERS	Contains a row for each stored procedure or function parameter	
ROUTINE_COLUMNS	Contains a row for each column returned from a table value function	Don't get excited—this just applies to table value functions. I'll show you later how to retrieve stored procedure recordset structures.
CHECK_CONSTRAINTS	Contains a row for each accessible check constraint	
REFERENTIAL_CONSTRAINTS	Contains a row for each accessible referential constraint	
KEY_COLUMN_USAGE	Contains a row for each column used in an accessible key	

Table 2-4. Individual Information Schema View Contents, continued

VIEW	DESCRIPTION	COMMENTS
VIEWS	One row for each accessible view	
VIEW_TABLE_USAGE	Contains a row for each accessible table used in a view	Not used in tool.
VIEW_COLUMN_USAGE	Contains a row for each column used in each view	Not used in tool.
DOMAINS	Contains a row for each accessible user defined type	
DOMAIN_CONSTRAINTS	Contains a row for each constraint on an accessible user-defined type	

In the SQL-92 data extraction tool, I include features you're likely to use and don't attempt to incorporate everything from the schema views. For example, I ignore aspects such as character set, collations, and so on. If you're using these features, you'll want to extend the tool with a derived class or change the source code.

Using Constraints

Many of the information schema views are self-explanatory. However, the constraints are a bit confusing. Although column constraints are limited to check constraints, table-level constraints can be check, unique, primary key, or foreign key constraints. In all cases, the TABLE_CONSTRAINT view contains a row with the full name of a constraint in the Catalog/Schema/Name format and the constraint type. The constraint's type indicates where to go looking for more information.

Retrieving Check Constraints

Check constraints are the easiest type of constraint to retrieve. The full name of the constraint in the TABLE_CONSTRAINT view points to an entry in the CHECK_CONSTRAINT view. This view supplies the check clause as text. That's simple enough (see Figure 2-4, A).

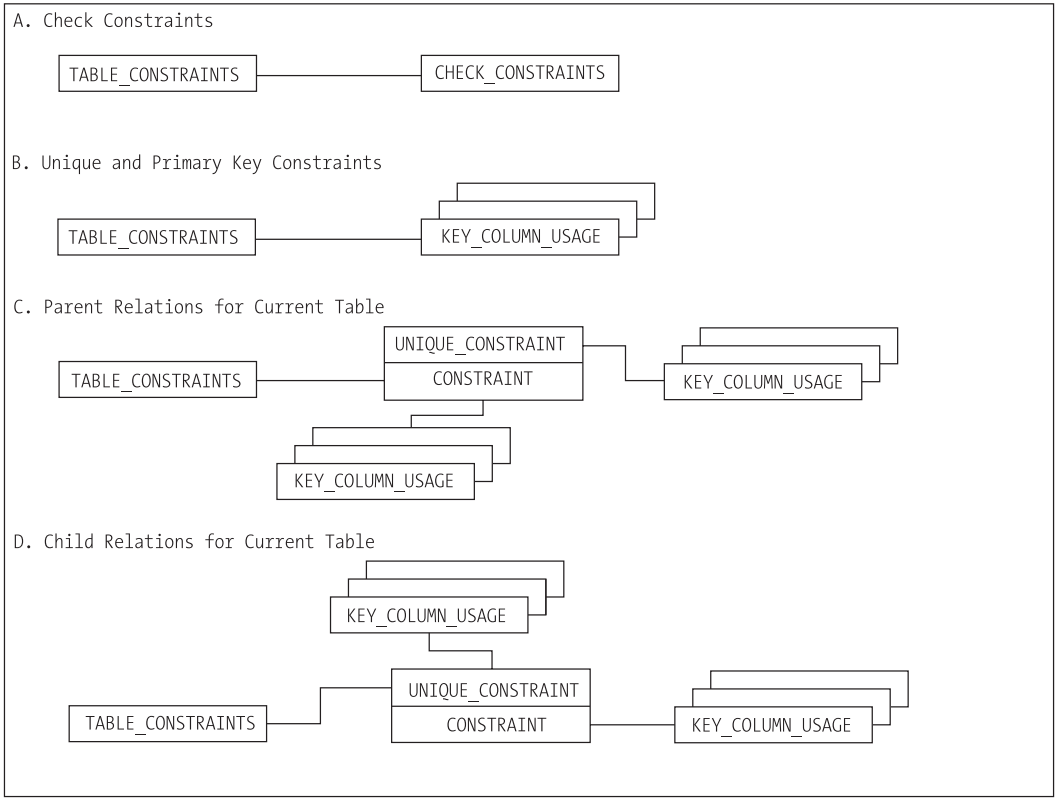


Figure 2-4. Schematics of how several information schema views are combined to retrieve information on constraints

Once your metadata includes the information from the check constraint of the SQL Server column, you can pass the constraint information out to a user interface tier to build its validation (such as a Web page that validates using JavaScript), to build middle tier validation, and/or build intelligent messages for your users on validation failures and other slick things.

Because you're using the constraint in a different language than the TSQL that defines the constraint, you'll have to parse to clean up the syntax. Dragons lurk in these waters. At the simplest, you'll have to do some parsing to remove square brackets and adjust naming in a consistent manner, including replacing keywords and any spaces that snuck into your column names. This won't be sufficient where SQL functions are used because .NET won't understand these

functions. It also won't be enough if your properties don't map directly to columns or the check is more complex. Problems generally get deeper as you look at UI-tier code where you also need to reference a middle-tier object within the constraint. It's easy enough for you to see the intent in the following check constraint, but how do you turn this into UI validation code?

```
<dbs:CheckConstraint Clause="([BirthDate] &lt; getdate())" />
```

It's actually a two-headed dragon that breathes fire from these depths. Attempting to handle all possible constraints eats too much development time. Handling some, but not all constraints, can result in lost constraints, which is very, very bad. It's especially bad because missing constraints are hard to catch in testing. Invalid constraints that raise compile time errors are just fine—good in fact. You still have a dragon, but he has such a bad case of the hiccups he can't sneak up on you.

Looking ahead to what generation actually does may clarify how you work with check constraints and the types of problems you're likely to encounter. In the middle tier, you can just insert the text of the constraint after the square bracket, and name parsing is complete. This results in the following:

```
If Not (BirthDate < getdate()) Then
   ' Respond to Validation Failure
```

This assumes `BirthDate` is a property of the current object, which is likely because you are performing this validation within a middle-tier object. The compiler raises an error on `getdate()` because .NET doesn't include this function, forcing you to add this method to your class. It's easy enough to include a standard block of wrapper functions corresponding to common SQL functions in all of your generated classes. Alternatively, you can transpose these functions into framework functions during metadata extraction, or you can prefix all SQL functions with a specific class name such as `SQLUtility.getdate()` and add the methods to this class.

Compiler errors point you to the specific types of constraint validation used in your application. Be cautious to ensure that you never lose a constraint, even if you get something weird in the constraint. If something goes wrong, either output the original string (which you anticipate raising a compiler error) or something that'll explicitly raise an exception such as the string `Error Parsing Constraint`.

TIP Endeavor to parse constraints such that they're fully complete and usable, or they'll cause a compiler error.

Retrieving Primary Key and Unique Key Constraints

It's only a little more convoluted to extract the unique and primary key constraints. Here, the constraint full name in the TABLE_CONSTRAINT view corresponds to one or more rows in the KEY_COLUMN_USAGE table, as shown in Figure 2-4, B. The KEY_COLUMN_USAGE table includes the full constraint name, the full table name (that you already had), and the column name and ordinal position, which are the two new pieces of information you need. There may be multiple columns as part of the primary key, and thus multiple rows in the KEY_COLUMN_USAGE view are possible. The KEY_COLUMN_USAGE view contains the columns for all the constraint types that use key columns, including parent and child constraints as well as primary key and unique constraints.

Retrieving Parent Relations

Relations between tables are called *referential constraints* and consist of a parent and child table. Child tables are also called *foreign key tables,* and parent tables are also called *primary key tables.* From the point of view of any single table, it can have any combination of parent and child relations.

The REFERENTIAL_CONSTRAINTS information schema view defines referential constraints. Each row in this view contains the full name to a parent table constraint and the full name of a child table constraint. The parent table is referred to in the view as the UNIQUE_CONSTRAINT and the child table simply as CONSTRAINT. You can retrieve both parent and child tables for any table from this view. In metadata, it's best to include the relation definition as part of both the parent and child tables' descriptions.

The foreign key relations of the TABLE_CONSTRAINT references the CONSTRAINT in the REFERENTIAL_CONSTRAINTS view. Effectively the current table is the child or foreign key table in the relation. The corresponding record in the REFERENTIAL_CONSTRAINTS view provides the full name of the parent constraint and table from the UNIQUE_CONSTRAINT fields of the view. With the full name of the parent constraint, you can grab the columns from KEY_COLUMN_USAGE in a manner similar to the primary and unique keys. This somewhat convoluted approach is illustrated in Figure 2-4, C.

TIP It's worth the trouble to add both parent and child relations to each table element in your metadata because it avoids convoluted XPath in your code templates. Convoluted XPath makes your code templates far less maintainable and far more sensitive to changes in metadata layout.

Child relations are all those relations with the current table listed as the parent table or UNIQUE_CONSTRAINT. To retrieve information on this constraint, you can search the REFERENTIAL_CONSTRAINTS view for all records with the current table as the UNIQUE_CONSTRAINT. Using the corresponding CONSTRAINT records, you can retrieve the table name and retrieve the keys for the child tables from KEY_COLUMN_USAGE. Figure 2-4, D, illustrates this relationship.

Retrieving Child Relations

Grabbing child relations is similar.

Modifying Mappings

Sometimes the information in the database schema isn't exactly the information you want to use. Maybe you need to use a different column name or a narrower type in the user interface. There are several ways to provide mapping. You can programmatically modify the structure, or you can use freeform data to override existing attribute values and rely on the merge mechanisms allowing later XML files to override data in earlier files. Chapter 6 shows a simplified ORM solution targeted at indirect mappings. If you're just changing a caption or type, overriding data with freeform metadata may be sufficient.

Extracting Stored Procedures Schemas

You can extract the stored procedure and parameter information from the ROUTINES and PARAMETERS information schema views. What's much harder to figure out is the intent of each stored procedure and what recordsets the stored procedure returns. The returned recordsets are important because these are often used to create business objects. Stored procedures are preferable to creating SQL statements as strings in your .NET code—also called *dynamic* or *inline SQL*. Inline SQL has significant problems, including security vulnerabilities, nonoptimal performance, and limited opportunities for reuse.

Determining Stored Procedure Intent

Knowing the intent of each stored procedure tells you how you'll use it in code generation. Will it define a business object because it's a select stored procedure? Will it be part of an updating strategy? Should it be totally ignored during code generation? You probably have some combination of nine kinds of stored procedures in your database as follows:

- Simple select stored procedures that retrieve a single root record
- Set-based select stored procedures to retrieve multiple records
- Insert stored procedures
- Update stored procedures
- Delete stored procedures
- Process stored procedures
- Administrative stored procedures
- Reporting stored procedures (select stored procedures used differently)
- Miscellaneous stored procedures

Although there isn't any difference in how you call them, I think it's helpful to differentiate between simple select procedures that return a single record and procedures that return multiple records. Note that this distinction is on the number of root records returned, not on the number of recordsets. Simple select stored procedures will often return associated child collections. Depending on the amount of preprocessing that has to be done, you can sometimes gain considerable performance benefits returning these multiple recordsets together. I'll differentiate these two types of stored procedures by calling them *select* and *setselect* stored procedures.

You're probably familiar with retrieve, insert, update, and delete stored procedures. You may also have process stored procedures that make changes to the database or retrieve scattered information that can't easily be considered a select and don't map to underlying tables. Most databases have some administrative stored procedures that are used for importing data, updating structures such as maintaining an internal calendar, and so on. Code generation can generally ignore these utility and internal stored procedures. Reporting stored procedures aren't much different from select stored procedures, except that they're more

likely to be denormalized and will almost always be read-only on the client. Few databases survive without a few stored procedures that are unique and strange to the history of that database. These are sometimes temporary or intended as temporary, and no one was quite sure they could delete them.

TIP Use your initials as a prefix for temporary stored procedures to clarify who is responsible for either deleting them or marking them as a permanent part of your database.

To generate code effectively, you have to know into which category each stored procedure falls. You can do this with either extended properties or an approach called *careful naming* discussed later in this chapter. Careful naming formalizes the stored procedure's purpose within its name, allowing you to grab it during metadata extraction. You might be able to parse the body of simple stored procedures to guess what the stored procedure does, but I advise against that approach because it gets too ugly too quickly with nontrivial stored procedures.

Retrieving Stored Procedure Recordsets

It's tricky to retrieve the structure of recordsets returned by stored procedures. It's not available from information schemas or from the system tables. To retrieve this information, you have to actually run the stored procedure and look at the output. If you look at Microsoft SQL Profiler, you'll see that Visual Studio uses this approach to come up with the return values for Server Explorer.

SQL Server includes a `SET FMTONLY ON` switch, instructing it to return empty recordsets with no data. This works in most cases, primarily failing if you have temporary tables and a few other scenarios. It's wise to also set the database to read-only prior to calling the stored procedure to minimize side effects from running these stored procedures or start and roll back a transaction.

The remaining problem is what parameters you're going to use when calling the stored procedure. In the SQL-92 metadata extraction tool, I solve this by passing default values for the type, such as zero from numeric values. For this to work, obviously your procedure can't blow up when default values are passed. I had one DBA include a consistently named optional parameter in all stored procedures that indicated whether I was extracting metadata. This Boolean parameter worked well, only requiring that one way or another the stored procedure return empty recordsets matching the runtime structure. The DBA could figure out internally the best way to do this. A similar approach would be to use a null value for one or more key parameters to indicate you're in metadata

extraction mode. If you can't use these approaches, you'll have to work out a scheme for storing the appropriate parameter values. You could store these in a separate file, store them in extended properties, or parse them from the opening comments of the stored procedure itself. Regardless of how you do it, this will be hard to maintain, so limit the number of parameters you need to assign values for code generation where possible.

Some stored procedures vary the structure of the recordsets they create based on parameter values. From a code generation point of view, not having predictable columns is disastrous. My first choice for a solution is to rewrite these stored procedures, perhaps having the stored procedures return empty data rather than omit columns when there are no values. My second choice is to create wrapper stored procedures for all the different variations. This way your code generation can access only the wrapper stored procedures, each of which returns consistent recordsets. If neither of these solutions works and your stored procedures need to return a variable structure, you'll have to determine how that variability will affect your business objects.

In some cases, such as pivot tables, the recordsets produced by stored procedures is inherently variable. These stored procedures aren't going to provide you with meaningful recordset metadata you can use to generate business objects. Your goal will be just to avoid misleading objects being built and develop other strategies, such as untyped DataSets for this kind of data.

Understanding ADO.NET Structures and Logical Tables

Running the stored procedures results in a set of ADO.NET DataTables. If you run the stored procedure with the `MissingSchemaAction` property set to `AddWithKey`, you'll get a recordset that includes the column names, sizes, and types. You can access this information through normal ADO.NET techniques.

That's nice information, but these returned columns almost always map to an underlying table and column. Previous sections showed how to extract even richer information from the underlying columns. To connect this information with the stored procedure recordset, you'll have to determine the mapping between the stored procedure and its underlying tables.

Each returned recordsets is a *logical table*. Logical tables often map directly to physical database tables. They won't map directly if the logical table consists of joined tables.

Inferring the Mappings

The mapping between the logical and physical tables can be explicit through extended properties or freeform data. But that's often not needed because the mappings can be inferred. These inference rules will almost always get you the right column. When they don't, you can override with manual mapping in

freeform metadata. As part of evaluating your database for metadata collection, evaluate where you have potentially ambiguous fields.

Selecting stored procedures that return a single recordset mapped to a single table aren't too hard. Because I love well-organized databases with names I can predict, I like the careful naming approach discussed later in the "Introducing Careful Naming" section to determine the physical table for selects that return a single recordset. For example, the stored procedure that retrieves data from the `Customer` table might be named *acc_Customer_Select*. Alternatively, you can use extended properties or freeform XML. You can apply extended properties to stored procedures and their parameters but not to returned recordsets or columns.

Things get a lot trickier when the stored procedure returns multiple recordsets or a recordset represents a table join. In the first case, you don't know much about each recordset, and in the second case, individual columns in the recordset are mapped to different tables. To map multiple recordsets to their underlying tables, you can extend careful naming to columns by having a column name that's the name of the table followed by *ID*, and using that column name to determine the physical table. This is probably how you already name your primary keys, so it's an easy rule to implement; however, you also have to place this column in a predictable location, usually the first occurrence of a field ending with *ID*. You'll also have to ensure that the recordset column names are the same as the underlying table's column names. This solution works if there's a single table mapped to each stored procedure recordset. Alternatively, you could store a comma-delimited list of table names as a stored procedure extended property.

When a recordset is comprised of joined underlying recordsets, you can gather a set of tentative recordsets by grabbing any column whose name is a primary key to another table. You can then walk these tables looking for each column name. Another approach is to prefix all of the column names with the corresponding table name and an underscore. This is the least ambiguous approach. It can be tedious to maintain by hand, but it's a breeze if you're generating the stored procedures. Yank these prefixing table names off the column name before exposing the data in a business object, which is also easy to do with code generation. Regardless of your overall approach, you need to use the concatenated table name approach when you have two columns that have the same name in different underlying tables with different metadata.

Working with SQL Server–Specific Metadata

Most of the information in your SQL Server database structure is available through the SQL-92 information schemas. Some of the information, specifically the auto-incrementing identity columns and extended properties, are SQL Server–specific extensions and not included in the SQL-92 standard or the information schemas. In the metadata extraction tool, these values are retrieved in the derived `SQLExtractMetadata` class.

Retrieving Identity Columns

Identity columns are SQL Server columns that auto-increment to ensure that each value is unique. These are often used to guarantee a unique primary key for tables. The type of each identity column is one of the integer types. Each identity column also has a start or seed value and an increment value. These default to `int`, `1`, and `1`. This metadata extraction tool just extracts which columns are identity columns from the type name returned by `sp_help` for the column.

Retrieving Extended Properties

The SQL function that returns extended properties can be a bit nonintuitive to use. Ultimately, it's giving you a recordset of name-value pairs. To get these for anything in the database, it uses a rather convoluted concept of levels. SQL Server refers to these three levels as *level0*, *level1,* and *level2* objects, but it's easier to think of them as *owner*, *object*, and *item*. Objects can be tables, stored procedures, and so on, and items can be columns, parameters, and so on. For each level there's a type and a name. You have to specify both the type and the name for levels above what you're requesting. For the level you're requesting, you can just specify the type to retrieve all extended properties of that type. If the first parameter is specified, it indicates the specific property to be returned. Thus, you'd retrieve the `Caption` parameter of the `CompanyName` column using the following:

```
SELECT * FROM ::fn_listextendedproperty('Caption', 'User', 'dbo', 'Table',
                'Customers', 'Column', 'CompanyName')
```

As an example of retrieving higher-level data based on partial parameters, the following returns all of the extended properties for tables in the database:

```
SELECT * FROM ::fn_listextendedproperty(null, 'User', 'dbo', 'Table', null,
                                        null, null)
```

Note that I specified the user as `'dbo'` at the highest level, but only the type at the current level (`'Table'`).

Introducing Careful Naming

I settled on the phrase *careful naming* for this section because of my lack of comfort calling it *anal naming*, but it means about the same thing. Names within your database can carry the information you need to build metadata if you do the following:

- Accept the premise that a name carries meaning.
- Allow that meaning to be overloaded with detailed rules.
- Commit to being very careful, almost fanatical, in your naming.

NOTE Careful naming is most important for stored procedures.

Do your stored procedures already have names that indicate what they're doing and what they're doing it to? Most databases I've seen are awfully close to careful naming, except for two important aspects. They don't use explicit clause delimiters, and they always seem to include a handful of exceptions. These exceptions really mess things up, and careful naming is really a commitment to follow a set of rules absolutely. It's a 100 percent thing. If you rely on it for determining intent during metadata extraction and then do something in an unexpected way, you'll generally have an extra or missing object, method, or property.

If careful naming doesn't sound like a good fit for your workgroup, another option is to store the task and identify information for each stored procedure in extended properties of the stored procedures or external freeform XML that can be merged with your metadata. The benefit of careful naming is that everyone can see the intent quickly and find what they want even after stored procedures proliferate in your database. Chapter 7 shows how to use code generation to build stored procedures, and this makes it easy to follow careful naming rules. However, your DBA may already be quite good at making and following detailed rules.

Implementing Naming Delimiters

If you change your stored procedure naming to a careful naming scheme, the most important feature you'll introduce is delimiters. Humans do very well with Pascal or camel casing because we gather meaning even when rules are broken.

Capitalization is a hard way to delimit meaning to a computer because humans are imperfect in applying it. You have no problem understanding the intent of these samples:

```
nwdCustomersSelect ' nonambiguous
nwdOrderDetailsSelectSpecialCase ' potentially ambiguous
```

versus the intent of these:

```
nwd_Customers_Select ' nonambiguous
nwd_OrderDetails_Select_SpecialCase ' never ambiguous
```

If you really hate underscores, you can work with casing. You'll struggle with ensuring clarity to both the computer and humans, especially with multipart names such as *OrderDetails*. You'll also need to avoid table names that contain your selected keywords such as *Insert*, *Select*, *Update*, *Delete*, or *Process*.

Implementing Naming Rules

The specific naming rules you use aren't essential. They could be different from mine, as long as you can articulate and follow them. The key things you're looking for are the intent (category) of the stored procedure, the subset of your database where it's applicable if you have a large database, and specifics that allow you to make a meaningful unique name for your middle-tier classes.

The naming rules I use for stored procedures consist of four portions, or clauses separated by underscores. Careful naming relies on being able to nonambiguously separate the clauses of the name—a task made easy by the underscore delimiters:

```
< subset abbrev >_< logical table name >_< type >_< specifier >
```

This format breaks down into clauses as follows:

subset abbrev: Designates major subsets of the database functionality, such as *acc* for accounting. These abbreviations should be three or possibly four characters. You can use a prefix such as *utl* (for utility) or *prv* (for private) to indicate those stored procedures that shouldn't be exposed in metadata. This is a common way to organize categories of stored procedures, and you can use this information to organize your business objects into multiple assemblies if desired.

logical table name: Set of columns that generally map to a single business class and one or more physical database tables. You can also think of a logical table as any unique combination of columns. A single physical table might map to several logical tables that each containing a different subset of columns. You'll also use different logical tables when you have joins producing different recordset structures. Each logical table needs a unique name—the table name of the physical table returned when there's a one-to-one mapping.

type: Type of the stored procedure. This will be Select, SetSelect, Insert, Update, Delete, Process, Report, or Internal. Report stored procedures are generally Select stored procedures but are used differently (or not used) in metadata.

specifier: Additional information that also makes the stored procedure name unique. Depending on how you build your stored procedures and how complex your database is, there's a good chance the other rules won't be sufficient to generate unique naming. For example, you'll frequently have specific variations of select stored procedures with different search criteria such as this:

```
nwd_ Customers_Select_ByAccountNumber
```

or this:

```
nwd_ Customers_Select_ByNameLastFirst
```

This last section of a specifier is optional and will probably appear in some, but not all, of your stored procedure names.

If you like to suffix your stored procedures with a consistent suffix such as *_sp*, you can also do that. The metadata extraction tools ignore this suffix.

If you put these rules together, the intent of each of the stored procedures in your database will be clear, and you can avoid the additional hassle of maintaining information about the intent of your stored procedures. If you aren't doing this now and have a significant database with stored procedures calling other stored procedures or other applications using your database, changing your stored procedure naming is a nearly impossible job. In that case, you probably want to maintain this information in freeform data or extended properties.

Creating Freeform Metadata

Freeform metadata is contained in XML files that contain anything you want to include for generating code. You'll merge this information into other sections of data. Merging into existing sections is important because you need to access metadata without regard to its source. For example, if you need the caption for a column, you don't care whether it was entered via an extended property or freeform metadata, and you certainly don't want to rewrite your code generation templates just because you changed how you entered the captions. Thus, you don't want to base the structure or organization of your metadata file on where information is coming from because it may change over the project life cycle. You accomplish this by using parallel structures in each metadata file and performing a merge that combines attributes within elements themselves—what I call an *intimate merge*. The "Merging Metadata" section discusses the mechanics of an intimate merge.

Creating Column Details

Because the major purpose of freeform metadata is to merge with other metadata, its design is rather easy when it's merged into an existing section. Freeform metadata files contain sections that mimic the structure of the metadata section into which you intend to merge it.

For example, the freeform metadata for adding captions, descriptions, and regular expression (regex) validation to database columns looks like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<dbs:Databases xmlns:dbs="http://kadgen/DatabaseStructure">
  <dbs:Database Name="Northwind">
    <dbs:Tables>
      <dbs:Table Name="Customers">
        <dbs:TableColumns>
           <dbs:TableColumn Name="CustomerID"
                Caption="ID"
                RegEx="AlphaNumericPunc">
          </dbs:TableColumn>
          <dbs:TableColumn Name="CompanyName"
                Caption="Company"
                Desc="Company Name to appear on shipping labels"
                RegEx="AlphaNumericPunc">
          </dbs:TableColumn>
```

This allows a nonambiguous merge into the main `DataStructure` element and its descendants. Note that you can enter the same information using

extended properties. I suggest for the sake of others that you adopt one approach consistently or at least be able to articulate what's where.

Defining Stored Procedure Intent

As shown earlier, the intent of each stored procedure is important in understanding how it'll be used in building your application. Each business object knows how to select its data from the database, update existing data, insert data for new instances, and delete data if appropriate. Mapping the intent of each stored procedure allows the business or data access objects to know what stored procedure to use for each operation. You can store this information in the procedure name using careful naming or enter it using extended properties. Like the column information, you can alternatively enter it via freeform metadata. If you have a naming system that works on many but not all of your stored procedures, the freeform approach works particularly well in overriding a few exceptions:

```
<dbs:Databases xmlns:dbs="http://kadgen/DatabaseStructure">
  <dbs:Database Name="Northwind">
    <dbs:StoredProcs>
      <dbs:StoredProc Name="nwd_Customers_Select" ProcType="Select" />
    </dbs:StoredProcs>
</dbs:Databases>
```

Providing UI Information

In addition to being able to build business objects, you'll also see how to build user interfaces in Chapters 9 and 10. Some aspects of your user interface don't have a good payback for code generation because there's a lot of variation and because existing visual tools make manual layout of user interfaces easy. Other user interface elements—including utility forms, prototypes, and supporting elements such as menus—can be created via code generation.

Metadata samples in earlier sections were easy to design because they parallels existing data structures. The following example defines a role-authorized menu structure and is a different scenario because you have free rein in how you design the structure. You want to design metadata looking forward to the code generation process. If you're designing a menu structure, when you look forward to code generation, you might think, "Ah, I'll be creating a main menu that will contain nested menu items. Each item may have an authorization role associated with it,

and each role has the potential to run a different method when the user clicks the menu item." Articulating the goal leads to this:

```
<?xml version="1.0" encoding="UTF-8"?>
<ffu:UIStucture xmlns:dbs="http://kadgen/FreeFormForUI.xsd">
  <ffu:WinApplication Name="Main">
    <ffu:MainMenu>
      <ffu:Menu Name="MainFile" Caption="File">
         <ffu:Menu Name="FileOpen" Caption="Open">
            <ffu:MenuAllow Role="All" TypeName="" MethodName="FileOpen"/>
         </ffu:Menu>
         <ffu:Menu Name="FileNew" Caption="New">
            <ffu:MenuAllow Role="Managers" TypeName="" MethodName="MgrsFileNew"/>
            <ffu:MenuAllow Role="Admins" TypeName="" MethodName="FileNew"/>
         </ffu:Menu>
      </ffu:Menu>
    </ffu:MainMenu>
  </ui:WinApplication>
  <ui:WebApplication>
  </ui:WebAppliation>
</ui:UIStucture>
```

If you recall earlier discussions about metadata not including dynamic data that might be changed at runtime, you might challenge this example by asking whether the permissions change at runtime. In this particular mythical application, the permissions associated with each role are set in the design with only the members of each role changing at runtime.

NOTE In Appendix A, I suggest you stick namespaces on only the root node within your XML files. The exception to this rule is when child nodes have a distinct and self-contained purpose. In freeform metadata, each child node beneath the root corresponds to a distinct subset of data, and it's appropriate to include namespaces and their prefixes on the child nodes.

Managing Freeform Metadata Files

Consider how you want to manage your freeform metadata files. You'll probably want to keep these XML files in a single centralized directory, and you'll certainly want to keep them under source control.

Two considerations guide how you split up the freeform information into multiple freeform metadata files. You'll see in the section "Merging Metadata" that the last file merged overrides any previous information if they collide. The order of merging files is important, and you can't have some parts of a single freeform metadata file merge first and other parts of the same file merge last. Once you've split up files on this basis, consider splitting based on who will edit the files if your group has clear roles and different people will be editing different freeform metadata files—this makes it easier to manage source control. You can also split up metadata based on the major metadata header. If you aren't clear at this point how to split up your freeform metadata files, then you can leave them as one file. The only reason to split it up is to make it easier to manage, and all other things being equal, one file is easier to manage than several files, particularly during maintenance.

CAUTION From a strict technical perspective, it isn't difficult to reorganize your freeform metadata files late in the project. From a strategic perspective, I discourage this. Because data from the last file wins on any collisions, you could have one or more very small changes, from a misspelled caption to more serious problems, arise because metadata changed as a result of the merge order changing. Also, it's important to use source control on these files, and late reorganization makes it difficult to maintain a long-term history.

Using Skip Attributes

One of the common things I hear when I talk to people who have experimented with code generation is that they found not everything should be treated the same way during code generation. Often, certain tables, columns, stored procedures, and so on should be skipped during generation.

You may be tempted to have a narrow, easy solution that one way or another marks metadata items to be skipped. That could just mean adding a `Skip` attribute to appropriate nodes. That doesn't work because skipping often depends on the specific context. For example, in a stateful Windows.Forms application, exposing the `TimeStamp` is just confusing to later programmers, but the value may be passed to the browser and back in a stateless data flow pattern in ASP.NET. Allowing your metadata to become dependent on context greatly diminishes the value of metadata as an independent unit of information about your application. So I encourage you to develop a small set of attributes that specify the purpose of the category and avoid the `Skip` attribute altogether.

Table 2-5 gives you some ideas; I doubt you'll want to use all of these, but you can select the ones that make sense for your project.

Table 2-5. Suggested Attributes to Manage Skipping Items During Code Generation

ATTRIBUTE	PURPOSE
`Obsolete`	No longer used by any operations
`Private`	Not exposed by objects in any context
`StatefulPrivate`	Only exposed in a stateful environment
`StatelessPrivate`	Only exposed in a stateless environment
`Large`	Ignored for tasks that don't use large objects such as images
`ReadOnly`	Lets you provide a different class for read-only tables
`ReportsOnly`	Allows you to provide a different class design for reports

TIP Don't use generic skip attributes, but instead categorize elements by adding extra attributes and allow templates to skip as appropriate.

Using an XSD

Freeform metadata isn't really a free-for-all. You don't have a lot of flexibility in the structure because the merge process and code output will rely on the specific structure you create. XSD is the perfect tool for defining the structure, or schema, of your freeform metadata. This allows you to validate your XML before you use it and makes editing easier because Visual Studio .NET uses XSDs for IntelliSense, autocompletion, and warnings from the background compiler.

Validation is especially important when you're using XPath, which you'll use in all three types of code generation. As you'll see in the Appendix A, XSLT trusts you completely and will raise no errors if something is missing, misspelled, in the wrong place, and so on. Important information might fail to be used. You can alleviate this problem by using XSD to validate any manually entered or edited data.

NOTE Hooking up XSDs for use by Visual Studio is covered in Appendix A. You can infer an XSD from an XML file using XSD.EXE directly or right-clicking the XML file in Visual Studio and clicking Create Schema.[13] You can build a master XML file that contains all the elements and attributes you plan to use as a basis for your schema inference, or you can create a schema off the most complete file you happen to have and maintain the XSD by hand. If you find the XSD format intimidating, the approach of inferring your XSD from a master XML file works well. Just don't use both approaches because you'll wreck the edited XSD.

Merging Metadata

Merging metadata into a monolithic metadata file combines identical elements for seamless use in code generation. The merge process is complicated by one additional twist. The organization of the metadata—where things are found—should be independent from the metadata source. This means information from different sources can't just be appended to the same file but must be seamlessly intermixed in an intimate merge.

In Chapter 3 you'll see how to combine multiple processes, including merge processes, into one script. You'll see how to use a script to determine the order in which files are merged. Doing this at the last minute allows you to handle the monolithic file as a transient and leave it out of source control. This is really helpful when you have several people doing independent code generation on different parts of your application or closely related applications. Even though the file is transient, you need a copy of the most recent version for debugging, so you'll save it to disk.

There are two types of metadata files: those that contain a root element corresponding to a section in the output metadata file and those that contain a different root element that contains the attribute `FreeForm="true"`. This small difference indicates a broader difference in the history of the files. Although it isn't technically required, files corresponding to a single metadata header are programmatically created. Files where the root node wraps one or more child nodes corresponding to section headers are manually created as freeform metadata files. Including the high-level root, even when you're editing a single section, allows you to add additional sections with minimal disruption. Combining several section headers in a single freeform file helps keep files from getting too small and scattered.

13. If your XML document runs into problems with the XSD.EXE generator, you might want to download the InferSchema tool from GotDotNet.com.

The merge is cumulative. It adds new child elements where they're missing and new attributes to existing elements. If the same attribute appears for the same element in multiple merge source files, the last one wins and will overwrite the previous values. The expectation is that you'll merge manual freeform metadata last and allow it to overwrite any values extracted programmatically. This allows you to smoothly override values, but be sure you debug against the monolithic metadata file.

CAUTION Because each metadata file can overwrite values in other metadata files as they're merged, use a copy of the monolithic file to check input during debugging.

The merge process is short and recursive, so I'll walk through the code. The `MergeFreeForm` class has a single public method that accepts an XML output document and a filename. It loads the file and checks the `FreeForm` attribute. If it doesn't exist or doesn't have a value of `true`, then the root is treated as the metadata section header. If the attribute is `true`, then the second-level nodes are treated as the section headers. This is the only difference between the two types of source files:

```
Public Class MergeFreeForm
   Public Shared Sub Merge( _
                  ByVal outDoc As Xml.XmlDocument, _
                  ByVal fileName As String)
      Dim mergeDoc As New Xml.XmlDocument
      Dim rootNode As Xml.XmlNode
      mergeDoc.Load(filename)
      rootNode = mergeDoc.ChildNodes(1)
      If Tools.GetAttributeOrEmpty(rootNode, "FreeForm") = "true" Then
         ' If its a freeform file, regardless of the root element name,
         ' attempt to merge all child nodes of root
         For Each node As Xml.XmlNode In rootNode.ChildNodes
            MergeNode(outDoc.ChildNodes(1), node, Nothing)
         Next
      Else
         MergeNode(outDoc.ChildNodes(1), rootNode, Nothing)
      End If
   End Sub
```

The `MergeNode` method is recursive. It searches the target file for an element matching the source node's element name and a matching value of the `Name` attribute (if the `Name` attribute is present). This step is simple and unambiguous because it assumes precisely the same structure in the source and target node. If the `SelectSingleNode` method doesn't find the node already in the target, the node is added. If `SelectSingleNode` finds the node, `MergeNode` checks the target node for the presence of each attribute from the source node. If the attribute doesn't exist yet, it's created. If the attribute exists, its value is updated:

```
Private Shared Sub MergeNode( _
                ByVal outParent As Xml.XmlNode, _
                ByVal node As Xml.XmlNode, _
                ByVal nameAttrib As Xml.XmlAttribute)
   Dim nsmgr As New Xml.XmlNamespaceManager(outParent.OwnerDocument.NameTable)
   Dim predicate As String = ""
   nsmgr.AddNamespace(node.Prefix, node.NamespaceURI)
   If Not nameAttrib Is Nothing Then
      predicate = "[@Name='" & nameAttrib.Value & "']"
   End If
   Dim testNode As Xml.XmlNode = outParent.SelectSingleNode( _
                     node.Name & predicate, nsmgr)
   If testNode Is Nothing Then
      outParent.AppendChild(outParent.OwnerDocument.ImportNode(node, True))
   Else
      ' Node exists, add attributes, then children
      For Each attrib As Xml.XmlAttribute In node.Attributes
         If testNode.Attributes(attrib.Name) Is Nothing Then
            testNode.Attributes.Append(xmlHelpers.NewAttribute( _
                     testNode.OwnerDocument, attrib.Name, attrib.Value))
         Else
            testNode.Attributes(attrib.Name).Value = attrib.Value
         End If
      Next
```

CAUTION Attribute values are overwritten. The last value wins.

Finally, `MergeNode` recursively calls itself for all child nodes. If you aren't familiar with recursive algorithms, you might have to look at this a couple of times, but it's a nice example of a simple, effective recursive algorithm. You'll see

recursive algorithms frequently in this book because they're so handy when you're working with nested data such as XML documents:

```
            For Each childNode As Xml.XmlNode In node.ChildNodes
               MergeNode(testNode, childNode, childNode.Attributes("Name"))
            Next
         End If
      End Sub
End Class
```

Namespace usage might be a little confusing here. I'm assuming you'll prefix everything, including your freeform data. I anticipate you may use different namespaces for different sections of your metadata. Thus, the namespace manager is created for the current node. Namespace prefixes are discussed in Appendix A. The key factor here is that the XPath expression's prefix will be from the node you're searching for—from the source document you're merging from.

It doesn't matter whether this prefix matches the target prefix as long as the namespaces pointed to by both prefixes match. This code will work just fine even if you vary the prefix you're using for a specific namespace. But don't do that! A major problem will arise if you add any new elements. These new elements will have a prefix from the source and won't be understood in the context of the target document. You could work on solving this problem, but I can't imagine a reason you'd use different prefixes for the same namespace in different source documents. It'd be really hard for a maintenance programmer to keep up with the mental calisthenics of switching from one prefix to another.

TIP Use consistent namespaces.

Summary

Extracting metadata is the process of gathering information from metadata sources and organizing it as an XML file that's friendly to code generation. Available types of metadata include the following:

- Schema definitions such as XSD
- Databases such as SQL Server

- Web Services via WSDLs
- Manually entered freeform metadata
- External sources, such as mainframes, Excel, and so on
- Existing applications and source code
- Design tools such as UML

If you're working with XSD, it's helpful to convert the XSD to friendly XML prior to code generation.

To extract from SQL Server, Oracle, or another SQL-92 compliant databases, use the SQL-92 information schema views. Extracting most metadata from databases is straightforward, but stored procedure intent, recordset naming, and the mapping between columns and the underlying data columns are difficult. Going to the trouble of extracting this information provides richer information for code generation. You can get this information via careful naming and inference, extended properties, or manually entered freeform data.

You can use freeform metadata to include any additional metadata. This freeform metadata can be manually entered or programmatically derived. In the intimate merge of metadata described in this chapter, files that merged later can overwrite previously merged files. This allows you to incorporate manual overrides late in the merge process.

Additional Reading

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

- In addition to covering Web Services, *Building XML Web Services for the Microsoft .NET Platform* by Scott Short (Microsoft Press, 2002) contains some concise sections on other X-Stuff subjects, including XSD.
- *Professional UML with Visual Studio .NET* by Andrew Filev, Tony Loton, Kevin McNeish, Ben Schoellmann, John Slater, and Chaur G. Wu (Wrox, 2002) discusses using UML with Visual Studio and discusses Visio's limited code generation capabilities.

EXHEART

CHAPTER 3

Outputting Code

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.

Once you've prepared your metadata, the next step is to build templates and perform code generation. Your initial development and long-term maintenance both benefit from repeatable one-click code generation. You benefit from operational *simplicity* because you can largely forget about code generation details and template content during development. A maintenance programmer benefits from operational *consistency* because they can repeat your actions with little effort. The keys to this repeatability are placing as much code as practical in templates and using a tool to perform the multiple steps of code generation.

Managing the complexity of sophisticated code generation is one of the key solutions this book provides. You can manage this complexity by combining the following three approaches:

- Splitting code generation into five distinct steps with five corresponding principles (covered in Chapter 1)
- Using a tool to manage the multistep process of code generation (covered in this chapter)
- Building templates that reflect best practices (covered in Chapters 6–10)

NOTE Building templates to hold the structure of your code is trivial for trivial cases. When it looks complex, keep in mind that I'm unwilling to relegate code generation to simple or trivial cases. I believe your code generation techniques should be able to handle any coding problem you choose to address and produce the code you want. My job is to tell you the possibilities; you get to tune in the level of complexity that makes sense for your project by using as much or as little as you want.

This chapter provides a preliminary look at a tool, the *harness*, to manage the multistep process of code generation, and Chapter 5 and Appendix C go on to cover this tool from different perspectives. Specifically, Chapter 5 contains more detail on using the harness, and Appendix C discusses its code.

Once you've seen how you'll perform code generation via the tool, you can focus on how you create individual templates. The bulk of this chapter explores template creation, covering aspects of brute force, XSL Transformations (XSLT) templates, and the CodeDOM in additional detail.

Understanding the Elements of the Code Generation Harness

One of the great parts of software development is that it thrives on a basic human trait—laziness. This isn't irresponsibility, carelessness, or a lack of personal hygiene. It's looking for the easiest way to get a job done well. You can work hard to cut corners in a responsible manner, and your source code often thrives as a result. Code reuse is a classic example, but object-oriented programming, code generation, refactoring, and other techniques benefit from the same trait.

You'll perform a large number of steps in code generation. You're likely to have the following:

- Several extraction, morphing, and merging steps to prepare your metadata, as discussed in Chapters 2 and 6
- Five templates to create stored procedure scripts (Create, Retrieve, Update, Delete, and SetSelect)
- An installation step for each stored procedure script you generate

- At least four middle and data tier templates (generated objects, generated collections, editable objects, and editable collections)

- Up to ten separate User Interface (UI) templates

Rather than individually running these steps, you can build on your responsible laziness and create a *one-click* approach to code generation that easily accomplishes all of these steps. The frightening alternative is code generation that follows a two-page checklist. To transition from that two-page checklist to one-click processing, you need a code generation harness that processes a script consisting of individual instructions called *directives*.

Avoiding Too Many Steps

Running a script lets you weave dozens of code generation steps together and run them with a single mouse-click. This lets you precisely repeat the generation process with a single click. Code generation must be repeatable. You'll want to easily repeat it next week, and someone marginally familiar with the language and application may need to repeat it in three or four years to incorporate a database or business rule change. If they can't do that, they'll cut corners and hack around your solid design. Now, the real nightmare: A programmer changes a generated file directly when trying to get a job finished. Eight months later, someone else regenerates and introduces potentially catastrophic bugs in a part of the application no one expected to change—so it wasn't even in the test plan! Code generation doesn't have to be like that.

If you've played with the ADO.NET strongly typed DataSet, you've experienced the opposite of one-click generation—the stepwise process of generating code that can turn into a two-page checklist. First, you drag the tables or stored procedures onto the design surface. This generates an XML Schema Definition (XSD). You manually add relations and might adjust name mappings or add annotations. These changes modify the XSD. Then you select Generate Dataset from the context menu, and code generation creates the strongly typed DataSet. You have to do this for every DataSet—most of the steps each time you regenerate!

Forget that! Your goal is one-click generation. You put a lot into building your application. The scripted harness preserves your important choreographed steps through rounds of even not-so-bright maintenance programmers. Listen to your laziness gene and ensure that *you're* using a one-click approach. That's the only way to build stable code generation. Then ensure that the harness and everything it accesses is a permanent part of your project. Let your script and

templates change for new projects, but split off them off as a semistatic part of each project. Change them only if needed because of a conscious decision and as part of the project's evolutionary cycles, backed by testing. It's not just convenience; it's essential. Planning for the long term includes intelligent one-click code generation based on a script.

Understanding Script Directives

The script you'll use for one-click code generation is an Extensible Markup Language (XML) file that contains a series of *directive* elements. Each directive is an XML element that describes a single, atomic step run by the code generation script. You can create and edit the script file through the harness or through an XML editor. To support editing in either manner, the XML script used by the harness follows an XSD schema that you can extend.

NOTE Appendix A shows the XSD schema used by the harness script as an example. This lets you look at its details, even if you aren't already familiar with XSD.

Each directive includes many attributes that specify different aspects of processing one atomic step. Child elements of the directive logically group attributes based on what they relate to. For instance, there's a set of attributes related to processes that produce multiple files. The `MultiPass` child element contains selection and naming criteria that can be used by XSLT generation, brute-force generation, CodeDOM generation, and SQL scripts. This grouping of attributes within child elements allows abstraction of the child element definition for use by multiple directives, significantly easing the maintenance burden for the XSD. For example, the basic information required of all directives is contained in a child element named `Standard` that contains attributes such as a human-friendly `Name`. It also includes a `Checked` attribute indicating whether the directive is checked in the TreeView of the harness, where all checked elements are processed.

Groups of directives are called *sections*. Sections let you group processing based on any convenient criteria. You might group directives by the generation step they correspond to or by the processing performed by different people, or you might group directives to provide alternate scripts for testing. You can turn processing on and off for entire sections when you run the generation harness. This is an example of a single section with a single directive and three child elements:

```
<kg:Section OutputDir="VBCode/Test">
   <kg:Standard Name="PreProcessing" Checked="true" />
   <kg:CreateMetadata Server="localhost" Database="Northwind">
      <kg:Standard Name="NorthwindExtraction" Checked=" true " />
      <kg:SinglePass OutputFile="[BasePath]Metadata Files\Northwind.xml" />
      <kg:OutputRules HashOutput="false" OutputFileType="XML"
                      OutputGenType="Overwrite" />
   </kg:CreateMetadata>
</kg:Section>
```

In this sample, the directive is `CreateMetadata`. The `CreateMetadata` directive and the `Section` each contain `Standard` child elements that contain the name and checked status. The checked status indicates whether the item appears checked in the TreeView and whether that step will run as part of code generation. The `CreateMetadata` directive contains two additional child elements, each of which contains a logical group of information required for processing. There are about ten directives for full-featured code generation, as well as several groupings of attributes in child elements. Chapter 5 discusses specifics of each directive and child element.

Using the Code Generation Tool

The code generation tool does only two things. It puts a friendly face on editing XML, and it performs code generation by executing your script. To perform code generation, the tool executes code based on the XML directive elements that make up your harness script. This code may be in an external assembly you reference in a script directive.

The tool's XML editing feature is generic and based on the XSD. The attributes of the XML harness script become controls on the harness's form, and the child elements become grouping boxes.

NOTE I don't promise it'll edit any XML file you throw at it, but it'll edit XML you need for scripting code generation directives and may edit other XML files if you have XSDs for them. It's an object-oriented design, so if you have a similar need, you can customize this form through inheritance. The harness user interface also uses some cool dynamic techniques you can walk through in Appendix C.

You start the tool by double-clicking CodeGenerationHarness.exe in the Tools subdirectory.[1] Selecting File ➤ Open lets you open a script such as Harness3.xml in the Chapter 3 subdirectory. When you open this file, you'll see a TreeView containing sections and directives. As you select different nodes, that node will appear for editing in the right pane. Figure 3-1 shows the user interface for this tool. From the File menu you'll be able to do familiar actions such as save and create new files. New files are created based on the XSD. The standard XSD for code generation is called *KADDrivingMetada.xsd* and is in the XFiles subdirectory of the book's code directory.

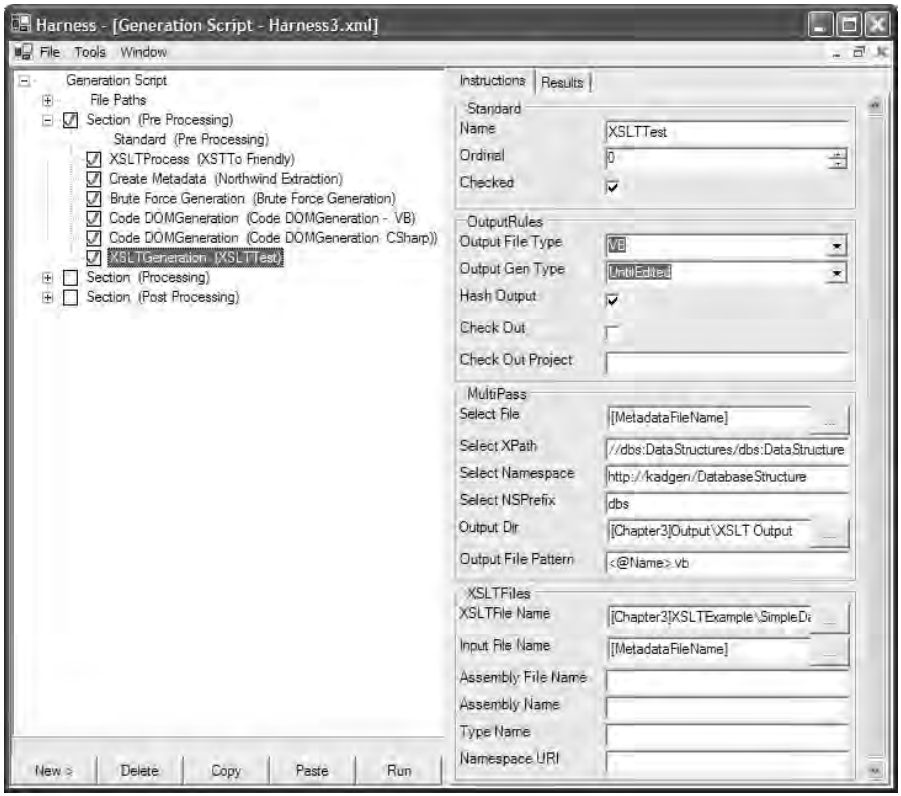


Figure 3-1. The code generation harness with a sample XML file open to an `XSLTGeneration` *directive*

When you click the Run button, all the currently checked directives are processed. After processing is complete, the Result tab contains any errors or warnings that occurred during processing.

---

1. Feel free to create a shortcut and put it on your Desktop or Start menu. The downside of XCopy deployment is that this doesn't happen automatically.

NOTE If you modify XSLT files and save the updates, code generation will use the updated files the next time it runs, regardless of whether the harness has remained open. However, you won't be able to compile assemblies that you reference as processes until you close the harness. This happens because the assemblies remain loaded. You have to stop and restart the harness when you make changes to brute-force or CodeDOM templates.

Understanding Other Features of the Generation Harness

An overview of harness features will help answer questions that may arise as you explore template details later in this chapter. For example, in the later discussion on using template parameters, you might wonder how context-sensitive parameters are passed from the harness. Information on harness features will also let you use the generation harness to test the templates in this chapter. This overview is simplified, and Chapter 5 offers additional information on these features.

Working with Abstract File Paths

The generation harness uses abstractions for file paths and names. This allows the same script to be used to start another project by changing only path definitions. Many of the file paths appear multiple times within the script, and this also provides you with a straightforward way to update all occurrences quickly. Square brackets surround abstract `FilePath` filenames, such as the `BasePath` in this code:

```
OutputDir="[BasePath]Chapter 3\Output\XSLT Output"
```

The `FilePath` elements of each harness script contain the definitions for the abstract file paths used by that script. You can nest abstract paths and use multiple abstract paths in a sequence.

NOTE File paths are relative to the harness script file location. This allows users with different file configurations to use the same harness, as long as the project directory has the same organization.

Providing Parameters

The harness retrieves the parameters accepted by templates during the generation process. It does this via reflection for brute-force and CodeDOM templates and by running an internal stylesheet against XSLT templates.

NOTE To retrieve parameters from an XSLT stylesheet, the stylesheet is opened as the input XML document, and an internally stored XSLT stylesheet is run against the template stylesheet. The internal XSLT retrieves all `<xsl:param…>` elements that are children of the `<xsl:stylesheet…>` (or `<xsl:transform…>`) element in the template stylesheet.

CAUTION Parameter names are case sensitive.

The harness itself provides three special parameters. If your template accepts any of these parameters, the harness will pass the values. The three special parameters are as follows:

- `fileName` passes the name of the output file. This would generally be used in a header comment.
- `genDateTime` passes the date and time the generation is to perform. You could use this in a header comment, but it causes files under source control to be checked in every time you run code generation, so this isn't recommended.
- `nodeSelect` passes the current node of the file. This isn't valid for XSLT processing, but it's important in brute-force and CodeDOM generation to determine the XML input file context for the code generation (which node you're processing).

In addition to these special parameters, the harness checks the node it's processing in the input XML metadata for attributes matching the names of any additional parameters your template accepts. You'll usually pass something identifying the current node, such as the name of the current table or object.

CAUTION The match between parameter names and the attribute name in the XML input node is case sensitive.

TIP Pass identifying values as parameters that allow the template to find the node to process in the metadata. If the `SelectFile` is the metadata file (the normal case), the template will be processing this node. It's generally easier to retrieve additional values from the metadata than pass them as parameters. (The "Looking at the Harness Script" and "Introducing the Target and Metadata Files" sections explain this in more detail.)

CAUTION No error occurs if attributes matching your parameters aren't found.

Overwriting Files

Managing file overwriting is a critical part of code generation. It's important to overwrite the correct files without overwriting the wrong files, particularly those you've modified by hand. Overwriting rules are defined in the harness script, so you don't have to make any decisions during generation unless something goes wrong. In Chapter 5, you'll see how to manage file overwriting through the `OutputGenType` attribute of the `OutputRules` child element. This allows you to specify whether to overwrite files and log errors if the harness encounters manually edited files. This feature ensures you can meet the goal of providing protection for manually edited files.

To provide this level of control, the harness needs to know whether a programmer has manually edited each file. You might think about using a date/time stamp, but in addition to the ugliness of tracking it across time zones, it isn't reliable, especially because source control frequently changes the file time stamps. A better solution is to place a *hash marker* within the file header comments. A hash marker is a numeric value that has an extremely high probability of uniquely representing the underlying value—in this case representing the file's text. The initial hash corresponds to the original file. If a hash calculated later doesn't match, then someone has edited the file. Chapter 5 explains hash markers in more detail.

The hash marker is the weird sequence of characters at the top of a file; if you edit or remove them or the surrounding comments and markers, you'll get code generation errors. This is an example of a hash marker:

```
%%%###%%%65c708f97193ef7405269d998b960f54%%%###%%%
```

Extending the Harness

This book is the start of your code generation adventures, not the sum total of them. There's no way I can predict what you'll need to do; there isn't even a way for *you* to predict what you'll need to do. The provided directives do everything I thought of; I trust your creativity, so I designed the harness with extensibility in mind. You can incorporate new templates using all the current harness capabilities (generating output from templates, collecting metadata, running processes, running SQL scripts, and so on) without extending the harness. If you need more capabilities, you can extend it. I provided this feature so you could incorporate every step of code generation as part of your one-click code process, even if I didn't provide the directive. To create new directives, you'll need to do the following:

1. Design the XML for the directive (what attributes does it need?)

2. Create a .NET assembly that performs the task in a static (`Shared`) method

3. Specify the assembly, type, and method name in the XSD along with any other information you defined

4. Add this directive's definition to the XSD

5. Create a new entry in your XML harness script for the new item

That's it. If your method doesn't run, you may need to look up (in the .NET Help) how to attach a debugger to an executing process to debug your code. The problem will probably be with the parameters passed. In designing the schema for your extension, you can use any existing child element—all of which are defined via XSD-named complex types. (Appendix A further discusses named complex types and simple types and how to restrict entries to a list.) You can also create new named complex types. Combo box entries are simple types restricted to a list. You can add new items to existing lists or create new lists.

CAUTION Changes to your XSD won't affect your editing in Visual Studio unless the updated XSD is in the project directory or in the magic directory [Program Files]\ Microsoft Visual Studio .NET 2003\ Common7\Packages\schemas\xml and you restart Visual Studio. Appendix A has more information about doing this.

Dynamic Techniques

Does it seem odd that a key tool in a book on code generation doesn't use code generation but instead uses its wild and crazy sister, data-driven dynamic techniques? Code generation just wasn't the easiest way to build the code generation harness. A few reasons why dynamic techniques are a better fit for this tool are as follows:

- The application is simple, just displaying the attributes for XML nodes and cycling through these nodes for processing.
- Performance isn't an issue.
- A sophisticated user interface isn't needed.
- The interface can be inferred from available information (the XSD).
- The structure of the user interface is under external control (the XSD, not the harness).
- The task of the application is fixed, so there's no reason, other than XSD changes, to compile and deploy the application.
- Maintenance programmers already need to understand reflection because of the way templates run.

NOTE If you're interested in how I used dynamic techniques to build this tool, refer to Appendix C. It walks through the code, explaining the dynamic techniques.

Looking at the Harness Script

You'll use scripts containing directive elements when you run the harness. I'll discuss details of scripts and directives in Chapter 5, but a look at the XML illustrates how the XML directives control code generation. The XML corresponding to the directive displayed in Figure 3-1 is as follows:

```
<kg:XSLTGeneration>
   <kg:Standard Name="XSLTTest" Checked="true" Ordinal="0" />
   <kg:OutputRules HashOutput="true"
      OutputFileType="VB"
      OutputGenType="UntilEdited" />
   <kg:MultiPass SelectFile="[MetadataFileName]"
      SelectNamespace="http://kadgen/DatabaseStructure"
      SelectNSPrefix="dbs"
      SelectXPath="//dbs:DataStructures/dbs:DataStructure/dbs:Tables/dbs:Table"
      OutputDir="[Chapter3]Output\XSLT Output"
      OutputFilePattern="&lt;@Name&gt;.vb" />
   <kg:XSLTFiles InputFileName="[MetadataFileName]"
      XSLTFileName="[Chapter3]Test\SimpleDataContainer.xslt" />
</kg:XSLTGeneration>
```

The `Standard` element contains the information common to all directives. The `OutputRules` contains overwriting information. In this case, code generation creates a hash and overwrites any existing file, unless a human edited the file.

The `MultiPass` child element contains information about how multiple output files are created based on a single directive. The `SelectNamespace` and `SelectNSPrefix` support the `SelectXPath` in selecting nodes. The `SelectFile` attribute tells the harness which XML file to use in selecting nodes. Code generation creates files in the `OutputDir` using the `OutputFilePattern`. This pattern is XML-like so uses angle brackets to define replacement tokens in the filename. However, you can't directly output the less than (<) sign via XSLT, so this attribute uses the escaped version `<` and `>`. Examples of the resulting filenames are Customers.vb and Project.vb.

The `XSLTFiles` element contains information about the XSLT transform. This includes the input file and the XSLT transform filename. Brute-force and CodeDOM sections of the script use identical `Standard`, `OutputRules`, and `MultiPass` child elements. Instead of `XSLTFiles`, the brute-force generation uses the `Process` directive. With the `Standard`, `OutputRules`, and `MultiPass` child elements represented by ellipses, the brute-force directive is as follows:

```
<kg:BruteForceGeneration>
...
<kg:Process
   AssemblyFileName="[Chapter3]BruteForceExample\bin\BruteForceExample.dll"
   TypeName="BruteForceExample"
   MethodName="GetStream"
   AssemblyName="" />
```

The similar CodeDOM directive is as follows:

```
 <kg:CodeDOMGeneration TargetLanguage="VB">
...
 <kg:Process
   AssemblyFileName="[Chapter3]CodeDOMExample\bin\CodeDOMExample.dll"
   TypeName="CodeDOMExample"
   MethodName="BuildGraph"
   AssemblyName="" />
```

NOTE The `Process` directive uses the `AssemblyFileName` attribute if the class is in an external assembly and uses the `AssemblyName` attribute in the less common case where the class is part of the harness.

Although these directives may initially seem complex, they provide a great deal of consistency between different directives, which makes maintenance easier. Even if you're sticking with one generation mechanism (recommended), you'll use multiple directives because you'll create metadata, merge metadata, transform metadata, generate code, run SQL scripts, and so on.

Examining Code Generation Mechanics

Now that you've seen an overview of the harness you'll use to run code generation, you'll take a close look at the templates that output code. Regardless of the mechanism you're using, I call what holds your code pattern a *template*. When you're using brute-force or CodeDOM generation, the template is a .NET method; when you're using XSLT code generation, the template is an XSLT stylesheet.

The following sections look at how you work with the templates in each of the three techniques for generating code. I approach each technique differently because the core mechanism differs. In XSLT code generation, I leave the XSLT basics for Appendix A and explore how to take sample code and convert it to an XSLT template. The section "Exploring Details of Brute-Force Generation" is slim because the beauty of this technique is that it's simple and uses techniques that are probably already familiar to you. Working with the CodeDOM is complex, and although that section just flutters across the surface, that section is long, complex, and won't be relevant to all readers. There's just so little information on using the CodeDOM available today that the section seemed necessary for those readers who will use, make a decision on using, or simply want to understand the CodeDOM. Appendix D has additional details on issues you may encounter if you work with the CodeDOM.

NOTE You may find creative ways to combine these approaches, such as your own tokenizing scheme. I keep the approaches separate to show the fundamentals of each. You can pull from each if you've devised a hybrid approach. I'm not a fan of tokenizing schemes because they don't have the power of XSLT, so they're hard to use in complex code scenarios.

Introducing the Target and Metadata Files

Metadata files are an XML description of your application's data. Target files are sample source code files that look like what you want to generate. You'll build templates that output code files that match your target file using XML metadata as input. You'll compare the output with the target as an initial test of your templates.

For reference, Listing 3-1 shows part of the XML input file used in this chapter's XSLT and brute-force code generation examples. This XML describes a single data table and is part of a larger metadata file containing descriptions for each table, stored procedure, view, function, and user-defined data type in the database. I removed privilege information and replaced the details of most columns with ellipses to shorten the listing. The `Table` element and each `TableColumn` element contain many attributes. These attributes provide details about the table or column you'll use during code generation.

Listing 3-1. Part of the XML Input File Used in This Chapter

```
<dbs:Table Name="Customers" OriginalName="Customers" SingularName="Customer"
          PluralName="Customers" DisplayName="CompanyName" Caption="Customer"
          DeletePriority="" InsertPriority="0">
   <dbs:TableColumns>
      <dbs:TableColumn Name="CustomerID" OriginalName="CustomerID"
         Ordinal="1" Default="" AllowNulls="false" SQLType="NChar"
         NETType="System.String" MaxLength="5" IsPrimaryKey="true"
         Caption="ID" RegEx="AlphaNumericPunc"/>
      <dbs:TableColumn Name="CompanyName" OriginalName="CompanyName" Ordinal="2"
         Default="" AllowNulls="false" SQLType="NVarChar"
         NETType="System.String" MaxLength="40" IsPrimaryKey="false"
         Caption="Company"
          Desc="Company Name for shipping labels and correspondence"
         RegEx="AlphaNumericPunc" />
      <dbs:TableColumn Name="ContactName" OriginalName="ContactName" Ordinal="3"
         Default="" AllowNulls="true" SQLType="NVarChar"
```

```
            NETType="System.String" MaxLength="30" IsPrimaryKey="false"
            Caption="Contact" RegEx="AlphaNumericPunc" />
         ...
      </dbs:TableColumns>
      <dbs:TableConstraints>
         <dbs:PrimaryKey>
            <dbs:PKField Name="CustomerID" Ordinal="1" />
         </dbs:PrimaryKey>
         <dbs:TableRelations>
            <dbs:ChildTables>
               <dbs:ChildTable Name="Orders">
                  <dbs:ChildKeyField Name="CustomerID"
                     Ordinal="1" />
               </dbs:ChildTable>
               <dbs:ChildTable Name="CustomerCustomerDemo">
                  <dbs:ChildKeyField Name="CustomerID"
                     Ordinal="1" />
               </dbs:ChildTable>
            </dbs:ChildTables>
         </dbs:TableRelations>
      </dbs:TableConstraints>
      <dbs:ExtendedProperties>
         <dbs:ExtendedProperty Name="Desc"
            Value="Contains customer address and contact information" />
      </dbs:ExtendedProperties>
   </dbs:Table>
```

Listing 3-2 shows portions of the target code file generated from the metadata shown in Listing 3-1. Ellipses indicate where I clipped redundant sections. I purposely showed only two columns. You can see the rest in the download or mentally extrapolate.

NOTE This sample contains both a class for the collection and the individual objects in the same file to illustrate more code generation techniques. You can include related classes in the same file, or you can create a separate file for each class when you generate code as shown in Chapter 8. This sample is deliberately different from the example in Chapter 8. This gives you template examples that create a greater variety of output code.

Listing 3-2. The Target File Used to Create Templates for All Three Code Generation Types

```
Option Strict On
Option Explicit On

Imports System
Imports KADGEN
Imports System.Data

#Region "Description"
' Orders.vb
#End Region

Public Class OrderCollection
   Inherits CollectionBase

#Region "Constructors"
   Protected Sub New()
      MyBase.New("OrderCollection")
   End Sub
#End Region

#Region "Public and Friend Properties, Methods and Events"
   Public Overloads Sub Fill( _
                ByVal OrderID As System.Int32, _
                ByVal UserID As Int32)
      OrderDataAccessor.Fill(Me, OrderID, UserID)
   End Sub
#End Region

End Class

Public Class Order
   Inherits RowBase

#Region "Class Level Declarations"
   Protected mCollection As OrderCollection
   Private Shared mNextPrimaryKey As int32 = -1

   Private mOrderID As System.Int32
   Private mCustomerID As System.String
   ...
#End Region
```

```
#Region "Constructors"
   Friend Sub New(ByVal OrderCollection As OrderCollection)
      MyBase.new()
      mCollection = OrderCollection
   End Sub
#End Region

#Region "Base Class Implementation"
   Friend Sub SetNewPrimaryKey()
      OrderID = mNextPrimaryKey
      mNextPrimaryKey -= 1
   End Sub
#End Region

#Region "Field access properties"

   Public Function OrderIDColumnInfo As ColumnInfo
      Dim columnInfo As New ColumnInfo
      columnInfo.FieldName = "OrderID"
      columnInfo.FieldType = gettype(System.Int32)
      columnInfo.SQLType = "int"
      columnInfo.Caption = ""
      columnInfo.Desc = ""
   End Function

   Public Property OrderID As System.Int32
      Get
         Return mOrderID
      End Get
      Set(ByVal Value As System.Int32)
         mOrderID = Value
      End Set
   End Property

   Public Function CustomerIDColumnInfo As ColumnInfo
      Dim columnInfo As New ColumnInfo
      columnInfo.FieldName = "CustomerID"
      columnInfo.FieldType = gettype(System.String)
      columnInfo.SQLType = "nchar"
      columnInfo.Caption = ""
      columnInfo.Desc = ""
   End Function
```

```
    Public Property CustomerID As System.String
        Get
            Return mCustomerID
        End Get
        Set(ByVal Value As System.String)
            mCustomerID = Value
        End Set
    End Property
    ...
#End Region
End Class
```

The following sections show how to create this file via XSLT and brute-force code generation. These example will show how to output Visual Basic .NET (VB .NET) code. The Apress Web site includes parallel examples that output the code in C# files. Chapters 4 and 6 have examples outputting Structure Query Language (SQL) stored procedures.

Exploring Details of XSLT Code Generation

XSLT can output any type of text, including code in any language. You saw an introduction to how this works in Chapter 1, and this example dives deeper, including showing template organization and conditional code segments.

NOTE XSLT code generation relies on XSLT, so if you're unfamiliar with this technology, you'll want to review Appendix A.

Creating a Class

The best way to learn about XSLT code generation is to convert some sample code into an XSLT template. There's a custom version of Order.vb in the XSLTExample folder of the Chapter 3 code file. I'll convert this sample file into a working template and test it across the tables of the Northwind metadata. You may want to fire up Visual Studio and walk through this process as you read the text.

Open a new stylesheet and name it *SimpleDataContainer.xslt*. Create the folder in the Chapter 3/Test directory. This directory already exists if you unzipped the code. If you have updated the default stylesheet as described in

Appendix A, you're ready to go. If you didn't, you'll need to add the `xsl` prefix to the namespace, add the preserve space element, add the entry-level template, and update the stylesheet to reflect the `xsl` prefix. The metadata file created with the metadata extraction tool (discussed in Chapter 2) uses the `dbs` prefix, so you'll also need to add this namespace and prefix. Before you start, your stylesheet should contain the following (bolded items represent changes from the .NET default):

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                xmlns:xs="http://www.w3.org/2001/XMLSchema"
                xmlns:dbs="http://kadgen/DatabaseStructure" >
<xsl:import href="..\..\XFiles\Support.xslt"/>
<xsl:output method="text" encoding="UTF-8" indent="yes"/>
<xsl:preserve-space elements="*" />
```

I'm using a supporting stylesheet imported as a separate file. This stylesheet contains utility templates that provide code reuse within XSLT. The processor has to find this utility template. You'll need to modify this path if your directory layout differs from that in the download. I used relative paths to simplify moving your project.

XSLT provides both `xsl:import` and `xsl:include` mechanisms for adding XSLT files. They have different behaviors when template names conflict. You want to avoid conflicting template names to keep others sane, even if XSLT is flexible on the issue. Import has behavior that's more predictable if you accidentally create a conflict, which is why I use it, but it has to be the first XSLT element in the file.

Specifying XSLT Parameters

This template, along with many of the templates you'll build, takes three parameters: the output filename, the date/time of file generation, and the name of what you're generating. The earlier "Providing Parameters" section explains how the code generation harness supplies parameter values.

Within the XSLT template, you retrieve parameters using this:

```
<xsl:param name="Name"/>
<xsl:param name="fileName"/>
<xsl:param name="genDateTime"/>
```

You can access these parameters from any template within the stylesheet, including templates in the XSLT file accessed using `xsl:import`.

Creating the Entry-Level Template

The purpose of the *entry-level template* is to start processing the stylesheet. All other templates in the stylesheet should have either a name or a mode to clarify their purpose. The entry-level template is brief and just shifts the context within the XML metadata file to the node you're processing. In this entry-level template, the template processes table nodes with a `Name` attribute that matches the `Name` parameter passed:

```
<xsl:template match="/">
   <xsl:apply-templates select=
        "/dbs:DataStructures/dbs:DataStructure/dbs:Tables/dbs:Table[@Name=$Name]"
                          mode="BuildClasses" />
</xsl:template>
```

The `xsl:apply-templates` directive indicates that you want to process any template matching the select criteria. The XSLT engine processes every matching node with every template applicable to the `dbs:Table` node that also has a mode of `BuildClasses` (there's only one). If the XPath statement in this code doesn't make sense to you, review the XPath section of Appendix A.

Creating the High-Level Processing Template

When doing code generation, it's easiest to have the entry-level template process (or run) a single *high-level processing template*. The high-level processing template lays out the structure of the output file. I came up with the terms *entry-level template* and *high-level processing template* to make it easier to describe the purpose of these templates. You're unlikely to find them used in other documentation.

The high-level processing templates make extensive use of additional templates. Templates act as subroutines if called using the `xsl:call-templates` directive. When called using the `xsl:apply-templates` directive, they provide what's ultimately a looping operation. In XSLT, the subroutine style templates called using the `xsl:call-templates` directive are called *named templates*. The looping style templates called using the `xsl:apply-templates` directive are called *match templates*. It makes no difference whether these templates exist in this XSLT file or the files accessed via `xsl:import` or `xsl:include`[2] because these additional XSLT files become part of the main template during processing.

---

2. It makes no difference whether you access supporting stylesheets via `xsl:import` or `xsl:include` unless there's a conflict. You can check the MSDN Help for these `xsl` elements for more information about how conflicts are resolved when you use them.

Just as in procedural programming, you want to know what each template does, so keep the focus of each template as narrow as practical and describe what the template accomplishes in its name or its mode. The high-level processing template, started by the entry-level template, looks like this:

```
<xsl:template match="dbs:Table" mode="BuildClasses">
   <xsl:call-template name="FileOpen">
      <xsl:with-param name="imports" select="'KADGEN, System.Data'" />
   </xsl:call-template>
Public Class <xsl:value-of select="@SingularName"/>Collection
   Inherits CollectionBase
   <xsl:call-template name="CollectionConstructors" />
   <xsl:call-template name="PublicAndFriend" />
End Class

Public Class <xsl:value-of select="@SingularName" />
   Inherits RowBase
   <xsl:call-template name="ClassLevelDeclarations" />
   <xsl:call-template name="Constructors" />
   <xsl:call-template name="BaseClassImplementation" />
   <xsl: call-template name ="FieldAccessProperties" />
End Class
</xsl:template>
```

The structure of the output file emerges in the high-level processing template, which outputs a header and two classes—a collection class and a row class. Although these classes are more sophisticated than the example in Chapter 1, they're still a simplified class to focus on the template structure. Chapter 8 offers a complete middle-tier template.

The first thing output to the file is a standard block defined in the `FileOpen` template. Although I'm not including the contents of this template, its name gives a good idea of what it accomplishes—outputting the header comments, option statements, imports, and so on that appear at the top of the output file. Within the code of this file, the collection class inherits from the `CollectionBase` class and contains the output of two named templates. The row class inherits from a different base class and calls four named templates.

Building the XSLT Template on Your Own

You can build this template yourself by copying the entire target file (Orders.vb) into your new template. Replace the header with a call to a template named `FileOpen` that already exists in Support.xslt. If you want to build Support.xslt as well, copy the header into a template in your shell Support.xslt. You can clean that up later. Then place an `xsl:call-template` directive for each region in the high-level processing template. Give these templates names that match your region name, with spaces removed (you can either use capitalization, as I did, or use underscores). This is the structure of your high-level processing template. You hardly had to think to do it, and any two people within your workgroup would've created identical files.

Now create a named `xsl:template` for each `xsl:call-template` and copy the contents of the region into the corresponding template. Now you're ready to use `xsl:value-of` to replace values with ones retrieved from the XML input file and use `xsl:if`, `xsl:choose`, `xsl:for-each`, and `xsl:apply-templates` to provide logic to your templates.

Regions are important when you're debugging XSLT and brute-force code templates. The compiler will find a problem in your output (or you'll determine where a change needs to be made), and you have to trace this back into your template file. By far the easiest way to do this is using regions to organize both your source code and templates.

Once you've got named templates corresponding to each region in your output code, scan the high-level processing template for any values that wouldn't be the same for all output files. These are the items you need to replace using the `xsl:value-of` directive. In this case, there are two, both part of the class names. The original lines are as follows:

```
Public Class OrderCollection
...
Public Class Order
```

You can replace whole words or any part of words. Whether a name is singular or plural needs special attention. The metadata provides attributes for `Name`, `OriginalName`, `SingularName`, and `PluralName`. Although that may seem like a lot of names, it's really the only way to ensure you're getting the right name at the right location. (Table 2-2 shows the different names available for each table.) In this case, you want the singular name in both places:

```
Public Class <xsl:value-of select="@SingularName"/>Collection
...
Public Class <xsl:value-of select="@SingularName" />
```

Creating Named Templates

Named templates can contain simple or complex output. To create the first named template, create a new `xsl:template` within your stylesheet and copy the code for the first region into it. It'll look like this:

```
<xsl:template name="CollectionConstructors">
#Region "Constructors"
   Protected Sub New()
      MyBase.New("OrderCollection")
   End Sub
#End Region
</xsl:template>
```

Look for anything that wouldn't be valid in another output file—all the stuff that changes. The only changing value here is the text `OrderCollection`. You can replace this with an `xsl:value-of` directive supplying the `SingularName` attribute. The resulting template is as follows:

```
<xsl:template name="CollectionConstructors">
#Region "Constructors"
   Protected Sub New()
      MyBase.New("<xsl:value-of select="@SingularName" />Collection")
   End Sub
#End Region
</xsl:template>
```

Named templates aren't always this simple to create. When you copy in the `Public and Friend…` region of the Orders.vb file, you have this:

```
<xsl:template name="PublicAndFriend">
#Region "Public and Friend Properties, Methods and Events"
   Public Overloads Sub Fill( _
                 ByVal OrderID As System.Int32, _
                 ByVal UserID As Int32)
      OrderDataAccessor.Fill(Me, OrderID, UserID)
   End Sub
#End Region
</xsl:template>
```

You often need to know the intent of the various aspects of your sample file. In this case, `OrderID` is the primary key of the table, and the application passes a `UserID` to all `Fill` methods. This is complicated by the fact that Northwind doesn't contain a primary key for all tables.

NOTE I didn't see the problem with missing primary keys in my crystal ball. I created the template without worrying about this and found compiler errors when there weren't any primary keys.

TIP Code generation is easier if you use a single-column primary key on all tables.

Variables can simplify processing, but XSLT variables are weird critters. You assign a value when you create them, and then you can never change this value. That doesn't sound like a variable in any language I've used, but it's consistent with some basic principles of XSLT (discussed further in Appendix A). This `PublicAndFriend` template uses variables of two different types. The first is a string that contains the name of the primary key. An XPath statement retrieves the `Name` attribute of the `TableConstraint` element's `PrimaryKey` child element of the current node (the table). The context for this XSLT is the table node:

```
<xsl:template name="PublicAndFriend">
#Region "Public and Friend Properties, Methods and Events"<xsl:text/>
   <xsl:variable name="primarykeyname"
            select="dbs:TableConstraints/dbs:PrimaryKey/dbs:PKField/@Name" />
```

To create a corresponding method declaration, you need the type, as well as the name, of the primary key. To find this, you can use the name of the primary key, now held in the `$primarykeyname` variable, to retrieve the matching node within the column definitions (see the XML metadata in Listing 3-1). This code assigns a single node to the `$primarykey` variable. This node is the `TableColumn` node corresponding to the primary key (context is the table node):

```
<xsl:variable name="primarykey"
         select="dbs:TableColumns/dbs:TableColumn[@Name=$primarykeyname]" />
```

NOTE This template assumes you'll use single-column keys and ignores any additional keys. You'll learn later how to deal with complex primary keys.

TIP Use the column node when you need any details on the column, such as its type.

The table may have no primary key, and in that case, the class needs a `Fill` method with only the `UserID` parameter. The directives between `xsl:if` and `/xsl:if` execute only if there are any primary key nodes. When this block executes, output from this block includes the name and type of the primary key:

```
Public Overloads Sub Fill( _<xsl:if test="$primarykey">
          ByVal <xsl:value-of select="$primarykey/@Name"/> As <xsl:text/>
         <xsl:value-of select="$primarykey/@NETType"/>, _</xsl:if>
          ByVal UserID As Int32)
```

NOTE When testing a variable containing a set of nodes, the test results in `true` if there are any nodes in the set.

Outputting the call to the `Fill` method presents similar issues and solutions. The `SingularName` attribute of the table node makes up part of the name of the data access layer class. The `xsl:if` directive is again used to include the primary key field name as a parameter only if it exists:

```
           <xsl:value-of select="@SingularName"/>DataAccessor.Fill(Me<xsl:text/>
            <xsl:if test="$primarykey">, <xsl:value-of
select="$primarykey/@Name"/>
            </xsl:if>, UserID)
   End Sub
#End Region
</xsl:template>
```

Managing Whitespace

This template also provides good examples of whitespace management. Unfortunately, you have to think about whitespace when you write XSLT templates. You might have thought I was careless (or downright dumb) in the layout of the XSLT directives in the previous section, but this layout provides the correct whitespace in the output.

Whitespace that's output is called *significant whitespace*. Whitespace is significant, meaning that it's output, when the whitespace is adjacent to any text that's output. When any two XSLT directives are adjacent to each other with only whitespace between, the processor assumes that intervening whitespace was for your convenience in reading the template and ignores it. That's called *insignificant whitespace*. Put another way, only whitespace that isn't between XSLT directives is output.

NOTE The meaning of whitespace is different in the generated VB .NET and C# code. In VB .NET, the type of whitespace often has meaning, particularly the Carriage Return/Line Feed (CRLF) character at the end of each line, the spaces before line continuation characters, and so on. C# has few places where the type of whitespace has semantic meaning for the compiler. However, programmers reading the code will appreciate your efforts at whitespace management in either language.

Because of whitespace behavior, if you write a method declaration such as this:

```
Public Overloads Sub Fill( _
        <xsl:if test="$primarykey">
         ByVal <xsl:value-of select="$primarykey/@Name"/> As System.Int32,
        </xsl:if>
```

then two new lines (two CRLF pairs) will be output, one after the underscore and the other before the `ByVal`. The result would be this:

```
Public Overloads Sub Fill( _

         ByVal OrdersID" As System.Int32
```

with the blank line causing a VB .NET compiler error. Take another look at the template lines if you don't see why this problem would occur.

Sometimes you want to break a line in the template at a location illegal in the output, such as this example. I could just toss in another line-continuation underscore, but that would output ugly code, and I want output to be easy to read, without artifacts of code generation. In the earlier example, I swallowed my desire to put the `xsl:if` test on a separate line. Alternatively, I could've tossed in an empty `xsl:text` directive, which is simply a meaningless XSLT tag in this

context. The `xsl:text` directive can appear at either the end or beginning of the next line, depending on which is needed to wrap the offending whitespace between two XSLT directives. In this case, you can put it either after the underscore or before the `ByVal`, but not both:

```
Public Overloads Sub Fill( _ <xsl:text/>
        <xsl:if test="$primarykey">
         ByVal <xsl:value-of select="$primarykey/@Name"/> As System.Int32</xsl:if>
```

TIP Whether you move directives onto the ends of lines such as this approach or use `xsl:text` at the start of lines is sometimes matter of style. But in other cases, conditional and looping constructs dictate where you place the `xsl:text` element.

The XSLT processor generally ignores whitespace within directives. I have the extra demands of making text readable on this page because you can't scroll the page if the text gets too wide. This results in extra `xsl:text` elements in the book's code, but it also gives me a chance to clarify where you can wrap within directives. You can generally split anywhere in a directive other than in a quoted string or within an element or attribute name, such as:

```
<xsl:variable name="primarykeyname"
        select="dbs:TableConstraints/dbs:PrimaryKey/dbs:PKField/@Name" />
```

Sometimes you'll need to output whitespace between directives. You might need to add tabs, spaces, or new line characters. These need to be escaped character sequences placed within an `xsl:text` directive. Appendix A shows examples of adding whitespace using escape characters. Support.xslt also contains a template that outputs new line characters.

XSLT whitespace is a pain. But as you become familiar with it, you'll find it workable, even if your XSLT templates will never have the grace of alignment you expect from your generated files.

TIP Solve whitespace issues so your generated code is logically aligned. This will save lots of problems in reading your code later.

Using Looping, Conditional Variables, and Other Fun Stuff

The `ClassLevelDeclaration` named template illustrates a couple of new things. The code target starts with this:

```
#Region "Class Level Declarations"
   Protected mCollection As OrderCollection
   Private Shared mNextPrimaryKey As Int32 = -1

   Private mOrderID As System.Int32
```

This goes on with a full list of fields that correspond to the columns in the table.

The first three lines of the template that outputs this code uses techniques you've already seen. You can output the list of the fields using an `xsl:for-each` directive. The `xsl:for-each` directive loops through each node in the specified set and outputs the class-level declarations:

```
<xsl:template name="ClassLevelDeclarations">
#Region "Class Level Declarations"
   Protected mCollection As <xsl:value-of select='@SingularName'/>Collection
   Private Shared mNextPrimaryKey As Int32 = -1
   <xsl:for-each select="dbs:TableColumns/*">
      <xsl:if test="string-length(@NETType)>0">
   Private m<xsl:value-of select="@Name" />
            <xsl:text/> As <xsl:value-of select="@NETType" />
      </xsl:if>
   </xsl:for-each>
#End Region
</xsl:template>
```

NOTE In every case where an `xsl:for-each` directive works, an `xsl:apply-template` with a mode also works, and vice versa. In your favorite programming language, you gained confidence in your proficiency at separating parts into subroutines. Similarly, you'll quickly become confident in determining whether to use an `xsl:for-each` or an `xsl:apply-templates`. If you avoid `xsl:apply-templates`, you'll build difficult-to-read XSLT templates with deeply nested loops. But, too many templates containing only one or two lines can also be difficult to read. As a rough guideline, if the match template would contain fewer than three lines, or the calling template would be reduced to fewer than about five lines, I use `xsl:for-each`. Otherwise, I use `xsl:apply-templates`.

The `xsl:if` handles the case where the metadata doesn't include a `NETType` attribute. The `NETType` attribute is derived from the SQL type during metadata extraction, and the extraction process doesn't yet support some SQL types such as `image` and `varbinary` types that exist in the Northwind database. They don't support them because I don't anticipate displaying employee pictures. Because the metadata contains empty strings for these unsupported types, the generated source code would result in .NET compiler errors. Again, my crystal ball didn't tell me this. I ran the template, found complier errors, and tracked them back to this problem.

Creating Match Templates

Match templates provide a mechanism to segment your XSLT stylesheet and simultaneously loop through a set of nodes. This is very handy. Using match templates instead of `xsl:for-each` directives will not only show the world that you're a competent XSLT jock but also will result in cleaner, more readable stylesheets.

The Orders.vb file contains a *`columnName`*`ColumnInfo` method and a *`columnName`* property for each column in the table. This function and property are shown near the end of Listing 3-2. The *`columnName`*`ColumnInfo` method returns an object containing information about the column. This type of information is frequently helpful to UI programmers using the data container class. The `columnName` property wraps the actual data.

Building a match template is much like building a named template. The key differences are that named templates run only once each time they're called and don't change the context within the XML metadata input file. Match templates, on the other hand, may run zero, one, or many times each time they're called. Match templates execute in the context of the selected node—the current member of the node list. This is similar to the `xsl:for-each` directive's behavior.

To build this template, copy the method and property blocks for any one of the columns from the target file into the template and identify the changeable items. These changeable items are retrieved from column's information in the metadata file, which is the current context when the output file is generated.

This template handles the problem with missing .NET types a little differently. Rather than skipping the code, it outputs explanatory comments within an `xsl:choose` directive if the `NETType` attribute is missing or empty:

```
<xsl:template match="dbs:TableColumn" mode="ColumnMethods" >
   <xsl:choose>
      <xsl:when test="string-length(@NETType)=0">
      ' TODO: Column <xsl:value-of select="@Name"/> is not included because it uses
```

```
        ' a SQLType (<xsl:value-of select="@SQLType"/>) that is not yet supported
        </xsl:when>
        <xsl:otherwise>
    Public Function <xsl:value-of select="@Name"/>
             <xsl:text/>ColumnInfo As ColumnInfo
        Dim columnInfo As New ColumnInfo
        columnInfo.FieldName = "<xsl:value-of select="@Name"/>"
        columnInfo.FieldType = GetType(</xsl:text><xsl:value-of select="@NETType"/>)
        columnInfo.SQLType = "<xsl:value-of select="@SQLType"/>"
        columnInfo.Caption = "<xsl:value-of select="@Caption"/>"
        columnInfo.Desc = "<xsl:value-of select="@Desc"/>"
        Return columnInfo
    End Function

    Public Property <xsl:value-of select="@Name"/> As <xsl:text/>
                <xsl:value-of select="@NETType"/><xsl:call-template name="NewLine"/>
        Get
            Return m<xsl:value-of select="@Name"/><xsl:call-template name="NewLine"/>
        End Get
        Set(ByVal Value As <xsl:value-of select="@NETType"/>)
            m<xsl:value-of select="@Name"/> = Value
        End Set
    End Property
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

NOTE The XPath expressions used by the `xsl:value-of` directive allows you to access any piece of information in the XML metadata file, even if it's nowhere close to the node you're currently processing. If you manage to find a problem you can't solve with XPath, Microsoft's implementation of XSLT supports both script and calling back to .NET objects. The sky is the limit of what you can do with XSLT—but it won't always be easy.

Supporting Stylesheets

Supporting stylesheets allow you to isolate generic templates that you can reuse in different XSLT stylesheets. For example, the SimpleDataContainer.xslt stylesheet uses the `NewLine` template in the Support.xslt file. Using this template doesn't save

much typing, but it keeps you (and later programmers) from having to remember the hexadecimal values of the carriage-return and line-feed characters:

```
<xsl:template name="NewLine">
   <xsl:text>&#x0d;&#x0a;</xsl:text>
</xsl:template>
```

As another example, you'll frequently need to control commas in lists. This is an example of when you might need to output this:

```
{0, 1, 2}
```

You can control the comma at the beginning or the end of the list, but the `CommaIfNotFirst` template is theoretically more efficient than `CommaIfNotLast`:

```
<xsl:template name="CommaIfNotFirst">
   <xsl:if test="position()!=1">, </xsl:if>
</xsl:template>

<xsl:template name="CommaIfNotLast">
   <xsl:if test="position()!=last()">, </xsl:if>
</xsl:template>
```

NOTE XPath has a number of useful functions. If you have trouble finding references to them in the .NET Help, look up the "xslt, reference" section in the index. This will get you into the fairly good XSLT documentation. Appendix A also discusses XPath functions.

The SimpleDataContainer.xslt stylesheet uses the `FileOpen` template in Support.xslt to supply a consistent header at the top of the output files, which you can adjust to your liking. When you make a change, the change will appear in all stylesheets importing or including Support.xslt. This template needs to run as part of a stylesheet that has the `fileName` stylesheet parameter or variable. If not run with this available, you'll get XSLT errors. If you aren't sure this parameter will be available, explicitly pass it as a template parameter because not passing a template parameter isn't considered an error. Accessing a variable that doesn't exist, however, is an XSLT error.

The `FileOpen` template outputs standard `Option Strict`, `Option Explicit`, and `Imports System` statements into your output file's header. The `imports` parameter passes additional imports as a comma-delimited string. The named template

RecursiveImports breaks this string down and outputs the Imports statement. The template then outputs a comment block containing the filename:

```
<xsl:template name="FileOpen">
   <xsl:param name="imports" />
Option Strict On
Option Explicit On

Imports System
<xsl:call-template name="RecursiveImports">
   <xsl:with-param name="imports" select="normalize-space($imports)"/>
</xsl:call-template>
#Region "Description"
' <xsl:call-template name="StripPath">
     <xsl:with-param name="fname" select="$fileName" />
 </xsl:call-template>
#End Region
</xsl:template>
```

TIP You could include the generation date and time, but that would cause source control to always see the file as new and unnecessarily mark the file as changed. Because of this complication, I suggest you don't include the date and time in the header.

Using Recursive Templates

How do you break down a comma-delimited string if you can't reassign the values of variables? You use a recursive template that creates a new variable every time you call it. This is an advanced aspect of XSLT that can be handy at times. String manipulation such as the previous two samples is a convenient place to use recursive templates. The RecursiveImports template outputs an Import statement for each namespace in a comma-delimited list:

```
<xsl:template name="RecursiveImports">
   <xsl:param name="imports"/>
   <xsl:variable name="remaining" select="substring-after($imports,',')"/>
   <xsl:choose>
   <xsl:when test="string-length($remaining) > 0">
```

```
Imports <xsl:value-of select="normalize-space(substring-before($imports,','))"/>
      <xsl:call-template name="RecursiveImports">
         <xsl:with-param name="imports" select="$remaining"/>
      </xsl:call-template>
   </xsl:when>
   <xsl:otherwise>
Imports <xsl:value-of select="normalize-space($imports)"/>
   </xsl:otherwise>
   </xsl:choose>
</xsl:template>
```

TIP I included this example so you have a reference of when you need recursion. Don't worry if you get a little lost in it—you don't need to use recursion often.

The `imports` parameter contains the comma-delimited string. The template creates a new variable named `remaining` that contains everything after the first comma. An `xsl:choose` directive tests whether the `remaining` variable contains anything. The `remaining` variable would be empty if the `imports` parameter ended with a comma or didn't contain a comma. If the `remaining` variable isn't zero length, the template outputs an `Imports` statement with the contents before the comma. The `normalize-space` XPath function is basically the same as the .NET `String` class's `Trim` method. After outputting the `Imports` statement, the template calls itself, passing the contents of the `$remaining` variable as the `imports` parameter. This effectively calls the `RecursiveImports` for every item in the comma-delimited string, with the `imports` parameter chopped off at the first comma each time. The last time it's called, the `imports` parameter contains no comma, the `xsl:otherwise` directive is processed, and the template simply outputs the `Imports` statement.

The trick to creating recursive templates (or using recursion in any language) is to provide a clear end to the recursion. In this case, you chop off string elements until there's no comma left. If you fail to provide this clear end point, the recursion is endless, or unbounded.

TIP When processing XSLT in .NET, if you encounter endless recursion, your application will appear to freeze up (the action will take significantly too long). If the debugger is running, you can break using Ctrl+Break and end the process.

You can use the same approach to strip a path off a full filename, resulting in just the filename. In the file header, the name of the file is useful. The directory where you generate it may not be its permanent location, so it isn't included. The `StripPath` template is otherwise similar to the `RecursiveImports` template:

```
<xsl:template name="StripPath">
   <xsl:param name="fname"/>
   <xsl:variable name="remaining" select="substring-after($fname,'&#x5c;')"/>
   <xsl:choose>
   <xsl:when test="string-length($remaining) > 0">
      <xsl:call-template name="StripPath">
         <xsl:with-param name="fname" select="$remaining"/>
      </xsl:call-template>
   </xsl:when>
   <xsl:otherwise>
      <xsl:value-of select="$fname"/>
   </xsl:otherwise>
   </xsl:choose>
</xsl:template>
```

NOTE `\` is the escape sequence for the backslash character. You have to escape the backslash in at least some cases.

Capping Off XSLT Code Generation

The previous sections took XSLT code generation out of the theoretical to show how you can use it in the real world with advanced XSLT techniques. To make XSLT code generation easy, you'll want to use the following:

- A single entry-level template accessing the root
- A single high-level processing template called in the context you're processing
- A named template (`xsl:call-template`) for each region in the output after you organize your sample code into regions
- Additional nested named and match templates (`xsl:apply-templates`) as regions become longer and more complex
- `xsl:value-of` for inserting values

- `xsl:if` when a conditional block has only one option and `xsl:choose` when there are multiple conditional blocks
- Match templates for any looping elements unless the output is nearly trivial, in which case you should use an `xsl:for-each`
- A separate supporting stylesheet to contain reusable utility templates

You'll see more samples of XSLT code generation throughout the book. I use it for examples because it's more concise than the other code generation mechanisms.

Exploring Details of Brute-Force Generation

Brute-force code generation is the simplest way to output code. It just spits out code to an open stream or writer. Because there are many variations on streams, there are different approaches to doing brute-force code generation. The approach I'll demonstrate uses the same metadata input file used for XSLT code generation. Instead of transforming it with XSLT, the metadata is loaded into the XML Document Object Model (DOM) where you can manipulate it for code generation.

TIP Brute-force generation doesn't impose a high degree of order on your code generation—it's easy to be sloppy. Moving from manual development to the more abstract approach of code generation can be a big step, so don't make it more difficult with obscure or inconsistent techniques.

NOTE I'll outline one approach in detail here. If you're using a variation of brute force, use this section for comparison.

All three of the code generation mechanisms rely on streams when run through the code generation harness. This allows hashing tools to access the output stream and insert a hash marker so you can later tell if the file has been edited manually. Other than hashing, XSLT and CodeDOM code generation don't use readers and writers. They use other .NET tools for their processing. Brute-force code generation outputs code directly to a stream, so you'll have to be familiar with writers. Readers aren't important in this context because you're writing to a stream, not reading from one. See the "Understanding File Streams" sidebar for more information on working with streams.

Understanding File Streams

For an understanding of streams, maybe a good place to start is Help for the `Stream` class. The description of the class is that it "provides a generic view of a sequence of bytes."[3]

That reminds me of a joke: A pilot lost in a plane low on fuel with limited visibility shouts to a guy on the roof of a building to ask, "Where am I?" The guy on the building says, "You're in a plane." The pilot flies around and lands at the airport. His terrified passenger asks how he found the airport. "That answer was absolutely correct and not terribly helpful, so I figured that was Redmond."[4]

Anyway, further exploration of streams shows this description of streams to be absolutely correct, even if it doesn't seem terribly helpful at first glance. Data—all the data you work with—is ultimately a bunch of bytes. These bytes might be in a file, an explicit memory buffer, a string, your console, and so on. The actual storage of the bytes is called the *backing store,* and regardless of which backing store you use, it's still just a bunch of bytes.

Streams are pipes into the backing store. Streams are named in terms of the backing store they lead to, such as `FileStream` or `MemoryStream`. Streams are pipes, so they don't themselves hold anything—although in some cases, such as the `MemoryStream`, the backing store is somewhat hidden behind the stream. Exactly what you stick into a stream depends on the encoding it supports. Like real-world pipes, a stream can generally transfer information in either direction.

.NET also provides fittings for the end of the pipes called *readers* and *writers*. These fittings are specific to either reading or writing, like a spigot with a backflow fitting. Readers and writers don't depend on where the data is going—in other words, they don't care what's on the other end of the pipe. Readers and writers provide special features you can use to get data pushed into or pulled out of the stream. They provide specialized syntax for the type of data you're converting to or from bytes. This can be as simple as Unicode encoding or as complex as an interface that understands HTML (such as the `System.Web.UI.HTMLTextWriter`). For XSLT and CodeDOM generation, the stream is just returned from the process and passed on to the hash tools.

NOTE There's more on using hashing code output in Chapter 5.

The hash tools create a reader and a writer to work with the stream. Because it's inserting something into the stream, it reads from one stream and outputs to

3. .NET 2003 Help: Description of the `Stream` class.

4. I like the joke, but the Microsoft documentation actually does a decent job of covering many parts of the complex .NET Framework.

another. This is analogous to adding dye to a stream. You'd run water from the tap into some sort of a mixing bucket, and then you'd pour it into another pipe leading to what you were doing with the dyed liquid. You wouldn't try to stuff the dyed liquid back into the pipe from which the clear liquid came. The full `ApplyHash` routine is as follows:

```
Public Shared Function ApplyHash( _
                ByVal inStream As IO.Stream, _
                ByVal commentText As String, _
                ByVal commentStart As String, _
                ByVal commentEnd As String) _
                As IO.Stream
   Dim s As String
   Dim reader As New IO.StreamReader(inStream)
   Dim writer As New IO.StreamWriter(New IO.MemoryStream)
   Dim hashstring As String
   Dim fullHeaderMarker As String = commentStart & HeaderMarker & commentEnd

   inStream.Seek(0, IO.SeekOrigin.Begin)
   s = StripHeader(reader.ReadToEnd, fullHeaderMarker)
   hashstring = CreateHash(s)

   writer.WriteLine(fullHeaderMarker)
   writer.WriteLine(commentStart & commentEnd)
   writer.WriteLine(commentStart & commentText & commentEnd)
   writer.WriteLine(commentStart & commentEnd)
   writer.WriteLine(commentStart & HashMarker & hashstring & HashMarker & _
                    commentEnd)
   writer.WriteLine(fullHeaderMarker)
   writer.Write(s)
   writer.Flush()

   writer.BaseStream.Seek(0, IO.SeekOrigin.Begin)
   Return writer.BaseStream
End Function
```

This function also shows how you can access the underlying stream for a reader or writer and how you can use the `Seek` method to reset the position within the file.

Initially streams might seem nonintuitive, but they provide a granular approach to byte processing. In the `ApplyHash` routine, it makes no difference what type of stream the data comes from. Another version of this routine could take the output stream as a parameter, making the task of applying the hash independent of the backing store for the data coming in or passed back out.

Using the IndentTextWriter

Although there are numerous writers within .NET that you could use for brute-force code generation, the best one is snuggled inside the `CodeDOM.Compiler` namespace. The `IndentTextWriter` manages indenting spaces for you. You can easily increase and decrease the indent of the output code using the `Indent` property:

```
Dim stream As New IO.MemoryStream
Dim inwriter As New CodeDom.Compiler.IndentedTextWriter( _
            New IO.StreamWriter(stream))
inwriter.Indent += 1
```

You can also skip indenting with the `WriteLineNoTabs` method:

```
inWriter.WriteLineNoTabs("#Region " & DQ & regionName & DQ)
```

TIP You'll be outputting a lot of double quotes; creating a constant for them with a short name such as `DQ` will save some typing.

NOTE When you concatenate to strings as part of your code generation, consider using a `StringBuilder` rather than just concatenating pieces onto a string. .NET strings have a lot of cool qualities, but they require copying on every assignment. If you're doing more than about six assignments that concatenate (multiple concatenations within a single assignment have no performance hit), you'll gain performance if you append using a string builder. This performance advantage becomes significant with more than a few dozen assignments.

Creating a Class

Creating a class using brute-force code generation has similarities with the XSLT approach to code generation. In both cases, you can use regions to organize both your output file and your template, and you can supply a different method to output each region. You start with an entry-level method that's the rough equivalent to the XSLT entry-level template.

Creating the Entry-Level Method

The entry-level method is specified in the `Process` attribute of the directive in the harness script file. The harness calls this method and expects it to generate a stream containing the output code. The entry-level template has to be a shared (C# static) method. In this case, the entry-level method is `GetStream`.

The `GetStream` method first creates a `MemoryStream` for output, then an `IndentedTextWriter`, and some other variables:

```
Public Shared Function GetStream( _
                ByVal Name As String, _
                ByVal fileName As String, _
                ByVal genDateTime As String, _
                ByVal nodeSelect As Xml.XmlNode) _
                As IO.Stream
   Dim stream As New IO.MemoryStream
   Dim inwriter As New CodeDom.Compiler.IndentedTextWriter( _
                New IO.StreamWriter(stream))
   Dim nodeList As Xml.XmlNodeList
   Dim nodeColumn As Xml.XmlNode
```

`GetAttributeOrEmpty` is a generic utility method that returns the attribute specified or an empty string if the attribute isn't present in the XML DOM element. You'll need a namespace manager when you reference a namespace (via a prefix) in the input XML. The namespace would generally be referenced with a prefix in arguments passed to the `SelectSingleNode` or `SelectNode` methods of the XML DOM. A utility method in the `Tools` class creates the namespace manager:

```
Dim singularName As String = Utility.Tools.GetAttributeOrEmpty(nodeSelect, _
            "SingularName")
Dim nsmgr As Xml.XmlNamespaceManager = _
            Utility.Tools.BuildNameSpaceManager( _
                  nodeSelect.OwnerDocument, "dbs", False)
```

CAUTION Namespaces can be hard to handle, especially in .NET, which is unfriendly to the default namespace (as discussed in Appendix A).

The `GetStream` method then calls the `FileOpen` method contained in a separate support file. Placing this method in a central location allows reuse by other

brute-force templates. Outputting an empty string using the `WriteLine` method produces a blank line:

```
Support.FileOpen(inwriter, "KADGen,System", fileName, genDateTime)
inwriter.WriteLine("")
```

Another supporting method, `WriteLineAndIndent`, outputs a line of text and then indents the writer. Changeable information is inserted either as a variable or as a direct lookup of information in the metadata XML. The following outputs the class declaration using a variable (`singularName` is declared previously):

```
Support.WriteLineAndIndent(inwriter, "Public Class " & singularName & _
             "Collection")
inwriter.WriteLine("Inherits CollectionBase")
```

Similar to the XSLT code generation, it's a lot easier to debug if you call a method for each region. The collection class contains two regions so the high-level method calls two other methods to output the core of the class:

```
CollectionConstructors(inwriter, nsmgr, nodeSelect)
PublicAndFriend(inwriter, nsmgr, nodeSelect)
Support.WriteLineAndOutdent(inwriter, "End Class")
```

The `WriteLineAndOutdent` method is similar to the `WriteLineAndIndent` method, and it decreases the indent by one and then outputs the specified text.

Outputting the row class is much like outputting the collection class. The next code uses a `For Each` construct to loop through the columns and calls a method for each column:

```
inwriter.WriteLine("")
inwriter.WriteLine("")
Support.WriteLineAndIndent(inwriter, "Public Class " & singularName)
inwriter.WriteLine("Inherits RowBase")
ClassLevelDeclarations(inwriter, nsmgr, nodeSelect)
Constructors(inwriter, nsmgr, nodeSelect)
BaseClassImplementation(inwriter, nsmgr, nodeSelect)
Support.MakeRegion(inwriter, "Field access properties")
nodeList = nodeSelect.SelectNodes("dbs:TableColumns/*", nsmgr)
For Each nodeColumn In nodeList
   ColumnMethods(inwriter, nsmgr, nodeColumn)
Next
Support.EndRegion(inwriter)
      Support.WriteLineAndIndent(inwriter, "End Class")
```

Flushing the stream after you've finished output ensures that the stream has emptied any buffers and all information is safely in the stream before the writer goes out of scope. Not all writers need to be flushed, but it's a great habit to be in because it's essential with many of them:

```
   inwriter.Flush()
   Return stream
End Function
```

Looking at a Sample Region

I'll skip over the `Constructors` region method because it doesn't present anything new and instead will walk through one of the more complex region methods. Just like the XSLT templates, the `PublicAndFriend` region outputs the primary key only if it exists. Passing the namespace manager parameter means you don't have to re-create it if methods need it for any XML processing:

```
Private Shared Sub PublicAndFriend( _
                ByVal inWriter As CodeDom.Compiler.IndentedTextWriter, _
                ByVal nsmgr As Xml.XmlNamespaceManager, _
                ByVal node As Xml.XmlNode)
   Support.MakeRegion(inWriter, _
                "Public and Friend Properties, Methods and Events")
```

The `SelectSingleNode` statement retrieves the primary key element of the `TableConstraints` element that's stored in the `nodeTemp` variable. If this node is found, the value of its `Name` attribute is assigned to the `primaryKeyName` variable:

```
Dim nodeTemp As Xml.XmlNode = node.SelectSingleNode( _
             "dbs:TableConstraints/dbs:PrimaryKey/dbs:PKField", nsmgr)
Dim primaryKeyName As String = ""
Dim primaryKey As Xml.XmlNode
If Not nodeTemp Is Nothing Then
   primaryKeyName = Utility.Tools.GetAttributeOrEmpty(nodeTemp, "Name")
End If
```

This XPath expression in the next `SelectSingleNode` uses the `primaryKeyName` to retrieve the `TableColumn` node with a `Name` attribute matching the `primaryKeyName`:

```
primaryKey = node.SelectSingleNode( _
             "dbs:TableColumns/dbs:TableColumn[@Name='" & primaryKeyName & _
             "']", nsmgr)
Support.WriteLineAndIndent(inWriter, "Public Overloads Sub Fill( _")
inWriter.Indent += 4
```

Once you've got the `primaryKey` node, you can use it in a simple conditional to output a parameter for the primary key value only if there's a primary key. You can use the `Write` method to output a partial line without a CRLF:

```
If Not primaryKey Is Nothing Then
   inWriter.Write("ByVal " & primaryKeyName & " As ")
   inWriter.WriteLine(Utility.Tools.GetAttributeOrEmpty(primaryKey, _
             "NETType") & ", _")
End If
```

The remainder of the function reuses the same techniques you've already seen:

```
   inWriter.WriteLine("ByVal UserID As Int32)")
   inWriter.Indent -= 4
   inWriter.WriteLine("ByVal UserID As Int32)")
   inWriter.Write(Utility.Tools.GetAttributeOrEmpty(node, "SingularName") & _
                  "DataAccessor.Fill(Me")
   If Not primaryKey Is Nothing Then
      inWriter.Write(", " & primaryKeyName)
   End If
   inWriter.WriteLine(", UserID)")
   Support.WriteLineAndOutdent(inWriter, "End Sub")
   Support.EndRegion(inWriter)
End Sub
```

Creating the Support Template

The `FileOpen` method outputs the same file header as the XSLT code generation template. You can see additional supporting methods and the template code that outputs the remainder of the sample in the download from the Web site. The `fileName` and `genDateTime` are explicitly passed as parameters. The `Split` method of the framework splits the `import` parameter at commas to output separate `Imports` statements and uses a framework method to retrieve the filename from the full path:

```
Public Shared Sub FileOpen( _
                ByVal inWriter As CodeDom.Compiler.IndentedTextWriter, _
                ByVal import As String, _
                ByVal fileName As String, _
                ByVal genDateTime As String)
   inWriter.WriteLine("Option Strict On")
```

```
    inWriter.WriteLine("Option Explicit On")
    inWriter.WriteLine("")
    inWriter.WriteLine("Imports System")
    For Each s As String In import.Split(","c)
       inWriter.WriteLine("Imports " & s)
    Next
    MakeRegion(inWriter, "Description")
    inWriter.WriteLine("'")
    inWriter.WriteLine("'" & IO.Path.GetFileName(fileName))
    inWriter.WriteLine("' Last Genned on Date: " & genDateTime)
    inWriter.WriteLine("'")
    inWriter.WriteLine("'")
    EndRegion(inWriter)
End Sub
```

Winding Up Brute-Force Code Generation

That's it for brute-force code generation. You can extend these ideas to build any code you want. The key to success is breaking the process down into discrete steps as follows:

- Break code outputting away from metadata extraction.
- Split outputting into individual template classes that each output a single file pattern.
- Within each template class, use an entry-level method.
- Use a different method for each region.
- Use additional methods as regions become longer and more complex.
- Use supporting methods for reuse wherever possible.

Exploring Details of the CodeDOM

CodeDOM generation is interesting for a number of reasons. It's a full abstraction of the nature of programming languages. In other words, it breaks code down into the parts of byte speech—the grammar of your programs. When you

write code to this abstraction, a .NET or third-party provider[5] can build output code in any target language. It's also cool that Microsoft provided something developed for its own use. Visual Studio uses CodeDOM generation for the strongly typed DataSet, the class created for each ASPX file, the `CodeDomSerilizer`, and a few other purposes. If you look at the problems that arise in outputting code for entirely different languages, it's an amazing piece of software in itself.

NOTE The `CodeDOMSerializer` also uses the CodeDOM. So, if you're working with `CodeDOMSerializer`, everything in this section applies, and you may find this tutorial helpful. I'm not discussing details of the `CodeDOMSerializer` because it isn't related to application-wide code generation. Instead, it allows you to control the code produced by Visual Studio when it creates the region labeled Windows Form Designer generated region in your form. If you're interested in using this, you can find articles on the Web that discuss how to tie it to your class using attributes and other unique aspects of convincing Visual Studio to generate your code correctly. Search for CodeDOMSerializer at MSDN or on Google to find more information.

The CodeDOM has the narrow purpose of outputting code in any language that has a language provider, and Microsoft supplies providers for C#, VB .NET, and J#.[6] Unfortunately, the effort you have to go through to achieve this flexibility—the real cost to you—is high. For almost all applications, settle on a language and use one of the other mechanisms for code generation.

NOTE You might not want to flip past this section just yet. If you've never chopped up code to see it in the perspective of its atomic grammar units—the adverbs, nouns, and adjectives that make up the grammar of common code—it's a useful exercise to explore the CodeDOM. You'll get past differences such as VB .NET's declaring variables with the name followed by the type versus C# declaring the type followed by the name (`Dim iVar As System.Int32` versus `System.Int32 iVar`). You'll also discover some of the things your language does for you that exceed interpreting syntax because those things don't work well via the CodeDOM. Universities often require a compiler course for Computer Science (CS) graduates so they understand the underlying programming "parts of speech" and how they're put together different ways in different languages. Maybe it'll help you find a deeper Zen connection with your code. On the other hand, there's a chance that you'll never use the information in the rest of this chapter.

5. There aren't any third-party providers yet, but the CodeDOM is designed for new languages to be incorporated into it.

6. If you're generating code for J#, please read the "CodeDOM, For Java Language" section in the .NET Help. Among a few other details, you have to reference the language-specific compiler.

In this section, you'll find out just how ugly it can be to output via the CodeDOM. If you're thinking about using the CodeDOM for code generation, you'll want to be realistic about how much work it takes to use the CodeDOM and the limitations on what you can accomplish. You'll spend considerably more than twice as much time writing and maintaining the code, compared to the other code generation mechanisms. You'll see a payoff only when you need to maintain absolute symmetry between your C# and VB .NET output or when maintaining parallel testing is impractical. If that's the case, you'll want to use the CodeDOM, and you'll find a real shortage of information about it. Because of this void, this is a core tutorial to understanding the underlying technology. This basic understanding will open up the CodeDOM Help to you, which in most cases isn't half bad once you know what you need. I won't walk through what it takes to create a full class, but you'll see enough basics to work through building one if you decide to continue with the CodeDOM.

NOTE CodeDOM creates plain-vanilla code based on common syntax. Each compiler does a few back flips to make your life as a developer easier. The CodeDOM supports basic syntax, not these extra features. For example, the CodeDOM doesn't support the `Handles` clause of VB .NET or its supporting `WithEvents`. Appendix D covers VB .NET and C# features that you can't do in the CodeDOM. Much of the missing stuff is specific to one of the compilers and not supported by the other. If you need something other than this plain-vanilla code, you'll have to extend the CodeDOM, which isn't for the faint of heart (or you'll need to use snippets, which are a really nasty construct).

NOTE CodeDOM is a write-only technology. If you want to read the structure of existing code, you'll need to parse the code or explore the automation model of Visual Studio .NET. Named the `FileCodeModel`, the Visual Studio automation model resides in the `EnvDTE` namespace. For C#, the `FileCodeModel` is read/write, but it's read-only for VB .NET (the current trade-off for some cool VB–-specific development environment features).

Like the other code generation approaches, there's no certainty that the code you output will compile. Although the nature of the CodeDOM means certain types of gross syntax errors aren't going to happen, the CodeDOM leaves you plenty of room to output code that won't compile in one or more languages or won't do what you intended when it compiles.

CAUTION The CodeDOM will allow you to do dumb things without any complaint. Because of the variety of invalid code it'll happily generate, test your output by compiling in multiple compilers. It's not a bad idea to do a code review of a few output files with any of the code generation techniques, but this is especially important with the CodeDOM.

Using an Object Hierarchy

At the root of the object model is the *CodeDOM graph*. The CodeDOM graph is a tree of objects that describes elements of your code in abstract terms. The graph is contained in an object called a `CodeCompileUnit`, which describes your code in terms of its elemental parts—expressions, statements, members, and so on. Figure 3-2 shows the tree approach of the CodeDOM graph. Each CodeDOM graph outputs one file, but it can contain multiple classes. Using multiple CodeDOM providers, you can create output for this code in multiple languages.

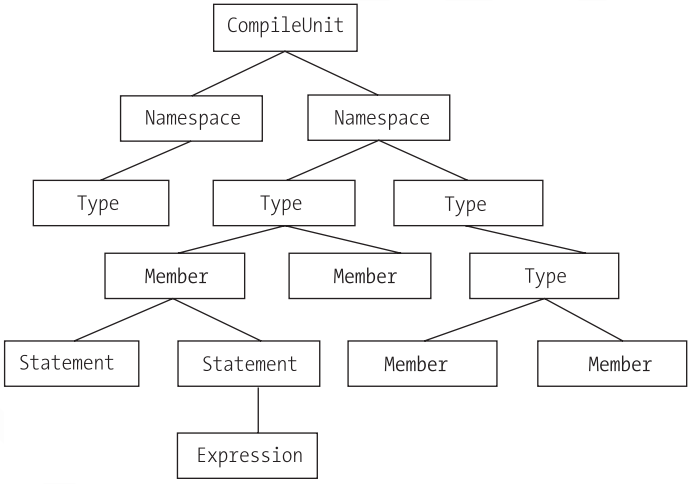


Figure 3-2. The CodeDOM is a tree of code elements.

NOTE It's a little confusing that the terms CodeDOM graph and CodeCompileUnit are so closely related. To be precise, the CodeDOM graph is the abstract hierarchy of code elements, and the `CodeCompileUnit` is the specific class whose objects contain this hierarchy. But in practice, you can probably treat these terms as being synonymous.

NOTE The CodeDOM parallels reflection in that they both work with abstractions of your code. But the CodeDOM is primarily a mechanism for creating and outputting code, and reflection is primarily a mechanism for retrieving information about an existing assembly. Reflection is also assembly focused, but the CodeDOM is organized in terms of .NET namespaces—with a collection of namespaces making up the CodeDOM graph.

Figure 3-3 shows key elements of the CodeDOM. Namespaces contain type declarations, types contain members, and so on. I excluded a lot of information to keep this figure simple. The interesting part of the CodeDOM starts with the `CodeMemberMethod`. The `CodeEntryPoint`, `CodeConstructor`, and `CodeTypeConstructor` derive from `CodeMemberMethod`. The `CodeConstructor` is the instance constructor, and the `CodeTypeConstructor` is the `Shared` or `static` constructor for the type. These four methods, along with the closely parallel `CodeMemberProperty`, contain `Statements`. At the statement level, you begin to break your code down to a molecular level that may be unfamiliar to you. Looking at a human language analogy, the statement is the sentence, the method is the paragraph, the class is the chapter, and the assembly is the book or volume. Applications can consist of many volumes.

Understanding Types

Types in .NET is a general term referring to classes, enums, data types, structures, and interfaces. Type declarations in the CodeDOM graph are represented as objects of the `CodeTypeDeclaration` class. Use the `IsEnum`, `IsStruct`, and `IsInterface` properties to indicate the category if it isn't a class (which is the default).

CAUTION `CodeTypeMember` has an `Attributes` property for values of the `MemberAttributes` enumeration. These values are appropriate for members and aren't intended for use on types. The `CodeTypeDeclaration` class also contains a `TypeAttributes` property for values of the `TypeAttributes` enumeration that are appropriate for types. Be sure you use only `TypeAttributes` for the `CodeTypeDeclaration` to avoid later compiler errors. It may seem odd that the `CodeTypeDeclaration` has both `Attributes` and `TypeAttriubutes`, but within the inheritance hierarchy, `CodeTypeDeclaration` derives from `CodeTypeMember`. This inheritance design is essential to let you nest types as members within other types but leads to this strange artifact.

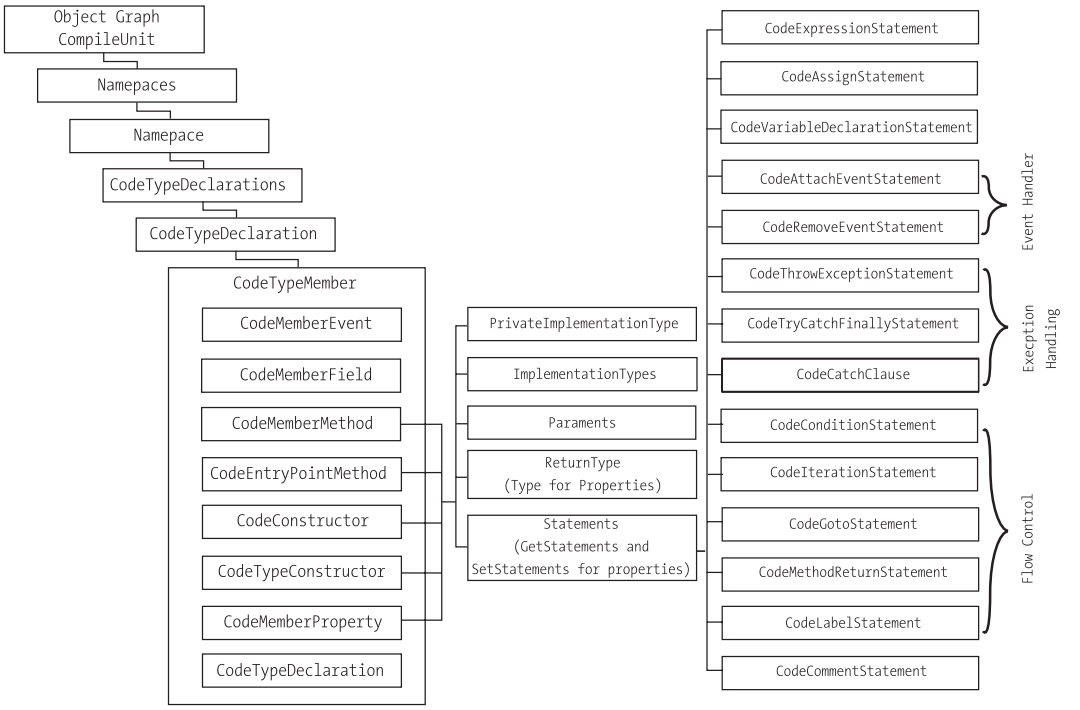


Figure 3-3. An object model of part of the CodeDOM

Understanding Members

Types contain *members*. For example, your .NET code might have a `Book` type (class or structure) that has a `Price` property and a `SetInventory` method as two of its members. When you're building a CodeDOM graph, you specify the kind of member to create by the class you instantiate. These classes are shown in Figure 3-4, and all derive directly or indirectly from the `CodeTypeMember`. To keep the figure focused on the overall design, Figure 3-3 doesn't show that `CodeEntryPointMethod`, `CodeConstructor`, and `CodeTypeConstructor` actually derive from `CodeMemberMethod`. `CodeMemberMethod` derives from `CodeTypeMember`.[7]

7. All of the objects contained in the `CodeCompileUnit` ultimately derive from `CodeObject`.

CAUTION The CodeDOM is all about outputting syntax. It isn't about correct code. So, in building the CodeDOM graph, you can reference objects and methods that don't exist. You have to output your code and let the compiler find out where you messed up. For example, a constructor isn't valid on an interface, but the CodeDOM won't complain if you provide one. However, the compiler will raise errors.

Understanding Statements

Members contain *statements*. The statement in code corresponds to the sentence in human language. This is the core of meaning. And just like human sentences, code statements perform different tasks.[8] The CodeDOM includes many different types of statements that perform different tasks. Looking at Figure 3-4, you'll see 14 different statement types. There's a comment statement, five statements to control program flow, three for exception handling, two for attaching and removing event handlers, one for variable declarations, one for assignments, and one for using expressions as statements. All statements reside in methods.

NOTE You often have declarations at the top of .NET classes. These may appear to be statements, but they're actually fields. Fields are members of the class, and statements can only appear within method bodies.

Understanding Expressions

Statements are made up of *expressions*. Expressions break the meaning of code down to the next level. The expression is similar to a clause in a human language. I can say "Understanding Expressions," and I haven't written a complete sentence, but I've placed words together in a way that conveys meaning. Within the CodeDOM, a clause may be the reference to a specific variable, invoke an expression, access language operators such as casts or `GetType` (`typeof` in C#), and so on. Figure 3-4 illustrates the range of expression types available.

8. In human languages, sentences can be commands, questions, informational statements, opinions, quotations, and so on.

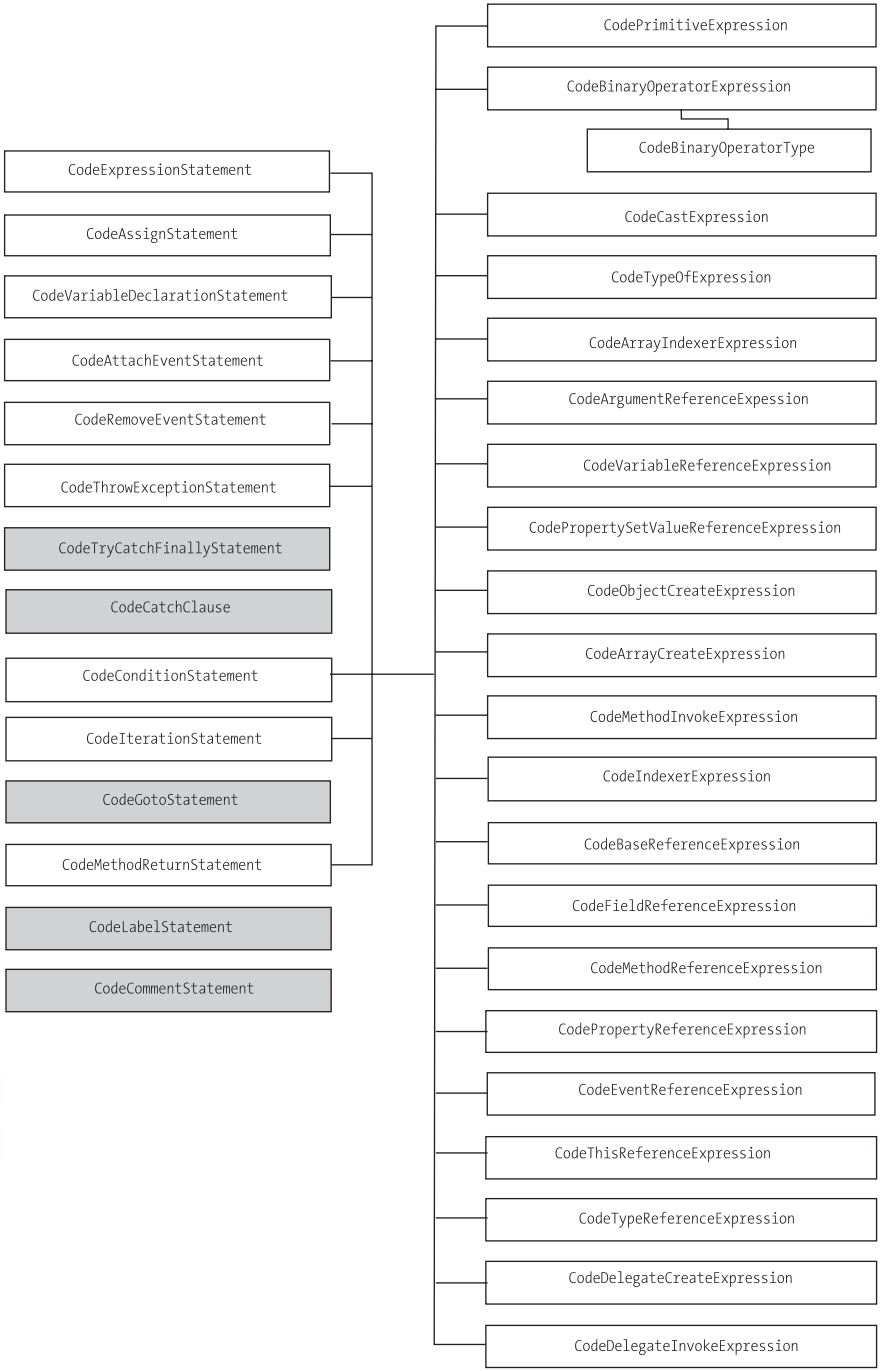


Figure 3-4. Statements and expressions supported by the CodeDOM

In human language, a clause might stand alone and be grammatically correct in some contexts. For example, I used "Understanding Expressions" as the main title of this section. The same thing occurs within .NET. An example of an expression that can stand alone is invoking a method that doesn't return a value—calling a `Sub` in VB. The expression needs a container, and you can use the `CodeExpressionStatement` class as a container for these simple expressions.

NOTE Where the CodeDOM expects an expression, it'll accept any of the expression types, but the output code may not compile.

Most statements use expressions. The statements that don't use expressions are gray in Figure 3-4. In addition to their use in statements, expressions often use other expressions. For example, an expression could call a method that took an argument that was itself the return value of another method:

```
obj.MethodA(obj.MethodB(j))
```

The code to create this line in the CodeDOM is a bit intimidating, but it illustrates how expressions work within the CodeDOM. A `CodeExpressionStatement` holds the expression. The expression is a `CodeMethodInvokeExpression` that invokes `MethodA` on the variable `obj`. The single parameter passed is another `CodeMethodInvokeExpression` that invokes `MethodB` on the variable `obj`. The code is as follows:

```
entry.Statements.Add( _
                New CodeExpressionStatement( _
                   New CodeMethodInvokeExpression( _
                      New CodeMethodReferenceExpression( _
                         New CodeVariableReferenceExpression("obj"), _
                        "MethodA"), _
                      New CodeMethodInvokeExpression( _
                         New CodeMethodReferenceExpression( _
                            New CodeVariableReferenceExpression("obj"), _
                            "MethodC"), _
                            New CodeVariableReferenceExpression("j")))))
```

NOTE A compiler error will occur when you compile the output code if `j` or `obj` aren't declared and/or initialized.

Building a CodeDOM Graph

To show how to build CodeDOM templates, I'll start with CodeDOM code that creates literals in your output code. Then I'll show how to create the structure of the CodeDOM graph and finally how to hook them together with CodeDOM code that creates method and class objects that later create your application code. The code in this section is cumulative. For example, I declare the `CodeCompileUnit` here and use it throughout later sections of the chapter. The compile unit will contain the CodeDOM graph that holds the rest of the code specifics. You declare the compile unit that contains the CodeDOM graph with this:

```
Dim compileUnit As New CodeDom.CodeCompileUnit
```

You enter expressions, such as primitive expressions, using the syntax of the language you're using when you create the CodeDOM graph—in this case, VB .NET. The code you output depends on the CodeDOM provider you're using.

Using CodePrimitiveExpressions

The most basic type of expression is a `CodePrimitiveExpression`. You can't just say "Hello World" or 42 with the CodeDOM. You have to transform everything, including literals, into an object. A few examples of primitive expressions are as follows:

```
Dim exp As New CodePrimitiveExpression("Hello World")
Dim exp2 As New CodePrimitiveExpression(42)
Dim exp3 As New CodePrimitiveExpression(3)
Dim s As String = "Sam"
Dim exp4 As New CodePrimitiveExpression(s)
Dim exp5 As New CodePrimitiveExpression(True)
Dim exp6 As New CodePrimitiveExpression(Nothing)
```

Primitives are literals—a basic unit of programming. It's impossible to imagine any nontrivial application that doesn't use literals. Although it might initially look confusing, it's perfectly legal to use a variable to define the value of a `CodePrimitiveExpression`, such as the definition of the literal `"Sam"`.

The previous examples that create primitives for `True` and `Nothing` may be the most interesting. They illustrate the primitive expression entered in the syntax of the language you're using to create the CodeDOM graph—in this case, VB .NET. When this value is output in the target language, it may not appear the same. Regardless of the language you use to create the CodeDOM template, if the target language is C#, the code provider outputs `true` (lowercase) and `null`. If the target language is VB, the code provider outputs `True` (mixed case) and `Nothing`.

Declaring Variables

Another basic type of expression is a local variable declaration. To produce code that will compile, you'll have to declare local variables before you use them, as either a variable or a parameter. Examples of variable declarations with and without initialization are as follows:

```
 ' Declare an integer variable named iSum
Dim decl As New CodeVariableDeclarationStatement( _
          GetType(System.Int32), "iSum")
' Declare an integer variable named iValue initialized to the expression Exp2
Dim decl2 As New CodeVariableDeclarationStatement( _
          "System.Int32", "iValue", exp2)
' Declare an object of the stream class named stream
Dim decl3 As New CodeVariableDeclarationStatement( _
          GetType(System.IO.Stream), "stream")
' Declare a string variable named fileName with the value Test.txt
Dim decl4 As New CodeVariableDeclarationStatement( _
          GetType(System.String), "fileName", _
          New CodePrimitiveExpression("Test.txt"))
```

Within the `CodeVariableDeclarationStatement`, you can declare the type of a variable either using the type (`GetType` is the VB equivalent of the C# `typeof` operator) or via a string. If you declare types using a string, you lose IntelliSense and compiler support in catching your typing mistakes. You can avoid using strings when you're working with framework types, but sometimes you're referencing unavailable types, such as referencing types in the code you're generating, and you'll have to use a string. If a third parameter is supplied, it's used to initialize the variable.

You create references to variables using the `CodeVariableReferenceExpression` method shown later.

NOTE The name `CodeVariableReferenceExpression` has nothing to do with whether the underlying variable represents a value type or a reference type. It's just the CodeDOM way of referencing a variable.

Building the Code Structure

Before using these declarations, take a step back and look at the structure of the method that builds the CodeDOM graph. The compile unit contains namespaces,

namespaces contain types (a class in this case), and types contain members (a method in this case):[9]

```
Dim nSpace As New CodeNamespace("CodeDOMTest")
compileUnit.Namespaces.Add(nSpace)
' Create a class to hold code
Dim clsStartup As New CodeTypeDeclaration("Startup")
nSpace.Types.Add(clsStartup)
' To run as an executable, you'll need a method that's an entry point
Dim entry As New CodeEntryPointMethod
entry.Name = "Main"
clsStartup.Members.Add(entry)
```

If you don't want a namespace declared in your output, just use an empty string for the namespace name. You always have to include the namespace object, but if it doesn't have an explicit name, it isn't output.

Once you've created an object to represent the method that the `CodeDOMProvider` will later output, you can stuff code into it. Code consists of statements. You create an object representing each method and add to its `Statements` collection. Statements will be output in the order they appear in the `Statements` collection:[10]

```
entry.Statements.Add(decl)
entry.Statements.Add(decl2)
entry.Statements.Add(decl3)
entry.Statements.Add(decl4)
```

Outputting Statements

The next code creates a reference to the system console class and outputs one of the primitive expressions defined earlier. This is similar to code in the "Hello World" example in Chapter 1:

```
Dim rExpConsole As New CodeTypeReferenceExpression( _
                GetType(System.Console))
Dim stmt1 As New CodeExpressionStatement( _
                New CodeMethodInvokeExpression(rExpConsole, _
                   "WriteLine", exp))
entry.Statements.Add(stmt1)
```

9. The code in this section is cumulative; `compileUnit` was declared earlier in the section "Building the CodeDOM Graph."

10. `decl1`, `decl2`, `decl3`, and `decl4` were declared earlier in the section "Declaring Variables."

Similarly Named Classes Can Be Confusing

Working with types is sometimes confusing. Types are generally, but not always, classes. The word `GetType` is a keyword and a method, and different .NET Framework classes sometimes have similar names. Some of the classes with the most confusing names support the CodeDOM and the automation model of `EnvDTE`. To keep these straight, watch the namespace with which you're working. For example:

- `CodeTypeReference`: Inherits from `CodeObject` and contains a reference to a type. Use this within other CodeDOM objects to declare variables, parameters, and so on.
- `CodeTypeReferenceExpression`: Inherits from `CodeExpression`. Use this to call static (`Shared`) methods.
- `CodeTypeRef`: Part of the Visual Studio .NET automation model and not used in CodeDOM programming.

`CodeTypeReference` and `CodeTypeReferenceExpression` are in the `System.CodeDOM` namespace. `CodeTypeRef` is in the `EnvDTE` namespace. Help lists `CodeTypeRef` and other members of the `EnvDTE` namespace as `CodeTypeRef` objects instead of the more common listing as a class entry.

Outputting Assignments

An assignment consists of two main parts: what you're assigning to and the value you're assigning. For example, to create the assignment `iSum = 0`, you'd use this:

```
Dim stmt2 As New CodeAssignStatement( _
                New CodeVariableReferenceExpression("iSum"), _
                New CodePrimitiveExpression(0))
entry.Statements.Add(stmt2)
```

You'll often assign the results of a *binary expression*. A binary expression is an expression with two operands such as addition. It consists of three parts—the left operand, the operator, and the right operand. In the next example, the left operand of the binary expression is a previously created primitive expression, and the right operand is a newly created primitive expression. The code output by this fragment assigns the result of `42 + 23` to the `iSum` variable:

```
Dim stmt2 As New CodeAssignStatement( _
                New CodeVariableReferenceExpression("iSum"), _
                New CodeBinaryOperatorExpression(exp2, _
```

```
                        CodeBinaryOperatorType.Add, _
                        New CodePrimitiveExpression(23)))
entry.Statements.Add(stmt2)
```

So far, the CodeDOM samples make up about 40 lines of template code. What does the resulting code output look like? It can output code in any language with a CodeDOM provider. Because the generating code is in VB .NET, I'll show the output in C#. After I removed the CodeDOM's standard file opening comments, the output is as follows:

```
namespace CodeDOMTest {
    using System;

    public class Startup {

        public static void Main() {
            int iSum;
            int iValue = 42;
            System.IO.Stream stream;
            string fileName = "Test.txt";
            System.Console.WriteLine("Hello World");
            iSum = 0;
            iSum = (42 + 23);
        }
    }
}
```

NOTE While the declaration was defined using the system type (System.Int32), it is output using the language specific keyword—`int` in C# or `Integer` in VB .NET.

Wow! Not very impressive for 40 lines of code! But that's life with the CodeDOM. In my experience, templates using the CodeDOM code contain at least three lines of complex code for every line generated.

Accessing Enums

Enums in .NET are created as special classes derived from `System.Enum`, with each enum value declared as a read-only field. You access enum values by treating them as fields:

```
Dim enumValue As New CodeFieldReferenceExpression( _
                New CodeTypeReferenceExpression( _
                      GetType(System.IO.FileMode)), _
                "Create")
```

Often you'll be referencing enums that aren't part of the framework and aren't available during generation. One way this can happen is to reference enums in the files you're outputting. Because the class isn't yet available, you'll only be able to use a string to reference the enum.[11]

TIP To create a new enum in a class, create a new member with `IsEnum` set to `True`. Use the `CodeFieldDeclaration` to declare individual enum values.

Creating Objects

Objects are generally created and assigned to a variable, either as part of the variable declaration or as a separate statement in the output file. The next code initializes the previously created stream variable by assigning to it to a new file stream:

```
Dim stmt3 As New CodeAssignStatement( _
                New CodeVariableReferenceExpression("stream"), _
                New CodeObjectCreateExpression( _
                      GetType(System.IO.FileStream), _
                      New CodeVariableReferenceExpression("fileName"), _
                      enumValue))
entry.Statements.Add(stmt3)
```

Declaring Array Variables

You can declare arrays in two different ways. You can create a normal variable with an array type, and you can use an overload of the `CodeTypeReference` expression. The second parameter to `CodeTypeReference` is the rank of the array, which allows you to declare arrays with multiple dimensions:

```
' Declare an array
entry.Statements.Add( _
                New CodeVariableDeclarationStatement( _
```

11. The string approach always works, but it doesn't offer strong typing benefits such as IntelliSense and compiler checks.

```
                GetType(System.Int32()), "aInts"))
' Shows option for type declaration
 entry.Statements.Add( _
                New CodeVariableDeclarationStatement( _
                New CodeTypeReference( _
                      New CodeTypeReference(GetType(System.Int32)), 1), _
                "a2Ints"))
```

You might find examples using square brackets within strings to declare arrays, such as `"System.Int32[]"`. Although understood by the CodeDOM, this isn't the preferable way to declare arrays because you don't get any strong typing benefits.

CAUTION It's often convenient to create variables in your CodeDOM code that hold references to expressions. Be cautious with these variables. You'll often place one reference to an instance in the `Statements` collection (either directly or through another object) and retain another reference to the same instance. If you make changes, the previous entry in the `Statements` collection will also reflect those changes. You can create new objects, or you can clone the objects as you put them into the CodeDOM. This is just normal object-oriented behavior, but it's different from the other two code generation mechanisms that output code as your template runs. The CodeDOM builds a complex object hierarchy and later outputs your code. If you forget this detail, you'll make a mess of your output.

Creating Arrays

You create arrays that can be assigned to previously declared array variables using the `CodeArrayCreateExpression`:

```
entry.Statements.Add(New CodeAssignStatement( _
                New CodeVariableReferenceExpression("a2Ints"), _
                New CodeArrayCreateExpression("System.Int32", 10)))
Dim varAInts As New CodeVariableReferenceExpression("aInts")
entry.Statements.Add(New CodeAssignStatement( _
                varAInts, _
                New CodeArrayCreateExpression("System.Int32", _
                   New CodePrimitiveExpression(0), _
                   New CodePrimitiveExpression(1), _
                   ' Additional assignments snipped
                   New CodePrimitiveExpression(9))))
```

Assigning Array Values

Accessing individual values within the array requires a `CodeArrayIndexerExpression`. This creates output such as `aInts(iValue)` in VB .NET or `aInts[iValue]` in C#. The next example assigns the fourth[12] element in the array to the previously declared variable named `iValue`:

```
entry.Statements.Add( _
                New CodeAssignStatement( _
                      New CodeVariableReferenceExpression("iValue"), _
                      New CodeArrayIndexerExpression( _
                            varAInts, New CodePrimitiveExpression(3))))
```

Okay, enough simple stuff. Let's look at something a little more complex. What would it take to output the following code?

```
Dim i as Integer
aInts(i) = aInts(i + 1)
```

After declaring the variable `i`, create an expression variable pointing to `i`. This and the previously created expression variable pointing to the array save some typing:

```
entry.Statements.Add( _
                New CodeVariableDeclarationStatement( _
                      "System.Int32", "i", _
                      New CodePrimitiveExpression(0)))
Dim varI As New CodeVariableReferenceExpression("i")
```

Outputting the target code requires array indexers for both the left and right operands of the assign statement. The left operand uses the variable reference expression pointing to the array and the variable reference expression pointing to `i`. The right operand also uses the variable reference expression for the array. For the array index, it uses a binary operator expression and the addition operator. The left operand of this binary operator expression is the variable reference expression for `i`, and the right is a primitive expression for 1:

```
entry.Statements.Add( _
                New CodeAssignStatement( _
                      New CodeArrayIndexerExpression( _
                            varAInts, varI), _
```

12. .NET arrays are zero based.

```
New CodeArrayIndexerExpression( _
      varAInts, _
      New CodeBinaryOperatorExpression( _
         varI, CodeBinaryOperatorType.Add, _
         New CodePrimitiveExpression(1)))))
```

This doesn't output a particularly complex line of code. It outputs a short, rather ordinary line of code, and the CodeDOM template is beginning to feel convoluted. If you understand how this piece of code works, though, you'll be able to extend it to more complex code statements on your own.

Exploring Other Features

In addition to the structure of the CodeDOM, a couple of features cut across all the different layers of the CodeDOM. These include `UserData`, which pass any information to the language-specific compiler, and code snippets, which include literal code in your output.

Using UserData

`UserData` lets you enter information that the language-specific provider uses to alter details of how it outputs code. This is a great feature. Unfortunately, the current providers support few `UserData` items, and many that they do support are undocumented.

NOTE Appendix D shows how to control `Option Strict` and `Option Explicit` in VB.NET output using `UserData`.

`UserData` appears at all levels of CodeDOM objects (compile unit, namespace, type, member, statement, and expression). In all cases, `UserData` is an `IDictionary` object allowing a string key and an object value. Only the language-specific provider that's looking for a `UserData` entry will use it. Language compilers ignore any `UserData` items they don't know about. If you extend CodeDOM providers, keep `UserData` in mind as the appropriate mechanism for the template programmer to communicate information to your specific provider.

Using Snippets

Literal snippets are evil. Okay, maybe they aren't quite evil, but they're terribly close. Snippets output language-specific code. The code contained in the

snippet is output as literals—it isn't interpreted by the CodeDOM, which is why it's language specific.

Snippets may seem cool, but a short time after you do anything with literal snippets you'll smack yourself on the head and do the Homer Simpson "D'oh!" I hope the reason you're using the CodeDOM is to provide language-specific output from a language-neutral source. Snippets output literal code segments, and literal code segments are inherently language specific. If you're doing language-specific stuff, you're breaking language neutrality, so why use the CodeDOM? Snippets are easy to grab when you can't figure out how to make the CodeDOM do something, and certainly that's their purpose. But before resorting to them, exhaust other ways to accomplish what you need. You can insert snippets in numerous locations in your code, and there are a handful of different snippet classes to provide the correct type of snippet for different uses, such as `CodeSnippetTypeMember`, `CodeSnippetStatement`, and `CodeSnippetExpression`.

Although it's ugly, passing a flag that indicates which language you're currently outputting and inserting different snippets based on the language is possible, but it makes the CodeDOM graph itself language specific, so you have to regenerate the graph for each language.

NOTE The current version of the code generation harness regenerates the CodeDOM graph although my primary intention was to simplify the harness, not to make it easier to output snippets.

Understanding the Nature of Output

The CodeDOM is about the syntax it outputs and makes no attempt to see if the output is sensible. For example, the CodeDOM recognizes the difference between variables and arguments and provides `CodeVariableReferenceExpression` and `CodeArgumentReferenceExpression`. However, both C# and VB .NET happen to use the same syntax for both variables and arguments. Both the variable and the argument class output the same thing, the output will compile, and you won't see an issue unless someone uses your template to create output in a language that differentiates between the variables and arguments. In this particular case, I can't think of a language that differentiates how arguments and variables are referenced, so you could argue it doesn't matter. But that's really the issue I'm describing—the CodeDOM allows you to be as sloppy as you'd like, and your only feedback is later compiler errors or improperly behaving applications, which may occur down the road and undermine your vision of a single template for all output languages.

NOTE There are numerous other cases where the CodeDOM and the compiler won't choke on what you're outputting, but the CodeDOM classes are improperly used. Although this may seem to make life easier, I think it actually makes it harder to work with the CodeDOM. You don't know if it's correct unless you carefully review your code by examining it manually. That's tedious and error prone. The best you'll be able to do is get to know all of the classes in the CodeDOM, particularly those in Figures 3-3 and 3-4. In some cases, such as switching the `CodeVariableReferenceExpression` and the `CodeArgumentReferenceExpression`, the output is identical and you could probably ignore the issues. In other cases, such as switching the `CodeBinaryOperatorExpression` and the `CodeAssignStatement`, the output may work in one language but not all .NET languages.

Using Keywords

Each CodeDOM language provider understands its own keywords and how to mark them when they're used as variable names. C# escapes them using the at (@) sign, and VB .NET surrounds them with square brackets as in `[New]`. If you'd prefer to modify keywords, such as adding a leading underscore, you'll have to do that yourself. Because the CodeDOM language provider handles escaping, you can use any names that match your style guidelines.

Using Parentheses

`CodeBinaryOperatorExpressions` places parentheses around its output. You can control parentheses with the order you build up these expressions, but you can't explicitly output parentheses unless you use snippets.

Testing in All Target Languages

It's absolutely essential that you test in all your target languages. I've been unable to create something that executed differently in the two core languages once it compiled in both (with `Option Strict On`[13]) other than the `Shadows` issue (discusses in Appendix D) and the scope issues on private interfaces. That doesn't mean it isn't possible; if you can create something, send it to me because it should be reported as a bug. This consistency says a great deal about how robust and stable the CodeDOM is, but it's not surprising because Visual Studio uses the CodeDOM in multiple places.

13. I suggest you don't count on C# operator overloading and VB .NET implicit casts (used when you set `Option Strict Off`) to always produce the same results.

Although I haven't created code that did different things once it compiled, I can create dozens of different ways to output code that'll compile in one but not both languages—and that doesn't even touch on warnings because C# catches unused variables and inaccessible code, and the VB .NET compiler doesn't. I haven't used the CodeDOM in J#. It's essential that you test the compile in all target languages early and often to catch problems and adjust your habits. You also need to test the resulting code. Microsoft used the technology in a relatively narrow usage, and there aren't many other people out there shaking it down.

Working with the CodeDOM

I've just fluttered over the top of the CodeDOM, focusing on what it means to work with code as an abstract language-independent entity. Although this may seem natural to you if you took a compiler course, it's a different way of looking at code than day-to-day creation in one or a small number of languages. Appendix D covers additional issues, incompatibilities, and limitations of the CodeDOM.

Most of the tips for efficient use of brute-force generation apply to the CodeDOM. You can't output regions without snippets, so you may want to replace regions with comments; however, you'll still want segments of your code output by individual methods. To maintain language neutrality in the CodeDOM, do the following:

- Avoid using snippets.
- Test your ability to compile the output in all target languages early and often.
- Test your component or application in all target languages.
- If you can't follow the first tip, provide alternate code for all target languages and re-create the CodeDOM graph for each language.

If nothing else, you can hand this second part of this chapter to your boss if she thinks you should use the CodeDOM. Comparing these three methods of code generation illustrates just how expensive CodeDOM is for developing applications.

Summary

One-click code generation allows anyone to precisely regenerate your code now and in the future. Reproducible one-click generation relies on a script. The script used in this chapter is a set of XML directives.

The keys to success for XSLT code generation are as follows:

- A single entry-level template accessing the root.
- A single high-level processing template called in the context you're processing.
- A named template for each region in the output after you organize your sample code into regions.
- Adding nested named and match templates while regions become longer and more complex.
- Using match templates for any looping elements unless the output is nearly trivial.
- A separate supporting stylesheet to contain reusable utility templates.

The keys to success for brute-force code generation are as follows:

- Break metadata extraction away from code outputting.
- Split outputting into individual classes that each output a single file pattern.
- Within each class, use an entry-level method.
- Use a different method for each region.
- Use additional methods while regions become longer and more complex.
- Use supporting methods for reuse wherever possible.

The CodeDOM abstracts your code sufficiently to generate code in any one of several languages. This abstraction makes working with the CodeDOM extremely complex. Seriously evaluate whether the extra time involved in working with the CodeDOM is justified for your application.

Additional Reading

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

- *XSLT: Programmer's Reference, Second Edition* by Michael Kay (Wrox, 2001)
- "Microsoft .NET CodeDom Technology: Part 1" by Brian J. Korzeniowski at `http://www.15seconds.com/issue/020917.htm`
- "Microsoft .NET CodeDom Technology: Part 2" by Brian J. Korzeniowski at `http://www.15seconds.com/issue/021113.htm`

EXCERPT

EXHEART

CHAPTER 4

Writing Handcrafted Code

Principle #4: You embrace handcrafted code by isolating and protecting it.

Generated code won't be enough to complete your project. It won't provide unique details to your individual classes or the component architecture that forms the backbone of your application. Significant segments of code, including startup code, utility functions, and exception reporting, occur only once within your application so aren't a good candidate for templates. This *framework* code, combined with your templates, creates the backbone of your application.

You'll also handcraft code to provide unique behavior or personality to many of your business classes and may have complex forms in your user interface that are easier to handcraft than generate. Recognizing handcrafted code lets you isolate it. Isolating it allows you to protect it when you regenerate, and the easiest way to isolate handcrafted code is by placing it in separate files. In this chapter, you'll see how to transparently interweave handcrafted code and generated code to create your fully functional application.

You already know how to write handcrafted code—you've been writing it since you started programming. This chapter extends ideas about handcrafted code in two key ways. You'll learn how to categorize handcrafted code and how to protect it across regeneration. Categorizing handcrafted code lets you organize it in a way that separates code based on your architecture from code implementing your application logic. You'll want your code organization to reflect this distinction to facilitate pattern reuse in different projects.

This chapter may also bring your overall perception of code generation into focus. Although you've seen individual aspects of code generation, you'll probably understand it fully only after you've seen how generation fits with your handcrafted code as the final piece of this new style of application development.

The word *keystone* comes from how stone arches, such as those in medieval bridges, are built. Each side is constructed. Each half is heavy and impressive but

requires temporary supports because the incomplete arch has little strength. It's not until the final keystone that joins the two portions is in place that the arch acquires its strength. Handcrafted code is the keystone of code generation. It fits between the patterns and application logic to complete the structure. It's hard to understand what code to autogenerate until you know what code to handcraft. It's hard to know where to place your autogenerated code until you understand where to place your handcrafted code. Once you understand this, you'll be ready to lay the roadbed to finish the bridge in Chapter 5, which covers details of how you interact with the peripherals of code generation such as testing, source control, and so on.

Effectively Using Handcrafted Code

If your application is so simple and predictable that your autogenerated code doesn't need to be specialized, there might be an adequate commercial solution already available, and you might not have the opportunity to write the application. Although most developers can use the same word processing software, for example, the nature of business applications and how particular companies work is often unique. This set of inconsistencies, even annoyances, is what keeps software developers employed. If you can do your job well, custom software is a cost-effective way for businesses to differentiate themselves from each other and compete. I'm a passionate believer in custom applications tailored to solve individual business problems. Understanding unique business processes and encapsulating these processes in specialized software can give your company an edge in today's market, and handcrafted code is part of that customization. Application development based on code generation provides two ways to customize an application—through handcrafted code (discussed in this chapter) and through metadata (discussed in Chapters 2 and 6).

Metadata vs. Handcrafted Code

When it's practical to generate your code from metadata and templates, you can reuse your solution. Solving the problem once with metadata and templates lets you implicitly apply it throughout your application or organization. However, it isn't practical to create new patterns and templates for every detail in your application. Technically, you could base everything on patterns by including conditional statements based on a particular condition such as a table name within either your template or your metadata extraction tool. But what would be the point? You'd inhibit the ability to move the pattern forward to your next

application and introduce mind-numbing complexity with no gain. Instead, you can encapsulate special cases in handcrafted code that you integrate with your patterns.

So how do you draw the line between generating code and handcrafting it? Table 4-1 shows four different scenarios and the benefit of generating code, handcrafting it, and copying it in from another location as a starting point for customization. Code that you create via copying falls into the handcrafted category because you'll manually maintain it.

Table 4-1. Benefits of Different Approaches to Code Creation

HANDCRAFTED CODE SCENARIO	GENERATION BENEFITS	WRITE-ONCE BENEFITS	COPY/PASTE BENEFITS
New code that will clearly be written only once.	None	Appropriate solution	Copy/paste isn't applicable because the code doesn't yet exist.
Current design shows single use, but the pattern might be reused.	Consistency and smooth transition if reused	Quick solution	N/A
Pattern will be used once in each application but reused in organization.	Consistency and ability to update when pattern evolves	None	This is quick way to provide initial consistency solution and independent evolution.
Pattern reused within project.	Consistency and ability to evolve as internal patterns change	None	None.

NOTE Single occurrence patterns include utility methods.

Table 4-1 shows several important points. If code will only appear once, it shouldn't be in a template because the purpose of a template is to create multiple versions of similar code. It may be unclear ahead of time whether code will be reused. You know how you plan to use the code in your current application, but you don't know whether you'll reuse it. There's a balance here: If you generate every pattern you might possibly reuse, your templates will be unnecessarily large and difficult to manage. If you don't generate features you do reuse, inconsistencies will creep into your application and between different applications in your organization. You can make this decision based on the following:

- How likely is it that you'll reuse the pattern?
- Which type is more logical from an organizational perspective?
- Where is someone else likely to look for the code?
- Can you describe a generic rule articulating when to reuse the pattern?

If you're likely to reuse the code, if you can describe it in terms of a generic rule, and if it's logically part of a template, put it in a template even if you're not sure how or when you'll reuse it.

TIP When you aren't sure whether you'll reuse a pattern, take the minimalist approach and handcraft it. Postpone what may never happen.

As an example of how the decision between adding a feature to a template or handcrafted code can be fuzzy, many applications concatenate people's full names. You may want to encapsulate this in your template if your organization has specific concatenation rules that apply to all applications. If you also standardize on how you name database fields, the presence of a `FirstName` and `LastName` field can indicate that the resulting class should have the standard set of full name methods. That's a business rule you use across application and you can apply in a generic manner in your templates. If you don't have consistent field names, you can still add this functionality to your templates in a rule-based manner with extended properties on appropriate fields.

Alternatively, your organization may vary how it concatenates names in different departments. Without consistency across applications or within different objects of the current application, it isn't appropriate to include the full name concatenation in your template, so you should handcraft at least the implementation of the full name methods. You can use rules to provide an interface rather

than an implementation. For example, you could offer an `IFullNameProvider` interface used throughout your organization and implementations specific to each department.

It's worth noting that I can't come up with examples that you'd always handcraft and never include in a template. The line is unclear because I can't predict whether the basis for a particular functionality is something such as "the `Customer` object of the accounting application" or "any object that contains a `FirstName` and a `LastName` property." If the basis is generic like the latter, include the pattern in your templates. Otherwise, handcraft it.

Understanding Application Frameworks

I've skimmed over the idea of frameworks so far in this book because I wanted you to have a vision of code generation independent of your previous ideas about what a framework is.

The word *framework* has been used in many contexts over the years. In relation to code generation, I use the phrase to describe what you carry forward from one application to another. The framework contains features that are common between diverse applications such as an application for a corporate legal department and one for an organic fertilizer company. The framework will include your templates and the components that make up the backbone of your applications. Ideally, if you have a framework and a set of metadata, you can create the first cut of your application in a matter of minutes. Once you've gotten that quick start, you'll have more work to do. Frameworks rarely fit perfectly; there may be superficial issues that make your user interface layers quite different, or you might even use different delivery mechanisms (such as Windows or the Web). The point of a framework is to have the appropriate chunk ready to move forward and allow the framework to evolve with your architecture. If your architecture evolves between applications, it's appropriate for your framework to evolve. If your organization works to implement a standard architecture across applications, your framework will be consistent. Ultimately, your framework is a tool for implementing your architectural patterns.

By delaying the discussion of frameworks, I hope I've provided a clear understanding that the set of components and templates you create can be separate from the generation harness and from the metadata. There's some dependency because the template must know the structure of the metadata, and the harness has to access templates in a standardized way. That means you need a standard interface for the harness and a standard schema for metadata. But the contents of the template, the structure of the application, and the framework itself can be independent, so they *can be interchanged*. In Chapter 11's

"Interchanging Templates and Frameworks" section, you'll find a discussion of how interchanging templates is likely to affect the programming industry and see how much power you'll have at your fingertips if industry leaders can decide on a common XML Schema Definition (XSD) for application metadata and one or a small number of interfaces for accessing templates.[1] But you don't have to wait for that. You can create your own internal standards for consistent templates today. This is key to the evolution of your own framework, including the ability to retrofit deployed code–generated applications with future best practices.

TIP The easiest way to create standards for code generation is to leverage or build on the code and patterns in this book, particularly using the tools in Chapter 3.

TIP Provide clear interfaces and templates that work with a standard metadata structure. Your goal is to build a framework that you can transfer to your next application.

You're also ready to extend historic ideas of frameworks by segmenting them. You work in a world of three-tier architectures. You can create independent frameworks for your data access layer, middle tier, and user interface layer. Assuming you create standard interfaces between the tiers, separating them allows you an incredible amount of flexibility, now and in the future.

TIP Think in terms of generic compartmentalized frameworks with components. Plan to move your application forward in pieces or in total.

1. Using XSL Transformations (XSLT) templates provides a mechanism for consistent template access to a large degree, but there are still issues about handling callback objects. For other template styles such as brute-force or CodeDOM generation, a consistent interface is an important part of being able to use multiple generation harnesses. Although the generation harness from this book's Web site is adequate for your use, there's a great deal of work that vendors will be able to do in terms of creating more powerful front ends for template generation and processing.

Creating Task-Based Categories of Handcrafted Code

Frameworks require that you separate your application logic from the framework patterns. Separating your application logic from your pattern is relatively easy when you're looking at metadata and templates. Metadata defines application details, and templates encapsulate patterns. It's trickier when you're looking at handcrafted code. You don't have this natural delineation and have to impose the distinction. Understanding handcrafted code lets you organize it, separate application and framework logic, and minimize later problems evolving your code. You can categorize segments of your handcrafted .NET code into eight basic types based on what each segment accomplishes in your application. Imposing these categories on your own code will help you separate the pattern-based framework code from the code implementing your application logic. This splits your handcrafted code into two distinct categories—code you'll carry forward and code that's an integral part of the business logic of the current application.

Many applications don't include all eight types, but you can place handcrafted code into one of these categories with surprisingly little overlap between the categories as follows:

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

Running Startup Code

Startup code is the stuff you run when your application starts up. In a console application, this is just the `Sub Main` that serves as the entry point. Web applications can have startup code when the server starts up, and Windows applications

can provide their own `Sub Main` entry point or start via one of the application's forms. As discussed in the "Starting a Windows Application" sidebar, I recommend you write your own Sub Main for .NET applications because it improves your control of global error handling and can even remove the requirement that the startup form remain loaded for the life of a Windows application.

Starting a Windows Application

.NET provides shortcuts to start your Windows application by allowing you to specify a startup form. Unfortunately, you can't customize global exception handling when you use a default startup form, and you can't keep your application alive when the default form closes. To fix these shortcomings, use a class such as the following:

```
Option Strict On
Option Explicit On

Imports System
Imports System.Windows.Forms

Public Class Startup
   Public Shared Sub main()
      Dim frm As Windows.Forms.Form
      frm = New HarnessForm
      Dim exceptionHandler As New GlobalExceptionHandler
      ' Adds the event handler to for exceptions
      AddHandler Application.ThreadException, _
               AddressOf exceptionHandler.OnThreadException
      Windows.Forms.Application.Run(frm)
   End Sub

   Public Class GlobalExceptionHandler
      Public Sub OnThreadException(ByVal sender As Object, _
                  ByVal t As System.threading.ThreadExceptionEventArgs)
         ' Handle the exception event
      End Sub
   End Class

End Class
```

Each application has only one set of startup code. Code that occurs once isn't a good candidate for code generation, so startup code is handcrafted code. Along with exception handling code, startup code is likely to be consistent between your applications and may even be part of an organizational guideline for building applications. Code you'll reuse across different applications can be part of your framework. Looking at the startup code in the "Starting a Windows Application" sidebar, you'll see that the only variable item is the name of the startup form. You'll have to use the criteria in the earlier "Metadata vs. Handcrafted Code" section to evaluate whether it's appropriate to write once, copy/paste, or apply a template-based pattern using application wide metadata. If you copy/paste this code as part of your framework components, you can make it more generic by calling a shared (C# static) method within your application or by using a consistently named startup form such as `MDIParent`. You can even pull the name of the startup form from a configuration file and create it via reflection, but you'll want to be careful that an error in the configuration file doesn't cause your application to fail at startup without a message.

Incorporating Exception Reporting

You can incorporate exception reporting into your framework. Exception reporting is one of three aspects of exception handling—using `Try/Catch`, creating custom exceptions, and reporting exceptions. `Try/Catch` is interspersed in your code, and the context of the surrounding code determines whether it's handcrafted or autogenerated.

TIP Using `Try/Catch` and raising exceptions should be as simple as possible because they're often part of your application's logic. Whether it's in handcrafted code or autogenerated code, limit your use of Try/Catch to where you're doing something in the `Catch` or `Finally` block.

TIP Although it isn't documented, Visual Basic .NET (VB .NET) supports using a `Throw` statement without parameters.

You're likely to have custom exceptions in both your application logic and your framework. The "Organizing Custom Exceptions" sidebar clarifies how you can organize custom exceptions.

Organizing Custom Exceptions

Custom exceptions are the heart of .NET exception handling. They provide a container for information about the exception and a flexible object-oriented way to recognize exceptions that are thrown.

In your code-generated application, exceptions will fall into three broad categories—Common Language Runtime (CLR)[2] exceptions, framework-related exceptions, and application-specific exceptions. I suggest an inheritance hierarchy such as the following:

```
System.Exception
   System.SystemException
      Example: System.InvalidCastException
      Example: System.Data.SQLClient.SQLException
   System.ApplicationException
      YourNamespace.frameworkNameException
         YourNamespace.exceptionName
      YourNamespace.applicationNameException
         YourNamespace.exceptionName
```

This pattern lets you later recognize and separately catch exceptions arising from .NET classes, your framework, and your application logic.

The third aspect of exception handling is exception reporting. Your exception reporting class is a special type of component. It's unique because it's highly desirable to allow your exception reporting to behave differently at different phases of your project life cycle. You can replace programmer-friendly "give me all you've got" type dialog boxes appropriate during development with polite user-friendly "would you like to send an email?" type dialog boxes when you move into your testing and delivery cycle.

NOTE The generation harness at the Apress Web site includes an interesting exception reporting tool useful to developers. This tool presents a TreeView of nested exceptions that's particularly valuable when you're working with XSLT or reflection. It also presents the call stack in a friendly grid that defaults to hiding framework entries in the callstack. Chapter 5 discusses the output of this tool.

2. The CLR is often called the *.NET Framework*, but this phrase can be confused with the code generation frameworks discussed in this chapter.

The sky is the limit on what you do in the exception reporting component. You're likely to have different exception reporting tools for Windows, Web, compact devices, and console/Windows service applications. The style of exception reporting (how polite and vague your user communication is) in use at a given time will often be driven by configuration files, especially on the Web.

If you're an experienced developer, you may read this section and say, "Duh! I already know that!" That's a core point I want to make as you dive into handcrafted code. Code generation is a new tool in your arsenal and worth reading a book about. But all the tools you currently have, such as components and other techniques for code reuse, are still critical to your success. What you'll gain is a clearer understanding of what makes up handcrafted code and how to organize it to be a full partner with your generated code. You're enhancing and building on existing skills.

Using Components

Components are classes that have a specific set of duties. They may be simple classes, components derived from `System.ComponentModel.Component`, or user interface elements. Components are different from utility methods because each component has a coherent purpose and a set of related tasks it performs, such as managing connections. Components also frequently maintain state. See the "State, Stateful, and Stateless" sidebar in Chapter 10 for a discussion of state.

You might include components in all tiers of your application. For example, in the data tier, components might provide generic management of parameters or connections. Or they might provide a single (or small number) of database access points. In the user interface tier, they're likely to provide specialized user interface behavior, especially as custom controls. Framework components are less likely in the middle tier because it's made of application specific business objects, but you may still have them to represent specialized units of behavior, such as a business calendar, that are used throughout your organization.

Using Utility Methods

Utility methods are stateless methods that do specific jobs. You may have a library of utility methods that you've used for a long time. These methods may include tasks that the CLR doesn't do (or doesn't do quite the way you want) and CLR functionality you've wrapped to simplify access. In VB .NET, you can house utility methods in either a module or a class with shared members. In C#, utility methods will be static members of a class.

NOTE Stateless methods are ones that take the values they work with as parameters.

You may have already isolated utility methods and already moved generic utility methods between applications. Not all utility methods are generic and used by multiple applications—some are specific to the application. The goal when you incorporate utility methods in your code generation is to have at least two files or libraries. One has generic tools useful in the framework, and the other has application-specific tools. Using separate libraries makes it much easier to recognize and manage dependencies.

TIP If you've moved from another language such as VB 6, review your utility methods to see where they offer functionality now provided in the CLR. Where you find overlap, you can remove your method and replace it with the CLR call, document the potentially subtle differences within your utility method, or use your utility method as a wrapper for the CLR methods. There's a good chance you'll implement a combination of these methods, depending on the individual utility method.

Using Code with Insufficient or Interspersed Patterns

The point of code generation is to get your job done faster, not to beat your head against the wall of some ivory tower. Sometimes the code is just easier to write by hand. Although you'll see how to combine autogenerated and handcrafted code in the same class later in this chapter, sometimes the code is so variable and interweaved that it isn't worth the effort to isolate the handcrafted code to allow regeneration. As you work with code generation, your skills in using patterns and metadata will increase, and you'll be able to incorporate code generation in a wider segment of your application.

Stored procedures, data access layers, and business objects are generally easier to generate than presentation layers, which take specialized techniques discussed in Chapters 9 and 10. Some presentation layers don't have a predictable format that can be applied to most forms or pages in the application. Although you may be able to generate many of your forms (especially maintenance forms), you may need to handcraft complex forms.

Providing Reusable Functionality with a Base Class

Sets of generated classes often share functionality. When you're using object-oriented programming, you can move this common functionality into a base class and avoid redundantly coding or generating it. Figure 4-1 shows a common

inheritance hierarchy for generated code. In this pattern, a base class derives from `System.Object` or another base class. This class is handcrafted and part of your framework. Your generated class derives from this base class, and a handcrafted class with specific customizations derives from the generated class.

Just like other inheritance patterns, the base class declarations and implementations are common to all derived classes. It's handcrafted because each type of base class (such as `BusinessObject` or `BusinessObjectCollection`) occurs only once in the application. The class generated for each entity (such as the `Customer`, `CustomerCollection`, or `Order`) is generated from metadata expressed through templates. There's a unique version of this class for each entity in the application. Finally, one derived handcrafted class corresponds to each generated entity class. Each entity in the application is made up of the common base classes, the generated class, and the derived handcrafted class.

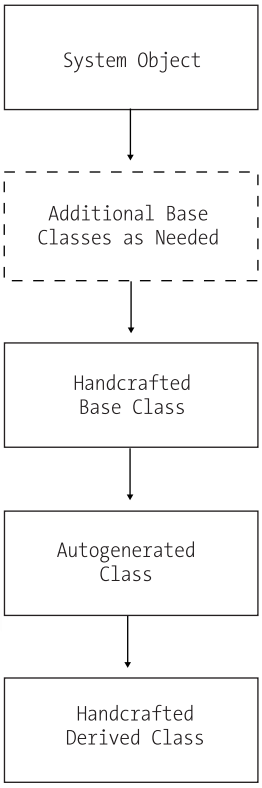


Figure 4-1. Bracketing your generated classes with a common handcrafted base class and a unique handcrafted derived class for handcrafted specialization provides great flexibilty in your generated application.

NOTE The word entity describes a granular, vertical slice of the application. For example, business objects encapsulating the Northwind database would probably contain entities such as `Customers`, `Orders`, `Suppliers`, `Products`, and so on.

Interfaces and Base Classes

Both interfaces and base classes allow you to test whether a class is a particular type using VB .NET's `TypeOf` or C#'s `Is` operators. This lets you and the compiler each check whether your use of the class matches its declarations.

You may have heard that the difference between an interface and a base class is that the interface doesn't contain an implementation. This is true, but it's also helpful to consider the base class as the "is a" relationship while the interface is a "can behave as" relationship. For example, my dog is a mammal, and she can behave as a companion (or a guard or a pest). I like this distinction because base classes without implementations are sometimes appropriate. This occurs when the derived class is inherently part of the base class, and the base class may have future implementation. Similarly, it's sometimes appropriate for an interface to define no methods or properties.

NOTE Without multiple inheritance, you don't want to "use up" your single base class unnecessarily, but `System.Object` doesn't offer much to build on as a base class.

Providing Class-Specific Code Functionality with a Derived Class

Generated classes often require customization. This customization provides each class its unique personality and both enhances pattern-based functionality and adds new functionality. You may extend the class, perhaps by adding a concatenated full name, or you may replace generated functionality with something unique such as supplying a nondefault validation routine. In both cases, inheritance is your primary tool.

As shown in Figure 4-1, each entity can have a derived handcrafted class that gives it a unique personality. This is the location for extending the generated class and altering its default behavior. You do this by adding new methods and properties and by overriding existing methods and properties. The original shell of the derived handcrafted class may be created via code generation, but the class exists as a home for your manually customized code. You'll probably edit at least some of your derived handcrafted classes. They're part of the application logic, not the framework, but the template that creates them is part of the framework.

NOTE The distinction between application-specific and framework can be confusing. The template for the handcrafted derived class is part of the framework. It determines the structure and interface of the derived class. The code generated from the template is part of the application logic because many customize it.

You specify which methods of the generated class you can override within the template. In C# you use the `virtual` keyword, and in VB.NET you use the `Overridable` keyword to specify that you can override the member.

TIP There's a small performance penalty when you invoke a virtual or overridable member. This occurs because the runtime must check whether the method has been overridden by the current object. This performance hit is small and benefits of virtual methods outweigh the performance penalty whenever it may be logical to override the member. However, it isn't a good idea to make everything in your application virtual. More important than performance, using both virtual and nonvirtual methods tell later programmers what you anticipate them overriding. If you can guess what they're likely to want to do, and you can provide the preferred mechanism for accomplishing it, sealing other avenues will clarify your intent.

NOTE Code generation gives you additional flexibility here. If you later need a member to be overridable, you can make the change in the base class or the template and won't need to make the change in numerous locations.

Providing Debug Support

In addition to code that's part of your application, you may have code designed to support your debugging process. For example, you might have a method you can call from the Immediate window that displays information about a DataSet, relation, or business class. To run this type of code in the Immediate window, you need to include it in your application. This is another type of handcrafted code, and if you use this style, you'll probably include this code as a separate class library within your framework to provide consistent tools between different applications.

Working with Handcrafted Code

To leverage code generation via templates and a reusable framework, you need to organize your code based on two independent criteria. These criteria are whether the code is handcrafted or generated and whether it's part of the framework or part of your application logic. Figure 4-2 shows how these two independent (also called *orthogonal*) criteria results in four categories of code. It also shows what kinds of code tend to go into each grouping.

Code generation allows your framework and your applications to evolve independently. This is one of the most important benefits of adding code generation to your development experience, and you'll need to pay careful attention to handcrafted code to make it work well. Moving into code generation is a great time to consciously address reuse and clearly distinguish between the pattern of how your application is structured (the framework) and the details of the application itself. Table 4-2 summarizes the categories of handcrafted code and whether each is logically part of the application or framework.

Table 4-2. Whether Categories of Handcrafted Code Are Part of the Framework or Application Code

CATEGORY	APPLICATION	FRAMEWORK
Startup		✓
Exception reporting		✓
Components		✓
Utility methods	✓	✓
Code with insufficient or interspersed patterns	✓	
Class specific code	✓	
Base class functionality		✓
Debug support		✓

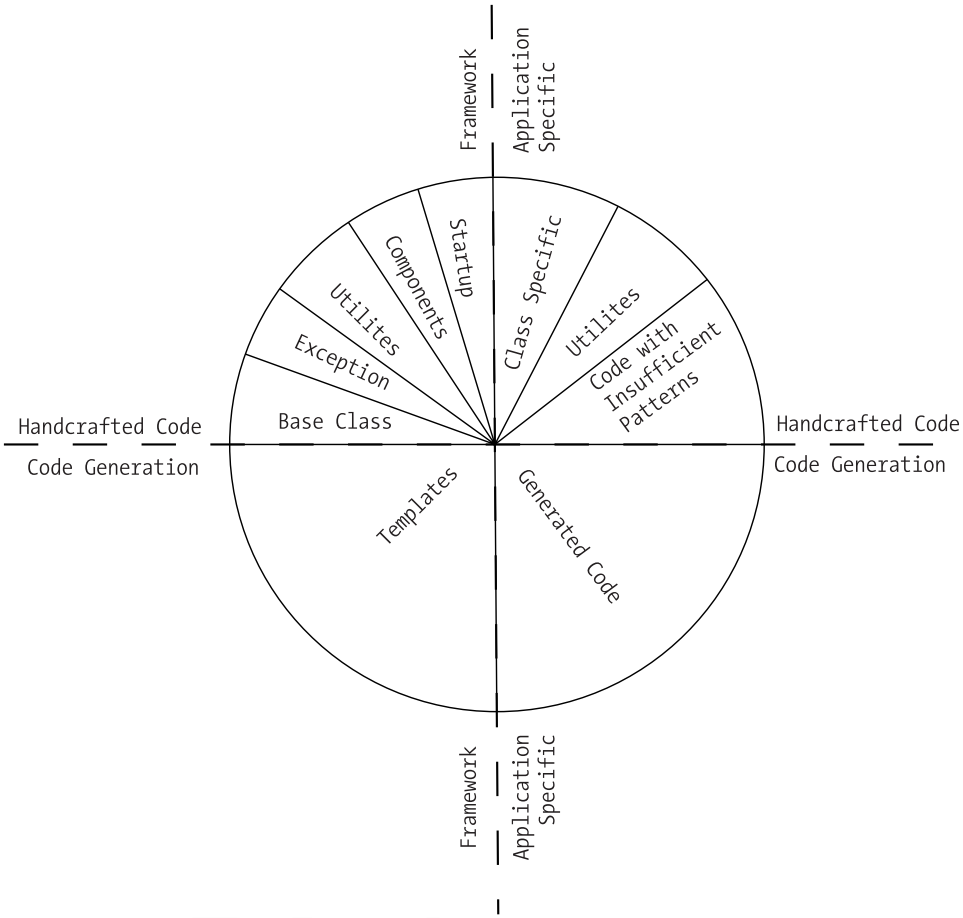


Figure 4-2. Two independent criteria result in four kinds of code in your application.

Once you've designed the distinction between your framework and application, you'll need a plan for physically isolating the code. You'll want different categories of handcrafted code in different files both to facilitate moving patterns between projects and to clarify the purpose of each set of methods. The purpose of the set of methods in each file is important to maintain the distinction between framework and application code throughout the life of your application. Where possible, also use separate directories as discussed in Chapter 5.

You'll also want to decouple your framework from the nonframework (application logic) portions of your application. The easiest way to accomplish these goals is to place the pattern-based code in separate assemblies—Dynamic Link

Libraries (DLLs). This works well for components and utility methods. It may require a little extra effort to maintain the separate DLL, but it forces you to clarify the interface between different pieces. This effort pays off when you move the pattern into your next application.

The downside of using separate assemblies is that you have to increase the scope of many elements from `Friend` to `Public` because `Friend` is only valid within assemblies. This means any programmer using the class will have access to the methods and won't be able to distinguish between methods they should use and methods that are `Public` because of the application architecture. How important this issue is for a given method depends on how sensitive its tasks are. If the member does something such as return a connection string, it's a big problem.[3] To make separate assemblies work, you'll have to carefully segment your application so that things you want to hide aren't passed between assemblies.

NOTE There's a slight performance penalty for passing between assemblies, even when they're contained within the same process. The important word here is slight. The performance hit isn't enough to worry about in almost any scenario.

Exploring Mechanisms for Isolating Handcrafted Code

Categorizing handcrafted code naturally leads to how you isolate it. Isolating handcrafted code from generated code is essential. The key to isolation is generally putting this code in separate files. You don't want to write tools that split up intermingled handcrafted and autogenerated code. Even with techniques such as regions, complex parsers are fragile to human changes and errors with very poor payback.

It's easy to isolate the majority of handcrafted code. Of the eight categories of handcrafted code, only class-specific code generally requires special isolation techniques. The challenge of isolating class-specific handcrafted code arises when you need to logically combine separate files for runtime use. For code that's a logical extension of a generated file, the inheritance features of .NET are a good solution. Unfortunately, inheritance isn't available for stored procedures, JScript, VB 6, and perhaps other languages you may be interested in generating. I'll show you some other techniques and why the difficulty of integrating handcrafted code

3. Although you want to use the proper scope and keep `Private` methods and properties `Private`, don't think `Private` methods prohibit programs outside your class from using methods. A programmer can use reflection to run methods of any scope, and the Component-based, Scalable, Logical Architecture (CSLA) architecture presented by Rockford (Rocky) Lhotka in his book *Expert One-on-One Visual Basic .NET Business Objects* (Apress, 2003) and discussed in Chapter 8 uses this feature.

makes VB 6 one of the most difficult languages to generate. Happily, I've got some novel approaches for customizing stored procedures and JScript. Concentrating on isolating handcrafted code through inheritance and in stored procedures will give you enough depth to create your own solutions for more difficult handcrafted scenarios.

There are six basic mechanisms for isolating handcrafted code from generated code when you need to combine them into a single runtime logical unit (a class, stored procedure, and so on):

Inheritance: You can use inheritance to separate handcrafted and generated code into separate classes in separate files. This is the best solution to creating classes that combine generated and handcrafted code.

Partial classes: Planned for the Whidbey release of .NET, partial classes allow the same class to be implemented over multiple files. As you'll see in Chapter 11, partial classes offer opportunities to tweak code generation for some scenarios but don't radically change the overall process.

Regions: Within .NET code, regions let you place generated code directly within a class. Because the region is in a class the user is editing, it's easy for them to accidentally edit the generated region, which is bad.

Replaceable stock elements: Stored procedures don't offer inheritance, but they offer important opportunities for generating code. You can define a small number of locations where customization can occur in most stored procedures. Using careful naming you can specify additional nested stored procedures to run at these locations.

Two-phase approach: You may already be creating JScript code on the fly through ASP.NET. The ASP.NET code that generates the JScript can itself be generated. This two-step generation offers the ability to combine handcrafted and generated code in JScript.

Comment markers: As a last resort, you can include comment markers that bracket code to overwrite, or code to preserve. This approach, like regions, is problematic because it's difficult for the programmer customizing the application to know what they can and can't edit within the file. However, this approach allows you to do code generation in any text-based language that supports comments.

The following sections look at these approaches in more detail. The discussion of inheritance goes to greater depth because you'll primarily use this technique. Sections on the other mechanisms are intended to help clarify your vision of the approach, not offer implementation details.

Understanding Inheritance

If you have inheritance available, this is the best tool to use to isolate handcrafted class-specific code. Inheritance allows you to create a single handcrafted base class, generate classes for each entity, and handcraft a derived class for each entity intended for customization by later programmers.

Exploring Object-Oriented Fundamentals

The reason inheritance works so well for protecting handcrafted code lies in Object-Oriented Programming (OOP) fundamentals. Inheritance allows you to split a class into two or more files—or a logical class into an instantiated class with one or more base class. These files can be any combination of generated and handcrafted classes.

Figure 4-3 shows a Unified Modeling Language (UML) class hierarchy similar to Figure 4-1. It differs in illustrating which items are sets of classes and which are generated. Only the shaded classes are regenerated. A single base class contains code used by all entities. Even if you don't currently have implementation code to put in this class, you'll generally want to include it because it gives you a common definition for all the other classes in the inheritance tree.

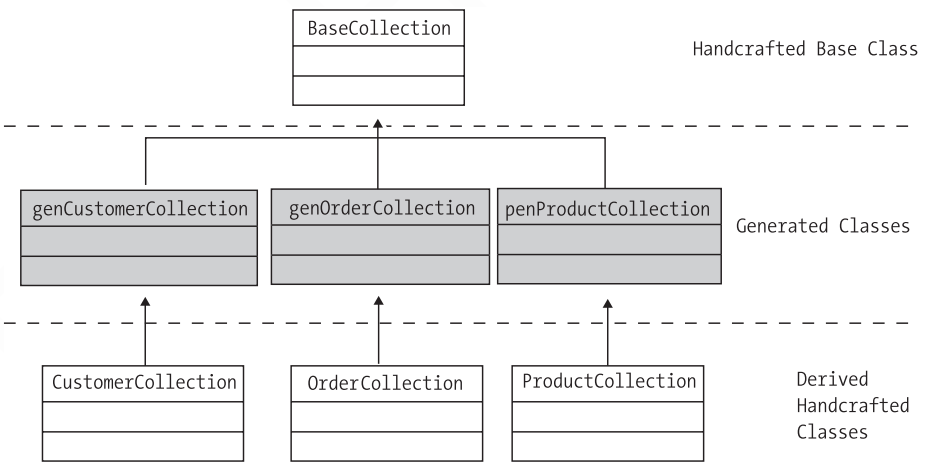


Figure 4-3. A more complex look at the handcrafted base, generated, and derived handcrafted class pattern showing where many or one derived class is expected

Instantiating the Most Derived Class

You have to create the derived handcrafted classes for each of your entities (such as the `CustomerCollection` in Figure 4-3) whether or not you currently have customizations planned. Depending on your application, only a small number of your classes may have customization in the derived handcrafted class. But you can't predict which of the classes will eventually contain this customization. If a programmer instantiated a base class, such as `genCustomerCollection`, and you later added customization, you'd have to alter all the code using the class, which would be expensive and wasteful. To enforce the use of the derived handcrafted class, make the generated class `MustInherit` (`abstract` in C#).

NOTE If you allow the generated class to be instantiated, later code in the derived handcrafted class will be ignored. Protect against this by marking classes you don't want instantiated with `MustInherit` where possible. You can't mark `Windows.Forms` base classes as `MustInherit` as explained in Chapter 9.

The handcrafted derived class, the generated class, and any additional base classes combine to form a logical entity. Programmers using your class will have no clue how you organized it, but this internal design maximizes what you can generate while ensuring your customizations are used. The performance hit of the extra class in the inheritance tree is negligible. Having it available helps future programmers understand where to put handcrafted code, and it allows you to maintain the flexibility provided by the base/generated/derived handcrafted class pattern. Because creating all the handcrafted derived classes by hand would be a pain, you'll initially generate these classes as empty shells. Chapter 5 shows how you can use hash markers to control overwriting handcrafted derived classes.

Looking at an Icky, Tricky OOP Detail

The handcrafted derived class you start with should be simple. Ideally, classes such as the `CustomerCollection` in Figure 4-3 are just an empty shell waiting for a programmer to customize it. You'll generate this class initially, but that's just to make it easier to start up your development. You'll want to keep it simple because it's designed for future programmers to edit, and if you include required code in the initial shell, they may accidentally break the class by deleting the required code. Sure, you can mark this code and make rules. You could even

develop a tracking system and hash the regions you want unchanged using the approach discussed in Chapter 5, but it's easier to start with an empty class that'll later contain only class-specific handcrafted code.

Although a good goal, this is hard to achieve because of shared (C# static) methods, including constructors. These are never virtual—in English that means you can't override them to provide special behavior.

The constructor is a special problem because both C# and VB provide compiler tricks to automatically call the base class constructor—but it has to be a parameterless constructor for this to work. If you have only parameterized constructors in your generated class, you'll need to include a parameterized constructor in your derived handcrafted class. Bang! You've included important code that a later programmer can delete. If at all possible, use a parameterless constructor to avoid putting a constructor in your handcrafted classes.

TIP Minimize or eliminate required code in handcrafted classes because a later programmer might delete this code. When you need code in the handcrafted class, including constructors and shared methods, you can force compiler errors when the method is missing by calling the method from generated code. These calls can be in dummy code that's never used at runtime.

Using Validation as an Example of Overriding Behavior

In many cases, you'll want to provide the same behavior for all of the classes created from a particular template although actual implementations of the behavior may differ. Validation in the middle tier is a simple, but important, example.

Validation, especially in business objects, is a required behavior for most business applications and the responsibility of the middle tier (the user interface might have prevalidation responsibility as well, but ideally that prevalidation is repeated in the middle tier). You'd like your User Interface (UI) code to be able to call a validate method and perform required validation or perform no validation if none is needed. The calling code needs no knowledge of the actual tasks the business object performs within the validate method, but there's a logical contract that the tasks should be all validation with no extra behavior. This independence (also called *decoupling*) means that the business object validation can evolve its behavior over time without affecting the calling code.

NOTE If all of your classes provide a set of methods and properties, it's logical to include the set as part of the expected interface for the class. You can do this with an interface or by adding the method to a base class. If you place it in an interface rather than the base class, you can implement the interface in the base, generated, or derived handcrafted class. Implementing the interface in the generated or base class ensures it's implemented and avoids some extra casts. Unless you need a specialized implementation in each derived handcrafted class, implement the interface and default members in the base class and allow the derived handcrafted class to override when needed.

You can provide a meaningful default implementation in generated classes because metadata defines a significant portion of your validation. Your metadata probably already includes data types and the maximum length of strings. You can add additional validation information such as ranges for numeric and regular expression (regex) expressions for strings. You can leverage this information to provide validation in either the base class or the generated class. The techniques are a little different.

To implement default validation in the base class, code generation creates rules about your validation in a `ColumnInfo` object property that contains fields such as numeric range, regex, maximum length, and other expected validations. The base class validation method checks these values. The advantage of this method is that you provide the validation information in a generic manner usable by both base class validation and user interface code. The downside, of this approach is that unexpected validations, such as a comparison with another column aren't easily incorporated.

To implement default validation in the generated class, simply place the validation into the code during code generation. The code generation templates can interpret any validation information available. The benefit of this is if the template can interpret the information, it can output it as validation code. Constraints in SQL Server can often be interpreted in a generic manner. The downside of this approach is that information such as the maximum length of strings isn't available for purposes other than middle tier validation unless you include it redundantly. If you redundantly include it, overriding the value becomes problematic.

NOTE In both base class and generated class validation, the validation information is hard coded into the generated class. This improves performance, keeps the application from changing unexpectedly, and allows a more limited database security role to be used at runtime. This means both approaches require regeneration to implement changes, and there's no meaningful performance difference.

You can combine base class and generated class validation. This lets you perform common validations—such as maximum length, numeric or date range, and regex in the base class—providing these important values to the UI if it requests the information. The generated class can handle additional database constraints, as long as the generation or metadata process can interpret the constraint. Finally, you can put validation that isn't contained in the database or isn't interpreted from database constraints in the derived handcrafted class.

Some validation depends on several fields and is difficult to express in metadata. An example of providing validation in a handcrafted method allows only a specific supplier to have unit prices of more than $5,000. In this approach, it's important that the handcrafted method first call the base class validation. This is an example of custom validation code that might appear in the handcrafted derived class:

```
Public Overrides Function Validate()
   If MyBase.Validate() Then
      If SupplierID <> 3 And UnitPrice > 5000 Then
         Return False
      End If
   Else
      Return False
   End If
End Function
```

NOTE You'd probably want to enhance this method to provide more information to the user interface layer, rather than simply a Boolean value indicating success or failure.

TIP Don't raise exceptions in middle tier code to signal normal occurrences such as validation failure. It's slow to throw and catch exceptions. Even worse, it forces certain behavior on bound controls, such as not allowing the user to leave a field in Windows.Forms. Even if you desire that behavior in your application, you'll want to manage this in the user interface, not the middle tier.

Ensuring Programmers Call Base Class Methods

If you're worried about programmers forgetting to call your generated class validation from the derived handcrafted class validation, you can enhance the design for this method. Instead of a simple overridable `Validate` method, you can seal the `Validate` method, ensuring the generated class's `Validate` method is always called. You can then provide an empty `Protected Overridable` method, perhaps named `DoValidate`, designed for the derived handcrafted class to override. The `Validate` method of the generated class always runs and calls `DoValidate`. Thus, both the generated and derived handcrafted validation logic are guaranteed to run. Because the generated class is empty, it doesn't matter whether programmers call the base class `DoValidate` method.

Although I'm describing this pattern in terms of validation logic, you can use the same pattern in any scenario where it's essential that a base class method runs. You can also solve this problem using events discussed in the section "Using Events to Provide Behavior."

Using Derived Methods to Extend Behavior

In addition to providing the specific implementations of methods that all classes in a particular pattern contain, many specific classes contain additional extended behaviors. An example is the concatenation of full names. Derived methods adding to base class behavior is common but very simple—you just add the new method. This is how you give your classes the specific personality that fulfills your application's requirements.

In some cases, a subset of your classes may support a particular interface. You can handcraft the declaration and implementation of interfaces in your derived handcrafted classes.

There are three ways to provide class-specific properties and methods. You can override behavior defined in your generated or base class, you can add new properties or methods, or you can use events.

Using Events to Provide Behavior

The third way to specialize the behavior of a derived handcrafted class is events. Events can be raised by a class and handled in external code or within derived classes. They let you provide behavior in response to something that happens within the class.

NOTE Because there's a little overhead associated with throwing and catching events, Windows.Forms provides On*eventname* methods that correspond to all (or at least almost all) of the events in controls and forms of the namespace. This is a great pattern to follow.

If the events either allow you to provide feedback (such as a cancel property) or react to an exception thrown in the event handler, you can affect the class behavior from the event handler. You can do this whether the event handler is inside or outside the class.

TIP Handling events outside the class is especially useful if your generated classes derive from the ADO.NET DataTable or DataRow. These classes provide untyped properties that access column information, data values, and so on. You can provide alternate strongly typed mechanisms for accessing the underlying information, but you can't block programmers from using the untyped properties directly. For example, you can't provide validation checks in the data value property set methods and ensure the validation code will run. However, you can leverage the `ColumnChanging` event to run your validation. You can count on the event firing regardless of how the user applies changes to the data.

TIP The issue with ADO.NET is that there are two paths to the same task. There isn't a mechanism to exclude the original untyped approach when you add the strongly typed one. You can take a lesson from ADO.NET and beware of offering multiple routes to accomplish key actions, and you can provide events where necessary so later programmers can plug in the holes.

For your own classes, you generally use events to allow code outside the class to respond to action within the class, but you can also use events when you want to ensure your base class functionality runs while still allowing programmers to specialize that functionality. You can do this with code like this:

```
Public Class TestEvents
   Event FooChanged(ByVal sender As Object, ByVal e As EventArgs)
   Private mFoo As Int32
```

```
    Public Property Foo() As Int32
        Get
            Return mFoo
        End Get
        Set(ByVal Value As Int32)
            mFoo = Value
            ' Critical actions
            OnFooChanged(New EventArgs)
            RaiseEvent FooChanged(Me, New EventArgs)
        End Set
    End Property

    Private Overridable Sub OnFooChanged(ByVal e As EventArgs)
    End Sub
End Class
```

Note that this allows external code to respond to the event while code in derived classes can either respond to the event or override the `OnFooChanged` method. However, they can't circumvent or modify the critical actions taken when `Foo` is set because the `Foo` property is sealed (you can't override `Foo`). The point of this approach is to allow external code and base classes to react to something while ensuring base class code in the property always runs.

Using Regions As an Alternative for Isolating .NET Handcrafted Code

So what about regions? Many of the Visual Studio designers isolate handcrafted code using regions, and if they're good enough for the Visual Studio teams, why not use regions for application code generation to isolate handcrafted from autogenerated code?

The Visual Studio designers have a significantly different role than your application code generation. The primary purpose of the Visual Studio .NET–generated code is to maintain values specified in the property dialog boxes. This is called *serialization* to code and is accomplished using CodeDOMSerializers. Serialization is significantly simpler than your application code.[4]

4. I don't want to oversimplify the work Windows.Forms and other Visual Studio generators do. They use the CodeDOMSerializer to output code that declares, instantiates, and sets properties for a variety of objects—some quite complex. They do this in a fully extensible pattern that has both default serializers for unknown classes and a mechanism for programmers to provide custom serialization—and they do this for any language that supports the CodeDOM. But declaring and instantiating objects and setting properties is just the tip of the complexity your application code generation will handle.

In your own code, using regions to isolate handcrafted code has the following drawbacks

- It's nearly impossible to track changes made by humans and changes made by autogeneration separately in source control.

- It's hard to effectively mark the code as generated because you can only mark the top of the region. This works for the Visual Studio generators because it's easy to recognize code generated primarily for serialization. Your own code won't have this distinction.

TIP Maintain an internal hash code for each region. Sooner or later a programmer will accidentally make edits inside the region, and the hash code allows you to recognize this and avoid overriting edited code.

- Parsing failures will cause invalid classes and be difficult for the later programmers to resolve. It'll be hard for them to know how to get the class looking like it can be generated again.

TIP Allow programmers to add a key phrase such as FORCE_OVERWRITE to the hash to cause the region to be overwritten, regardless of a hash code match.

With these caveats, you can use regions if you think the benefits outweigh these drawbacks. You'll have to trust programmers using the resulting class not to deliberately or accidentally edit your generated code. If they edit it, you'll smash over those edits or the programmer may break your code in a way that's unexpected. To replace regions, do the following:

1. Open the source code file as a stream.

2. Parse to find the region you'd like to replace.

3. Erase the existing contents of the region, possibly after checking a hash to ensure that no one has edited the code.

4. Replace it with the newly generated code.

TIP Consider modifying the hash code routines in Chapter 5 to ignore whitespace. Chapter 5 determines whether a human has edited a file, and even whitespace changes means they did. It's unlikely that a programmer working with generated regions will anticipate that changing whitespace would invalidate the hash code, especially since they may introduce the whitespace changes by simply reformatting. Ignoring insignificant whitespace is non-trivial because you have to determine what's significant, which differs by language.

Replaceable Stock Elements: Stored Procedures

When you're working with SQL stored procedures, there's no inheritance, so you'll need a different approach to isolating handcrafted code. The stored procedures you're customizing are generally simple, and you're in a database where *careful naming* (see Chapter 2) is appropriate. These two factors save the day. For example, a simple stored procedure might look like this:

```
/******
Name:       nwd_Customers_Select.prc
Summary:    Select information from nwd_Customers_Select
Created:    3/3/2003 7:24:39 PM
*******/
CREATE PROCEDURE nwd_Customers_Select
       @CustomerID NChar
AS
BEGIN
   SET NOCOUNT ON
        SELECT
        [Customers].[CustomerID],
           [Customers].[CompanyName],
           [Customers].[ContactName],
           [Customers].[ContactTitle],
           [Customers].[Address],
           [Customers].[City],
           [Customers].[Region],
           [Customers].[PostalCode],
           [Customers].[Country],
           [Customers].[Phone],
           [Customers].[Fax],
           [Customers].[timestamp]
   FROM     Customers
```

```
   WHERE
 [Customers].[CustomerID]= @CustomerID
END
```

TIP Use square brackets to support unusual table and field names, such as names containing spaces.

There are only three places where you can add code to this stored procedure. You can add code before the SELECT statement, add it after the SELECT statement, or replace the SELECT statement. Returning to the concept of careful naming, and assuming the metadata extraction gives you a list of stored procedures in your XML metadata file, you can check for a carefully named procedure that might be called at each of these key locations.

For example, you might create a rule that if a stored procedure named *storedProcName*_Custom_BeforeSelect (such as nwd_CustomerSelect_Custom_BeforeSelect) exists during code generation, it'll be called before the SELECT statement. If the original XSLT template for this stored procedure is as follows:

```
CREATE PROCEDURE <xsl:value-of select="$spname" />
   <xsl:apply-templates select="dbs:TableConstraints/dbs:PrimaryKey/dbs:PKField"
                        mode="PrimaryKeyParameters"/>
AS
BEGIN
   SET NOCOUNT ON
      SELECT
      <xsl:apply-templates select="dbs:TableColumns/dbs:TableColumn"
                           mode="SelectList"/>
   FROM    <xsl:value-of select="$Name"/>
   <xsl:if test="$hasprimarykey='true'">
   WHERE <xsl:call-template name="PKeyMatch"/>
   </xsl:if>
END
```

then you can add directives that check for the presence of specially named stored procedures and call them if they exist. The XSLT directives you'd add before the SELECT statement might look like this:

```
<xsl:choose>
   <xsl:when test=
     "ancestor::dbs:DataStructure/dbs:StoredProcs/dbs:StoredProc/@Name=concat(
                                          $spname,'_Custom_BeforeSelect')">
```

```
   EXEC <xsl:value-of select="concat($spname,'_Custom_BeforeSelect')"/>
         <xsl:apply-templates
                      select="dbs:TableConstraints/dbs:PrimaryKey/dbs:PKField"
                      mode="PrimaryKeyList"/>
   </xsl:when>
   <xsl:otherwise>
   /* To add custom code here, create stored proc named <xsl:text/>
       <xsl:value-of
           select="concat($spname,'_Custom_BeforeSelect')"/> with parameters
       <xsl:apply-templates
                      select="dbs:TableConstraints/dbs:PrimaryKey/dbs:PKField"
                      mode="PrimaryKeyParameters"/> */
   </xsl:otherwise>
</xsl:choose>
```

Here you're checking for the existence of the stored procedure in the input XML metadata file. If it's found, the stored procedure is called. If the stored procedure doesn't exist, then the template inserts a comment in the text that explains how to provide customization. If a `Custom_BeforeSelect` stored procedure exists, but no `Custom_ReplaceSelect` or `Custom_AfterSelect` exists, a resulting stored procedure looks like this:

```
/******
Name:       nwd_Categories_Select.prc
Summary:    Select information from nwd_Categories_Select
Created:    3/3/2003 7:24:39 PM
*******/
CREATE PROCEDURE nwd_Categories_Select
      @CategoryID Int
AS
BEGIN
   SET NOCOUNT ON
   EXEC nwd_Categories_Select_Custom_BeforeSelect
            @CategoryID
      /* To replace the SELECT statement with custom code, create a stored
         procedure named nwd_Categories_Select_Custom_ReplaceSelect with params
              @CategoryID Int */
      SELECT
          [Categories].[CategoryID],
          [Categories].[CategoryName],
          [Categories].[Description],
          [Categories].[Picture],
          [Categories].[TimeStamp]
```

```
    FROM    Categories
    WHERE [Categories].[CategoryID]= @CategoryID
    /* To add custom code here, create a stored procedure named
       nwd_Categories_Select_Custom_AfterSelect with parameters
           @CategoryID Int */
END
```

NOTE Providing customization for stored procedures makes verbose templates.

You can extend this technique to pass whatever parameters are defined in the optional stored procedure. To handle these parameters smoothly, the parameters of the nested stored procedure would need to map to parameters or variables of the same name in the base stored procedure.

TIP The simplest way to map nested stored procedure parameters to their counterparts in the base stored procedure is to give them the same name.

You can also extend the metadata to include a Transact-SQL (T-SQL) block that should be included in the base stored procedure for tasks such as setting variables or creating temporary stored procedures when you need to execute in the context of the base stored procedure.

Using a Two-Phase Approach: JScript

Another language you might want to generate code in is JScript (JavaScript). JScript is deployed in two primary ways—script files and script that's output as part of a Hypertext Markup Language (HTML) page. When the script is output as part of the HTML page, you can build the script as strings within .NET code. You can use inheritance with the .NET class and thus indirectly provide functionality via inheritance. To make this work well, you'll want to create blocks of JScript in individual overridable methods. You can provide individual methods that return fragments of JScript, and you can override the methods as appropriate. When using a two-phase approach such as this, it's easier to isolate handcrafted code.

When working instead with JScript files, it's relatively easy to generate the files for deployment on the client, and debugging these files is only slightly more

difficult than debugging nongenerated JScript.[5] However, it's difficult to customize functionality with handcrafted code for the same reasons it's difficult with VB 6 code discussed in the next section. You can customize Jscript code using comment markers, but this limited ability to customize code generated in JScript severely limits its practicality.

Using Comment Markers As a Last Resort: VB 6

You can generate code in any language. The difficulty for application code generation lies in seamlessly interweaving handcrafted and generated code. When you can't find another way to do it, you can use comment markers to customize code in any language as a last resort. In this approach, you surround lines of generated code with markers indicating that they're generated, allowing you to find these sections in the application's code. Because these hash markers aren't legal in the target language, output them as comments.

You have to be able to find these segments of code during regeneration because you need to overwrite the generated code, and only the generated code. To accomplish this, you can use comments to either mark internal segments you'll overwrite on regeneration or mark segments to preserves during regeneration.

TIP If you're using comment markers, be consistent with whether you mark the segments you want regenerated or mark the segments you want preserved. Regardless of which approach you take, you'll need to mark both the start and the end of each segment.

The generated code might look something like the following VB 6 code:

```
Option Explicit

Private Sub Form_Load()
    InitClass
End Sub

'/?? START OF GENERATED CODE - DO NOT EDIT THIS COMMENT OR THE FOLLOWING CODE
Private Sub InitClass()
    ' Autogenerated class initialization code
End Sub
'/?? END OF GENERATED CODE - DO NOT EDIT THIS COMMENT OR THE PREVIOUS CODE
```

5. Debugging JScript is always hard.

This code is problematic. If the generated code is long, future programmers may view a page of code that offers no indication that it's code they shouldn't edit. It's difficult to avoid this code looking like other code in the source code files. If the programmer alters or copies the marker, later code generation may overwrite edited code. To protect against this, you can use a hash code, either on the whole file with the handcrafted segments removed or on individual segments you're regenerating, depending on your marker approach. Either way, you've added one more thing the programmer can mess up, but it's preferable to overwriting edited code.

This approach can work, but the difficulty in insulating handcrafted code makes it a last resort. It requires too much discipline from later programmers. They can't even reorganize files or edit the marker comments. If you use this approach, you'll find its implementation similar to using regions, so you can use those steps to implement this approach.

Summary

Handcrafted code can be part of the framework or the application code. The combination of templates and certain types of handcrafted code create the framework for your application development. Isolating framework and application code lets each evolve independently and allows your framework to move to your next project while the metadata and handcrafted code of your application logic evolves to stretch the lifetime of the application and eventually carry this information forward to the next version.

Categorizing your handcrafted code makes it easier to separate framework and application code. There are eight categories:

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

Isolating handcrafted code from generated code allows you to regenerate without damaging edits. The following are six mechanisms for isolating handcrafted code:

- Inheritance
- Partial classes
- Regions
- Replaceable stock elements
- Two-phase approach
- Comment markers

The mechanisms that use separate files—inheritance and partial classes—are the easiest to use. Because partial classes aren't yet available, inheritance is currently the preferred approach.

EXCERPT

EXHEART

CHAPTER 5

Tying It All Together

Principle #5: Code-generated applications are high-quality applications.

You've seen how the big pieces of code generation fit together to build your applications. A handful of additional details allow you to achieve maximum effectiveness in your generation process. This chapter talks about the most important of those details—organizing files via the directory structure, building project files, incorporating source control, debugging, testing, and bringing your framework forward to start your next project. It also covers details of the code generation harness introduced in Chapter 3, including the specific directives you'll use in later chapters to manage your code generation. Taking care of these details helps you create high-quality applications with successful life cycles.

You'll probably come back to this chapter when you're solving specific problems—such as integrating source control or modifying the code generation harness. You might skim parts of this chapter if you find it difficult to understand without the context of a problem you're trying to solve. On the other hand, if you're ready for concrete answers to nagging concerns—such as how you manage project and solution files—you'll find those answers in this chapter. This chapter covers a lot of detail, and its goal is to give you the tools to match or exceed the quality of your current process when you're using code generation. Other chapters discuss how code generation increases the quality of your code, but development quality relies on more than quality code.

Getting Into the Harness Details

Significant benefits arise from one-click code generation. This simplifies your development and allows later programmers to repeat your generation steps

exactly—significantly improving the maintenance cycle. One-click code generation relies on a *code generation harness,* which is a tool that runs your code generation. Although you could build a code generation harness, the easiest approach would be to use the one I provide at the Apress Web site. This section is about using that code generation harness.

As you saw in Chapter 3, the harness strings a bunch of steps you define into a single process initiated by one button click. This process incorporates all of your automated steps—extracting metadata, outputting code, running SQL scripts, copying files into the correct directories, and so on. These steps are defined as a series of *directives*. Each directive describes a single step in the code generation process. The directives are Extensible Markup Language (XML) elements in an XML harness file. Each directive element has a number of child elements that indicate how to perform the task—including input files, rules for overwriting files, templates, and so on. Understanding individual directives and their attributes allow you to better understand the many pieces that make up code generation and let you better utilize the harness.

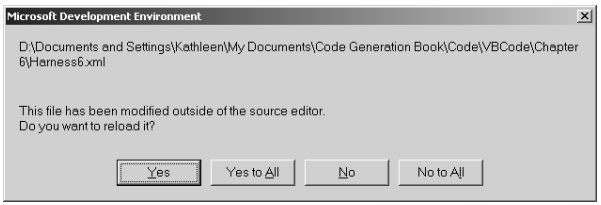
NOTE The code at the Apress Web site for Chapters 6 through 10 uses the code generation harness to run the examples. There's a harness file associated with each chapter to let you explore issues related to a single chapter in the book. When you're working on your own code generation, you'll build a single harness file that runs all your steps. The ReadMe file at the Apress Web site discusses building a single harness file to run your code generation process.

Appendix C discusses how I built the code generation harness. It's made of two projects. The base project is an XML editor. This editor isn't going to handle any XML you toss it, but it's a general-purpose XML editor that's dynamically driven from an XML Schema Definition (XSD). Derived from the classes of the base project, classes in the code generation harness add functionality. Specifically, the derived class adds the Run button and corresponding behavior. You can also edit the XML harness file in an XML editor (including Visual Studio's XML editor), which you might find helpful for some operations, such as merging the contents of two harness files.

TIP If you need an XML editor for another context, you could extend the harness tool for that purpose.

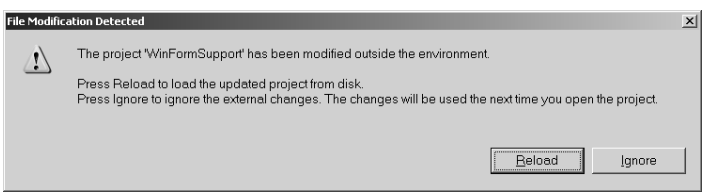
"Modified Outside the Source Editor" Dialog Boxes

Working with code generation, you'll often see two dialog boxes that appear when you change your files outside the Visual Studio editor. This happens when you have files in your application open for editing in Visual Studio and you regenerate those files. The following figure shows the most common dialog box. It's just telling you that you changed files that were open for editing. Well, duh! That's what code generation is for. You can check whether the file was edited in Visual Studio by seeing whether there's a star beside the filename on the tab that displays it. If you've edited in Visual Studio but not yet saved, and it's the first time you've edited the file, code generation doesn't know about your change and reloading the file will overwrite your changes. If the files haven't been changed in the editor, simply click Yes or Yes to All. If the files have been changes, you'll have to resolve the conflict.



CAUTION If you click No in the previous dialog box, there's a quirk in how Visual Studio handles the file. This occurs when you change a file outside Visual Studio, say No so you don't reload the file into the editor, and then save the file (either by saving directly or building the application). Scenario 1: If you make no further changes to the file within Visual Studio, you'll receive a confirmation dialog box about overwriting the existing file. Scenario 2**:** If you make changes to the file within Visual Studio, you won't receive a confirmation dialog prior to overwriting the file.

The next dialog box appears if you change your project file while the project is open. The wording is rather confusing, but if you know you made the changes because of regeneration, you can reload the project.



You want to do a Save All *before* regeneration. Code generation works only with the files on the disk. If your loaded files don't match the files on disk, you can wind up losing the manual edits in files currently open in the Visual Studio editors. Similarly, if you don't save an edited project file prior to reloading, you'll lose the edits you entered in the Project Properties dialog box and the changes to the included files when you reload the project.

Reviewing the Harness Building Blocks

Before looking at the individual directive types, I'll review the structure of the harness file introduced in Chapter 3. Under the root, harness files contain a `FilePaths` element and one or more `Section` elements. The `FilePaths` element contains child `FilePath` elements. Each `FilePath` element defines an *abstract filename*. Abstract filenames allow you to reuse the file path or name later in the harness file, facilitating reuse and making it easy to adjust your project directory structure. Paths are relative, so you shouldn't need to make changes for different developer's machine layouts, as long as the directory structure within the project is consistent.

Each *section* contains a set of *directive elements*. Each directive element is a single step and is the atomic unit of code generation. You can organize directives into sections in any manner helpful to your project. One way you could break this down is as follows:

- Extract metadata and perform Object Relational Mapping (ORM)
- Generate stored procedures
- Generate business objects
- Generate a Windows user interface
- Generate an ASP.NET user interface

Directives contain child elements. Each child element contains a set of attributes. The purpose of child elements is to organize the many attributes associated with each directive. Organizing them as child elements allows the definition to be reused—thus a `MultiPass` element, for example, is identical for any of the directive types that use it. If you look back at Figure 3-1 (in Chapter 3), each child element is displayed in a separate group box.

Introducing Basic Directive Types

The code generation harness performs three different types of actions—single-pass processing, multipass generation, and nesting scripts. The task described in a single-pass directive is run once each time the directive is encountered. There's generally one input file and one output file. Single-pass processing effectively says, "Go do this task, and come back when you're done."

Multipass processing is more complex. Multipass processes are performed multiple times, generally to output multiple files. For example, you might run a template to create a business object for every `orm:Object` element in your metadata file. Each business object is a separate code output file—each created by processing the template once. Multipass effectively says, "Go do this task on this series of nodes and come back when you've finished all of them."

Figure 5-1 illustrates the difference between single-pass and multipass attributes. It also shows how the steps of code generation are often iterative; the output of one step is the input for the next step. Metadata files may be used to generate files from a single template or several templates.

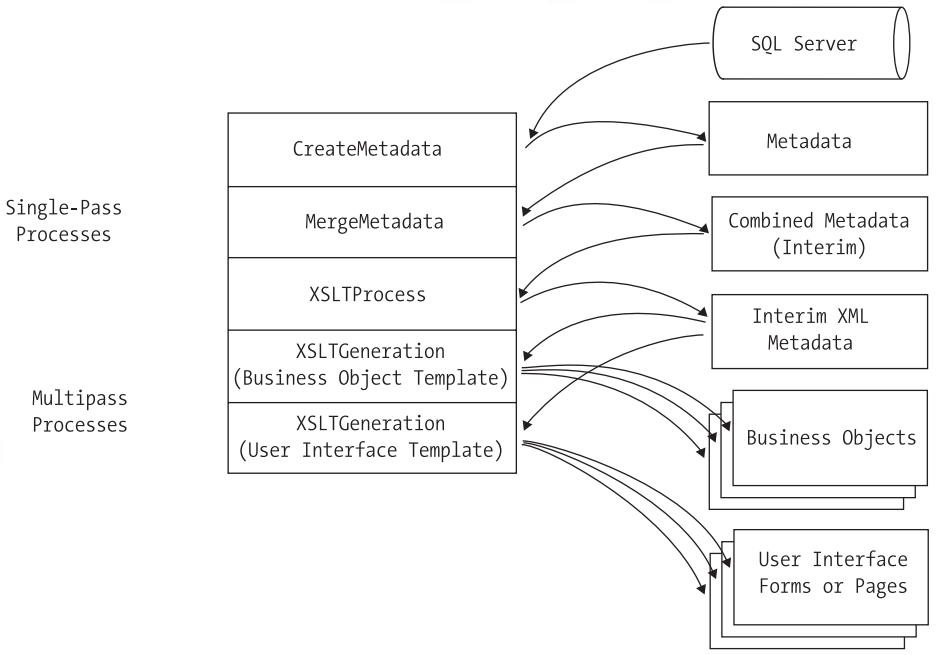


Figure 5-1. Several code generation steps illustrating the difference between single pass and multipass directives

Attributes on the contained `MultiPass` child element describe when the task is performed. These attributes include an input XML file that's used for selecting what to run, an XPath pattern that selects elements from the input XML file, and a pattern describing how to name output files. The harness performs the specified action once for each matching element, producing zero to many files as output. The `MultiPass` child element is discussed in the section "Introducing the MultiPass Child Element."

Nested scripts perform whatever actions are contained in the nested script. It just runs the directives contained in the specified script. You can find more information about nested scripts in the section "Using the NestedScript Directive."

Understanding Directives

The code generation tool supports nine different directives. The four multipass generation directives are `XSLTGeneration`, `BruteForceGeneration`, `CodeDOMGeneration`, and `RunSQLScripts`. The four single-pass process directives are `XSLTProcess`, `CreateMetadata`, `MergeMetadata`, and `CopyFiles`. The `NestedScript` directive runs nested scripts.

Because the directives are extensible, you can add new ones if needed. Table 5-1 shows the directives currently supported by the harness. It also shows which child elements each directive supports. The "Exploring Specific Directives" section later in this chapter talks about each of these directives, the differences between them, and how you'll use each one. Details about the individual directives will make more sense after you understand what each child element contains, so I'll cover that before moving into the individual directives.

NOTE This chapter talks about the specifics of the directives used with the code generation harness. It's your user's manual for using the harness. At the same time, it shows you what's really required for one-click code generation. If you evolve this tool or create your own, you'll have to address these same types of problems. The tool wraps this all up but leaves it extensible.

Table 5-1. Directives Supported by the Code Generation Harness

DIRECTIVE	DESCRIPTION	ATTRIBUTES	CHILD ELEMENTS
`XSLTProcess`	Processes an XSL Transformations (XSLT) file to create a single output		Standard, OutputRules, SinglePass, XSLTFiles
`RunProcess`	Runs any .NET process		Standard, OutputRules, SinglePass, Parameter, Process
`CreateMetadata`	Extracts metadata from database	`Server`, `Database`, `SkipStoredProc` `SelectPatterns` `SetSelectPatterns` `RemovePrefix` `LookupPrefix`	Standard, OutputRules, SinglePass, AlternateProcess
`MergeMetadata`	Combines two files	`BaseXMLFileName`, `MergingFileName`, `OutputXMLFileName`	Standard
`CopyFiles`	Copies files	`SourceFileName`, `TargetFileName`	Standard
`XSLTGeneration`	Processes an XSLT file multiple times create multiple output files		Standard, OutputRules, MultiPass, XSLTFiles
`BruteForceGeneration`	Calls an assembly's method (through reflection) multiple times creating multiple files		Standard, OutputRules, Process, MultiPass,
`CodeDOMGeneration`	Calls an assembly's method (through reflection) multiple times creating multiple files for a specific language	`TargetLanguage`	Standard, OutputRules, Process, MultiPass,

Table 5-1. Directives Supported by the Code Generation Harness, continued

DIRECTIVE	DESCRIPTION	ATTRIBUTES	CHILD ELEMENTS
`RunSQLScripts`	Processes a script file, normally to load a stored procedure	`Server`, `Database`	Standard, MultiPass, OutputRules
`NestedScript`	Runs a nested script defined in a separate harness XML file	`ScriptName`	Standard

Exploring Child Elements

Child elements provide standardized access to sets of attributes used during processing. In the user interface, each child element appears as a group box. Child elements are defined in the XSD, and several different directives may use the same child element definition. Reusing the child element definition simplifies the XSD and provides predictable behavior when you're working with the directives. For example, you can count on the `Standard` element to always contain the same three attributes. Table 5-2 describes the individual attributes defined in each child element. `Process` and `XSLTFiles` are logically incompatible as are `SinglePass` and `MultiPass`. The following sections discuss the specific meaning of each attribute.

Table 5-2. Child Elements for Code Generation Directives[1]

CHILD ELEMENT	DESCRIPTION	CONTAINED ATTRIBUTES
`Standard`	Standard attributes associated with all elements	`Name`, `Ordinal`, `Checked`
`Process`	Defines a process that can be run via reflection	`AssemblyFileName`, `TypeName`, `MethodName`

1. One cool thing about Table 5-2 is how I built it. I already had an XSD for the harness XML file to create the data-driven generation harness user interface. I wrote an XSLT that transformed the XSD into the proper format and copied that into Microsoft Word. Cool, huh?

Table 5-2. Child Elements for Code Generation Directives, continued

CHILD ELEMENT	DESCRIPTION	CONTAINED ATTRIBUTES
XSLTFiles	Defines an XSLT template and the appropriate input file	XSLTFileName, InputFileName, AssemblyFileName, TypeName, NamespaceURI
SinglePass	Defines a single-pass process with a single output file	OutputFile
MultiPass	Defines a multipass process with varying paramters and output filenames	SelectFile, SelectXPath, SelectNamespace, SelectNSPrefix, OutputDir, OutputFilePattern
OutputRules	Defines how hashing and comparison with existing files will occur	OutputFileType, OutputGenType, HashOutput
Parameters	Defines parameters for processes	Parameter

Introducing the Standard Element

All directives use the Standard child element. The Standard element contains the Name, Ordinal, and Checked attributes. The harness doesn't use the Name attribute. It exists for the convenience—no, the sanity—of programmers. If you have ten or twenty directives without names, you won't know which is which. Each Name attribute should describe the purpose of the directive within the current harness file, such as Generate Data Container or Generate Data Access Layer. Names should be unique, at least within the current section—again, to help keep you and follow-on programmers sane.

The Checked attribute determines whether the directive appears checked in the harness TreeView (refer again to Figure 3-1) and whether the harness will process the directive. The directive is processed if the value of the checked attribute is true. The Ordinal attribute determines the order in which the harness processes the directives within the section. An example of a Standard element is as follows:

```
<kg:Standard
          Name="GenerateDataAccessLayer"
          Checked="true"
          Ordinal="0" />
```

Introducing the Process Element

Several directives run a process—*methods*—in a .NET type and assembly. For example, the `CreateMetadata` directive uses a process to extract SQL metadata. In this case, the harness supplies a default process, which you can override. For example, you might want to override the default process if you've created a different metadata extraction tool to access Oracle. In other cases, such as brute force or CodeDOM code generation, you have to supply information about the .NET method to run. In these cases, the harness can't predict a logical default for what routine you'll want to run.

You'll specify a .NET method via its name, class name, and assembly for each process element. As mentioned in Chapter 3, the harness uses reflection to determine what parameters to pass to your routine. You should have values corresponding to these parameter names in the XML metadata node you're processing. Because these operations are stateless, the harness expects these methods to be shared (C# `static`).

The `Process` and `AlternateProcess` elements contain identical attributes. These attributes are `AssemblyFileName`, `AssemblyName`, `TypeName`, and `MethodName`. This is enough information for reflection to call the method. If you specify the `AssemblyName`, the harness will try to locate the assembly among currently loaded assemblies. If you specify the `AssemblyFileName`, the harness will load the assembly. If you don't specify either the `AssemblyName` or the `AssemblyFileName`, the harness will look for the `TypeName` in the current assembly (CodeGenerationSupport.dll). If you don't specify the type, the harness will look for the `MethodName` in the current type (the harness's `Generation` class). The `Process` and `XSLTFiles` child element can't be used together. An example of a `Process` child element is as follows:

```
<kg:Process AssemblyFileName=
          "[BasePath]Chapter 3\BruteForceExample\bin\BruteForceExample.dll"
        AssemblyName=""
        TypeName="BruteForceExample"
        MethodName="GetStream" />
```

NOTE `[BasePath]` represents an abstract path. This is discussed further in the later "Abstract Paths in Filenames" section of this chapter.

Introducing the XSLTFiles Element

The `XSLTFiles` element supports XSLT processing by defining the template file and XML input document. Both `XSLTProcess` and `XSLTGeneration` use this child element. `XSLTProcess` processes the XSLT once and outputs a single file. `XSLTGeneration` processes the XSLT multiple times, with different parameters and different output files. The `XSLTFiles` element can also contain attributes for `AssemblyName` and `TypeName`. If these are included, they're used for callbacks into your .NET code—called *extension objects* in XSLT terminology (see Appendix A for further discussion of extension objects). This technique is especially useful when you have functionality that you want to use in both your .NET processes (including metadata extraction) and in XSLT templates. An example of this is writing a .NET function that recognizes keywords that require special handling. By using a .NET function, you can access this functionality both from .NET and your templates. It also lets you drop into a familiar language to solve issues that are especially tricky in XSLT. When you use extension objects, you must assign a namespace for the extension objects in the XSLT file. You specify this namespace via the `NamespaceURI` attribute. The harness uses the `AddExtensionObject` of the `XSLTArgumentList` to define the namespace and extension object to use:

```
<kg:XSLTFiles
      InputFileName="[BasePath]Metadata Files\Northwind.xml"
      XSLTFileName="[BasePath]Chapter 3\Test\EmptyDataAccessor.xslt"
      AssemblyFileName=""
      AssemblyName=""
      TypeName=""
      NamespaceURI=""/>
```

CAUTION There are issues returning node-sets from .NET code called as extension objects in .NET. Be careful relying on this technique in current versions of .NET.

NOTE Using extension objects requires that your application is running with a security setting of full trust.

Introducing the SinglePass Element

Single-pass processing performs a task a once. The `SinglePass` element just defines the output files for this processing and is the simplest of the defined child elements. This element is used as a child of elements such as `XSLTProcess`, `RunProcess`, and `CreateMetadata`. Other child elements define other aspects of the task. An example of a `SinglePass` element is as follows:

```
<kg:SinglePass OutputFile="[BasePath]Chapter 3\Output\Table3-2.txt" />
```

Introducing the MultiPass Child Element

The four multipass code generation directives—`XSLTGeneration`, `BruteForceGeneration`, `CodeDOMGeneration`, and `RunSQLScripts`—face the same set of challenges. You need to cycle through metadata in a controlled manner doing *some action* for each occurrence of *something* in your metadata. For example, you might need to loop through all tables in a metadata file running a particular XSLT template for each table with the table name as an input parameter.

When you cycle through the metadata performing a task multiple times, you'll create multiple output files—one for each condition. For example, you might create a select stored procedure script for each table in your database. Multiple output files present two challenges. You have to know the context in the XML metadata for each pass, and you need a unique and logical filename for each output file. The `MultiPass` child element provides information to solve both of these problems.

To select the nodes to process, the harness uses the XPath expression specified in the `MultiPass` child element. These nodes are selected from the XML file specified in the `SelectFile` attribute. The XPath expression often requires a namespace defined by a prefix, so you can supply both the `SelectNamespace` and `SelectNSPrefix`.

NOTE In most cases, the XML file is the metadata file you're using for processing. Brute-force and CodeDOM generation pass the selected node to the method that outputs the code. For XSLT generation, the `SelectFile` could be a separate manual metadata file. You might do this to create user interface forms or pages on a subset of your business objects defined in manually entered XML file.

To create names for the output files, use a pattern defined in the `MultiPass` child element and information from the node being processed. Token replacement

lets you replace part of the filename with a value—often the `Name` attribute—from the selected node. Tokens are offset by angle brackets, which need to be escaped within XML using the `<` and `>` escape sequences. To avoid overwriting files, plan your naming so the filename is unique after the token replacement. An example of a `MultiPass` child element is as follows:

```
<kg:MultiPass
      SelectFile="[BasePath]Metadata Files\Northwind.xml"
      SelectNamespace="http://kadgen/DatabaseStructure"
      SelectNSPrefix="dbs"
      SelectXPath=
            "/dbs:DataStructures/dbs:DataStructure/dbs:Tables/dbs:Table"
      OutputDir="[BasePath]Chapter 3\Output\XSLT Output"
      OutputFilePattern="&lt;@Name&gt;DataAccessor.vb" />
```

The `OutputFilePattern` in this example would result in filenames such as CustomerDataAccessor.vb or ProjectDataAccessor.vb if the current nodes were `Customer` and `Project`.

Introducing the OutputRules Element

All of the directives that write output to files use the `OutputRules` child element. This element specifies whether to generate a hash code and the rules for overwriting files. Hash codes let you test whether anyone edited a file after it was created. They do this because the marker placed on the file when it's generated is more or less unique for that file. You can later check whether a new calculation of the files hash code matches the original. If they match, the file was almost certainly not edited. The `OutputGenType` determines the rules for overwriting files, and the section "Overwriting Files" in Chapter 3 discusses details about using this attribute. File creation/overwriting falls into four categories, each represented by a different `OutputGenType` as follows:

- **Once**: The file is generated only if the file doesn't already exist.
- **UntilEdited**: The file is generated unless an existing file has been manually edited. No error is recorded if the file has been edited. This requires a hash code, and if the hash code is missing, an error occurs.
- **Always**: The file is generated but won't overwrite an edited file. An error is recorded if the file has been edited or the hash code is missing.
- **Overwrite**: The file is always generated overwriting existing file, regardless of whether the file has been edited. No hash is required.

Directives marked to generate `Once` are easy to accommodate. If the file exists on the disk, don't regenerate it.

You'll create files during code generation that you expect programmers to edit. For example, Chapter 4 discussed derived editable files. Using `UntilEdited` lets you automatically incorporate template modifications in the majority of your classes as your template evolves early in your application development.

`Always` indicates that you don't expect the file to be manually edited, and manual edits represent an error. The difference between the `UntilEdited` and `Always` is whether finding a manually edited file is treated as an error condition. `UntilEdited` doesn't overwrite and generates a warning rather than an error. `Always` doesn't overwrite either but logs an error.

To make the `UntilEdited` and `Always` modifiers work, you have to know whether a programmer manually edited the file. A hash marker stored as a file comment can contain a representation of the file contents. Yeah, hashes are kind of ugly to look at, but they're very effective.

TIP You can use the hash code algorithms in the harness to write a program to report what files have been edited and therefore won't be generated. If you want a simpler approach, check the modified date of the files unless your source control tool munges the time stamp on files. This check is easy if the output of each template is in a separate directory, as described later in the chapter.

Code generation doesn't overwrite edited files unless you use the `Overwrite` generation type. Don't use this generation if you can avoid it! Overwriting when this generation type is used leaves little trace that edits were lost (source control may contain the changes). Table 5-3 shows what happens with each of these indicators in various file conditions.

CAUTION Don't use the `Overwrite` directive if you can avoid it. All other directives respect edited files. This directive will overwrite files regardless of their status and is intended for use in rare cases.

This is an example of an `OutputRules` child element:

```
<kg:OutputRules
      HashOutput="true"
      OutputFileType="VB"
      OutputGenType="UntilEdited"
      CheckOut="false"
      CheckOutProject="" />
```

Table 5-3. Action Taken with Different Indicators and File Conditions

INDICATOR	INTENT	FILE DOESN'T EXIST	FILE EXISTS, NOT EDITED	FILE EDITED
Never	Files not code generated	No action	No action	No action
Once	Files intended for immediate edit or with a very stable template. Rarely used.	Created	No action	No action
UntilEdited	Files intended for editing, but whose templates might change. Frequently used.	Created	File Overwritten	File preserved; warning logged
Always	Files intended for ongoing regeneration. Frequently used.	Created	File Overwritten	File preserved error logged
Overwrite	A safety hatch where hash is inappropriate. Rarely used.	Created	File Overwritten	File overwritten

Introducing the Parameter Element

The `RunProcess` directive often needs parameters. The `Parameter` child element contains the values for these parameters. Pipe characters (|) delimit the individual parameters. Each parameter value is passed through the methods that replace abstract paths—so you can use abstract filenames in these parameter values. Use commas to delimit members of array parameters and enclose the array in curly brackets. An example of a Parameter child element is as follows:

```
<kg:Parameter Parameter="[ProjectFile]|
                         [FileList]|
                         {Autogen,Editable}" />
```

Exploring Specific Directives

Table 5-1 shows the supported directives along with what attributes and child elements each supports, but the brief description in the table doesn't say much about when and how you use each one. Now that you've seen the information each child element contains, the following sections discuss each directive in more detail.

Using the XSLTProcess Directive

Use the `XSLTProcess` directive when you want to perform an XSLT transformation once, producing a single output file with a constant name. The `XSLTProcess` directive is a single-pass directive, and the `XSLTFile` child element specifies the XML input file processed and the XSLT file to use for the processing.

An example of how you'll use this directive is managing the metadata stream. Numerous steps affect your metadata each time you run your script, and many of these steps are `XSLTProcess` directives that iteratively make the metadata friendlier to code generation. Some changes, such as those that incorporate ORM information discussed in Chapter 6, are complex and require several `XSLTProcess` steps to complete. An example of the XML for an `XSLTProcess` directive is as follows:

```
<kg:XSLTProcess>
  <kg:Standard Name="Create Interim Object " Checked="true" Ordinal="0" />
  <kg:OutputRules HashOutput="false" OutputFileType="XML"
        OutputGenType="Overwrite" CheckOut="false" CheckOutProject="" />
  <kg:SinglePass OutputFile="[Temp]ORMObjectOutput.xml" />
  <kg:XSLTFiles InputFileName="[Temp]ORM2WithDBS.xml"
        XSLTFileName="[ORMTransform]ORMObjectOne.xslt" AssemblyFileName=""
        TypeName="Tools" NamespaceURI="http://kadgen.com/NETTools"
        AssemblyName="Utility" />
</kg:XSLTProcess>
```

Using the RunProcess Directive

Use the `RunProcess` directive when you want to run a .NET method that produces a single file. This directive is similar to the `XSLTProcess` directive. They're both `SinglePass` directives, and you use them to perform miscellaneous functions during your application code generation. An example of when you might need to do this is creating the list of files you'll include when building the project file—see

the section "Building Project Files." An example of the XML for a `RunProcess` directive is as follows:

```
<kg:RunProcess>
  <kg:Standard Name="BuildFileList" Checked="true" Ordinal="97" />
  <kg:OutputRules HashOutput="false" OutputFileType="XML"
                  OutputGenType="Overwrite" />
  <kg:SinglePass OutputFile="[FileList]" />
  <kg:Parameter Parameter="[ProjectDir]" />
  <kg:Process AssemblyName="Utility" TypeName="FileTools"
                  MethodName="GetFileListXML" AssemblyFileName="" />
</kg:RunProcess>
```

Using the CreateMetadata Directive

Use the `CreateMetadata` directive to retrieve metadata from a database. The directive itself contains a `Server` and `Database` attributes that specify what database to extract metadata from. Chapter 2 explains how critical metadata is in your code generation process. You can perform multiple metadata extraction steps across different databases, but each uses a separate directive. The SkipStoredProcedures attribute allows you to skip this time consuming step when you don't need stored procedure information. The SelectPatterns and SetSelectPatterns let you skip running other stored procedures that don't return values. If you prefix your database tables (as suggested in Chapter 2) you can remove this prefix by setting RemovePrefix to true. The LookupPrefix lets you identify lookup tables automatically if they have a unique prefix.

NOTE The harness includes a metadata extraction tool that probably extracts all the metadata you need if you're using SQL Server. If you want to run an alternate metadata process, you can specify it using the `AlternateProcess` child element.

An example of the XML for a CreateMetadata directive is as follows:

```
<kg:CreateMetadata Server="localhost" Database="PTracker"
         SkipStoredProcs="false" SelectPatterns="" SetSelectPatterns="">
   <kg:Standard Name="SQL Extraction" Checked="true" Ordinal="0" />
   <kg:OutputRules HashOutput="false" OutputFileType="XML"
         OutputGenType="Overwrite" CheckOut="false" CheckOutProject="" />
   <kg:SinglePass OutputFile="[Metadata]" />
   <kg:AlternateProcess />
</kg:CreateMetadata>
```

Using the MergeMetadata Directive

Use the MergeMetadata directive when you want to merge multiple XML metadata files into a single output. The output is often a new file, but the output could overwrite either of the input files. The directive contains the BaseXMLFileName and MergingXMLFileName. This directive specifies that you want to merge the MergingXMLFileName file into the BaseXMLFileName attribute. Values from the MergingXMLFileName file overwrite values of any matching attributes in the BaseXMLFileName file. An example of the XML for a MergeMetadata directive using abstract paths is as follows:

```
<kg:MergeMetadata BaseXMLFileName="[BasePath]ORMChapter8.xml"
         MergingXMLFileName="[Metadata]"
         OutputXMLFileName="[Temp]ORM2WithDBS.xml">
   <kg:Standard Name="Merge ORM" Checked="true" Ordinal="0" />
</kg:MergeMetadata>
```

Using the XSLTGeneration Directive

Use the XSLTGeneration directive when you want to process an XSLT template to create multiple output files. You'll use this process to generate code files if you're working with XSLT code generation. You'll often run this process across all of the nodes in your metadata that define tables or objects and create an output file for each table. You're likely to have multiple XSLTGeneration directives, each generating a set of files from a different XSLT stylesheet. For example, one stylesheet might create middle tier data containers while another creates stored procedures. Chapter 4 and Chapters 6–10 explore additional types of classes you might create, each using its own template. The MultiPass child element defines what nodes to process and how to name each output file. The XSLTFiles child element defines the XSLT file and XML metadata file to use. See the Chapter 4 section "Providing Parameters" for a discussion of the parameters passed to the XSLT stylesheet. The following is an example of the XML for an XSLTGeneration directive:

```
<kg:XSLTGeneration>
  <kg:Standard Name="Genned Business Objects" Checked="true" Ordinal="0" />
  <kg:OutputRules HashOutput="true" OutputFileType="VB"
        OutputGenType="Always" CheckOut="false" CheckOutProject="" />
  <kg:MultiPass SelectFile="[GenInputMetadata]" SelectXPath="//orm:Object"
        SelectNamespace="http://kadgen.com/KADORM.xsd" SelectNSPrefix="orm"
        OutputDir="[Output][BusinessObjects]Autogen\genCSLAEditSwitchable"
        OutputFilePattern="gen&lt;@Name&gt;.vb" />
```

```
  <kg:XSLTFiles InputFileName="[GenInputMetadata]"
        XSLTFileName="[XSLT][BusinessObjects]genCSLAEditSwitchable.xslt"
        AssemblyFileName="" TypeName="" NamespaceURI="" AssemblyName="" />
</kg:XSLTGeneration>
```

Using the BruteForceGeneration Directive

Use the `BruteForceGeneration` directive when you want to create files by calling the specified .NET method multiple times, passing different parameters to each call. You'll use this process to generate code files if you're working with brute-force code generation. The `BruteForceGeneration` directive uses the `MultiPass` child element to determine the data to process. It uses the `Process` child element to specify the method that performs processing. The harness uses reflection to open the specified assembly, find the specified type (class), and call the specified shared (C# `static`) method. The assembly and type default to the current assembly and type (the harness's Generation class) if they're not specified.

You might be tempted to output multiple code files from a single method call. This is a bad idea because it bypasses checking whether it's appropriate to overwrite the file and avoids the automatic hashing of appropriate files. The harness makes it easy to call your method once for every file you want to create, such as for every table defined in your metadata. It also makes it easy to call multiple methods, each using a different directive and creating a different file.

The following is an example of the XML for a `BruteForceGeneration` directive:

```
<kg:BruteForceGeneration>
  <kg:Standard Name="BruteForceGeneration" Checked="true" Ordinal="0" />
  <kg:OutputRules HashOutput="true" OutputFileType="VB"
        OutputGenType="Always" CheckOut="false" CheckOutProject="" />
  <kg:MultiPass SelectFile="[MetadataFileName]"
        SelectNamespace="http://kadgen/DatabaseStructure"
        SelectNSPrefix="dbs" SelectXPath=
                "//dbs:DataStructures/dbs:DataStructure/dbs:Tables/dbs:Table"
        OutputDir="[Chapter3]Output\BruteForce Output"
        OutputFilePattern="&lt;@Name&gt;.vb" />
  <kg:Process AssemblyFileName=
                "[Chapter3]BruteForceExample\bin\BruteForceExample.dll"
        TypeName="BruteForceExample" MethodName="GetStream"
                AssemblyName="" />
</kg:BruteForceGeneration>
```

Using the CodeDOMGeneration Directive

Use the CodeDOMGeneration directive when you want to generate code using the CodeDOM. The only difference between the CodeDOMGeneration directive and the BruteForceGeneration directive is the TargetLanguage attribute of the CodeDOMGeneration directive. Using the correct directive will also indicate your intent to a future maintenance programmer. If you want to generate code in multiple languages simultaneously, you'll have to use multiple directives. This means you'll be rebuilding your CodeDOM graph (discussed later in this chapter), which is less efficient than reusing the CodeDOM graph. However, the alternative is to increase the complexity of the harness user interface to allow multiple target languages or some other approach. You can make this modification if you're doing a lot of CodeDOM generation, but re-creating the graph also gives you a path for including language-specific snippets that are often necessary to overcome CodeDOM limitations. The following is an example of the XML for a CodeDOMGeneration directive:

```
<kg:CodeDOMGeneration TargetLanguage="VB">
    <kg:Standard Name="CodeDOMGeneration - VB" Checked="true" Ordinal="0" />
    <kg:OutputRules HashOutput="true" OutputFileType="VB"
          OutputGenType="Always" />
    <kg:MultiPass SelectFile="[MetadataFileName]"
          SelectNamespace="http://kadgen/DatabaseStructure"
          SelectNSPrefix="dbs" SelectXPath=
                "//dbs:DataStructures/dbs:DataStructure/dbs:Tables/dbs:Table"
          OutputDir="[Chapter3]Output\CodeDOM VB Output"
          OutputFilePattern="&lt;@Name&gt;.vb" />
    <kg:Process
          AssemblyFileName="[Chapter3]CodeDOMExample\bin\CodeDOMExample.dll"
          TypeName="CodeDOMExample" MethodName="BuildGraph" AssemblyName="" />
  </kg:CodeDOMGeneration>
```

Using the CopyFiles Directive

Use the CopyFiles directive to just copy files from one location to another. You might use this to copy individual support files from a central location. You can skip this and manually copy these files, but then you'll have to plan other strategies for recording how you built your application—it wouldn't all be in your code generation script, and you'd be starting down the road to that two-page checklist. As you'll see in Chapters 6 and 10, code generation gets really exciting when generic components, metadata, and handcrafted code come together to

build your application. `CopyFiles` helps you manage this complexity with a single button click. The following is an example of the XML for a `CopyFiles` directive:

```
<kg:CopyFiles SourceFileName="text.txt" TargetFileName="..\text2.txt">
      <kg:Standard Name="Copy File" Ordinal="0" Checked="true" />
</kg.CopyFiles>
```

Using the RunSQLScripts Directive

Use the `RunSQLScripts` to run scripts against a SQL Server database. This is a little different from other multipass processes. In this case, the `OutputDir` and `OutputFilePattern` define the output file created earlier in the generation process that should now be run against the SQL Server database. You could run these scripts manually (such as executing the scripts in Query Analyzer), but again it'd move you away from one-click code generation.

When you work with SQL, there are two hash codes. One is created by the `XSLTGeneration` or `BruteForceGeneration` output and represents the script file on disk. This ensures that you don't inadvertently overwrite the script. The other is created by the `RunSQLScript` when it runs the script against your database. This helps protect inadvertent edits of your stored procedures within your database (such as someone editing them via Query Analyzer). The following is an example of the XML for a `RunSQLScripts` directive:

```
<kg:RunSQLScripts Server="localhost" Database="PTracker">
  <kg:Standard Name="Run 'Retrieve' Scripts" Ordinal="0" Checked="true" />
  <kg:OutputRules OutputFileType="StoredProc" OutputGenType="Overwrite"
          HashOutput="true" CheckOut="false" CheckOutProject="" />
  <kg:MultiPass SelectFile="[SPInputMetadata]"
          SelectXPath="//orm:SPSet[contains(@Generate,'R')]"
          SelectNamespace="http://kadgen.com/KADORM.xsd" SelectNSPrefix="orm"
          OutputDir="[Output][StoredProc]Retrieve"
          OutputFilePattern=
                "&lt;orm:RunSP[@Task='Retrieve']/@Name&gt;.prc" />
</kg:RunSQLScripts>
```

Using the Nested Scripts Directive

Use the `NestedScripts` directive to run a harness script from within another harness script. This lets you avoid maintaining identical scripts, or portions of scripts. The only item you have to specify is the filename of the script to run. You

might use this if you have a process—such as a multistep ORM transformation process—that you want run in the same manner throughout your organization.

CAUTION The current version of the harness doesn't handle the progress bar or output dialog box well with nested scripts.

NOTE You'll see this directive in Chapters 6–10 where code in each chapter also runs scripts from earlier chapters. Each chapter produces a single type of output, and using nested scripts might suggest maintaining multiple scripts—one for each type of output. But, that's just to keep things aligned with the book content. For your own code generation, you'll want to use a single script, and the ReadMe file in the downloads from the Apress Web site explains how to create a single harness script file.

The following is an example of the XML for a `RunSQLScripts` directive:

```
<kg:NestedScript ScriptName="[Chapter6]Harness6NETGenerate.xml">
  <kg:Standard Name="Chapter 6 Operations" Checked="true" Ordinal="0" />
</kg:NestedScript>
```

Organizing Files via the Directory Structure

A stable and consistent directory structure will help your development, particularly if you work in a group. The harness that generates your code doesn't care where you put files and allows you to be a bit sloppy with your directory structure, as long as it can find the appropriate files. However, the directory structure you use contributes to how easily you can manage source control, synchronize files on multiple development machines, and move your framework forward to create new projects. Although you could organize your directories several different ways and still be effective, I'll stick to showing you one approach that's simple, works well, and is friendly to the tool that moves your framework forward to your next application.

Finding an Effective Directory Organization

Figure 5-2 shows an example of a directory structure. The framework copy tool shown later in this chapter builds new projects from your existing ones and works from a single root directory (although you can run the tool multiple times

across different directories). Files that you share between applications shouldn't reside in this directory or any of its subdirectories. The single application root contains a subdirectory for each project.

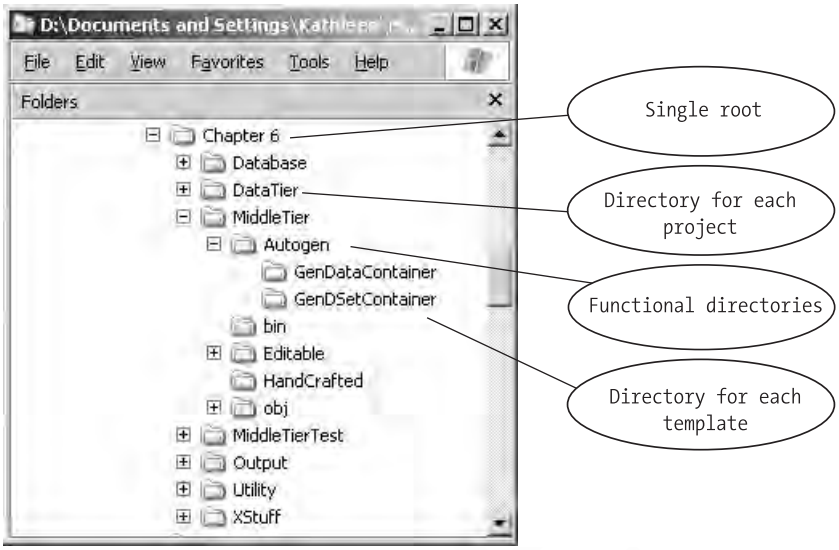


Figure 5-2. An example of a directory structure that works well with code generation tools

---

TIP At the top of Solution Explorer in Visual Studio, one of the graphics-only buttons is Display All Files. Another is Refresh. You may need to click Refresh after code generation to update the Solution Explorer list. You'll need to do this for each project that has new or deleted files. Display All Files will show all files and directories, including those not included in the project.

Within project directories, arrange files in separate subdirectories by the `GenType` such as Autogen, Editable, Handcrafted, and so on. This helps programmers quickly see how code generation will affect any file. The names of the directories aren't particularly important, as long as they're clear to programmers working on the project.

TIP Use a set of subdirectories you can live with easily. The framework copy tool will carry this structure forward, and being consistent across different projects in your organization will help you and later programmers.

Within each of these GenType subdirectories, use a separate subdirectory for each template. You'll generally want these directory names to match the template name, but the mapping is manual, so some variations, such as adding a prefix or suffix, are fine. Easily mapped names will significantly help programmers debug your applications. When they find a problem in a generated file, they can find the responsible template just by glancing at the file's position in the Solution Explorer.

TIP If you haven't worked with source files in subdirectories in Visual Studio, it's a great trick for managing large projects. The subdirectories appear in Solution Explorer, allowing you to control the number of source files visible at any time.

This depth of your directory structure impacts how easy it is to work with. If it's too shallow, too many files wind up in the same directory, which makes it difficult to find the files you want. Too deep, and you spend excess time opening subdirectories and looking for a particular file. I think this structure is a good compromise—not too deep, not too shallow, with the appropriate amount of useful information overloaded into directory names.

TIP If you have trouble finding a file in Solution Explorer, switch to Class View to find the file. Visual Studio synchronizes the Class View and Solution Explorer, so this trick works both ways. You can use the Class View and then switch to find the physical files location, and you can use Solution Explorer and then switch to find the location in the object hierarchy.

NOTE Chapter 9 introduces a variation of this directory structure that contains the AutoGen, Editable, and Layout directories.

Using Abstract and Relative Paths

Chapter 3 describes the concept of abstract paths. The `FilePath` elements of the harness contain mappings from abstract file paths to the corresponding real path. All paths in the harness are relative to the location of the XML harness

script file. This allows different developers with different overall directory layouts to use the same script, as long as the project directory is laid out the same. If the project directory structure is the same, it doesn't matter where this directory is located on the programmer's machine.

NOTE Without abstract paths, search and replace isn't reliable for path replacements in harness files because there are several ways you can represent a given path.

TIP If you want to use absolute paths, start your path names with a drive letter.

You'll use these abstract paths as shortcuts into your directory structure. Using abstract paths will save some typing—you won't have to worry about typing long directory paths correctly every time. However, the key element they provide is abstraction. The abstraction lets you change directory names in only one location. You won't want to change directory names often, even between projects because the filenames are relative to an application root. A stable predictable structure below this application root provides a level of consistency between projects.

Managing Template Projects

Where do you put your templates so you can easily load them in Visual Studio? If you're using brute-force or CodeDOM generation, your templates are in a separate Class Library project. If you're using XSLT code generation and you're editing your XSLT using Visual Studio, create a project to contain each logical set of templates. This can be any kind of project, but a class library or an empty project is logical.

You can organize your templates into a single project or multiple projects. Have the granularity of your organization meet or exceed the granularity of how you'll reuse your templates. In other words, if you expect to move your stored procedures as a group into your next project, you'll simplify the transition if you include the stored procedure templates in their own project and directory. I organize templates similarly to the chapters in the second part of this book—separate projects for ORM metadata preparation, stored procedures, business objects, and each user interface platform. You can use templates from multiple projects in the same harness script.

TIP Don't include your templates with the application project you're building. Errors in both the templates and the generated code will appear together in the task list, making it difficult to isolate specific types of errors.

NOTE If you're editing XSLT files in Visual Studio, you can just open XSLT templates using File ➤ Open File. This can become a pain because they're opened in the Miscellaneous node of Solution Explorer. This file mode doesn't allow you to rename files, doesn't indicate the path of the flies, and removes files from the list in an apparently random pattern. Other problems include poor Task List support and the inability to create new files. You can solve these problems by creating a project for sets of XSLT templates and a solution that combines these projects for display and editing.

Building Project Files

You can automate the maintenance of project files (vbproj and csproj files), but in the context of a real project it's not as easy as you might expect. The problems arise in determining the list of files to include. You could use all .vb or .cs files in a directory and its subdirectory, but problems arise in at least the following situations:

- **Referenced projects**: You'll often reference additional projects that project file maintenance will have to support.
- **Handcrafted files**: You're likely to add additional handcrafted files that you need to include in your projects.
- **Extra files in the directories**: The directory containing your project may contain extra files that shouldn't be included in the project.

Because these conditions tend to exist in every project, you'll have to solve them to effectively automate the updating of your project file. If you don't provide automatic updating; you can instead manually ensure that all of your generated files are included in the project. That can work well if your project files don't

change often or you want to control what's included in the project (you may generate files you aren't yet ready to include in the build). I manually update my own projects and solutions but wanted to explain how to automate this. You can also automate the compile if that seems helpful to your project. I don't cover automating the compile because I prefer to compile within Visual Studio. That's because the Task List gives me a quick avenue for tracking down any problems that arise.

NOTE It's a bit trickier to automate your solution files, and you don't gain much. Once solution files are created, they're relatively easy to maintain. The section "Moving the Project Forward" later in this chapter explains how to create a new solution file from an existing project.

The key to automating project files is to update existing files, not generating them from scratch. This lets you respect any enterprise templates or customizations to the default projects. Before jumping into how to build project files, I'll review the structure of a project file so you'll understand the target output.

Reviewing the Project File Structure

Project files are XML files that contain, among other things, an element for each contained file. You can load the project file in the XML DOM and update the section that contains the included files. You can use file paths to indicate which groups of files you want to automatically include.

NOTE Visual Basic and C# project files are similar. I'll show the Visual Basic version, and if you're interested in C#, you might want to open a C# project file in Notepad to compare.

The project file opens with a root element that identifies the project as a Visual Studio project; Visual Studio and .NET compilers use the version to ensure the source code is a version they can compile. The project element includes a unique project GUID. This GUID identifies the project within Visual Studio and SourceSafe, and it's important that it's unique. To maintain unique GUIDs, you need to avoid copying project files. The later "Building a New Project from Your Framework" section discusses this GUID further and shows a mechanism for

creating a new project file with a unique GUID from an existing project. This element of project files looks like this:

```
<VisualStudioProject>
   <VisualBasic ProjectType="Local"
                ProductVersion="7.10.3052"
                SchemaVersion="2.0"
                ProjectGuid="{1F6D9FE2-CEAD-4909-855F-CA4573BED599}">
```

CAUTION Don't copy project files because you'll have redundant GUIDs that can cause problems with Visual Studio and Visual SourceSafe.

The `Build` element of the project file contains a `Settings`, `References`, and `Imports` element. The `Settings` node contains a number of attributes that I'll skip to simplify this discussion. Within the `Settings` section, you'll find a configuration element for each of the build configurations you've defined:

```
<Build>
   <Settings [[ ... ]] >
      <Config Name="Debug" " [[ ... ]] />
      <Config Name="Release" [[ ... ]] />
   </Settings>
```

NOTE Ellipses in these listings show where I've left out attributes. You can load the vbproj file into Notepad to see all the attributes included for your projects.

The `References` element lists all the assemblies referenced by the project. It references assemblies by name if they're outside the current solution and references projects within the current solution by the project GUID. The `References` element looks like this:

```
<References>
   <Reference Name="System" AssemblyName="System" />
   <Reference Name="System.Data" AssemblyName="System.Data" />
   <Reference Name="System.XML" AssemblyName="System.Xml" />
```

```
    <Reference Name="DataTier"
            Project="{9BC19437-19C5-47F5-AA6F-7D379940B8B5}"
            Package="{F184B08F-C81C-45F6-A57F-5ABD9991F28F}" />
    <Reference Name="Utility"
            Project="{7E5A07E6-48EA-4B71-9A09-CEECC9378F65}"
            Package="{F184B08F-C81C-45F6-A57F-5ABD9991F28F}" />
    <Reference Name = "Office"
            Guid = "{2DF8D04C-5BFA-101B-BDE5-00AA0044DE52}"
            VersionMajor = "2"
            VersionMinor = "1"
            Lcid = "0"
            WrapperTool = "tlbimp" />
  [[ ... more files ]]
</References>
```

The `Imports` element contains any project-wide `Imports` statements set in the Project Properties dialog box and doesn't appear in C# projects:

```
     <Imports>
      <Import Namespace = "System" />
    </Imports>
</Build>
```

It's easiest to manage the information in the `Settings`, `Imports`, and `References` sections via the tools in Visual Studio once the project is set up. You don't want to generate these settings because they rarely change, and you don't want to overwrite the changes programmers to make in the Project Properties dialog box. The easiest way to leave the programmer in control is to leave this stuff alone, except when you're creating a new project as discussed in the section "Moving Your Project Forward." Alter only the contents of the `Files` element's child node `Include`. Update this list to include files in your project. The `Include` node looks like this:

```
      <Files>
        <Include>
            <File RelPath="AssemblyInfo.vb" SubType="Code"
                      BuildAction="Compile" />
            <File RelPath="XMLTest.xml" BuildAction="Content" />
            [[ ... more files ]]
        </Include>
    </Files>
  </VisualBasic>
</VisualStudioProject>
```

Each file includes its name as a path relative to the project file location, a `SubType`, and a `BuildAction`. The `BuildAction` indicates whether the compiler should include the file or whether it should be deployed as part of the project content. Options are `None`, `Compile`, `Content`, and `EmbeddedResource`. See the Visual Studio help files for more information about when to use each.

Building Project Files

You'll perform the following three processes to build the project file:

- **Build file list**: Create the file list.
- **Merge file lists**: Merge a manual metadata file containing exceptions.
- **Run a template**: Run a brute-force template that outputs the project file.

These three actions combine to update the project file. You can create the file list from the list of files in a specified directory and its subdirectories. The files you want in your project may not exactly match the files that exist in the directory. Particularly during early development, some files may not yet be ready to support a clean build or you may have temporary files that you don't want to include. Because maintaining the manual file list takes extra work, you'll want to keep the directories free of extraneous files when you can, but the process supports ignoring specific files or adding additional files. Once the file list is correct, a brute-force template updates the file section of the project file.

Like other automated process, updating the project file list takes metadata and a template. Creating the metadata for this process involves creating a list of files in an XML format. I use a brute-force approach to generate the project files. There are actually two input XML files—the list of files and the existing project file.

I'll start by showing you the directives that direct the process in the XML harness file. Then I'll discuss the individual steps.

Directing the Process

The process of building the project file uses six `FilePath` elements to specify directories and files. These specify three files. The `ProjectFile` is the project file you're updating. I suggest you use a backup scheme, which I'm doing by creating a new file specified in the `OutputProjectFile` abstract filename. Instead, you might use the `CopyFiles` directive to create a backup and overwrite your input project file. The interim FileList.xml file is the output from `Build File List` directive and is

also the input to `Merge File Lists` directive. The ManualFileList.xml contains any exceptions:

```
<kg:FilePaths>
  <kg:FilePath Name="BasePath" Path=".." File="" />
  <kg:FilePath Name="ProjectDir"
            Path="[BasePath]Chapter 8\BusinessObjects" File="" />
  <kg:FilePath Name="ProjectFile" Path="[ProjectDir]"
            File="BusinessObjects.vbproj" />
  <kg:FilePath Name="FileList"
            Path="[BasePath]Chapter 5\Temp" File="FileList.xml" />
  <kg:FilePath Name="ManualFileList"
            Path="[BasePath]Chapter 5" File="ManualFileList.xml" />
  <kg:FilePath Name="OutputProjectFile"
            Path="[BasePath]Chapter 5" File="Test.vbproj" />
</kg:FilePaths>
```

NOTE Because Chapter 5 doesn't create any full projects, I'm using the Chapter 8 directory structure for the example.

Each of the three actions that are part of updating the project file requires a directive in the harness XML file. The first directive is a `RunProcess` that creates the XML file list. It runs the `GetFileListXML` in the `FileTools` class of the `Utility` assembly. This directive is as follows:

```
<kg:RunProcess>
  <kg:Standard Name="Build File List" Checked="true" Ordinal="97" />
  <kg:OutputRules HashOutput="false" OutputFileType="XML"
                  OutputGenType="Overwrite" />
  <kg:SinglePass OutputFile="[FileList]" />
  <kg:Parameter Parameter="[ProjectDir]" />
  <kg:Process AssemblyName="Utility" TypeName="FileTools"
                  MethodName="GetFileListXML" AssemblyFileName="" />
</kg:RunProcess>
```

Note that the ordinal is set to 97 because you need this process to run near the end of processing for the section after any the files have been generated. This directive uses several of the capabilities discussed earlier in this chapter, including abstract paths and parameters. Providing the `AssemblyName` instead of the `AssemblyFileName` uses a previously loaded assembly.

The second directive merges any manually entered metadata into the XML file list you created via the previous directive. As discussed in the section "Providing Overrides," this step is optional if you don't exclude any files, but leaving it in place with comments in the XML file may clarify how a later programmer can exclude files:

```
<kg:MergeMetadata BaseXMLFileName="[FileList]"
                  MergingXMLFileName="[ManualFileList]"
                  OutputXMLFileName="[FileList]">
  <kg:Standard Name="Merge File Lists" Checked="true" Ordinal="98" />
</kg:MergeMetadata>
```

The merge initiated by this directive overwrites the `FileList` with values from the updated file list.

The final process actually updates the vbproj file. It can overwrite the existing project file, which can be currently open in Visual Studio (see the sidebar "'Modified Outside the Source Editor' Dialog Boxes" for a discussion of the Visual Studio behavior):

```
<kg:RunProcess>
  <kg:Standard Name="Update Project File" Checked="true" Ordinal="99" />
  <kg:OutputRules HashOutput="false" OutputFileType="XML"
                  OutputGenType="Overwrite" />
  <kg:SinglePass OutputFile="[OutputProjectFile]" />
  <kg:Parameter Parameter="[ProjectDir]BusinessObjects.vbproj|
                           [FileList]|
                           {Autogen,Editable}" />
  <kg:Process AssemblyName="CodeGeneration" TypeName="ProjectSupport"
                  MethodName="UpdateProjectFile" AssemblyFileName="" />
</kg:RunProcess>
```

That's it. Now you have the tool to automatically maintain your project files.

Building the FileList

You build the file list by retrieving a list of files in the project directory and its subdirectories. You'll merge this list with your manual file list—which allows you to exclude or add files. Then you'll use the list to update the files specified in the project file.

TIP To view XML files easily, open them in Internet Explorer, not Visual Studio. That's because Visual Studio doesn't yet support collapsing elements. You can uncheck the Reuse Windows for Launching Shortcuts option on the Advanced tab of Internet Explorer's Internet Options dialog box to easily open each XML file in separate instances of Internet Explorer.

The following is an example of part of the generated FileList.XML file with the `FullName` attributes removed:

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<FileLists>
  <FileList StartDir="..\Chapter 8\Output\BusinessObjects">
    <Dir Name="BusinessObjects" ">
      <Dir Name="Autogen" >
        <Dir Name="genCSLAEditCollection" >
          <File Name="genAssignments.vb" Ext=".vb"
                  Attributes="Archive" CreationUTC="9/8/2003 2:39:40 PM"
                  LastWriteUTC="10/7/2003 2:44:36 PM" Length="5064" />
          <File Name="genProjectAssignments.vb" Ext=".vb
                  Attributes="Archive" CreationUTC="9/8/2003 6:47:27 PM"
                  LastWriteUTC="10/7/2003 2:44:37 PM" Length="5176" />
```

NOTE I removed the `FullName` attributes for clarity.

FileList.XML contains a node for every file descendant from the start directory, organized by subdirectory. Within each directory, there's a `File` element for each corresponding file. This is exactly what you need to update the `Include` element of the vbproj file.

Building the File List XML

You can manually create the XML describing your files and directory structure, or you can automate it. I'm not sure there's any benefit to automating your project file updates unless you also automate creating the XML file list because it's easier to include/exclude files inside Visual Studio than separately maintaining an XML file. If you automate file list creations, you'll also need a mechanism to override entries in the automated list for special cases.

A RunProcess directive initiates retrieving the list of files into an XML document. Like other process directives, attributes specify an assembly, type, and method. Because the public shared method is in the Utility assembly, the assembly is already loaded. This method called GetFileListXML is in the FileTools class and outputs the file list as an XML document. The GetFileListXML method takes a single parameter that indicates where to start the search. It returns the XML using a stream, which allows standard file creation within the harness.

The GetFileListXML method creates the XML document and adds FileLists as the root node. FileLists contains a set of FileList elements. The FileList element includes the directory where the search starts. The DirectoryInfo object for the starting directory is passed to the FileListXML, which returns an XML node to complete the document. The code of the GetFileListXML method is as follows:

```
Public Shared Function GetFileListXML( _
            ByVal startDir As String) _
            As IO.Stream
   Dim xmlDoc As New Xml.XmlDocument
   Dim node As Xml.XmlNode
   Dim dir As New IO.DirectoryInfo(startDir)
   Dim stream As New IO.MemoryStream
   xmlDoc.AppendChild(xmlDoc.CreateXmlDeclaration("1.0", "utf-8", "yes"))
   node = xmlDoc.AppendChild(xmlHelpers.NewElement(xmlDoc, "FileLists"))
   node = node.AppendChild(xmlHelpers.NewElement(xmlDoc, "FileList"))
   node.Attributes.Append(xmlHelpers.NewAttribute(xmlDoc, "StartDir", _
                    startDir))
   node.AppendChild(FileListXML(dir, xmlDoc))
   xmlDoc.Save(stream)
   Return stream
End Function
```

FileListXML is a recursive routine. Each time you call it, it returns an XML node that represents the current directory and its subdirectories. Recursion handles any depth of nesting in the subdirectories—although I recommend you avoid deep nesting. Recursion leaves this method short and sweet. FileListXML recursively calls itself for each subdirectory and adds an element containing file information.

```
Private Shared Function FileListXML( _
            ByVal dir As IO.DirectoryInfo, _
            ByVal xmlDoc As Xml.XmlDocument) _
            As Xml.XmlNode
   Dim node As Xml.XmlNode = xmlHelpers.NewElement(xmlDoc, _
                  "Dir", dir.Name)
   Dim dirs As IO.DirectoryInfo()
```

```
    Dim files As IO.FileInfo()
    node.Attributes.Append(xmlHelpers.NewAttribute(xmlDoc, _
                   "FullName", dir.FullName))
    dirs = dir.GetDirectories
    files = dir.GetFiles()

    For Each d As IO.DirectoryInfo In dirs
       node.AppendChild(FileListXML(d, xmlDoc))
    Next

    For Each f As IO.FileInfo In files
       node.AppendChild(FileInfoXML(xmlDoc, f))
    Next
    Return node
End
```

NOTE I can imagine having other reasons to use the file list as input metadata, so I added a handful of attributes that might be useful in other scenarios.

The `FileInfoXML` method returns a node containing information such as name, length, and so on. The values important for building the project file are the name and the extension.

Providing Overrides

Once you have the basic file list, you may want a way to exclude files from the project. Ideally, you'd like to keep these directories free of any extraneous files. But it's simple enough to provide a mechanism for excluding files when the need arises. To do this, support an `Exclude` attribute and use `MergeMetadata` to incorporate a manually created metadata file including this attribute for files you want to skip. As discussed in Chapter 2, XML files merged with `MergeMetadata` need to match the structure of the file you're merging them with. For example, to exclude the genShippers.vb file, use a manual XML metadata file that looks like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<FileList>
   <Dir Name="MiddleTier" >
      <Dir Name="Autogen" >
         <Dir Name='GenDataContainer'>
```

```
            <File Name='genShippers.vb' Exclude='true'/>
         </Dir>
      </Dir>
   </Dir>
</FileList>
```

After the merge, the genShippers node of the file list includes the Exclude attribute, and the template checks for this attribute.

Running the Template

The template used to create the project file is a brute-force template. This is the easiest way to merge the new file list into the existing project file. The process directive calls the UpdateProjectFile method in the ProjectSupport class of the GenerationSupport assembly. This method takes three parameters. The first parameter is the project file that you'll update. If it's the same as the output file in the SinglePass child element of the directive, the existing file will be overwritten. The second parameter is the FileList.XML file containing the directory structure.

The third parameter is a list of the directories containing the files to include. The method removes all of the current entries for files in these directories from the project file and then adds back entries for the files currently in these directories. This parameter uses the array support feature of the generation harness for parameters passed to processes. Curly brackets must surround the comma-separated list of directories. If you organize your project as suggested in the section "Organizing Files via the Directory Structure," this parameter will be {Autogen,Editable}. The UpdateProjectFile method starts by splitting the parameter into parts and opening the project file and FileList.xml:

```
Public Shared Function UpdateProjectFile( _
            ByVal projectFileName As String, _
            ByVal fileListXMLFileName As String, _
            ByVal directoryList As String) _
            As IO.Stream
   Dim prjXML As New Xml.XmlDataDocument
   Dim nodes As Xml.XmlNodeList
   Dim stream As New IO.MemoryStream
   Dim reloadDirs() As String
   Dim listXML As New Xml.XmlDocument
   Dim nodeInclude As Xml.XmlNode
   directoryList = directoryList.Substring(1, directoryList.Length - 2)
   reloadDirs = directoryList.Split(","c)
   listXML.Load(fileListXMLFileName)
   prjXML.Load(projectFileName)
```

This method has to keep two XML documents straight. The `listXML` variable contains the list of files, and `prjXML` contains the `vbproj` file. The process makes all of its changes to the `Include` node, so a variable holds a reference to that node:

```
nodeInclude = prjXML.SelectSingleNode("//Files/Include")
nodes = prjXML.SelectNodes("//Files/Include/*")
```

Looping through the current file nodes lets you delete existing files if they're in one of the subdirectories you're updating. This code leaves any other nodes unchanged:

```
 ' Delete files in reload dirs
 For Each node As Xml.XmlNode In nodes
     For Each dirName As String In ReloadDirs
        If Utility.Tools.GetAttributeOrEmpty(node, "RelPath"). _
                     StartsWith(dirName) Then
           nodeInclude.RemoveChild(node)
        End If
     Next
Next
```

To add the files that currently exist in the managed subdirectories, the second loop processes the files in the file list—using the XML file list you just created as metadata for this process. If the file has the `Exclude` attribute set to `true`, it isn't included. `LoadVBFiles` adds the appropriate files to the `Include` node of the project:

```
   ' Add back the files in these directories recursively
   For Each dirName As String In reloadDirs
      nodes = listXML.SelectNodes( _
                     "//FileList//Dir[starts-with(@Name,'" & _
                     dirname & "')]/*")
      For Each node As Xml.XmlNode In nodes
         If Not Utility.Tools.GetAttributeOrEmpty(node, "Exclude") = _
                     "true" Then
            LoadVBFiles(node, nodeInclude, _
                     Utility.Tools.GetAttributeOrEmpty( _
                        Node.ParentNode, "Name") & "\")
         End If
      Next
   Next

   prjXML.Save(stream)
   Return stream
End Function
```

LoadVBFiles is a recursive method that works down the directory structure, adding all of the files it finds directly to the Include node. The relative path reflects the location in the directory. As you work down the directory structure, the fileNode parameter reflects the current file or directory, the includeNode is always the Include node of the vbproj, and the code concatenates relDir to reflect the current fileNode directory. The first loop recursively calls LoadVBFiles for each subdirectory, and the second loop appends a node for each file to the Include node of the project. MakeNewVBIncludeNode actually creates the node to append:

```
Private Shared Sub LoadVBFiles( _
            ByVal fileNode As Xml.XmlNode, _
            ByVal includeNode As Xml.XmlNode, _
            ByVal relDir As String)
   Dim dirNodes As Xml.XmlNodeList
   Dim fileNodes As Xml.XmlNodeList
   relDir &= Utility.Tools.GetAttributeOrEmpty(fileNode, "Name") & "\"
   dirNodes = fileNode.SelectNodes("Dir")
   fileNodes = fileNode.SelectNodes("File[@Ext='.vb']")
   For Each dNode As Xml.XmlNode In dirNodes
      LoadVBFiles(dNode, includeNode, relDir)
   Next
   For Each fNode As Xml.XmlNode In fileNodes
      includeNode.AppendChild( _
            MakeNewVBIncludeNode( _
               includeNode.OwnerDocument, _
               relDir & Utility.Tools.GetAttributeOrEmpty(fNode, "Name"), _
               Utility.Tools.GetAttributeOrEmpty(fNode, "Ext")))
   Next
End Sub
```

MakeNewVBIncludeNode creates a node with attributes for the subtype, which will be Code for all of these files, and the buildAction, which will be Compile:

```
Private Shared Function MakeNewVBIncludeNode( _
            ByVal xmldoc As Xml.XmlDocument, _
            ByVal fileName As String, _
            ByVal ext As String) _
            As Xml.XmlNode
   Dim node As Xml.XmlNode = xmldoc.CreateElement("File")
   Dim subType As String
   Dim buildAction As String
   node.Attributes.Append(Utility.xmlHelpers.NewAttribute(xmldoc, _
               "RelPath", fileName))
   Select Case ext.ToUpper
```

```
        Case ".VB"
            subType = "Code"
            buildAction = "Compile"
    End Select
    node.Attributes.Append(Utility.xmlHelpers.NewAttribute(xmldoc, _
                "SubType", subType))
    node.Attributes.Append(Utility.xmlHelpers.NewAttribute(xmldoc, _
                "BuildAction", buildAction))
    Return node
End Function
```

NOTE If you're adding files to your project that aren't source code that will be compiled as part of your project, you'll need to alter this code to use the correct `SubType` and `BuildAction`.

Maintaining Project Files by Hand

If you think that's a lot of bother, it's worth a look at the alternative of maintaining the project file by hand. You can just use the Show All Files icon at the top of Solution Explorer, include all files that don't yet appear in the project, and manually delete the files that show the triangular error icon because they're missing. If you aren't adding and removing files very often, this approach can work, but it'll take a minute or two each time you check for changes to the directory. You'll want to work carefully because source control relies on the project file. If you're missing files, you won't have them in source control.

TIP Visual Studio allows you to include and exclude files by directory. A quick way to update your project to include new generated files when you're following the previous directory suggestions is to simply exclude and then include the Autogen and Editable subdirectories.

CAUTION Visual Studio has a strange quirk with deleted files. You can have a "ghost" file in Solution Explorer that doesn't actually exist on the disk. When this occurs, you can delete these files within Visual Studio, but it may seem strange to have to confirm the deletion of a nonexistent file.

NOTE When Visual Studio says, "This file will be permanently deleted," it actually means it'll move the file to your Recycle Bin.

Incorporating Source Control

Source control performs numerous tasks that keep your development sane. Source control provides at least the following:

- A repository for the team to access the same copy of your application source code
- A mechanism for marking who is using each file so programmers don't overwrite each other's changes
- A centralized location for backups
- A history of the source code evolution

Almost all modern source code control offers additional useful features. These include finding differences between files, merging changes, and branching histories.

NOTE There are many tools available for source control—I'm covering SourceSafe because it's what I use. If you're using something else, you'll be able to use similar techniques.

NOTE MSDN has an article on SourceSafe best practices, which you can access via the search *SourceSafe Best Practices* on MSDN.

TIP If a search fails on MSDN, try Google. You can access some articles on MSDN via a Google search that the MSDN search engine fails to find. Go figure.

Code generation changes the source control picture in several ways. You'll be creating lots of files automatically. You have extra parts of your project—at least code generation templates and a set of harness directives. Also, you're far more likely to have multiple people generating the same files than you'd have multiple people manually editing the same files. Because different people will probably be generating code, you'd like to keep the files checked in except for the short time you're actually generating code—the shortest time possible.

It's important to keep the extra files that are part of code generation—but not part of your project—in source control. Templates will evolve, and you'll need to track those changes. Multiple people may need to edit templates, and you'll want to ensure they don't bash over each other's changes.

NOTE If you don't automate source control, and you use the directory structure described previously, manually checking out files isn't too difficult, but it's time consuming and prone to mistakes. If you use a directory structure that doesn't isolate the generated files, you can assign a single person (or dummy person) and leave files checked out to that person. While this works, you won't know who ran the code generation or have a record that changes were made via code generation.

TIP The checked in files will be read-only, which will remind your team not to edit generated files.

Automating Source Control

Automating your source control can help you manage project issues. Unfortunately, different teams use different source control tools, and you may not be using the same one I use. I use Visual SourceSafe, and that's what I'll show you how to automate. SourceSafe has two mechanisms for automation—the command line and a COM-based automation model. The command line doesn't work well for the type of automation you need for code generation, so I automated it via the COM interface.

If you're using another source control tool, this section will give you ideas about how to automate it, but you'll have to write the automation tools yourself. Also, my automation technique is simple, and you may find ways you'd like to extend the functionality. I discuss most of the code of the tool to help you understand and evolve it.

Figure 5-3 shows the object hierarchy for the source control classes. The abstract base class defines the expected interface with a number of `MustOverride` methods and sets the class to be `IDisposable`. Each source control backend is in a separate assembly to minimize problems with references to missing files. I've implemented only the SourceSafe version, but you can add parallel versions and change how the SourceSafe class is instantiated.

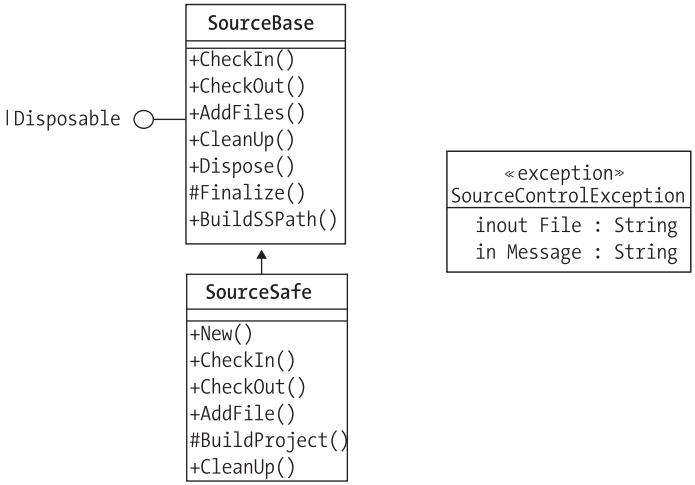


Figure 5-3. A UML diagram showing the relationship between different classes in the source control automation

Initiating Source Control Tasks

Two attributes on the `OutputRules` child element specify whether source code will be automated as part of the code generation harness. These two attributes are a Boolean attribute named `CheckOut` and a string attribute named `CheckOutProject`. `CheckOut` indicates whether the harness should check the generated files in and out, and `CheckOutProject` indicates the position of the files within the SourceSafe project structure. An example of the OutputRules child element with these attributes is as follows:

```
<kg:OutputRules HashOutput="false" OutputFileType="XML"
          OutputGenType="Overwrite"
          CheckOut="true" CheckOutProject="[SSMetaData]" />
```

NOTE `CheckOutProject` supports abstract paths.

Three source code automation tasks are performed during code generation. These are checking files out, checking them in, and adding them to source control if they don't yet exist. Looking at the source code for one directive (`XSLTProcess`) shows how a method checks files in and out:

```
If ShouldRun(outputFile, node) Then
   If SSCheckOutFile(node, basePath, outputFile) Then
      Try
         stream = CodeGenerationSupport.XSLTSupport.GenerateViaXSLT( _
                          XSLTFileName, inputXML, extObjects)
         madeFile = MarkWriteAndClose(stream, node, outputFile)
      Finally
         SSCheckInFile(node, basePath, outputFile, madeFile)
      End Try
   End If
End If
```

If the `ShouldRun` method indicates that the step should be performed, the file's checked out. Both `ShouldRun` and `SSCheckOutFile` log errors and skip the generation if they encounter problems. The results pane of the generation tool displays the errors logged by these methods. `SSCheckOutFile` returns `True` unless the check out fails; specifically, it returns `True` if it successfully checks the file out or the file doesn't exist.

NOTE The harness determines whether to perform code generation prior to checking the file out. You don't want to check files that you aren't generating out and back in because someone may be editing these files.

If code generation successfully checks out the file, you want to ensure it checks the file back in. The `Finally` clause of the `Try` block guarantees this. A stream holds the results of whatever code generation method is used (`GenerateViaXSLT` in this case), and `MarkWriteAndClose` writes out the file. It doesn't write out empty files, so the Boolean `madeFile` variable indicates whether a file was written. This is necessary because you can't check in or create a nonexistent file.

The `SSCheckOutFile` method (shown next) of the generation class determines the path to the file position in the SourceSafe database. The working file path on your disk doesn't match the project directory of SourceSafe exactly. The `CheckOutProject` attribute of the `OutputRules` child element (of your harness XML file) specifies the subproject location in SourceSafe. This path generally contains abstract paths. With these adjustments made, `SSCheckOutFile` calls the

CheckOut method of the source control object. The SSCheckOutFile method of the Generation class is as follows:

```
Protected Overridable Function SSCheckOutFile( _
                      ByVal node As Xml.XmlNode, _
                      ByVal workingPath As String, _
                      ByVal file As String) As Boolean

      Dim status As Utility.SourceBase.ItemStatus
      Dim checkoutProject As String = GetSSCheckOutProject(node)
      Dim ret As Boolean
      If (Not checkoutProject Is Nothing) AndAlso _
                 checkoutProject.Trim.Length > 0 Then
         checkoutProject = Utility.Tools.FixPath(checkoutProject, _
                 mLocalSettings.GetBasePath(), mDocPath, mNodeFilePath)
         If mSourceControl Is Nothing Then
            ' TODO: Figure out whether this is an error
            Return True
         Else
            status = mSourceControl.CheckOut(file, workingPath, checkoutProject)
            ret = (status = Utility.SourceBase.ItemStatus.CheckedOutToMe Or _
                         status = Utility.SourceBase.ItemStatus.DoesntExist)
            If Not ret Then
               LogError("Couldn't check file out of SourceSafe", file)
            End If
            Return ret
         End If
      End If
   End Function
```

NOTE You might notice the TODO item in this source code. I try not to leave many of those in source code, but this is a special case. You'll probably consider failing to instantiate your source control object to be an exception and want to throw one. On the other hand, I don't want readers of this book to be unable to use code generation because they don't use SourceSafe. Adjust that to provide the behavior you want.

The CheckOut method of the source control object returns the source control file status as an Utility.SourceBase.ItemStatus enum. Valid values for the

`ItemStatus` enum are `CheckedOut`, `CheckedOutToMe`, `NotCheckedOut`, or `DoesntExist`. If the file is checked out to the current user or doesn't exist, code generation can be successful and `SSCheckOutFile` returns `True`. If the check out failed, code generation will fail, so the method logs an error and returns `False`.

`SSCheckInFile` is similar. It also determines the SourceSafe path to the file representation and calls a method in the source control object:

```
Protected Overridable Sub SSCheckInFile( _
                    ByVal node As Xml.XmlNode, _
                    ByVal workingPath As String, _
                    ByVal file As String, _
                    ByVal addFile As Boolean)
   Dim status As Utility.SourceBase.ItemStatus
   Dim checkoutProject As String = GetSSCheckOutProject(node)
   If (Not checkoutProject Is Nothing) AndAlso _
                checkoutProject.Trim.Length > 0 Then
      checkoutProject = Utility.Tools.FixPath(checkoutProject, _
                mLocalSettings.GetBasePath(), mDocPath, mNodeFilePath)
      If mSourceControl Is Nothing Then
         ' TODO: Figure out whether this is an error
      Else
         status = mSourceControl.CheckIn(file, workingPath, checkoutProject)
         If status = Utility.SourceBase.ItemStatus.DoesntExist Then
            If addFile Then
               SSAddFile(node, workingPath, checkoutProject, file)
            End If
         End If
      End If
   End If
End Sub
```

The `CheckIn` method of the source control object returns the status after the check in. This should be either `NotCheckedOut` or `DoesntExist`. If the file doesn't exist, it's time to add it. Because `SSAddFile` expects that you'll call it from `SSCheckIn`, it accepts a precreated source control path and simply calls the source control object:

```
Protected Overridable Sub SSAddFile( _
                    ByVal node As Xml.XmlNode, _
                    ByVal workingPath As String, _
                    ByVal checkoutProject As String, _
                    ByVal file As String)
```

```
    If checkoutProject.Trim.Length > 0 Then
        If mSourceControl Is Nothing Then
            ' TODO: Figure out whether this is an error
        Else
            mSourceControl.AddFile(file, workingPath, checkoutProject)
        End If
    End If

End Sub
```

Each of the directives that generate code call these three methods to manage source control.

Accessing the SourceSafe Wrapper Object

As shown in Figure 5-3, the SourceSafe wrapper object supports half a dozen methods. The ones of most interest are `CheckIn`, `CheckOut`, and `AddFile`. To accomplish these key tasks, you need to also build and tear down a SourceSafe COM object.

NOTE Documentation for SourceSafe automation is very limited. The reference I used is "Microsoft Visual SourceSafe OLE Automation" on MSDN (currently at `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnvss/html/msdn_vssole.asp`). It's a rather old article, but it does contain the key information.

NOTE You access SourceSafe through the SSAPI.DLL. If you need to update this reference within your harness project, look for this file in [Program Files]/[Visual Studio]/Common directory.

Starting Up SourceSafe

You can instantiate the SourceSafe object, but an empty SourceSafe object isn't useful, so you'll also need to open a database. I do this in the constructor. This

constructor takes a `srcSafeINI` file that could point to a different database, but this isn't yet fully supported. You can pass an empty string to open the default. Consistent with other databases in this book, I'm relying on default security in SourceSafe:

```
Public Class SourceSafe
   Inherits Utility.SourceBase

   Private mSSDatabase As New SourceSafeTypeLib.VSSDatabaseClass
   Private mRoot As SourceSafeTypeLib.VSSItem

   Public Sub New(ByVal srcSafeINI As String)
      mSSDatabase.Open(srcSafeINI)
      mRoot = mSSDatabase.VSSItem("$/")
   End Sub
```

TIP If you'd like to track all changes because of code generation separately from manual changes, create a dummy code generation user and modify the constructor to use this account.

This prepares the class for checking files in and out and adding any new files to SourceSafe.

Using the Class

To check out a file, you need the full path to it within SourceSafe. The file passed contains the path to the file on disk. `BuildSSPath` modifies the file variable to point to the SourceSafe location:

```
Public Overrides Function CheckOut( _
                  ByVal file As String, _
                  ByVal baseLocalPath As String, _
                  ByVal baseProject As String) _
                  As Utility.SourceBase.ItemStatus
   file = BuildSSPath(file, baseLocalPath, baseProject)
   Dim item As SourceSafeTypeLib.VSSItem
   Dim status As Utility.SourceBase.ItemStatus
   item = Me.GetItem(file)
```

```
    status = Me.GetItemStatus(item)
    If status = Utility.SourceBase.ItemStatus.NotCheckedOut Then
        item.Checkout()
        status = Me.GetItemStatus(item)
    End If
    Return status
End Function
```

The fashion under COM was to spring an error when something didn't exist in a collection, so this is how SourceSafe communicates the lack of a file. In .NET, you use a `Try/Catch/Finally` block with no implementation in the `Catch` clause to swallow this exception. The `GetItem` method hides this ugliness. The `GetItemStatus` method translates from the SourceSafe enum to an enum specific to code generation that's defined in `SourceBase`. This has two benefits—you can add new entries such as the `DoesntExist` enum, and all source control objects can use the same status values.[2] It's important to use a consistent enum since the method passes it back to the `Generation` class. If the file is currently available for check out, the method attempts to check it out and returns the status after the attempt.

The `CheckIn` method handles similar issues. It builds the SourceSafe path, retrieves the item, and determines its status. The file may not exist, in which case the calling code will make a later call to create it in SourceSafe. If it's checked out, you need to check it back in. If it's neither, something funny is going on, and throwing an error will let the calling code sort it out:

```
Public Overrides Function CheckIn( _
                    ByVal file As String, _
                    ByVal baseLocalPath As String, _
                    ByVal baseProject As String) _
                    As Utility.SourceBase.ItemStatus
    file = BuildSSPath(file, baseLocalPath, baseProject)
    Dim item As SourceSafeTypeLib.VSSItem
    Dim status As Utility.SourceBase.ItemStatus
    item = Me.GetItem(file)
    status = Me.GetItemStatus(item)
    Select Case status
        Case Utility.SourceBase.ItemStatus.DoesntExist
            ' Do nothing
```

2. It also hides the poorly named `IsCheckedOut` method. This method is poorly named by current conventions. Methods that start with `Is` or `Has` are now expected to return a Boolean. Instead, `IsCheckedOut` returns a status.

```
        Case Utility.SourceBase.ItemStatus.CheckedOutToMe
           item.Checkin("Checked in via code generation")
           If Me.GetItemStatus(item) <> _
                     Utility.SourceBase.ItemStatus.NotCheckedOut Then
              Throw New Utility.SourceControlException("CheckIn Failed", file)
           End If
        Case Else
           Throw New Utility.SourceControlException( _
                     "Attempted to check in file you didn't have checked out", _
                     file)
    End Select
    Return status
End Function
```

The `AddFile` method is similar, so I'm not including it here. It builds the SourceSafe project structure using the recursive `BuildProject`. `BuildProject` splits off the first part of the pathname—for example, `CodeGeneration` for `CodeGeneration/AutoGen/TemplateName`. `BuildProject` creates this SourceSafe subproject and then passes the remaining portion of the path (`AutoGen/TemplateName` in the example) recursively to itself.

Extending Source Control

Once you have source control working on your project, you can look at ways to leverage it. There are several additional issues you can address as follows:

- Managing source control for the generation harness in the context of the project
- Further protecting files from accidental editing via security settings and source control accounts
- Allowing the programmer to include a comment on what was changed in the generation
- Leaving files checked out that were initially checked out

In addition to having your project in source control, you'll want to keep your code generation under source control. The sidebar "Key Decision Point: When to Reuse Your Code Generation Harness" discusses other aspects of managing your code generation harness.

Key Decision Point: When to Reuse Your Code Generation Harness

You'll have to decide how to link the generation harness to each particular project. Either you can have a separate copy of the code generation harness for each project or you can reuse the same code generation harness. If you treat an individual copy of the generation harness as part of your project, you get a great deal of stability within the project. There's no chance of a change in the harness for another project mucking up your templates. On the downside, that leaves you with multiple copies of a key tool to manage. If you manage a single of copy of your generation harness, management is greatly simplified. However, changes in the generation harness (such as the addition of a new required attribute) could break your code generation for an existing project.

This problem isn't specific to code generation, you have the same problem on any project that uses component development. What's unique about code generation is that you might not make changes to the project for months or even years, and when you do make them, you may unexpectedly find the process of even creating and compiling your project compromised.

The best solution is to keep a single copy of the code generation harness but maintain long-term connection between the project and the copy of the specific harness code that generated the project. You can do this either by storing a copy of the code generation with the source control for each project or by maintaining a long-term history of the code generation harness in source control and labels when it's used for specific project generation cycles.

You can also use source control to supply an extra layer of protection to your generated files. If you provide a dummy user that performs source control, you can limit access to generated files to only this user. That would prohibit human users from even checking out the files. You'd do this by supplying a username and password to the constructor of the SourceSafe class.

I haven't provided a mechanism for adding user comments when files are created or checked in through the code generation harness. This would be a good addition to the source control automation. I just never came up with a user interface that didn't seem to be a burden for the programmer (okay, I admit I'm really bad about making source control comments).

My source control automation checks generated files back in, regardless of whether the programmer previously had them checked out. This is inconsistent with my usual "if you open a door, close it" philosophy, but I was unable to come up with a scenario where a generated file should remain checked out. If you find such a scenario, add a flag that tracks the previous status of the file.

Debugging

Debugging is a big topic in itself, and this section will just highlight aspects of debugging that are important to code generation. Debugging code generated applications can be more difficult than debugging other applications. This happens because code generation relies on two techniques that can be difficult to debug—XSLT and reflection. But to a greater extent, it's because problems in your code can arise from several different locations as follows:

- Handcrafted code
- Metadata
- Templates
- Harness directives
- Generation harness

Sorting out the source of a particular problem can be a challenge, but if you work systematically back from the symptom, you'll solve the problem. The sidebar on the scientific method describes one systematic approach. Code generation further complicates things because it makes debugging a two-step process. You first find the problem in your application code. If the problem is in generated code, you then need to debug the generation process to fix the template.

Getting a Valid Compile

Getting your generated code to compile without errors is nontrivial. Regardless of your generation technique (XSLT, brute force, or CodeDOM), you're writing code without a background compiler and without the ability to do frequent builds. Your template abstracts the actual code you're creating, but the degree of this abstraction varies with the generation technique. Writing source code as an abstraction without feedback is likely to be imperfect. You'll generate your code, attempt to compile, and resolve the compiler errors. This project is much easier if you copy tested code into your templates as the basis for your abstraction. But the good money says you'll still make a few mistakes.

This can be overwhelming at first because you're likely to exceed the limit on the number of errors the compiler will display, which is more than 100. That's because you'll see so many copies of the same exception.

TIP When you're faced with a lot of compiler errors, start at the top of a file containing errors. Invalid namespaces, inheritance, and implements problems can create a large number of errors for each file, and thus a huge number of compiler errors, as the result of a single, easily fixed typo.

TIP If you sort compiler errors by double-clicking the description, you'll be able to find errors that you have many copies of and solve those first. If you sort on the filename, you can clean up specific files first. In both case, you find the error and return to the template to solve the problem.

The majority of changes you'll make in response to compiler errors are simple typographical errors that the compiler error describes reasonably well. You'll have to trace the location back into the template, generally narrowing the location down using the region. My strategy for compiler errors is to try to fix fewer than about half a dozen simple things in the template between regenerations. I tend to lose track if I make too many changes.

NOTE The code generation harness loads the DLLs containing the brute-force and CodeDOM templates. The harness doesn't unload these DLLs after you run a single generation cycle. Thus, you can't recompile your assembly until you stop the code generation harness.

Debugging Your Application

Code generation doesn't lead to any fundamental differences in how you track down problems in your application. You'll debug problems in your application in the same manner you do now. However, you may want to be more deliberate in making experimental changes that you think might fix your problem.

TIP Use a zip file or source control or another approach to store a snapshot of your application prior to debugging. If it's new code with a bug, you may want to use a zip file rather than checking it in to source control to avoid pushing your problems onto your coworkers. But you need a snapshot to ensure that you can reliably clean up any changes you make while debugging, regardless of whether the code is generated. In generated code, keep a snapshot of the template as well.

When you find a problem in generated code, don't just change the code. If you change the generated code, you'll wind up with a generation error during your next generation or your changes overwritten (depending on the `GenType` of the directive generating the code). Instead, trace the problem back to the template. You'll generally face one of two scenarios. The template may be doing what you expect—but it happens to be wrong. Or, you're a bit mystified about why the template is producing the specific code. The second case means you have to debug the generation process, which the next section "Debugging the Generation Process" covers.

The rest of this section covers resolving problems in your application that you've traced to a template and need to fix there. You'll either change the template or handcraft code to solve the problem or add a new rule to your code generation. You need to make a conscious decision about which approach to take. Although any of the three may fix your bug, you need to maintain a broader vision and select a solution that makes sense when the template is run over similar files.

> If the problem is specific to the current generated file, such as special case handling of a particular method, you'll want to override the method and provide a handcrafted solution. Sometimes this will also involve changing the template to make the method overridable.
>
> If the problem would occur for all code generated from the template, you'll just fix the template. You'll take advantage of code generation and apply the change to all occurrences as soon as you regenerate.
>
> Too often, the issue lies in middle ground. It doesn't occur for all generated code from the template, but it occurs for multiple versions. This indicates that you need additional metadata. This new metadata needs to differentiate between cases that should incorporate the change and those that shouldn't. It'll probably be obvious whether you have the metadata and just need to incorporate it in your template, whether it's reasonable to capture it, or whether it's best to enter it manually.

The goal of code generation is to make your life easier. If something occurs in two or three classes, and you don't have the metadata, you can fall back on procedural techniques. You can override the template to call a simple method that simply calls a handcrafted utility function that gets the job done for you.

The Scientific Method and Debugging

Debugging code generation is much easier if you have a strategy or clear approach. If you debug badly, making poorly thought-out changes until the problem goes away, code generation will highlight your problems and make your debugging quite difficult. But debugging doesn't have to be a difficult or complex process.

Nancy Folsom introduced me to using the scientific method for debugging. I argued at first, insisting my techniques were unlike these. I became a convert as I came to understand that the core of this approach is disproving possibilities until the solution is apparent—a process of elimination. The following description is from an article Nancy and I coauthored about debugging in .NET.[3]

The scientific method is useful for everyday problem solving. Use the method to observe a problem, investigate when it happens and how it looks, hypothesize causes, experiment to test hypotheses, and analyze alternatives. Apply the method iteratively to complex problems to refine your understanding until a solution is apparent.

1. *Observe a problem to develop a problem statement. "Dates don't work" is vague, whereas "I can enter dates for last year, and that's wrong" is clear.*
2. *Investigate the problem. Observe when it happens and how it looks, and weed out irrelevant observations by listing the steps necessary to reproduce the problem.*
3. *Hypothesize one or more explanations. Pick the most likely, or the easiest, to test first.*
4. *Test your hypothesis with an experiment designed to disprove it. If you can't disprove your hypothesis, you're not done. You might have to rule out other possible explanations.*
5. *Analyze the results. Was your hypothesis wrong, or is the solution clear? Consider alternatives: Patch, rewrite, work around, or do nothing if the cure is worse than the disease. Use the project budget, schedule, and scope to determine the best course of action.*

The methodology is compartmentalized in theory; frequently, in practice, it's not. Even simple debugging includes all these steps, although they can be transparent. Notice the steps don't include implementing a solution. For a sufficiently complex solution, approach the fix the same as you would writing any maintenance code. If you use source code control, you can roll back the changes you made while experimenting, make the correction, then test only the fix. While this seems like overkill, it can avoid the difficulty of tracking down a bug that's introduced as a side effect of making other changes.

This process allows you to quickly get rid of the low hanging fruit. If my screen is blank, I'll look first to see if the computer is getting power, regardless of how unlikely I think it is that the box has lost power. I think in a simple way, but there's an important background process going on. I don't actually check whether the computer has power by pulling out a test meter or by sticking wet fingers in the socket. Instead, I look at the light on the front of the computer.

---

3. "Debug VS.NET Applications" by Nancy Folsom and Kathleen Dollard. *Visual Studio Magazine*, September 2002 (`http://www.fawcette.com`)

The background process is something such as, "Maybe the computer doesn't have power, and if the light on the front of the computer is on, that explanation is wrong." In scientific terminology, the first part—"Maybe the computer doesn't have power"—is the hypothesis. The check of the light is the experiment.

The light being on disproves the hypothesis. The light being off doesn't prove the hypothesis. A variety of problems within the computer, including a fried power supply or a broken wire, might also cause the problem. I'll formulate another hypothesis, such as "The power cable got disconnected" and crawl under my desk to perform the experiment—observing the cable. I'll formulate a series of ideas and devise the simplest test that can disprove each idea.

The example of the blank monitor mimics debugging. You'll generally want to start with the symptom and work back into core components of your application, in the same way I'd work toward my fuse box to sort out the blank screen problem—working from my computer to the source of the problem. The scientific method's formalization of this process contributes that you're disproving your ideas. State the problem as a positive or negative so you can *disprove* it. There are four core benefits of this strategy: You minimize time running down blind alleys, you work with simple test that don't distract you from keeping a broad perspective, you avoid a cognitive rut focusing on a single possible solution, and *any* coherent process keeps you from making wild guesses. You may do things that are just as intuitive and illogical with this approach, but you'll learn from each step and move coherently toward the right solution.

TIP Remember each test is associated with ruling out a single possibility and often has no bearing on the universe of other possibilities.

The scientific method works in addition to your good judgment and intuition. If there was a strong burning smell or the overhead light went out at the same time the screen went blank, I'd take a different approach to sorting out the blank screen problem. If you see what's wrong with your application, by all means go fix it. That's fixing your application, not debugging it. This technique is for problems that don't have obvious solutions—the ones that leave you scratching your head.

TIP You may want to add some extra `Try/Catch` for debugging. This allows you to break within the method where an exception occurs and access the method's local variables.

Effectively Using Visual Studio's Debugging Tools

Few people use the Visual Studio debugging tools effectively. There's an incredible array of easy to access functionality. You won't use it all, but you have to experiment to find which features match your style. You've probably already found the call stack, and you can track down SQL Server and remote debugging information when you need it. A few commonly overlooked features are using conditional breakpoints, turning break points off without removing them, running code in the Immediate window (at runtime only) and the Watch windows. The Watch window contains the same information as the Autos and Locals windows, but you can control the content and order. The four Watch windows let you maintain multiple sets of watches just by dragging stuff onto them. Each Watch window can reflect a different place in your code or a different problem. If you're in the habit of typing or copying/pasting things into the Immediate window to check values at runtime, try dragging it into the Watch window instead. You can edit it just as easily, and it's updated in real time. For background on Visual Studio debugging, you can read John Robbins' book *Debugging Applications for Microsoft .NET and Microsoft Windows* (Microsoft Press, 2003).

Debugging the Generation Process

Sometimes you'll be mystified by how a particular segment of code was generated or, more often, why something is missing. The problem could arise from metadata, templates, harness directives, or the generation harness. Tracking down the culprit can seem overwhelming. Here the scientific method, or a process of elimination, helps out. To get you started, Table 5-4 lists common problems and some ideas of where to start tracking the problem.

NOTE This is a book that empowers you to do your own code generation. As part of this, I've provided tools to illustrate the process. These tools may work well for you, but they aren't as robust as commercial tools and may fail in some scenarios. You have the source code to all of the code generation tools, so you can track down the problem and fix it—then please let me know where I can improve the tools. Including the code generation harness in this table acknowledges that you may find problems in the code associated with this book. If you do, I hope you'll let me know about it at `Kathleen@gendotnet.com`. Conversely, I hope you'll keep updated with the latest versions of the tools at the Apress Web site.

Table 5-4. Common Problems and Possible Solutions

ISSUE	THINGS TO CHECK	THINGS THAT ARE UNLIKELY
Harness failing (will not open the harness XML file)	• Check that harness XML file validates against the specified schema. • Code generation harness. • Invalid xsi:schemaLocation in the harness XML file.	• Templates • Metadata
Harness raises exceptions when run	• An incorrect process or alternate process. • Missing attributes. Generation harness	
Files missing, or not regenerated, and files in the wrong place	• The output dialog box's Count Created, Count Selected, and Count Output columns. • The results tab of the generation process. • Harness directives particularly the `MultiFile` attributes and abstract paths. • Metadata, particularly missing nodes. • Templates, particularly the condition on the `select` condition at the entry level. Generation harness.	• Problems deeper in the template
Portion of file missing	• Check parallel output files if they're also wrong, check templates and then metadata, and check the reverse if other output files are correct. • Metadata, particularly missing nodes and incorrect attributes such as `PrimaryKey`. • Templates, check that individual subroutines and XSLT templates are being run. • Templates, check conditionals and particularly XPath statements.	• Harness directive problem • Generation harness

Table 5-4. Common Problems and Possible Solutions, continued

ISSUE	THINGS TO CHECK	THINGS THAT ARE UNLIKELY
Monolithic metadata issue	• Check source metadata prior to any merges. • If a merge problem, check whether the merge data was placed in the wrong place or not included.	• Harness directive problem • Generation harness Templates
Merge data in wrong place	• Check the structure of the metadata sources, particularly any manual metadata.	• Harness directive problem • Generation harness Templates
Merge data in source, but file missing in output	• Check the harness directive. • Check that the metadata sources are present.	• Generation harness Templates
Merge data missing from source	• Check underlying source through the information schema views. • Metadata extraction tool, using the information in Appendix B, set break points, and walk through the tool.	• Harness directive problem • Generation harness Templates

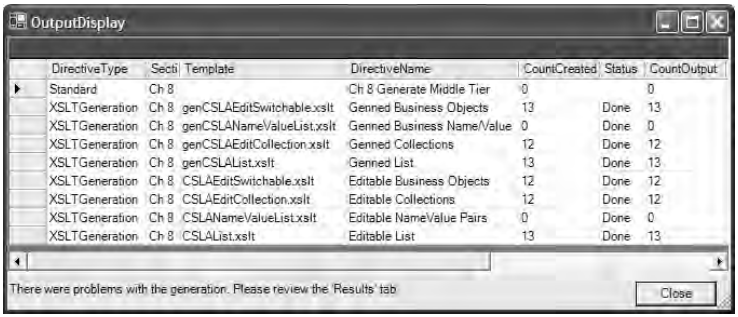
Debugging Generated Output

Figure 5-4 shows the display you'll see if generation is successful using the harness; it includes the number of files generated and output. This information will often help you track down the source of a problem as follows:

- If no files are selected (CountSelected), the problem is probably in the `SelectXPath` attribute of the `MultiPass` directive.

- If the files are created (CountCreated), but no files are output (CountOutput), the problem is probably in the XPath expression the entry-level template uses to select nodes to process.

- If the files were selected, but not created, the files probably had an `Always` or `UntilEdited` generation type and had been edited, or they couldn't be checked out of source control. See the Results tab for more information.



DirectiveType	Secti	Template	DirectiveName	CountCreated	Status	CountOutput
Standard	Ch 8		Ch 8 Generate Middle Tier	0		0
XSLTGeneration	Ch 8	genCSLAEditSwitchable.xslt	Genned Business Objects	13	Done	13
XSLTGeneration	Ch 8	genCSLANameValueList.xslt	Genned Business Name/Value	0	Done	0
XSLTGeneration	Ch 8	genCSLAEditCollection.xslt	Genned Collections	12	Done	12
XSLTGeneration	Ch 8	genCSLAList.xslt	Genned List	13	Done	13
XSLTGeneration	Ch 8	CSLAEditSwitchable.xslt	Editable Business Objects	12	Done	12
XSLTGeneration	Ch 8	CSLAEditCollection.xslt	Editable Collections	12	Done	12
XSLTGeneration	Ch 8	CSLANameValueList.xslt	Editable NameValue Pairs	0	Done	0
XSLTGeneration	Ch 8	CSLAList.xslt	Editable List	13	Done	13

Figure 5-4. The dialog box that appears when code generation is successful

Figure 5-5 shows the display you'll see if generation fails because of an exception. The comment in the upper left is intended to point to probable issues. It may sometimes be correct although I haven't yet included all of the information in Table 5-4. The tree below it shows the exception as a nested tree. This is particularly helpful when you're debugging XSLT or reflection—both of which store valuable information in various nested exceptions. The lower-left tree shows the directive that failed, allowing you to quickly check what was running when the exception occurred. ToolTips show the full text of truncated entries.

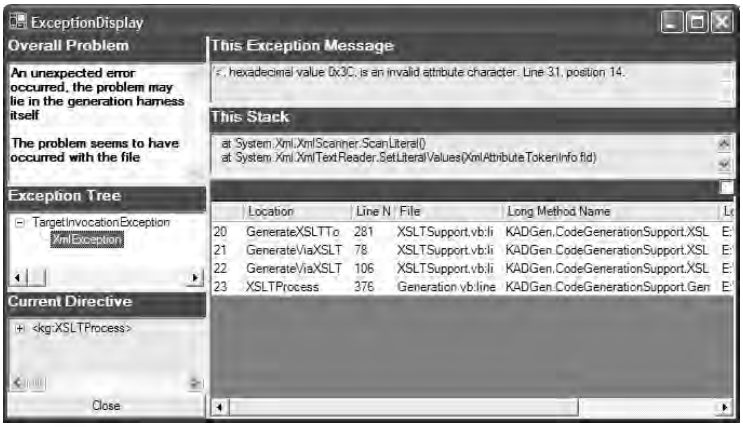


Figure 5-5. The dialog box that appears when code generation fails

The right panels contain the information corresponding to the exception currently selected in the middle tree on the left. The top-right panel shows the error message. The middle-right panel shows the callstack the way Visual Studio normally displays it. The bottom-right panel displays the call stack as a grid. Note that the line number and file are easy to spot. There's also a checkbox that allows you to display the entire callstack or to exclude framework entries.

Debugging Metadata

As shown in Table 5-4, metadata problems can arise from incorrect manual metadata, a merge problem, or an issue with a metadata extraction tool. It's important you save your interim monolithic metadata file for debugging. Use the monolithic metadata for debugging, even though you're building it on the fly and not storing it in source control, as discussed in Chapter 2.

You'll generally be using visual inspection to look for problems in the metadata. There are several ways to view these large files. You can use Internet Explorer, which is probably your default if you double-click the file. This gives you expand/collapse of nodes, but you can't edit. You can use Visual Studio, which lets you edit but doesn't expand/collapse. Or you can use a third-party tool that does both.

Debugging Harness Directives

Harness directives are frequently a source of problems, but they're easily fixed. If the harness application is failing when you open the harness file, your XML probably doesn't validate against the schema. You can evolve the schema, so you'll need to be sure that you're validating against the intended schema. Because the user interface of the code generation harness builds on the schema, problems with the user interface may also be based in schema problems, including the `xsi:schemaLocation` attribute of the harness file pointing to an invalid location.

CAUTION You can test whether XML is valid through the XML menu that appears in Visual Studio when you're editing an XML document. But this test *doesn't* respect the `xsi:schemaLocation` attribute that's the standard way to specify a schema. The validation follows the IntelliSense rules for finding the schema file—looking first in the project directory and then in the "magic" directory for schemas, [Program Files]/[Visual Studio]/Common7/Packages/schemas/xml. It certainly won't rule out a validation problem to validate against the wrong schema in Visual Studio.

If you successfully load the harness XML file but it fails on Run, the problem may still lie in directives. Harness directives that select values (such as those that use `MultiPass` elements) must use valid XPath expressions, and reflection-based processes (such as those that use the `Process` element) must call available .NET methods. Also, referenced input files must exist. Failure of the harness may result in either a runtime error or the failure of some templates to generate any code. In both cases, you probably have an invalid attribute on a directive.

Debugging Generation Harness

The generation harness included with this book is effective, but it isn't overly robust. If you give it improper directives, it may fail. This is the trade-off for the flexibility and extensibility of a tool built entirely on an XSD schema.

I've used this harness a good bit and evolved it for the context of this book. But you could still find bugs in it. If you find bugs in the generation harness, you've got the source code so you can fix the problem—but I hope you'll also let me know what problems you find so I can improve the version available (`Kathleen@gendotnet.com`).

Debugging Templates

Templates will be the most frequent source of generation errors. Whether you're using XSLT, brute-force, or CodeDOM generation, templates are code and code can have problems. The general approach to debugging each type of template is a little bit different. The key similarity is that you'll need to develop techniques to quickly trace code issues to the appropriate template location. Organizing your code into regions with a single template or method corresponding to each region makes the tracing process much easier, regardless of the template method.

Debugging Your XSLT Stylesheet

The XSLT stylesheet isn't tested for correctness until you open it to extract parameters as you execute code generation. In Visual Studio, you can't even validate it because the XML/Validate menu option is only available when you're editing a file with an XML extension. It does give you Task List entries; but note that these are mismatches with your schema, and available schemas are imperfect resulting in Task List entries for valid XSLT files. So, be patient and expect a couple of issues when you first generate.

TIP Be sure you've provided a version of XSLT.XSD to enable IntelliSense for XSLT. Appendix A discusses how Visual Studio uses XSDs to provide IntelliSense and where to find a copy of XSLT.XSD.

Reflection is used by the generation harness regardless of the template mechanism you're using. This makes it a bit harder for the generation harness to provide good exception information. To work around this, the harness includes `Try/Catch` blocks at key locations. If you place a break point in all locations where you're rethrowing the exception, you'll catch the exception close to the source—yep that's a bit ugly but effective.

XSLT exceptions will often contain interesting information in both the inner exception and the wrapping exception. Move among the nested exceptions in the middle-left panel of the exception dialog box (as shown in Figure 5-5) to see what information is displayed. As you get used to them, you'll find the exceptions .NET raises during XSLT transformations are rather good, but at first the information will seem strange. Table 5-5 contains some hints.

Table 5-5. Common XSLT Exceptions and Things to Look For

EXCEPTION MESSAGE	OFFENDING XSLT CODE	LOOK FOR
This is an unexpected token. The expected token is 'QUOTE'. Line 45, position 53.	`<xsl:value-of select=Name/>`	Missing quotes.
The '@' character, hexadecimal value 0x40, cannot begin with a name. Line 45, position 53.	`<xsl:value-of select=@Name/>`	Missing quotes.
The '$' character, hexadecimal value 0x24, cannot begin with a name. Line 45, position 53.	`<xsl:value-of select=$Name/>`	Missing quotes.
This is an unexpected token. The expected token is 'NAME'. Line 102, position 72.	`<xsl:value-of select="$Name"`	Missing close element. Note that this exception will point to the next non-whitespace character, which may not be on the same line. Also, note that this error will produce squigglies *unless* the missing close quote ends the line.
'//Menu[@Name=$Name' is an invalid XPath expression.	`<xsl:apply-templates select= "//Menu[@Name= $Name"/>`	Missing close square bracket. The exception message appears in the inner exception, its position appears in the outer exception.

Table 5-5. Common XSLT Exceptions and Things to Look For, continued

EXCEPTION MESSAGE	OFFENDING XSLT CODE	LOOK FOR
Message: "'xsl:value-of' cannot be a child of 'xsl:template' element."	`<xsl:valu-of select="$Name"/>`	Misspelling. This exception can also occur with improper nesting.
Message: "The 'xsl:value-of' start tag on line '102' doesn't match the end tag of 'xsl:template' in file 'file:///...SearchUC.xslt'. Line 114, position 3."	`<xsl:value-of select="$Name">`	Failed to close element. This exception can occur many different ways.
The contents of 'xsl:stylesheet' are invalid.	`</xsl:template> qwe<xsl:template>`	Text appearing between templates.
The '4' character, hexadecimal value 0x34, cannot begin with a name. Line 123, position 8."	`If 3 < 4 Then`	Less than must be escaped. In this case, `If 3 < 4 Then`.
'<', hexadecimal value 0x3C, is an invalid attribute character. Line 123, position 33.	`<xsl:value-of select= "position() <last()" />`	Less than must be escaped. In this case, `<xsl:value-of select="position()< last()" />`.
An error occurred while parsing EntityName. Line 128, position 89.	`var1 & var2`	Ampersands must be escaped.
'<', hexadecimal value 0x3C, is an invalid attribute character. Line 149, position 41."	`<xsl:if test= "$Child='true'> Function New<xsl:value-of select="$Name"/>`	The previous element's literal doesn't include a closing quote.

Table 5-5. Common XSLT Exceptions and Things to Look For, continued

EXCEPTION MESSAGE	OFFENDING XSLT CODE	LOOK FOR
The expression passed to this method should result in a NodeSet.	`<xsl:value-of select="ancestor::@Namespace"/>`	Incorrect syntax; use `<xsl:value-of select="ancestor::*/@Namespace"/>`. You'll get this message with a variety of syntax errors within any XSLT attribute that accepts an expression, including test and select.
'xsl:param' cannot be a child of 'xsl:template' element.	`<xsl:template name="CreateRunSP">Fred<xsl:param name="type"/>`	If you include an `xsl:param` directive, it must immediately follow the `xsl:template` directive. This message is confusing because of course an *xsl:param* directive is a child of an `xsl:template`; it has no other purpose.

In addition to responding to transformation exceptions, you'll probably encounter times when the template leaves out code you expect. If you're debugging through an XSLT debugger, use its features. If you're debugging through Visual Studio, you'll have to fall back on extra output statements. You can't send these output statements anywhere besides your standard output—meaning you'll have to put stuff into your generated code during debugging. If you add these as comments in your target language, they won't raise more compiler errors.

Start by focusing on the `select` attribute of `xsl:for-each` and `xsl:apply-templates` because they're a common cause of problems. Within the match template or `xsl:for-each` clause, add output text, perhaps `foo`. In examining `xsl:value-of` and `xsl:call-templates` clauses, add text around the clause to see if it isn't executing or if it's executing and returning an empty string. I generally surround the output with asterisks, so I can easily find and remove them later.

TIP When you add information to your output to aid in debugging, be sure you can recognize it later to remove it. Whenever possible, add text that results in compiler errors to ensure it's later removed. The simpler your strategy, the better.

Debugging Recursion

A particularly difficult aspect of XSLT templates to debug is recursion. You may never need recursion in your code generation. But because you can't reassign variables in XSLT, you may need it to solve even relatively simple problems such as breaking down a string into component parts. I'll walk through my experience debugging a particularly nasty recursion bug using Visual Studio .NET. This will give you some tips for solving similar problems.

In this particular problem, I had an XSLT template going into infinite recursion. In Visual Studio, the symptom was the transformation locking up but responding to a Ctrl+Break. I added the bold portions of the next template just for debugging. When I output text for debugging, I use text that'll be easy to find later, such as "garbage" and other words that won't be in my own code. Placing a key character such as the asterisk (*) before and after output text lets you know what spaces appear at the start and end of a string. If you're outputting several values, it's easier to identify what each one is if you prefix with an identifiable string—here I used garbage and trash.

When a template goes into infinite recursion and you stop it via Ctrl+Break, you don't get any output. So you have to explicitly halt the recursion and force the output to view the extra information. The easiest way to do this is by measuring the recursion depth and forcing recursion to end when it reaches a reasonable depth—four in this example. The `level` parameter in this template measures the recursion depth. Examining the output, I saw that the variable `remaining` was always the full name of the file, even though it clearly contained backslash characters:

```
<xsl:template name="StripPath2">
   <xsl:param name="fname"/>
   <xsl:param name="level" select="0"/>
   <xsl:if test="$level!=4">
      <xsl:variable name="remaining" select="substring-after($fname,'\')"/>
      <xsl:choose>
         <xsl:when test="string-length($remaining) > 0">
            garbage  <xsl:value-of select="$remaining"/>
            <xsl:call-template name="StripPath">
               <xsl:with-param name="fname" select="$remaining"/>
               <xsl:with-param name="level" select="$level+1"/>
            </xsl:call-template>
         </xsl:when>
         <xsl:otherwise>
            trash <xsl:value-of select="$fname"/>
            *<xsl:value-of select="$remaining"/>*
```

```
            </xsl:otherwise>
        </xsl:choose>
    </xsl:if>
</xsl:template>
```

Using the scientific method (although in a subtle way that I was barely conscious of), I thought, "Maybe I have to escape the backslash character; I'll add one and see if I still have the problem." If I got the same output, I would rule out the impact of escaping the backslash—disproving the hypothesis. Instead, I got the correct values for the `remaining` variable, which also provided the solution.

Debugging Brute-Force or CodeDOM Templates via Reflection

There are two issues with debugging reflection. First, the exceptions you receive are rather meaningless wrapper exceptions. The real information is in the `InnerException`, and I built the exception display to easily show these inner exceptions (as shown in Figure 5-5).

The second issue is that you're calling a method based on a string. The string can be mistyped, resulting in a runtime exception. You'll want to look for typos in method, class, or assembly names when you receive errors that indicate a method can't be found.

Once you correctly call the brute-force or CodeDOM template, you can debug it with normal .NET techniques. CodeDOM generation presents additional complications because you must determine whether the CodeDOM graph isn't being correctly built or the language provider isn't outputting the code you expect.

Managing the Project over the Long Haul

My philosophy for code generation is this: Get it going; get it right.

The first part of this means the tricks and tools that get your application started quickly. Software projects have momentum. You don't want to lose the energy of the project startup in details. The easiest way to quickly ramp up a project is to bring forward tools and rules from your last project, leveraging the lessons you've already learned. But there are so many lessons you can't expect to carry them forward in your head. You need tools to help you extract key elements from your past project.

Even as you're getting started on the new application, you can head toward doing things right. I don't believe in disposable prototypes because your time is

too valuable to create things you're planning to throw away. I do believe in prototypes, and I do believe in throwaway research projects. I don't mind building something that's risky and I *might* trash, but I don't want to spend time building something I *know* I'll trash. Using the techniques and tools in this book, you'll be able to evolve smoothly from prototype to working application. Part of *getting it going and getting it right* is leveraging past applications for startup and managing your projects properly from the start.

Although you may see the benefits of moving your project forward, it won't surprise me if you remain skeptical that you can accomplish it in a reasonable manner. *Reasonable manner* here means without so much extra work you decide not to bother. Moving the framework forward takes some work, but the following guidelines will make the process relatively painless. Note that the automation tool is just one part of the successful transfer process as follows:

- Use an existing project as the source.
- Have a postmortem of the previous project done by its team.
- Create an explicit strategy for what you'll reuse for the current project.
- Bring the framework forward with an automation tool.

In an organization, you can apply extra resources to smooth this process. You might formalize the process with additional tools, create additional documentation for project startup, or isolate a framework under the responsibility of a separate group. This section isn't about that sort of formalization. It's not a bad idea; I just trust you can formalize things on your own.

This section is about bringing forward what you already have in place from a previous project—with as little additional work as possible. You'll have to select which of your projects you want to base the new project on, and I'll ask you to document two steps that you're probably doing only in a casual manner. These are the postmortem of the project you'll reuse and a strategy for reuse within the new project. Then I'll show you a rudimentary tool to simplify the mechanics of moving your code forward. The team on the initial project does the postmortem, and the new team creates the reuse strategy.

Using an Existing Project

Unless you have the resources to maintain a separate framework, I suggest you start with a working project, probably your most recent project. This lets you skip a step of explicitly isolating your framework. Because that step is tedious,

any separate framework is likely to become outdated, even if you go to the trouble of creating it. If you have the resources to provide a separate team for infrastructure and let it stay in close cooperation with your development teams, fine. However, I'm interested in leveraging a previous project with almost no additional work.

The first step of the new project is to look around and determine the best source of a core framework. Right now, that's likely to be based on the samples in this book that are contained in the download associated with Chapters 6–10. You're likely to evolve these projects a good deal, but starting with something other than a blank page will help you get up to speed quickly.

Picking and choosing among frameworks for different tiers is valuable, but problematic. If you move entire solutions—including the framework for all tiers—you can automate some sticky issues such as updating the project GUIDs used for referencing other projects within the solution. If you move individual projects forward, you can't automate this but can do it manually.

Performing a Postmortem

You need to understand what you currently have before you bring it forward into a new project in a meaningful way. What are the strengths and weaknesses of the project you want to reuse? What would *that team* do differently if it could do start over?

The postmortem I suggest is a short document of one to eight pages depending on the project, number of issues encountered, verbosity of the team, and so on. Ideally, the whole team has an hour-long meeting to create the shell of the postmortem. The easiest way to do this is work on a whiteboard with three columns. Imagine you could start the project over knowing everything you know at the end of the project. The three columns are for what you'd do the same way again, what you think you'd do differently, and what you'd spend more time researching. These should be the shortest phrase the team can understand. Pay attention to framework issues, and minimize time spent on application issues. What were the lessons you learned and how might they apply to other projects?

Try for balance. Don't focus just on what was right, or focus just on what was wrong, because almost every project has lessons to be learned on both fronts. Before the meeting, assign a scribe to create the postmortem document. The scribe shouldn't be the person working with the whiteboard because they'll need to take additional notes. After the meeting, the scribe's job is to create a document based on the whiteboard items that will make sense to someone outside the project. They'll create a paragraph for each item on the whiteboard. Ideally, all the members of the team will review the document, and someone outside the project will review it for clarity to an outside party. Keep the document reasonably short. If there are complex issues that need more attention, add appendices. You'll use the postmortem later, at the start of other projects, but it's almost

impossible to create at that time. Get it done; get it out the door before your team is hopelessly distracted by new projects, and have a strategy for keeping these postmortem documents available to new project teams.

On the new project, you'll use the postmortem to set initial strategies and become familiar with the framework you're using and to remain aware of its weaknesses from the start of your project, so you can work to improve the framework as appropriate.

Developing a Strategy for Reuse

In bringing your solution forward, you'll probably consider whether to copy or share each of the following project categories:

- Code generation harness tool
- Generation harness XSD
- Components
- Utility library
- Project files
- Templates
- Harness XML file
- .NET solution file
- Directory structure

Sharing means you'll have a single copy of the item that you use in both the initial and new project. Copying means each project has its own copy of the item. You get to decide which of these items to copy into individual applications and which ones to share. This list is roughly arranged in the order of how likely you are to share the item (not reuse it) between projects. You wouldn't share the specific directory structure, solution files, or harness XML file. It's generally unwise to actually share templates because these are intimately associated with your application, and sharing templates means critical source code in your application could change unexpectedly. You'll reuse the template by copying them. You can update the templates based on a conscious decision, but you don't want them changing during maintenance on a schedule unrelated to the project using them.

You'll only share project files if you were sharing an underlying component contained in that project. You might copy utility libraries and components into each project or share them between projects. You're most likely to share your code generation harness and harness XSD between files. But you may elect to dedicate these to each project and not share them.

The benefit of putting something under source control as part of the project is that you have a separately managed unit that will only change when you change it for the project. The drawback of putting something under source control as part of the project is that you have a separately managed unit that will only change when you change it for the project. You wind up with a lot of copies to manage, and you don't get the latest and greatest tools automatically in the project. On the other hand, you won't have any surprising changes.

TIP There's a point in many projects where the number of changes drastically drops off and the application goes into a stable maintenance cycle. In some cases, this change occurs because you've solved the important issues. More often, the money runs out. It may be wise to copy items in into the project itself at that point to ensure a stable maintenance cycle.

The mechanics of handling items as part of the project or as a shared component are easy. If they're copied into the project, include them as a subdirectory under the project root. This allows both source control and the tool for creating new solutions to recognize them as part of the project. If they're shared, keep them in a central location.

Carrying Experience into Your Next Applications

The Apress Web site includes a tool to carry a project forward. I won't walk through the code, but I'll highlight important features of what the tool accomplishes. You can test the tool by moving the `InitialProjectWindows` or `InitialProjectWeb` example at the Web site to start development for one of your own applications. The `InitialProjectWindows` example combines the templates in framework in Chapters 6–9, and the `InitialProjectWeb` example combines the templates and framework in Chapters 6–8 and 10. I repeated the template and framework because I didn't want you to have to cut and paste from each chapter's examples to create your harness XML script file or to have your directories continue to be named for the chapters in this book. You can find details about `InitialProjectWindows` and `IntialProjectWeb` in the ReadMe file.

Understanding the Create Solution Tool

The tool has the following four basic tasks:

- Creating the directory structure
- Copying appropriate files
- Updating the project files to reflect new GUIDs
- Updating the solution files

Creating the directory structure is recursive but relatively straightforward. The tool assumes that you want all files in the specified directories, except project files (vbproj and csproj files).

NOTE You may want to extend the tool to skip copying files with other specific extensions.

The tool handles project files in a special way to avoid redundant GUIDs. Nonunique project GUIDs can lead to problems in Visual Studio and source control. The project and solution files contain project GUIDs, and the project files contain both their own GUID and the GUID for any projects referenced via the Projects tab of the Add References dialog box. To update all these GUIDs to new values, the tool opens the solution files and builds an array relating the old and new project GUIDs for all projects in the solution. The tool then updates GUIDs in the solution and each project file using the `Replace(oldGUID,newGUID)` method of the string containing the file contents. Replacing GUIDs in the project files updates the GUIDs both in the project definition and in any project references.

NOTE When creating your first project, be sure to place the XSD for the harness XML in the "magic" directory as described in Appendix A. This gives you IntelliSense for the harness XML file if you edit it in Visual Studio.

This tool won't fully create your project. Specifically, it'll probably copy some files from your previous project that you don't want. You'll probably want to do a Find in Files (from the Edit menu's Find and Replace submenu of Visual Studio) and search for several keywords that are likely in code specific to your previous project. You also may need to update the handling of components, paths within your code, and the harness script. But running this process helps automate the tedious process of getting a new project ready to run.

NOTE If you don't have any metadata issues, creating a new application using this tool, existing templates, and existing framework, you can take your project from initial start to running code in less than an hour or two.

Calling the Create Solution Tool

You call the create solution tool using the code generation harness. You'll use a `RunProcess` directive in the harness XML script that runs against the previous project. From a practical perspective, the new project doesn't exist yet, so you have to work with the old project.

Put this directive in a separate harness XML file and only run it when you want to create a new project. At that point, specify the directory you're copying from and where you want the project built and then run the process.

CAUTION Immediately after you successfully create the project, change the specified directory so you don't inadvertently overwrite a working project if you accidentally rerun the process.

The following is an example of the directive:

```
<kg:Section OutputDir="[Chapter6]FutureTest">
  <kg:Standard Name="Ch6 Moving Forward" Checked="true" />
  <kg:RunProcess>
    <kg:Standard Name="MakeNewSolution" Checked="true" Ordinal="0" />
    <kg:OutputRules HashOutput="false" OutputFileType="XML"
                    OutputGenType="Once" />
    <kg:SinglePass OutputFile="[BasePath]VBCode\Chapter6FutureTest" />
    <kg:Parameter Parameter="[Chapter6],
```

```
                    MiddleTier\MiddleTier.sln,
                    {ConsoleApplication1, bin, obj},
                    {Autogen,Editable}" />
    <kg:Process AssemblyName="CodeGeneration" TypeName="ProjectSupport"
                    MethodName="CreateNewSolution" AssemblyFileName="" />
  </kg:RunProcess>
</kg:Section>
```

Most of these directive attributes are familiar. The `Parameter` attribute contains a comma-delimited list of parameters to pass to the process. Table 5-6 describes each of these parameters.

Table 5-6. Parameters for the New Solution Tool

PARAMETER	DESCRIPTION
Root directory	Points to the root directory of the existing project and can contain abstract paths.
Solution filename	The name of the solution file to move forward.
Exclude directories	A comma-delimited array of directories to skip. Directories with these names will be skipped wherever they occur.
Empty directories	A comma-delimited array of directories to create but leave empty. Directories with these names will be emptied wherever they occur.

The empty directories will generally be the subdirectories that include generated code. If you have mixed generated and handcrafted code in individual subdirectories, the tool won't work.

Summary

You support the big picture of code generation by paying attention to important details about your code and development environment. These details include the following guidelines about directory structure:

- Separate handcrafted, editable, and generated files in different directories.
- Create a separate subdirectory for the output of each template.

The following guidelines will help you manage project files (.vbproj and .csproj files):

- Project files include a project GUID that should be unique for each project; don't copy project files.
- Automate the maintenance of the project's include file list in order to incorporate new files and discard unused ones if desired.
- Manually maintaining project files can also be effective.

The following guidelines will help you make effective use of source control:

- Keep everything under source control, including your handcrafted code, original metadata, templates, harness directives, and the generation harness.
- Make a conscious decision about which portions of your project are part of the project or shared between projects.
- Automate source control to simplify code generation

The following are a few of the debugging tips included in this chapter:

- Debugging generated code is a two-stage process; find the problem in your application code, and fix it in the generation process when the problem is in generated code or add a handcrafted fix.
- Use a clear process for debugging, such as a process of elimination based on the scientific method.

Your long-term success with code generation includes how well you can leverage your techniques between application development projects. The following steps will move your infrastructure and experience forward to a new project:

- Isolate your framework from your application code so you can move your framework forward.
- Perform a postmortem on your applications to identify the strengths and weaknesses in the framework used.
- Strategize what portions of the project you'd like to move from previous projects.
- Automate the process of starting a new project based on previous projects.

Additional Reading

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

- Try *Debugging Applications for Microsoft .NET and Microsoft Windows* by John Robbins (Microsoft Press, 2003).

- To learn more about the exception dialog box used by the code generation harness, see my article "Display Dissected Exceptions" in the November 2003 edition of *Visual Studio Magazine*, also available online at `http://www.fawcette.com`.

EXHEART

EXHEART

Part Two

Code Generation in Action

FINAL

EXHEART

CHAPTER 6

Mapping Database Structure into Business Objects

Map between your database structure and business objects to allow variations between these structures, now and as your application evolves.

Although your business objects are designed to interact with database information, your business object structure may not map directly to your database structure—or this mapping may become more complex as your application evolves. Variations in mapping might involve hidden housekeeping tables, semi-obsolete columns, and joined views of your data, as well as other issues. For example, you may have classes that don't expose a child to keep your client-side objects from becoming huge and unwieldy.

To provide any of these behaviors, you need to a mechanism for describing the mapping between your database structure and your business objects. This mapping is generally called Object Relational Mapping (ORM). Chapter 2 skipped over this mapping aspect of metadata to give you a greater understanding of code generation before tackling this complexity. Now that you've seen the importance of metadata in code generation, you're ready to face that your business object structure and your database structure may not be identical or may not be identical over the life of your project. Instead, most applications are somewhere between a wide divergence and some minor exceptions in the object structure. You probably

don't want to commit to exact symmetry. Instead, you can use the Chapter 2 metadata as one of two key inputs into the mapping process. The other input is the manually created ORM file that I'll introduce in this chapter. The ORM file is an Extensible Markup Language (XML) file that defines the mapping to your business objects.

There are commercial ORM tools that handle a variety of scenarios with front ends that help you describe the mapping. I'll take a different approach on two fronts. First, I won't give you a User Interface (UI). Instead, I'll ask you to understand the data behind the ORM process by focusing on the XML that makes up the ORM description. I want you to understand how the process works at a high level, rather than understand the UI of a particular tool. Second, this isn't a full-blown ORM tool. Writing such a tool is a huge effort, and I don't think you either want to wait three years to read this book or pay a few thousand dollars for it. The main distinction between this tool and a full-blown commercial quality ORM tool is that I don't solve all scenarios. For example, I designed the *structure* for multiple databases, multiple assemblies, and other complexities, the actual *implementation* focuses on a single database.

I'm not trying to solve side cases—if you have side cases, you probably understand what you need and how you should solve your specific problems. Instead, my focus is on making the common cases as simple to work with as practical while helping you understand what sometime appears to be a mystical ORM process. Most people don't use commercial ORM tools because they don't need that complexity. A simplified ORM tool, such as the one here, will help many projects.

NOTE Although this is an ORM mapping, it's simplified to cover the problems you'll generally encounter in application development. I won't attempt to cover all the scenarios high-end ORM tools cover—including support for heterogeneous database backends.

But what does an ORM tool have to do with code generation? Simple—ORM is all about metadata. The process consists of taking the database metadata you created in Chapter 2, combining it with ORM descriptions, and producing a metadata view of your business objects. This chapter walks you through the ORM description and the process of transforming these two inputs into your code generation metadata. You can look at this process at three levels—strictly as a user of the process, as a user with a basic understanding, and at the detailed knowledge you'd need to make modifications. This chapter works at the first two

levels; I'm not including detailed walkthroughs because that would detract from the overall goal of code generation. If you need to move to the detailed level to alter the tool, the section "Morphing ORM into Generation-Friendly Metadata" will help you find the right part of the code for the change you'd like to make.

ORM defines the metadata for both stored procedures and business objects. As discussed in the section "Incorporating ORM," creating the stored procedures alters the database and therefore the input to the ORM transformation process, so you'll run it a second time to ensure you're using the updated stored procedure information.

Creating the Object Relation Mapping (ORM) Solution

What's the problem ORM sets out to solve? You've got a relational database that's highly optimized for storing and retrieving information. You've got a set of business objects you've independently optimized to express your business problems in a coherent manner. These differing goals result in parallel but different representations of your data. The degree of variation between your object structure and database structure varies greatly by the domain you work in, by your personal or organizational style, and by the database history. Don't be alarmed if your mapping is a direct mapping. The point of ORM is to free you from the *requirement* that your objects and database tables be identical and remain identical throughout the life of your application.

NOTE If you perform a direct mapping and your database changes, these changes will be forced onto your business objects, potentially changing the calling code throughout your application. One of the biggest benefits of ORM is protecting against this kind of disruption by letting you control how your business objects evolve.

Once you've mapped between your relational database and your business objects, information flows smoothly between them. ORM is a plumbing issue. Once you've got the plumbing in place for morphing the metadata, later programmers won't need to think about the specific ORM techniques, though they may have to maintain nondirect mappings. Figure 6-1 illustrates the process of translating your manual ORM definition and your database metadata into the input for code generation. The "Morphing ORM into Generation Friendly Metadata" section covers the morphing process in more detail.

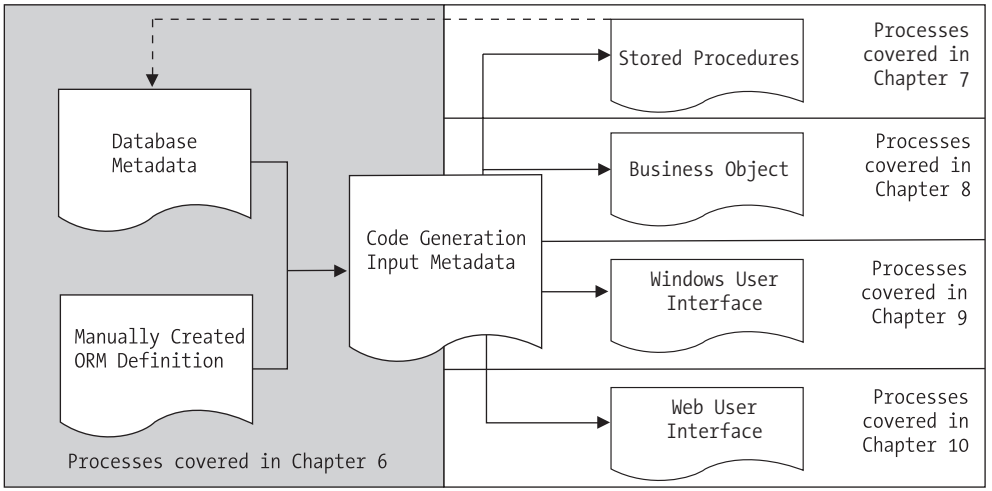


Figure 6-1. A high-level view of the transformation of database structure metadata and ORM file to create code generation input metadata

NOTE Objects represent a unique set of columns. Generally, objects map to stored procedures that themselves map to tables and columns. Although my samples use stored procedures, there's no requirement that you use stored procedures rather than dynamic SQL with code generation.

Exploring the Benefits of ORM

Early in your development process, ORM may seem like a stumbling block—another infrastructure piece standing between you and full-scale project development. I've certainly done my share of projects without it, but in retrospect I tucked the ORM process away into other tasks. For example, my stored procedures and careful naming performed simple mapping. That's okay, but it didn't allow the team to visualize the ORM mapping, and it wasn't very well suited to control growth as the projects evolved. The ORM process I'll describe uses a simple XML description, and you'll use this description to create your stored procedures (which are a key part of mapping). This gives you a view of your mapping that will be up-to-date because it's a code generation source.

Other benefits of the specific approach I'll take to ORM are as follows:

- Make it easy to define the mapping using only a few lines of XML in the ORM file when the mapping is direct.
- Maintain automatic flow-through of database structure changes whenever possible.
- Produce metadata that's very friendly to code generation.
- Allow you to generate even relatively complex stored procedures.
- Control the data visible to the template programmer.
- Allow customization of any element.
- Set important aspects of the resulting code via simple flags.

CAUTION The ORM transformation engine needs to understand your data relationships. This is easy if you define logical relations within your database structure because they're already in the metadata described in Chapter 2. If you don't elect to include data relations in your database, you'll have to create an additional step that adds the relations to the database metadata prior to ORM inference. You'll probably do this by creating an XML file that defines your relations and running an XSLT transform to incorporate this into your database metadata. For inference to be correct, it's got to have this information.

Simplifying ORM Mapping

I want the process of manually maintaining the ORM definition to be as simple as possible. I want to infer the object structure whenever the mapping between the database and your business objects is direct. This not only makes your life easier, it also allows database changes to automatically flow through to the business objects. Explicitly entered information blocks this process. If you explicitly specify each individual column, the system can't know whether you intended to include the new column information. Similarly, if you delete a column from the

database that you've explicitly included in your ORM description, would you want the ORM tools to remove it?

Automating the Flow of Database Structure Changes

Obviously, being able to add a table or column and automatically include this in the business objects when you next generate is a huge gain during development and an even bigger gain during later parts of your application life cycle. Although you sometimes need to override the database structure, the more direct the mapping, the easier it is to pass database changes through. If you generically request `AllColumns`, then it's quite easy to flow modified columns through your ORM model into your business objects because it's obvious you want all the columns.

Creating Code Generation-Friendly Metadata

At the end of the ORM morphing process, you want metadata that's very friendly to code generation.[1] Generation-friendly metadata significantly simplifies the generation templates, regardless of the mechanism you're using for generation. In this case, you'll be doing code generation into two languages—Transact SQL (T-SQL) for stored procedures and VB .NET or C# for .NET classes. The input for these targets differ because stored procedures focus on tables and stored procedure structures, where .NET code focuses on objects.

NOTE The ORM process I use builds stored procedures from the ORM mapping information and then builds an object model from the stored procedures. This two-step process allows handcrafted stored procedures to be reflected in the object model.

NOTE The .NET code focuses on objects whether you're creating middle tier or UI code. Middle tier code creates the objects, and UI code uses these objects and presents them to the user.

1. I call it *generation-friendly* metadata because it minimizes the number of complex XPath statements you need during code generation.

The easiest way to avoid compromises between the metadata needed for stored procedure generation and metadata for .NET objects is to provide separate XML files for each code generation set. You can hide this detail within the sequential process of the harness script, so programmers performing code generation don't need to be concerned with this complexity. You'll see more about the transformation process in "Morphing ORM into Generation-Friendly Metadata." I named my input file for stored procedures `SPInput.xml` and the input for .NET class generation `GenInput.xml`. You're free to use other naming, but I use these filenames to identify the different metadata files in this book.

NOTE You manually create an ORM file as an XML description of the exceptions to a direct database to object mapping. You'll morph this file into friendly metadata files as explained in the section "Morphing ORM to Generation-Friendly Metadata." This process creates a file named SPInput.xml that's friendly to stored procedure generation and a file named GenInput.xml that's friendly to .NET object generation and don't affect the structure.

Describing Complex Stored Procedures

You can either create stored procedures as part of code generation or use precreated stored procedures. Although stored procedure descriptions sufficient for generation aren't strictly part of ORM, this is a convenient place to build those definitions. If the mapping is direct and you use careful naming, the morphing process can infer the stored procedure descriptions so you don't have to explicitly describe them. When the mapping is indirect, or you don't want stored procedures to retrieve all child data (a common case), you'll have to specify a bit more information. The stored procedure description is fairly robust, allowing joined tables, multiple recordsets, and so on.

Controlling Metadata Visibility

There's another benefit of providing separate metadata input for stored procedure and .NET object generation. Because they're separate, you can remove database details from the metadata used for .NET object generation. This greatly

simplifies the XML structure the programmer who is building and maintaining the business object templates has to deal with and prohibits the template programmer from accessing unexpected database details. This is especially important because he might be able to accomplish the same thing by directly accessing the table structure information, which will break if an indirect mapping is later introduced.

NOTE You can hide irrelevant structural details of the database from template programmers. However, the data access code they're creating needs the name, parameters, and recordset structure output from the stored procedures.

Interfaces are an important part of modern object-oriented programming. They're important because of the value in clearly defining boundaries or the way two bodies of data and functionality touch. This is important both inside applications and in the human/development interface. If a programmer creating templates works with an XML input file containing a variety of information that they aren't expected to use, this unnecessarily complicates their life. Think of your metadata structure as your interface with a later template programmer.

Overriding Inferred Details for Customization

You'll sometimes need to override database defaults. There are two reasons you might do this: You need to adjust the structure, or you need to add or override details. If you need to make structural changes (such as specifying the list of properties created) or supply information used during the transformation, use the main elements `Object`, `Assembly`, `Property`, and so on. If you need to provide attributes to extend or override any inferred details held in attributes, use the informational elements `ObjectInfo`, `AssemblyInfo`, `PropertyInfo`, and so on.

To do this, you can create information objects that parallel each significant ORM element. For example, the `AssemblyInfo`, `ObjectInfo`, and `PropertyInfo` elements parallel the `Assembly`, `Object`, and `Property` elements in the ORM XML file. This approach maintains the ability to flow through structural changes when you supply additional information.

It's easy to get confused here. You manually maintain the ORM XML file that contains some of the information on your application structure. The ORM transformation (or morphing) process combines this with database metadata to build metadata you'll use to build your application. Within the ORM file, you'll include

elements that define the application. These elements have names that describe what you'll build with them such as `Assembly`, `Object`, and `Property`. If you don't specify these objects, they'll generally be inferred from database information. The elements in your ORM file have a large number of attributes that describe their details.

NOTE The main elements such as `Object`, `Property`, and `Assembly` affect the structure of the morphed metadata and therefore the generated application. When you include these, the morphing process uses your structural description in place of information retrieved from the database. The parallel informational objects, such as the `ObjectInfo`, `PropertyInfo`, and `AssemblyInfo` objects, contain extra attributes that are merged into the inferred nodes just before code generation and don't affect the structure.

Setting Characteristics of Generated Code via ORM Flags

The output from your ORM transformation is the input for your application code generation templates for business objects and UI. These code generation templates can behave differently based on flags in the metadata—originating from your manually created ORM file. For example, in Chapter 8 you'll see templates that create code to either use ADO.NET transactions or use enterprise transactions (COM+). You can extend this powerful technique by adding support for new attributes if you need to provide alternate code for different scenarios through your templates.

Seeing ORM File Examples

The ORM definition file lies at the heart of the ORM transformation and defines exceptions to the direct mapping. Because you'll maintain the ORM file by hand, you want to use the smallest simplest version that works for your project. Not only does this make your life easier, the more you infer, the more your application structure will reflect the current status of your database structure making maintenance easier throughout your application's life cycle.

This ORM system design maximizes the amount of information you can infer and organize from the database based on the business object layout you request. Because of this inference, the final version you use for generating code contains many attributes, but the ORM file you maintain remains clean and

simple. At the same time, you can override any inferred attribute by supplying attributes to the parallel informational element.

NOTE How you use each attribute in code generation is up to the templates you use, and you may ignore many of the attributes. The goal of the morphing process is to present the most complete picture of your object model available, with minimal concern for how you'll use this information in templates. The sidebar "Key Decision Point: ORM Holding Object Details" discusses this further.

This section covers the following three key things:

- How you manage responsibility for whether you describe the structure manually or whether the morphing process infers the structure.
- How you handle four basic scenarios for specifying the business structure of your application. Your stored procedures, business objects, and UI code will reflect this structure via your templates.
- The mechanism for overriding inferred attributes.

The ORM file you create may contain sections for objects, stored procedures, and special or custom types.

TIP You can use any XML editor to maintain the ORM file (such as the Visual Studio XML editor). If you'd prefer to use a custom tool, you can write one or modify the code generation harness's UI to edit the ORM file based on its XML Schema Definition (XSD). You can use the design of the generation harness as a starting point for developing your own tool. Appendix C describes the internals of the generation harness.

Key Decision Point: ORM Holding Object Details

In the course of building the metadata you'll use for your stored procedure and middle tier code generation, you'll encounter a few key decision points. By identifying these decision points, and why I made the choices I did, you'll be prepared to make the right decision for your project. Obviously, one benefit of following my lead is that you can reuse the code I've included in the book. But it's not a one-size-fits-all world, and I want to identify some of these key decision points so you can evaluate the fit for your project.

One key decision point is how closely the metadata you build from your ORM will reflect the way you'll express your object model in .NET code. This metadata could remain vague, or it could be quite specific. For example, you could explicitly define each property, or you could retain a more structural view of columns. If you take this view to the extreme, even housekeeping properties of your object would be included in your metadata.

I think there's a logical middle ground. The metadata can contain all of the properties derived from your data while allowing the templates themselves to introduce housekeeping properties and methods, such as the current status of the object (unchanged, new, modified, or deleted).

The split between specificity of metadata and what the template introduces is a fuzzy one. Because you'd like to reuse the templates, this decision isn't localized to your current project but reflects the way your templates will be used over time. Pushing the determination of how a nuance will be covered (such as the multiple parent issue discussed in the sidebar "Key Decision Point: Unique or Shared Child Objects") into the metadata simplifies the template. But if you push the decision forward into the metadata, all templates that use the metadata must be able to interpret that metadata representation. If you push the decision into templates, you'll generally work harder to handle a specific scenario consistently within all of your templates.

My guideline is to push things into the metadata. The process of morphing the metadata occurs only once and tends to change less than the templates. Thus, I can solve complex problems once and reuse the logic.

Managing Responsibility

To infer a maximum amount of information while still providing a maximum amount of flexibility, you need to be able to specify what portions of the object structure you're maintaining and what portions of the structure you expect the morphing process to infer. Examples of the type of structural information you'll infer include the objects that make up your application and the properties of a particular object. There are four possibilities as follows:

- You specify no items (infer all objects or properties).
- You specify all the items you want.
- You specify some items but infer the rest managed automatically.
- You want no items created.

If you want all items inferred—meaning you don't want to specify any items—you can include the `All`*`itemtype`* element such as `AllObjects` or `AllProperties`. If you specify you want all of a particular item, the morphing process infers that you want all of its dependent elements as well. For example, if you request `AllObjects`, the morphing process infers that you want all `AllProperties` for each of the new objects.

NOTE This section discusses the mechanism for overriding the inferred *structure*. The later section "Managing Attributes" discusses overriding attributes, which you do somewhat differently using informational elements such as `ObjectInfo`.

If you add any items, even a single `Property` or `Object`, the morphing process assumes you're taking responsibility for the set of items, and morphing will not add any new items by default. This is specific to the context. For example, if you add a property to one object, you're declaring control of only that single object, and the morphing process still infers the properties of all other objects.

You'll often want to specify one or more items while having the rest inferred. You'd do this to handle a structural issue you want to override, such as whether to include a particular property (not to override attributes). You'll often use this technique to take responsibility for structural issues of some objects while having

the remainder of your objects inferred. Another example would be overriding the handling of one child collection while inferring any other child collections.

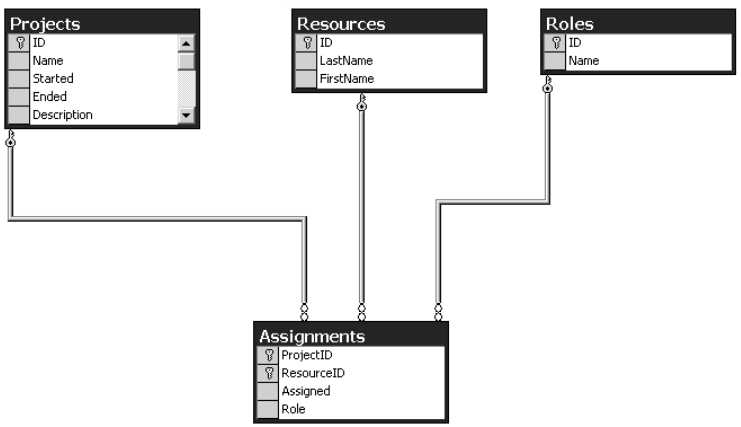
NOTE Structural aspects of your application are defined as *elements* in your ORM file. Details are defined using *attributes* on these elements.

You'll need the fourth approach—wanting no objects or properties—less often, but it's important when you want to create an object with no children. You can do this by including the `No*itemtype*` element such as `NoChildCollections`.

The examples in the following sections show how these elements and other features work in various mapping scenarios.

Introducing the Example Database

In Chapter 8 I'll explain why I decided to use the architecture from Rockford (Rocky) Lhotka's book *Expert One-on-One Visual Basic .NET Business Objects* (Apress, 2003). For consistency, I'll also use the database Rocky uses for these examples. This database describes a project tracking application and consists of four tables, as shown in the following figure. The Assignments table is a child of the Projects, Resources, and Roles tables. This database provides primary keys for each table, but it adds a few twists to standard primary keys. It uses a combination of identity and Global Unique Identifier (GUID) primary keys and uses a complex (two-part) primary key for the Assignments table.



CAUTION The variations in primary keys make for a great example but may not be the best approach for your real-world application. A design that provides GUID keys for parent tables, while using identity keys for child tables, will break if an unexpected grandchild table is added later. Also, two-part primary keys will require two-part foreign keys if you add an unexpected grandchild. For example, adding a Timesheet child table to the Assignments table would require a two-part foreign key to the Assignments table. To avoid future problems, settle on either identity columns or GUID columns and use them as single-part primary keys on any tables except those that couldn't logically have child tables (such as simple lookup tables).

TIP The benefits of GUID primary keys are that you can generate them on the client and thus avoid needing to cascade primary keys. GUIDs also work well with replication (something that may become more common with increasing use of intermittently connected devices using tools using SQL CE). The drawbacks of GUIDs are that they take more room in your database and are more difficult to manage. An example of the management difficulties is that you'd need to accurately supply a primary key to run select statements in Query Analyzer to determine details of a particular record—and it's harder to accurately type in a GUID than an identity column value.

Specifying the Simplest ORM Definition

The morphing process can fill in almost all of the blanks in creating metadata. The closer you are to a direct mapping between business objects and database structure, the better the inference in the morphing process and the less you have to specify in the ORM XML file. Listing 6-1 shows the simplest version of ORM.

NOTE The next six sections cover ORM definitions starting with the simplest versions and moving on to structural overrides. As you read the remainder of this section, keep in mind that you take on as much responsibility, or as little, for defining the mapping as appropriate for your application.

Listing 6-1. The Simplest Useful Version of an ORM XML File

```
<orm:MappingRoot xmlns:orm="http://kadgen.com/KADORM.xsd"
          MapDataStructure="PTracker"
          TransactionType="ADONET"
          Namespace="cslaTest"
          TransactionForRetrieve="false"
          RetrievePattern="wx_app[Object]_SelectOne"
          SetSelectPattern="wx_app[Object]_SelectAll"
          CreatePattern="wx_app[Object]_Insert"
          UpdatePattern="wx_app[Object]_Update"
          DeletePattern="wx_app[Object]_Delete" >
   <orm:Assembly Name="BusinessLayer" >
      <orm:AllObjects/>
   </orm:Assembly>
</orm:MappingRoot>
```

Most of the information in the simplest version of an ORM file is contained in the root. It's where you'll define your database name, and calling this attribute `MapDataStructure` indicates that it could be a data source other than a database (such as a Web Service). If you're using a different type of source, such as a Web Service, you may need to modify the morphing process to infer from another schema source, such as Web Service Description Language (WSDL), or manually maintain the entire ORM file.

TIP The ORM file's root element includes the namespace `http://kadgen.com/KADORM.xsd`. Complete details of the ORM definition file is contained in this schema.

You can define the attributes you see on the root element in this example (except `xmlns`) on any element, and the transformation process will use the attribute closest to the leaf node. For example, you could define the `MapDataStructure` on the `MappingRoot` and the `Assembly` elements, and the morphing process would use the `MapDataStructure` attribute defined for the `Assembly`.[2] Part of the morphing process copies these attributes into all of their child nodes for the convenience of the template programmer.

You can use the `TransactionType`, `Namespace`, and `TransactionsForRetrieve` in templates. These were the attributes I found useful in implementing the Component-based, Scalable, Logical Architecture (CSLA) templates in Chapter 8. There's nothing special or magic about these attributes. You can add any attributes you want to the root node and modify the morphing process to copy these into all child nodes. Then you can use these attributes in your templates. The pattern of allowing attributes to be defined on any element is important because it allows you to specify overridable defaults for any attribute helpful in your code generation templates.

TIP Visual Studio uses XSD to determine legal syntax for your XML file, and if you just add new attributes, Visual Studio will mark them as errors with red "squigglies." To fix this, you'll need to both add your new attributes to the XSD and place a copy of the new XSD in the directory Visual Studio uses for XSD files. That directory is [Program Files]\[Microsoft Visual Studio .NET]\Common7\Packages\schemas\xml.

Once you've specified the information in the `MappingRoot`, all you need is the single `Assembly` element to build your application if the mapping between your business objects and database is direct and you don't need to generate stored procedures.

Specifying Minimal Stored Procedure Information

To generate stored procedures, you need two additional pieces of information. You need to know what to generate, and you need to know the default privileges. The differences from Listing 6-1 are bold in Listing 6-2.

---

2. Note that I haven't tested multiple database scenarios. There are likely to be problems with multiple database scenarios where the same table or field name appears in multiple databases.

Listing 6-2. An ORM File Specifying Minimal Stored Procedure Information

```
<orm:MappingRoot xmlns:orm="http://kadgen.com/KADORM.xsd"
          MapDataStructure="PTracker"
          TransactionType="ADONET"
          Namespace="cslaTest"
          TransactionForRetrieve="false"
          RetrievePattern="wx_app[Object]_SelectOne"
          SetSelectPattern="wx_app[Object]_SelectAll"
          CreatePattern="wx_app[Object]_Insert"
          UpdatePattern="wx_app[Object]_Update"
          DeletePattern="wx_app[Object]_Delete"
          Generate="CRUDS">
   <orm:DefaultPrivileges>
      <orm:DefaultPrivilege Grantee="public" Rights="RS" />
      <orm:DefaultPrivilege Grantee="Admin" Rights="CRUDS" />
      <orm:DefaultPrivilege Grantee="Manager" Rights="RUS"   />
   </orm:DefaultPrivileges>

   <orm:Assembly Name="BusinessLayer" />
</orm:MappingRoot>
```

The `Generate` attribute on the `MappingRoot` element specifies which stored procedures you want to generate. This will generally be CRUDS for `Create`, `Retrieve` (which retrieves one record), `Update`, `Delete`, and `SetSelect` (which you could also think of as select all).

The `DefaultPrivileges` are the privileges you want for all stored procedures of your application if you don't supply special privileges. This example defines the privileges for three different roles. Privileges are also defined using the CRUDS pattern. In this example, all uses can retrieve data—either individually or as a group. Managers and administrators can update, and only administrators can create and delete records. You can also override these privileges by specifying them as described in the section "Applying Different Stored Procedure Privileges."

NOTE Including `DefaultPrivileges` and the `Generate` attribute allows the morphing process to create a section supporting stored procedure generation. The harness script determines whether you run templates to generate the stored procedures and the templates determine whether details such as `DefaultPrivileges` are used correctly.

Limiting Children: Avoiding the Mega-Object Problem

One of the problems you can encounter if you rely on inferred mappings is creation of mega-objects that are impractical to work with at runtime. For example, imagine that you've added a Timesheet table as a child of the Assignments table in the example database. You probably wouldn't want to load all of the Timesheets whenever you loaded a Project or Resource. I call this the mega-object problem, and it occurs because the morphing process can't infer sensibility—only what's defined in the database. You need to intervene and explicitly define that although a relation exists between Assignments and Timesheet tables in the database, you don't want that relationship expressed in your business objects. This is one of the situations where you'll have to take control of child collections.

The default morphing creates the definition for a `ProjectAssignment` object. The default object would contain children if there were any. If you wanted to block those children, you'd use the `NoChildCollections` element.

In this and later listings, I removed the attributes of the `MappingRoot` element and the `DefaultPrivilege` elements to shorten the listing. With the new elements marked in bold, the ORM definition to exclude children from `ProjectAssignment` and not generate an `Asssignment` object is as follows:

```
<orm:MappingRoot ...>
      <orm:Assembly Name="BusinessLayer" >
      <orm:AllObjects/>
      <orm:Object Name="ProjectAssignment">
         <orm:NoChildCollections/>
      </orm:Object>
      <orm:Object Name="Assignment" NoGen="true" />
   </orm:Assembly>
   </orm:MappingRoot>
```

In the sample database, the `ProjectAssignment` has no children (you can refer to the earlier sidebar "Introducing the Example Database"), so the `NoChildCollections` element is just for illustration and does nothing. You may occasionally want to include an extra `NoChildCollections` element like this to block future children from being included.

Another scenario you may encounter is a table that should be accessible only through parent objects. In the example database, the Assignments table should be accessible only through the `ProjectAssignments` and `ResourceAssignments` child collections of the `Project` and `Resource` objects (see the sidebar "Key Decision Point: Unique or Shared Child Objects"). If you don't also want the `Assignment` object created, you can use the `NoGen` attribute on the `orm:Object` element.

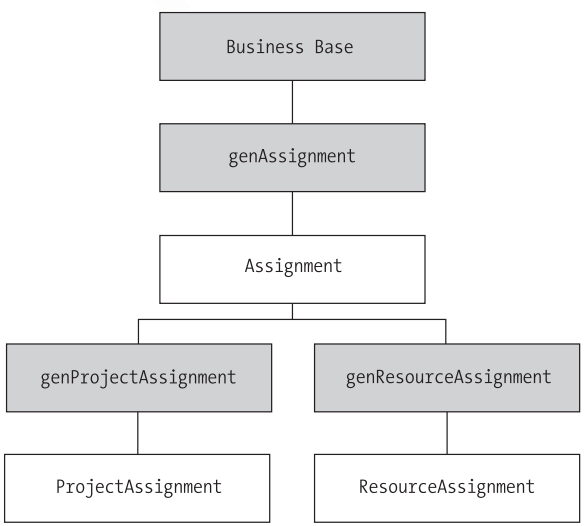
Key Decision Point: Unique or Shared Child Objects

A key decision point is whether you'll reuse child objects. In the case of the project tracking system, the Assignments table is a child of both the Resource and the Project tables. This results in two logical entities that use the same underlying physical entity—a logical *resource assignment* and a logical *project assignment* that each map to the physical Assignments table. You can also take a third approach where you build `ResourceAssignment` and `ProjectAssignment` objects (reflecting the logical view) with each deriving from an `Assignment` object (reflecting the physical view). As discussed in the sidebar "Key Decision Point: ORM Holding Object Details," you may reflect this detail in the code generation metadata input you create from the ORM process.

The benefit of the single `Assignment` object is that you can reuse a maximum amount of code. Much of this code is generated code, and it isn't as important to maximize reuse in generated code because the creation is automatic. The single-object approach becomes quite difficult when there are differences between the logical entities—such as a business rule that makes sense only with one of the logical entities. An example of such a business rule would be providing a resource name only on the `ProjectAssignment`. Other disadvantages are that you can't determine the objects purpose from its class name, and it's hard to hide the current parent as you usually would in a child object.

The benefits of the individual entities are that you can supply unique business logic in derived classes of each logical entity, differentiate an entities purpose by its class, and tune the child object to the context of the parent. The disadvantage is that you have to provide business logic in multiple places.

If you want to make things a bit more complex, you can solve these problems with the third approach described in the following figure. The shaded boxes in this figure are `MustInherit` classes.



Specifying Individual Properties

You may not want every object to include properties for all of the columns returned by the stored procedure. This may occur when there are large fields such as images returned. To manage this scenario, you'll define individual properties:

```
<orm:MappingRoot xmlns:orm="http://kadgen.com/KADORM.xsd" ...>
   <orm:Assembly Name="BusinessLayer" >
      <orm:AllObjects/>
      <orm:Object Name="ProjectAssignment">
         <orm:Property Name="ResourceID"/>
         <orm:Property Name="LastName"/>
         <orm:Property Name="FirstName"/>
         <orm:Property Name="Assigned"/>
      </orm:Object>
   </orm:Assembly>
</orm:MappingRoot>
```

In this case, I've excluded the `Role` property by listing only the other properties. This is an unusual need, but it illustrates the general approach of specifying the items you want, which you can also use when specifying objects and child collections. Using this technique blocks the ability of the morphing process to incorporate data structure changes automatically. To apply control by maintain the ability to flow changes through, you could extend the transformation process to recognize excluded elements and support a syntax such as `<orm:ExcludeProperty Name="Role"/>`.

Defining Special Types

It'd be nice if everything you needed was neatly available in the .NET system types, and you could easily map these types to your database types. Life isn't that easy today, and the next version of SQL Server (currently code-named Yukon) will complicate the picture because XML-based data types are likely to result in columns of data that are inherently more complex—such as a phone number field with a dictionary of phone numbers for an individual contact. Today mapping directly to .NET system types comes up short at least when you're dealing with nulls and images. Although the SQLTypes exist and can handle null values, they're difficult to work with and don't offer a lot of flexibility. To solve these problems, you may want to create your own classes that wrap specific data-handling behavior. An example of this sort of class is the `SmartDate` class Rocky

Lhotka included with his book *Expert One-on-One Visual Basic .NET Business Objects* (Apress, 2003). The core purpose of the `SmartDate` class is to handle null dates smoothly. It also demonstrates how to define wrapping classes for other special types you might create, such as wrappers for image data.

When you use special types, your business class will hold objects of the wrapper class instead of directly interacting with the underlying type. Well, that sounds good, but how will you map this during code generation? You can define the type for the property by using a SQL User-Defined Type (UDT), specifying the type for the properties in the `Property` or `PropertyInfo` element or by declaring all data of a certain type to use a specific wrapper class. I take the last approach in the Chapter 8 templates and map all date properties to the `SmartDate` class.

For code generation, you need several key pieces of information about the special type, and a `SpecialType` element holds this information. There are three times you'll need to interact with each wrapped type as follows:

- To get or set the business object property
- To retrieve data from the DataReader
- To put the data into a parameter for an insert or update command

The `SpecialType` element describes the mechanisms for these interactions:

```
<orm:SpecialTypes>
   <orm:SpecialType Name="SmartDate" >
      <orm:UseForAllType Name="System.DateTime"/>
      <orm:Property Retrieve="Text" Set="Text" AccessVia="System.String" />
      <orm:SafeDataRetrieve Name="GetSmartDate">
         <orm:SafeDataProperty Explicit="ToString" />
      </orm:SafeDataRetrieve>
      <orm:SafeDataStore Name="DBValue" />
   </orm:SpecialType>
</orm:SpecialTypes>
```

In the `Property` element, the `Get` and `Set` attributes specify the methods of the `SmartDate` class called to interact with values. The `AccessVia` attribute specifies the type of the business object's property that uses the `SmartDate` class.

Clear as mud? Looking at the resulting property may clarify it for you. The `Ended` property is a date field wrapped in a `SmartDate` object. `AccessVia` identifies that the `Ended` property as a `System.String`. The `Get` and `Set` attributes indicate that

interaction with the underlying mEnded class variables is via its Text method. The property procedure generated from this SpecialType element is as follows:

```
Public Overridable  Property Ended() As System.String
  Get
    Return mEnded.Text
  End Get
  Set(ByVal Value As System.String)
    If mEnded.Text <> Value Then
      mEnded.Text = Value
      MarkDirty()
    End If
  End Set
End Property
```

Pulling information from the DataReader presents a couple of challenges. You'll need to wrap the DataReader in a specialized DataReader class that recognizes the new type and can retrieve it using the SQL DataReader's underlying methods. The Name attribute of the SafeDataRetrieve element and the Explicit attribute of the SafeDataProperty element define this action.

In most cases, you'll retrieve a value using the method specified by the SafeDataRetrieve, but it won't be in the format, or of the type, required by the special property. You can specify a method to run via the Explicit attribute of the orm:SafeDataProperty element. Again, an example should clarify it for you. These elements result in this:

```
Me.PropertyName = sdr.GetSmartDate().ToString
```

NOTE You can't derive from the DataReader because it's NotInheritable (sealed in C#). Because you'll want your specialized DataReader class recognized as a DataReader, it should wrap a SQLDataReader object and implement the IDataReader interface.

NOTE Chapter 8 uses the architecture Rockford (Rocky) Lhotka presents in his book *Expert One-on-One Visual Basic .NET Business Objects* (Apress, 2003). This chapter describes Rocky's SafeDataReader class.

Note that this solution is generic. You can create a class that wraps other data needing complex behavior, such as exchanging images with the database, and provide a `SpecialType` description that allows code generation to work smoothly with your new class.

Applying Different Stored Procedure Privileges

Stored procedures are usually inferred from object definitions—which may themselves be inferred. There are at least three scenarios where you'll want to define additional aspects of stored procedures as follows:

- Applying different privileges to the stored procedure
- Creating joined tables
- Defining handcrafted stored procedures you don't want to generate

To apply unique privileges to a stored procedure, you create an explicit `SPSet` element and apply the privileges you'd like. The `SPSet` name must match the name in the `UseSPSet` child element of the `Object` element that'll use the stored procedure. The `UseSPSet` element may be either explicitly entered or inferred. The `SPSet` element must also be in a `Build` element.

The way security is implemented depends on the templates you use. In the templates in this book, these roles define SQL Server roles with execute privileges on the generated stored procedures. The same roles are used via Windows Integrated Security at runtime. Templates that use these security values are included with the Chapter 7 and Chapter 8 downloads.

Explicit privileges override default privileges, so this XML provides access to just the Admin role (an arbitrary SQL Server role), regardless of the defaults for the ORM file:

```
<orm:MappingRoot ...>
    <orm:Assembly Name="BusinessLayer" >
      <orm:AllObjects/>
   </orm:Assembly>
   <orm:Build>
      <orm:SPSet Name="Projects">
         <orm:Privilege Grantee="Admin" Rights="CRUDS"/>
      </orm:SPSet>
   </orm:Build>
</orm:MappingRoot>
```

The remainder of the stored procedure information, such as the `BuildRecordset`. The `BuildRecordset` element defines the contents of each returned recordset.

CAUTION The template, not the metadata, holds the ultimate responsibility for the privileges granted to a particular stored procedure. Setting the correct privileges in the metadata is an important part of the process, but you need to also ensure that your stored procedure generation uses this information properly and remember that privileges defined in your ORM file aren't used at all for handcrafted stored procedures.

Generating Joined Tables

Joined table definitions can get a bit complicated. A simple join between two tables involves specifying the SQL recordset, the two tables, and how they'll be joined. The XML that specifies a simple join is as follows:

```
<orm:MappingRoot ...>
   <orm:Assembly Name="BusinessLayer" >
      <orm:AllObjects/>
   </orm:Assembly>
   <orm:Build>
      <orm:SPSet Name="Projects">
         <orm:BuildRecordset>
            <orm:BuildTable Name="Projects" />
         </orm:BuildRecordset>
         <orm:BuildRecordset >
            <orm:BuildTable Name="Assignments" />
            <orm:BuildTable Name="Roles" />
            <orm:Joins>
               <orm:Join Type="Inner">
                  <orm:Left JoinTable="Assignments"  />
                  <orm:Right JoinTable="Roles" />
                  <orm:JoinOn Clause="Assignments.RoleID=Roles.ID"
               </orm:Join>
            </orm:Joins>
         </orm:BuildRecordset>
      </orm:SPSet>
   </orm:Build>
</orm:MappingRoot>
```

To illustrate how you can add complexity to your stored procedure definitions, this stored procedure returns two recordsets. The first recordset represents a single table for project information. The second recordset includes two joined tables for child assignments joined to roles. The ORM file defines the individual tables and then defines the join. This allows you to join two tables or join a table to another named join, such as this:

```
<orm:Joins>
   <orm:Join Type="Inner">
      <orm:Left JoinTable="Assignments"  />
      <orm:Right Join="AdditionalJoin" />
      <orm:JoinOn Clause="Assignments.RoleID=Roles.ID"
   </orm:Join>
    <orm:Join Name="AdditionalJoin">
      <orm:Left JoinTable="Roles"  />
      <orm:Right JoinTable="AnotherTable" />
      <orm:JoinOn Clause="Roles.ID=AnotherTable.ID"
   </orm:Join>
</orm:Joins>
```

CAUTION The templates of the generation process may or may not prohibit a Cartesian join. If you leave out the `JoinOn` element, you may build a stored procedure that returns millions of rows.

Specifying Handcrafted Stored Procedures

Sometimes you'll want to handcraft a particular stored procedure while generating your remaining stored procedures. You'll specify the stored procedure to run in metadata by using the `RunSP` element, such as this:

```
<orm:MappingRoot ...>
   <orm:Assembly Name="BusinessLayer" >
      <orm:AllObjects/>
   </orm:Assembly>
   <orm:Build>
      <orm:SPSet Name="Projects" Generate="CUDS">
         <orm:RunSP Name="wx_appProject_SelectOne" Task="Retrieve" /> />
      </orm:SPSet>
   </orm:Build>
</orm:MappingRoot>
```

Removing the R (Retrieve) from the Generate attribute of the SPSet indicates that code generation shouldn't create the stored procedure. The RunSP element indicates which stored procedure you're specifying via the Task attribute.

NOTE The name of the stored procedure is moved to the Object element by the morphing process based on the SPSet name. As part of the Object element, generated data access code will use the handcrafted stored procedure in a manner identical to how generated stored procedures are used.

Understanding the ORM Structure and Attributes

This section walks through some of the structure and attribute details of the ORM file as a reference. This section isn't intended for easy reading. I included it for you to return to when you're working with your ORM file and need more information than what's contained in the XSD.

There are five main elements of the ORM file (see Table 6-1). The MappingRoot contains several key attributes. DefaultPrivileges contains child elements for each role that should receive a set of default privileges. An Assembly element is the top of a hierarchy that describes your business objects. The Build element is the top of an optional hierarchy that described details of your stored procedures. Finally, one or more SpecialType elements describe special data handling.

Table 6-1. Main Elements of the ORM File

ELEMENT	ELEMENT DESCRIPTION
MappingRoot	Root element of ORM XML that also holds default values for several key attributes
DefaultPrivileges	Defines the default privileges for generated stored procedures
Assembly	Defines the business object structure
Build	Defines stored procedures to generate
SpecialType	Defines classes that hold data, particularly for null handling or tricky data types such as images

The `MappingRoot`, `DefaultPrivileges`, and `Assembly` elements are discussed in the sections that follow. The `Build` and `SpecialType` elements were discussed in the sections "Defining Special Types" and "Generating Joined Tables" earlier in this chapter. Because the `Build` and `SpecialType` elements contain many child elements and attributes that you'll rarely use, I haven't included a description of each element and attribute.

NOTE You're in charge of the attributes your templates recognize. You can extend or simplify the list of attributes I use. Just be sure to maintain an XSD schema and document how you support (or intend to support in the future) various attributes. This information is critical to maintenance programmers.

Specifying the MappingRoot

The root node of the ORM XML structure serves as the single root, holds the XML namespace for the elements in the file, and contains some important attributes. The attributes that are valid for the `MappingRoot` element can also appear on lower elements in the XML hierarchy. The value of the attribute that appears above and closest to the current context is used.

The attributes of the `MappingRoot` element hold a good deal of information about the overall code generation process. Table 6-2 shows the attributes appropriate for the `MappingRoot` element.

CAUTION The tables describing ORM structure in this chapter indicate only what information the ORM structure supports. Whether these issues are actually expressed in the templates or scripts is an independent issue.

Table 6-2. Attributes Appropriate for the `MappingRoot` *Root Element of the* `ORM` *File*

ATTRIBUTE	PROGRAMMER SPECIFIED	EXAMPLE	DESCRIPTION
`MapDataStructure`	Generally	`PTracker`	Defines the database to use. The attribute name reflects that your code generation might use a Web Service, XML file, or other source.
`TransactionType`	Generally	`ADONET`	Defines the type of transaction used. Supported transaction types are `Enterprise`, `ADONET`, or `None`. `Enterprise` indicates Enterprise Services transactions (previously known as COM+).
`Namespace`	Generally	`cslaTest`	Defines the namespace for the entity. You can use this in templates to place code in different .NET namespaces. In general you'll use one namespace per data structure, and the templates in this book don't use this attribute.
`TransactionForRetrieve`	Generally	`false`	Defines whether retrieve operations should be contained in a transaction. The default is `false`.
`RetrievePattern`	Generally	`wx_app[Object]_SelectOne`	The pattern for careful naming used in the database. Currently `[Object]` is the only token supported. The expectation is that each stored procedure will map to an object.
`SetSelectPattern`	Generally	`wx_app[Object]_SelectAll`	The pattern for careful naming used in the database. Currently `[Object]` is the only token supported. The expectation is that each stored procedure will map to an object.

Table 6-2. Attributes Appropriate for the `MappingRoot` *Root Element of the* `ORM` *File, continued*

ATTRIBUTE	PROGRAMMER SPECIFIED	EXAMPLE	DESCRIPTION
`CreatePattern`	Generally	`wx_app[Object]_Insert`	The pattern for careful naming used in the database. Currently `[Object]` is the only token supported. The expectation is that each stored procedure will map to an object.
`UpdatePattern`	Generally	`wx_app[Object]_Update`	The pattern for careful naming used in the database. Currently `[Object]` is the only token supported. The expectation is that each stored procedure will map to an object.
`DeletePattern`	Generally	`wx_app[Object]_Delete`	The pattern for careful naming used in the database. Currently `[Object]` is the only token supported. The expectation is that each stored procedure will map to an object.
`Generate`	Generally	`CRUDS`	Indicates which stored procedures should be generated. Abbreviations correspond to `Create`, `Retrieve`, `Update`, `Delete`, and `SetSelect`.

Setting Default Privileges

The `DefaultPrivilege` element indicates the default privileges for generated stored procedures. As shown in the section "Generating Stored Procedures," each `DefaultPrivilege` child element specifies a single `Grant` statement. These `Grant` statements execute in the order they appear in the document. Table 6-3 shows the attributes valid on the `DefaultPrivilege` elements.

NOTE Permissions for objects are determined directly from the stored procedures—whether the stored procedures are generated or handcrafted.

Table 6-3. Attributes Supported on the `DefaultPrivilege` *Element*

ATTRIBUTE	PROGRAMMER SPECIFIED	EXAMPLES	DESCRIPTION
`Grantee`	Default or Specify	`public` `Admin`	Defines the role the permission will be granted to.
`Rights`	Default or Specify	`CRUDS` `RS`	The rights to be granted—specified in the CRUDS format. The first example grants all rights to the role, and the second grants read only rights.

Specifying Assembly Details

Assembly elements contain child elements that describe an individual middle tier assembly. If you have several databases, you might want to split access to them into separate assemblies, but most applications will have a single middle tier assembly. Objects are part of assemblies, so you nest `Object` elements within `Assembly` elements. The assembly description is simple and shown in Table 6-4. Remember the `Assembly` element can also include the overall attributes described for the `MappingRoot`.

CAUTION I designed the structure to accommodate multiple databases and multiple assemblies so the structure itself could be extended. I haven't yet developed or tested multiple database or multiple assembly projects with this tool.

Table 6-4. Attribute Supported by the `Assembly` *Element*

ATTRIBUTE	PROGRAMMER SPECIFIED	EXAMPLE	DESCRIPTION
`Name`	Ignored	`BusinessLayer`	Name of the middle tier assembly. This isn't used in book's templates.

Using the AllObjects Element

The `AllObjects` element appears as a child of the `Assembly` element and indicates that the morphing process should create an object definition corresponding to each table in the specified database. This element can be used alone or in conjunction with specific `Object` elements when you'd like to define one or more objects but have the remaining objects inferred.

Setting Specific Object Elements

`Object` elements appear as children of `Assembly` elements. Each `Object` element describes a single business object. `Object` elements can include child elements that describe properties, child collections, and stored procedures used for data access. Table 6-5 shows the attributes that are valid for the `Object` element, in addition to the overall attributes described in the section "Specifying the MappingRoot."

Table 6-5. Attributes Supported by the `Object` *and* `ObjectInfo` *Elements*

ATTRIBUTE	PROGRAMMER SPECIFIED	TYPE	DESCRIPTION
`Name`	If element specified, it must include name	String	Defines the name of the object as used in its class name and to relate parent and child objects. The name must be unique within at least the assembly. If this attribute isn't present, nothing is generated.
`Islookup`	Often	Boolean	Whether the column should be treated as a lookup column. A lookup column is displayed as a combo box in most UIs.
`HideInUI`	Occasionally	Boolean	Templates that create UI forms and pages are expected to recognize this attribute and not generate a UI. In the templates with this book, objects marked `HideInUI` receive a `NoGen` attribute.

Table 6-5. Attributes Supported by the `Object` *and* `ObjectInfo` *Elements, continued*

ATTRIBUTE	PROGRAMMER SPECIFIED	TYPE	DESCRIPTION
`CollectionName`	Rarely	String	Defines the name of the collection to generate. If a `CollectionName` isn't included, a collection object isn't generated.
`Root`	Rarely	Boolean	Defines whether the object is available as a root object. This is always inferred as `true`.
`Child`	Rarely	Boolean	Defines whether the object is available as a child object. This is inferred as `true` if another object references this object as a child or the object is defined as a collection.
`CollectionRoot`	Rarely	Boolean	Defines whether the collection is a root collection. This is inferred to match the `IsRoot` attribute.
`CollectionChild`	Rarely	Boolean	Defines whether to create a child collection. This is inferred as `true` if another object references this collection as a child.
`ReadOnly`	Rarely	Boolean	Defines whether the object is read-only. This is inferred as `true` if there are no `Update`, `Create`, or `Delete` stored procedures in the `SPSet`.
`ChildOf`	Rarely	String	The name of the containment parent if the `Object` element was created as a child of another business object.

Each `Object` element contains either explicit or inferred `Property` elements and `ChildCollection` elements, as explained in the following sections. A number of other child elements exist but are almost always inferred.

NOTE You can define `Object` element attributes as extended properties of the corresponding table or stored procedure in a SQL Server database.

Using the AllProperties Element

The `AllProperties` elements indicate that the morphing process should create properties for each column returned by the `Retrieve` stored procedures. `AllProperties` is the preferable way to define properties because this allows changes in the database structure (such as new or deleted columns) to flow into the .NET objects. I can't imagine the scenario where it'd appropriate to combine `AllProperties` and `Property` elements for the same object.

Setting Specific Property Elements

In some cases, you'll want to define the specific properties that make up the object. You'll generally do this when you want to use a subset of columns and you don't want to alter the stored procedure. The drawback of specifying `Property` elements or attributes on `Property` elements is that they won't automatically be kept in sync with the database. Table 6-6 shows the attributes valid on `Property` elements.

Table 6-6. Attributes Valid on the `Property` *and* `PropertyInfo` *Objects*

ATTRIBUTE	PROGRAMMER SPECIFIED	TYPE	DESCRIPTION
`Name`	If element specified it must include name	String	Defines the name of the property as used in the object.
`IsRequired`	Occasionally	Boolean	Indicates to UI templates that an empty string isn't valid input. You could extend this to zero-value numerics.
`Caption`	Occasionally	String	The text to display next to the name. This is inferred by adding a space before all capitals in the column name. For many databases, this creates adequate captions.
`Ordinal`	Rarely	Number	The position within the retrieved SQL recordset.
`AllowNulls`	Rarely	Boolean	Whether null values are allowed in the column.

Table 6-6. Attributes Valid on the `Property` *and* `PropertyInfo` *Objects, continued*

ATTRIBUTE	PROGRAMMER SPECIFIED	TYPE	DESCRIPTION
`NETType`	Rarely	String	The name of the .NET type for the data in the column. This may be a wrapper class as discussed in the section "Defining Special Types."
`MaxLength`	Rarely	Number	For string data types, the maximum number of characters allowed in the field.
`IsPrimaryKey`	Rarely	Boolean	Whether the underlying column is part of the primary key.
`UseForDesc`	Occasionally	Boolean	Used to determine the columns for `SetSelect`.
`Empty`	Rarely	String	Inferred based on type and available for use in template development.
`Default`	Rarely	String	Inferred based on type and available for use in template development.
`UseForNew`	Rarely	String	Inferred based on type and available for use in template development.

NOTE For `Empty`, `Default`, and `UseForNew` attributes, record the value as a string, and code generation later places this string into your generated code. Because of this usage, you need to enclose strings in another set of double quotes, such as `Default=""default value""`.

NOTE You can define `Property` element attributes as extended properties of the corresponding table columns in your SQL Server database.

Using the AllChildren Element

In addition to defining properties, you can define child collections for your business objects. Because one-to-one child objects are far less common than child collections, I'm handling child objects as child collections containing a single item. At times, you may use the `AllChildren` element along with specific child elements. This allows you to define structural information about a single child while allowing its siblings to be inferred. You can even use this approach when you don't currently have sibling collections but might in the future.

Setting Specific ChildCollection Elements

Although it's fairly rare, in some cases you'll want to specify details about specific child collections. Table 6-7 describes the attributes valid for `ChildCollection` elements.

Table 6-7. Attributes Valid on `ChildCollection` *and* `ChildCollectionInfo` *Objects*

ATTRIBUTE	PROGRAMMER SPECIFIED	TYPE	DESCRIPTION
`Name`	If element specified it must include name	String	Defines the name of the child collection. This should match a child collection name of another `Object` element in the ORM file.
`ChildTableName`	Rarely	String	The name of the child.
`Ordinal`	Rarely	Number	Order in which the child collection's SQL recordset appears in the collection or recordsets returned by the stored procedure. If the first resultset returned is zero, the first child is one.

Summarizing the Rules for ORM Files

The most important rules for ORM XML file are as follows:

- If you define a `BuildRecordSet`, you must define its tables.
- If you define multiple tables in a `BuildRecordset`, you must define the join.

- Almost all elements require that you define a `Name` attribute if you define the element. Exceptions include `MappingRoot`, recordsets, and elements without a logical name such as `Recordset` elements.

Morphing ORM into Generation-Friendly Metadata

A morphing process transforms your ORM XML file and database metadata into the code generation input for either stored procedure creation or .NET business object creation. I'm including an explanation of this process because it's an important step in your overall code generation process, and it's helpful for you to have some feel for the complexity of the ORM morphing process when you decide whether to use ORM or a direct database mapping. Some of you will also need to modify the process to better fit your project's needs. However, this section is largely background reference; you could skip this section and still understand the code generation process.

Chapter 2 showed how to extract rich metadata from your databases. You'll combine this with an ORM definition file that describes the structure of your business objects and how they map to the database.

NOTE I use XSLT to transform the database XML to the generation input metadata. XSLT for transforming XML is considerably more complex than XSLT for code generation. XSLT for code generation is primarily a token replacement process, and XSLT for transforming XML is creating an abstract entity from scratch. In this case, it's somewhat more complex because the target is the result of multiple XSLT transformations, and there isn't a clear target for the interim XML files created. This is tough stuff.

Using XSLT for the transformation is convenient because XSLT has significant support for outputting XML and is designed for transforming information. Because XSLT works best if you pass through the data making a finite number of changes,[3] there are multiple steps to transforming the data. Each step does a specific set of tasks, and I'll share the reasoning behind each step in the sections that follow.

3. It may be possible to perform the transformation in fewer steps in XSLT through extensive use of node-set variables, but if you're going to work with that much complexity in individual stylesheets, you'd probably be better off using .NET code for the transformation.

TIP I haven't included the XSLT stylesheets in this section because I want you to focus on the process itself. If you expect to do significant modification to the transformation process, and you don't want a commitment to having some XSLT expertise associated with your project, you could rewrite the transformation in .NET code. If you do this, I suggest you build an object model that represents the structure and write similar steps to those described here for fleshing it out during the transformation. After you build this model, you can output it to XML.

The goals of the manually maintained ORM include supplying a lot of flexibility in the input. This means there's a near infinite number of input variations the morphing process may encounter. To handle these scenarios, the transformation process is extensive, and the output looks nothing like the input. This makes the transformation a tough job. But, by pushing complexity into the ORM process, both the ORM description file and code generation templates are as simple as possible. Although there are several XSLT transformation steps, and there's a lot of XSLT code, I simplified the process as follows:

- Each step does just a few tasks.
- The steps use variations on the same pattern.
- The process builds on techniques you've already seen, such as XSLT transformations, metadata extraction, and so on.

CAUTION This ORM transformation process depends on careful naming as described in Chapter 2. If your stored procedures are named in a less predictable manner, you'll have to do additional work to map stored procedures to the logical tables they represent.

NOTE One of the goals of the transformation process is to remove indicators of the source of a particular item. It really doesn't matter to a code generation template whether you manually entered information or infer it from the database. You don't want code generation templates to depend on any residual difference in source because the source might change over the life of the project with either more or less information inferred from the database.

Incorporating ORM

The overall process of code generating[4] your application will be something like this:

1. Prepare metadata.
2. Generate stored procedure scripts.
3. Apply stored procedure scripts to the database.
4. Generate middle tier code.
5. Generate UI code.

There are two complications here: First, generating stored procedures alters your database and invalidates the metadata you previously collected about stored procedures. The easiest way to solve this problem is to create metadata a second time. The second complication is that when you're using an ORM approach, preparing metadata is actually a two-step process—creating database metadata and then transforming it as described in this chapter.

NOTE Don't worry; you can't build an infinite logical loop here. That's because the metadata transformation (or morphing) process uses information about the database tables, and generating stored procedures obviously doesn't alter tables.

With the bold marking steps you perform twice—once before and once after you create the stored procedures—the process that your script will describe looks like this:

1. **Create database metadata.**
2. **Transform metadata for input to generating stored procedures.**
3. Generate stored procedure scripts.
4. Apply stored procedure scripts to the database.

4. All of this falls into step 3 of the five steps of the application code generation process described in Part One.

5. **Create database metadata.**

6. **Transform metadata for input to generating business objects and UI code.**

7. Generate middle tier code.

8. Generate UI code.

NOTE Transforming metadata—steps 2 and 6 in the preceding list—break down further and correspond to transformations 2–8 of the detailed list in Table 6-8.

The transformation process is logically a single process, but I break it down into multiple XSLT transformations to make the process easier. Table 6-8 describes the individual transformations that make up the morphing process as well as the order the transformations need to occur in. Brackets around input, and output filenames represent abstract filenames as discussed in Chapter 3 and indicate that they're used outside the transformation process, so their names may vary.

NOTE The names of the input files and output files in Table 6-8 are arbitrary and describe what I used in the sample scripts. Although I suggest you use this naming, nothing prevents you from using some other naming.

Table 6-8. Transforming Manually Entered ORM Description to Code Generation Input

STEP	PROCESS	XSLT TEMPLATE	INPUT FILE(S)[a]	OUTPUT FILE
1	CreateMetadata		None, uses database	[Metadata][b]
2	MergeMetadata		[Metadata] and [ManualORM]	ORM2WithDBS

[a] All input and output filenames in this table have an .xml extension that has been removed for clarity.
[b] For example—Northwind.xml.

Table 6-8. Transforming Manually Entered ORM Description to Code Generation Input, continued

STEP	PROCESS	XSLT TEMPLATE	INPUT FILE(S)	OUTPUT FILE
3	XSLTProcess	ORMObjectOne	ORM2WithDBS	ORMObjectOutput
4	XSLTProcess	ORMObjectTwo	ORMObjectOutput	ORMObjectOutputTwo
5	XSLTProcess	ORMBuildOne	ORMObjectOutputTwo	ORMBuildOutput
6	XSLTProcess	ORMBuildTwo	ORMBuildOutput	ORMBuildOutput2
7	XSLTProcess	IncorporateInfo	ORMBuildOutput2	[SPInputMetadata]
8	XSLTProcess	ORMFilterForGen	[SPInputMetadata]	[GenInputMetadata]

The output of each step feeds the next step. The business object metadata creation includes an extra step that removes information not needed in business object templates. This transformation makes the input used by the stored procedure and business object templates significantly different. The following sections discuss the purpose of each of these transformations.

NOTE The names bracketed in Table 6-8 are used outside the ORM transformation, so you might want to use alternate names. These names are defined via abstract filenames (discussed in Chapter 5) in the download script.

There are a total of six XSLT transforms. Two of these primarily create metadata about objects, two primarily create metadata for building stored procedures, one incorporates the *Element*Info elements discussed earlier (such as `PropertyInfo` and `ObjectInfo`), and one removes irrelevant database information to limit the information visible to programmers building middle tier templates. Multiple transforms are necessary because later transforms depend on information built by earlier templates.

NOTE You could alter the process to remove extraneous information for the stored procedure template developers as well.

Step 1: Creating Metadata

The first step to the ORM transform is retrieving fresh metadata from the database. Chapter 2 discusses retrieving database metadata. After you build stored procedures, you'll need to rerun this and step 2 to incorporate the new stored procedure information.

Step 2: Merging Metadata

You've manually created an ORM XML file, and you've retrieved database metadata in step 1. You combine these two files to create a monolithic file for further processing. Combining these files lets you easily use database information for fleshing out the full metadata file from the skeletal information provided by the manually created ORM file. This process uses the simple intimate merge described in Chapter 3.

Introducing Base Pattern Used in Steps 3–8

The same basic pattern is used for the four XSLT stylesheets ORMObjectOutput.xslt, ORMObjectOutputTwo.xslt, ORMBuildOutput.xslt, and ORMBuildOutputTwo.xslt. This pattern is to pass through the XML file copying all elements that aren't being altered. For those elements being altered within the current template, existing child elements are copied if they exist, and additional elements are inferred if appropriate. This process continues down the hierarchy of the element and child elements altered in the current template. I'll walk through portions of the ORMObjectOutput.xslt stylesheet.

Generically Copying XML Details

You can use XSLT recursion to copy XML without specifying excessive details about the XML. This is important to reduce the dependency of the XSLT on the specific XML details. The following XSLT code recursively copies XML elements and their attributes:

```
<xsl:template match="/ | @* | node()">
   <!-  Other stuff ->
   <xsl:copy>
```

```
        <xsl:apply-templates select="@*"/>
        <!-  Other attribute stuff ->
        <xsl:apply-templates select="node()"/>
        <!-  Other element stuff ->
     </xsl:copy>
     <!-  Other stuff ->
  </xsl:template>
```

The `xsl:apply-templates` directive recursively processes the entire template in the context of the root, child node (`node()`), or attribute (`@*`). You can put additional processing in place of the comments (`<!- OtherStuff ->`).

NOTE You must add attributes prior to adding text or child elements. Any attributes you add after text or child elements won't appear in your output.

The entry-level template of these stylesheets follows a pattern such as the following one from ORMObjectOne.xslt. This copies all elements and attributes from the input file except the `Assembly` element and its children and runs an additional template for the `MappingRoot` element:

```
<xsl:template match="/ | @* | node()">
   <xsl:choose>
      <xsl:when test="name()='orm:Assembly'" /> <!- Handled Below ->
      <xsl:otherwise>
         <xsl:copy>
            <xsl:apply-templates select="@*"  />
            <xsl:apply-templates select="node()"/>
            <xsl:if test="name()='orm:MappingRoot'">
               <xsl:call-template name="Assembly" />
            </xsl:if>
         </xsl:copy>
      </xsl:otherwise>
   </xsl:choose>
</xsl:template>
```

Templates processing specific portions of the input file call a template, such as the named template `Assembly`, to supply default attribute values and then copy any attributes in the input file. Because attributes overwrite previously created attributes, this gives the last data precedence which is generally data you entered in the manual ORM file:

```
<xsl:template name="Assembly">
   <xsl:for-each select="//orm:Assembly">
      <xsl:copy>
         <xsl:call-template name="AssemblyAttributes" />
         <xsl:apply-templates select="@*" />
         <xsl:apply-templates select="node()[name()!='orm:Objects']"/>
         <xsl:call-template name="Objects" />
      </xsl:copy>
   </xsl:for-each>
</xsl:template>
```

The object templates follow a slightly more complex pattern. Portions of this template were snipped to better illustrate the structure of the pattern. The `runall` variable is `true` if there are no `Object` elements in the input or the `AllObjects` element is present. Later in the template, this variable determines whether the transformation infers additional objects—beyond those defined in the input ORM file. The template then loops through existing objects copying their data:

```
<xsl:template name="Objects">
   <xsl:variable name="runall">
      <xsl:if test="count(.//orm:Object)=0 or .//orm:AllObjects" >true</xsl:if>
   </xsl:variable>
   <xsl:for-each select=".//orm:Object">
      <xsl:variable name="dsname" select=
               “ancestor-or-self::*[@MapDataStructure][1]/@MapDataStructure"/>
      <xsl:variable name="ds" select="//dbs:DataStructure[@Name=$dsname]"/_
      <xsl:variable name="table">
<<snipped for clarity >>
```

I’ve snipped a portion of the template here because determining the table is complex and not relevant to the template’s pattern. Only one table is expected, but the `ObjectAttributes` template uses an `xsl:for-each` directive because it needs to run in the context of that table’s element (from the database metadata portion of the file). The template then overwrites existing values with any defined in the input file (ultimately ones defined in the ORM file) and calls the `ObjectContents` template to create child elements. Finally, the template copies

any elements not already handled. The common pattern in the following fragment is the use of the `ObjectAttributes` and `ObjectContents` templates:

```
   </xsl:variable>
   <xsl:for-each select="$table" >
      <xsl:call-template name="ObjectAttributes"/>
   </xsl:for-each>
   <xsl:apply-templates select="@*" />
   <xsl:call-template name="ObjectContents" >
      <xsl:with-param name="table" select="$table"/>
      <xsl:with-param name="object" select="." />
   </xsl:call-template>
   <xsl:apply-templates select="node()[
                name()!='orm:ChildCollection' and
                name()!='orm:ParentObject' ]" />
</xsl:for-each>
```

If additional objects should be inferred, the `runall` variable will be `true`. If it's `true`, the template runs through the appropriate set of elements in the database metadata portion of the input file. In this case, objects are inferred from tables, so the template loops through the database's tables. For each table, the template determines if an `Object` element already exists and create it if needed:

```
<xsl:if test="$runall='true'">
   <xsl:variable name="dsname" select=
            "ancestor-or-self::*[@MapDataStructure_[1]/@MapDataStructure"/>
   <xsl:for-each select="//dbs:DataStructure[@Name=$dsname]//dbs:Table">
      <xsl:variable name="tablename" select="@Name" />
      <xsl:variable name="alreadyexists">
         <xsl:for-each select="//orm:Object[@Name=$tablename]">
            <xsl:if
            test="ancestor-or-self::*[@MapDataStructure][1]/@MapDataStructure
             =$dsname">true</xsl:if>
         </xsl:for-each>
      </xsl:variable>
```

CAUTION Sometimes it's difficult to perform a direct XPath test. In this case, I want to compare the nearest parent XML method that has a `MapDataStructure` attribute. To accomplish this, I loop through a set of candidates testing each one via a conditional. This technique works in defining variables, which you can use in conditional directives to manage the flow of your template.

If the `Object` element doesn't already exist, the template adds it in the `AddObject` template:

```
            <xsl:choose>
               <xsl:when test="starts-with($alreadyexists,'true')"/>
               <xsl:otherwise>
                  <xsl:call-template name="AddObject">
                     <xsl:with-param name="name" select="$tablename"/>
                  </xsl:call-template>
               </xsl:otherwise>
            </xsl:choose>
         </xsl:for-each>
      </xsl:if>
   </xsl:template>
```

The `AddObject` template doesn't need to copy existing child elements and attributes because it's creating a new `Object` element. It does need to call the same templates called earlier for existing elements: `ObjectAttributes` and `ObjectContents`:

```
<xsl:template name="AddObject" >
   <xsl:param name="name"/>
   <xsl:param name="childof"/>
   <xsl:param name="inherits"/>
   <xsl:element name="orm:Object">
      <xsl:call-template name="ObjectAttributes">
         <xsl:with-param name="name" select="net:GetSingular($name)" />
         <xsl:with-param name="childof" select="$childof" />
         <xsl:with-param name="inherits" select="$inherits" />
      </xsl:call-template>
      <xsl:call-template name="ObjectContents" >
         <xsl:with-param name="table" select="."/>
         <xsl:with-param name="object" select="/xxx" />
         <xsl:with-param name="allproperties" select="'true'" />
      </xsl:call-template>
      <xsl:element name="orm:AllProperties"/>
   </xsl:element>
</xsl:template>
```

NOTE The `ObjectAttributes` and `ObjectContents` templates were previously called with different parameters. When parameters are left off, default values are used.

This pattern has some important implications. Primarily, it isolates the locations where specific activities occur. For example, if you need to add an inferred `Assembly` element attribute, you'll make this change in the `AssemblyAttribute` template. Understanding this pattern will make it easier for you modify the process to better fit your own use.

NOTE The attributes for `Object` elements have to be added in two pieces, so there's an `ObjectAttribute` template and an `ObjectAttributesUpdate` template.

Step 3: Processing ORMObjectOne.xslt

This template adds almost all of the `Object` element attributes and child elements. It does most of its work by calling templates in ORMSupport2.xslt because several stylesheets reuse these support templates.

`Object` elements describe objects in your output, and this stylesheet adds child elements for the `UseSPSet`, `ChildCollection`, and `ParentObject` elements.

NOTE Additional attributes are added to the `Object` element using the `ObjectAttributeUpdate` template called from the `ORMBuildOne` stylesheet.

Step 4: Processing ORMObjectTwo.xslt

Once most of the information for the `Object` elements are built, you've got to check that all child collections have corresponding `Object` elements and add them if needed. That's the job of the ORMObjectTwo.xslt stylesheet.

Step 5: Processing ORMBuildOne.xslt

The `ORMBuildOne` template creates most of the child elements and attributes for the portion of the metadata used to create stored procedures. This portion of the metadata has an `SPSet` in the `Build` section of the metadata for each entity, such as project or invoice.

This template also adds some extra attributes to the `Object` elements. Some aspects, such as Booleans indicating whether the object is a child or root, can't be easily determined until you've created all objects, including child objects.

If Your Attributes Are Wrong...

One problem you can encounter in debugging transforms that create XML documents is attributes that don't contain the values you expect. Attributes can be a bit tricky to debug because later `xsl:attribute` directives overwrite any earlier values. This overwriting can occur within the stylesheet that creates the attribute—or in a later stylesheet if you're using a multistep transformation. In this case, you'll first want to observe your input and output of each transformation to determine where the problem first occurs.

TIP Although it's technically feasible to overwrite the input document with the output of a transform, this makes it difficult to pinpoint the source of a problem. Instead, create a temp directory to contain these interim files that you rarely expect to access.

If the specific stylesheet where the problem originates defines the attribute, the first step in isolating the problem is to create a new dummy attribute with a name you're sure you aren't using elsewhere. I often use the names of the Weasely children from the Harry Potter books. This gives me several names I can easily search for and clean up later. Repeat the assignment you're performing for the problem attribute so you assign an identical value to this new attribute. Examine this value in your output. If the values both contain the incorrect value, the problem is in how you create the value. If, instead, the original and dummy attributes have different values, something later in your processing overwrites the correct value. If the attribute is missing, it's likely that a later template simply fails to copy it.

Be careful when outputting debugging information as text or elements rather than as attributes. Because any `xsl:attribute` directives that appear after an `xsl:element` directive or after any text output are ignored, you may significantly affect the output of your template if you output text or elements. This may even "fix" your problem, but don't use this as a permanent fix because it disallows later changes to attributes. Instead, track down the extra `xsl:attribute` directives and fix them.

I ran into this problem in the ORM transfer. Usually I copy attributes by applying a general template such as the one shown in the "Generically Copying XML Details" sidebar. For example:

```
<xsl:call-template name="PropertyAttributes"/>
   <xsl:apply-templates select="@*"/>
   <xsl:apply-templates select="node()"/>
```

The problem I encountered was trying to copy attributes when I wanted to override attributes of the original element. Even empty strings were overwriting the inferred defaults. To solve this, I changed the template to this:

```
<xsl:call-template name="PropertyAttributes"/>
   <xsl:for-each select="@*">
      <xsl:if test="string-length(.)!=0">
         <xsl:copy/>
      </xsl:if>
   </xsl:for-each>
   <xsl:apply-templates select="node()"/>
```

This simple technique allowed control over what attribute values the output XML file contained.

NOTE Overwriting *elements* doesn't present the same problem with the in overwriting attributes. An element is a container for attributes and child elements. If the same element exists in either or both input files, the element exists in the output. If it exists in both input files, the resulting element will contain all attributes and child elements that are included in either input file. Thus, the element may be expanded on by the merge, but information is only lost when the same attribute appears in both files.

Step 6: Processing ORMBuildTwo.xslt

This template ties up several final details. It does the following:

- Adds parameters to the `RunSP` elements of the stored procedure section
- Adds `RunSP` elements to the `Object` elements in the business object generation section
- Adds properties to the `Object` elements

These tasks are dependent on the output of earlier templates. You can't easily determine stored procedure parameters until you've got the full list. You can't add the `RunSP` elements to the `Object` elements until you have them in the stored procedure section, and you can't add the properties until you know the output of the stored procedures.

Step 7: Processing IncorporateInfo.xslt

In the section "Overriding Inferred Details for Customization" earlier in this chapter, you saw how the informational elements (such as `ObjectInfo`) parallel the main elements and offer the capability to define any attribute without affecting the structural inference, such as `ObjectInfo` overriding `Object` element details. It's an important capability because it allows you to specify attributes without altering the overall business object hierarchy. However, incorporating this information is a bit tricky and is the task performed by `IncorporateInfo`. Your script should run `IncorporateInfo` after the inference transformations and just before the code generation steps because values in the informational elements should override attributes from any other source.

The code to copy the attributes is trivial:

```
<xsl:template match="*" mode="SetAttributes">
   <xsl:apply-templates select="./@*" mode="SetRootAttributes"/>
</xsl:template>

<xsl:template match="@*" mode="SetRootAttributes">
   <xsl:copy/>
</xsl:template>
```

This code relies on the ability to overwrite attribute values. Thus, the previous code overwrites any existing attribute values in the current output context with the attributes in the current document context. The output context in this case would be the main element such as `Object` or `Assembly`, and the current document context would be the corresponding informational element, such as `ObjectInfo` or `AssemblyInfo`. The trick is being at the right document context for the current output context.

To accomplish this, the IncorporateInfo.xslt outputs the main element attributes and then shifts the context to the corresponding informational element. With this context, it outputs the attributes from the informational element, overwriting any matching attributes in the existing element. Unfortunately, the XSLT for the context shift has to be explicit so results in a rather long and ugly stylesheet.

Step 8: Processing ORMFilterForGen.xslt

The last step in preparing metadata to create business objects is to strip extraneous information. The business object programmer shouldn't need direct access to the database or stored procedure building sections of the metadata. Removing this data makes things simpler for the programmer working on business objects.

Variables Containing Sets of Nodes

When working with complex XSLT, you'll frequently need to work with groups of nodes and need to assign these nodes to a variable. You can take three approaches as follows:

- Using a simple XPath expression
- Manually building a set of elements and converting them using the `msxsl:nodeset` extension function
- Using XSLT keys

If you can build an XPath expression that selects the nodes, the easiest approach is similar to this:

```
<xsl:variable name="test" select="//orm:Object[@Name=$name]//orm:ChildCollection
```

Although you can use conditionals in the XPath expression, you can't extend this to conditionally use different XPath expressions.

The second approach is to use the Microsoft extension node-set function. This converts a set of elements into a node-set. The key drawback of this approach is that you have copies of the elements, not references to them. Although you can do a deep copy and have access to descendant elements, you can't easily access ancestor elements.

The third approach requires use of an advanced XSLT feature called *keys*. You can see this technique in the ORMBuildTwo.xslt stylesheet. Keys let you associate one or more keys with selected elements in the input document. You can associate any number of keys with an element and any number of elements with each key value.

The trick to using keys for creating conditional groups of elements is to segment string keys. For example, in ORMBuildTwo.xslt I have two parallel node-sets defined via keys and a later conditional selects between them. So the start of the keys indicates the group—`In Object` or `In Stored Procedure`. I used two later parts of the string key to specify the `Assembly` and `Object` name or the `DataStructure` and `StoredProcedure` element names. Later in the template, conditionals create a variable that contains the correct key string, rather than attempting to directly create the set of elements. You won't need this trick often, but it does allow you to create a set of nodes from even complex criteria.

NOTE One approach that doesn't currently work well is returning a node-set from callbacks into .NET code. I hope this problems will be fixed in the future.

Using the Output

SPInputMetadata.xml is used in Chapter 7 to create stored procedures. GEnInputMetadata.xml is used in Chapter 8 to produce business objects, in Chapter 9 to produce Windows.Forms code, and in Chapter 10 to produce Web Form code. In all of these cases, nested scripts are used to call the actual Chapter 6 script.

Summary

ORM maps individual database tables and fields to business objects and properties. ORM isn't required for code generation, but using it frees you from the restriction that your business objects map directly to your database structure throughout the life of the project. The benefits of the ORM approach I take are as follows:

- Provide a simple model using only a few lines when the mapping is direct.
- Maintain automatic flow-through of database structure changes whenever possible.
- Produce metadata that's friendly to code generation.
- Allow you to generate even relatively complex stored procedures.
- Control the data visible to the template programmer.
- Allow customization of any element.

The ORM process combines the business object structure defined in the ORM XML file with database details contained in the database metadata file (described in Chapter 2) to create metadata tuned for code generation. The process of transforming the ORM and database metadata into metadata designed for code generation is complex and consists of multiple steps. You can generally ignore the complexity of this process when doing code generation.

Because stored procedure generation and business object generation has different needs, it's simplest to create two sets of metadata—one tuned for each generation type.

EXHEART

CHAPTER 7

Generating Stored Procedures

Stored procedures have the highest payback for code generation.

THE EASIEST CODE TO GENERATE is stored procedure scripts. Stored procedures tend to closely follow a pattern, have a high payback for code generation, and give rapid feedback. It's easy to execute the scripts that compile stored procedures and place them in your database right after you generate the scripts. Stored procedure scripts also tend to be relatively small and simple, making this the ideal place to start with code generation templates.

I'll use XSLT templates because I think the token replacement approach of XSL Transformations (XSLT) makes it easier to see the underlying processes. If you'd like to use brute-force generation, you'll be able to translate these templates into .NET code. You can't use the CodeDOM to output stored procedure scripts, however.[1]

Exploring Stored Procedure Generation

In this chapter, I'll show the process of generating stored procedures for Microsoft SQL Server. If you use another database backend, you'll need to modify this procedure and the templates to fit your backend. Most modern relational databases work in a similar manner, so the translation should be easy.

SQL Server stored procedures are written in Transact SQL (T-SQL), which is Microsoft's variation of Structured Query Language (SQL). If you use another backend, such as Oracle, you might find some variations in how SQL is used. The "Additional Reading" section at the end of this chapter provides several references for T-SQL.

---

1. Okay, if you worked hard enough and use snippets, you might be able to create stored procedure templates via the CodeDOM, but it'd be a silly approach.

Designing Your Database for Easier Code Generation

This sidebar gives you tips for making code generation easier. You don't have to follow this specific set guidelines because you can make code generation very flexible. For example, the sample database used in this book doesn't follow some of these guidelines (I applied code generation to an existing database). However, applying these guidelines will make code generation easier to work with when you can control the following factors:

- Use surrogate primary keys. This means your primary keys don't carry any meaning and exist merely for identification and maintaining relationships. You can still do code generation if you don't use surrogate keys.
- Use single column Global Unique Identifier (GUID) or identity (auto-increment) primary keys. GUIDs are a bit easier during code generation and a bit harder to manage within your database.
- Select one primary key strategy for all tables. An exception to this rule is that you probably want to use identity columns for simple lookup tables, even if you're using GUIDs for other tables. If you base a combination strategy (such as using GUIDs for tables you're replicating) on any other criteria, it's likely to change during the course of your project.
- Place the primary key as the first field in the table.
- Name all tables with the singular form of the entity, or name all tables based on the plural and anticipate some changes from the book's templates.
- Don't allow a singular and plural form of the same noun as table names in the same problem domain; for example, don't have a Project and a Projects table.
- Don't include the parent table name in child table names unless the child table exists to support a many-to-many relationship, in which case name it for its two parent tables. Have a consistent strategy for which appears first, such as alphabetical order.
- Name primary keys with the name of the table followed by `ID`.
- Name foreign keys with precisely the same name as the primary key they reference. If this results in redundant column names, such as a self-join or two fields referring to the same foreign key table, prefix the foreign key name with a word describing its purpose such as `ManagerEmployeeID` and `MentorEmployeeID`.
- Define relations in the database.
- Define simple constraints in the database.

Outlining the Process

Code generation templates for stored procedures create SQL script files. Script files are normal ASCII files that exist outside your database and contain instructions for SQL Server. In the case of stored procedure scripts, the script instructs SQL Server to create the stored procedure. Because you can't have two SQL Server objects with the same name, the script drops an existing stored procedure (if it exists) before creating the new stored procedure. The generated script also grants appropriate privileges to the newly created stored procedure.

NOTE *Dropping* something from SQL Server is synonymous with *deleting* it from the database. The different term is used because you *delete* rows (*rows* is synonymous with *records*) and you *drop* objects. *Objects* in SQL Server are any table, view, stored procedure, function, or user-defined type.

Because the script is outside the database, there are two steps to creating the stored procedure. You need to create the stored procedure script, and then you need to execute the script. You could use Query Analyzer to execute the scripts, but manually running each script is tedious and prone to error. To avoid this manual step, use the `RunSQL` process of the code generation harness to apply scripts to your SQL Server.

NOTE Although it's rarely an issue, people often set `QUOTED_IDENTIFIER ON` and `ANSI_NULLS ON` before running the stored procedure and reset them at the end.

If you set the `OutputRules` child element's `HashOutput` attribute of the directive you use to generate your stored procedure script to `true`, code generation will add a hash code at the top of the script. These funny characters and the markers that offset them from the remainder of your file let code generation later test whether a programmer has edited the script file. If the file was edited, the marker stored in the file won't match a newly calculated hash code. Placing this code in the script and setting the `OutputGenType` to `Always` protects your script file. This is an example of a hash code with its surrounding markers:

```
/* ***^^^^^^***||| %%%###%%%0c8c516a3f51b2dfa9518a92845f1b23%%%###%%% ***^^^^^^***|||/
```

It's also possible for a programmer to edit the stored procedure within SQL Server. The programmer can do this by running another stored procedure script,

which is easy to do through the Query Analyzer, Enterprise Explorer tools of SQL Server, or the Server Explorer of Visual Studio. To protect against changes made to the stored procedure within the database, the process run by the `RunSQL` harness directive to execute stored procedures also supports the `OutputRules` child element. This means you can specify the hash code and overwrite rules for both the stored procedure script and the stored procedure within the database.

NOTE You probably also want to keep these scripts under source control.

Building Stored Procedure Metadata

Chapter 6 shows how to develop metadata from the perspective of your business objects. A transformation process fleshes out a full description of your business objects, the information needed to build the stored procedures from the database schema, and any indirect mappings you create. The result of this morphing process is the SPInputMetadata.xml[2] metadata file that includes a `Build` section with the information for building your stored procedures. You'll use this to build stored procedures that are a critical piece in your Object Relational Mapping (ORM) strategy because they map the database information into a structure usable by your business objects.

The `Build` element of the SPInputMetadata.xml file includes a block for each logical entity such as Projects. These sections are called `SPSet` elements. The `SPSet` for the entity provides information for the five key stored procedures: create, retrieve, update, delete, and set select. Listing 7-1 shows a portion of the `Build` element of the SPInputMetadata.xml that defines a single set of stored procedures. The mapping in Listing 7-1 is indirect because the retrieve stored procedure returns two recordsets, one of which includes a join. Chapter 6 discusses some of the elements within this section. In this chapter, you'll see how to use this input to create stored procedures for your application.

NOTE When you're working with direct mappings, the ORM transformation process creates the `Build` section for you. Even when you're working with indirect mappings, much of the information in Listing 7-1 is created for you by the ORM transformation process. See the section "Generating Joined Tables" in Chapter 6 for information entered to produce this `Build` section.

2. You can call this file anything you like in your script. I'm using the name from the book's script for convenience.

Listing 7-1. The `SPSet` *with Information for Creating a Set of Stored Procedures for a Single Logical Entity*

```
<orm:SPSet Name="Projects" Generate="CRUDS" ObjectName="Project"
                  TableName="Projects">
    <orm:RetrieveParam Name="ID" Column="ID" SQLType="UniqueIdentifier"
               NETType="System.Guid" IsPrimaryKey="true" FKName="ProjectID" />
    <orm:WhereClause Clause="@ID = [ID]" />
    <orm:SetSelect TableName="Projects">
       <orm:SetSelectColumn IsPrimaryKey="true" Name="ID" Column="ID"
               SQLType="UniqueIdentifier" NETType="System.Guid" />
       <orm:SeSelectColumn Name="Name" Column="Name" SQLType="VarChar"
               NETType="System.String" IsPrimaryKey="false" MaxLength="50" />
    </orm:SetSelect>
    <orm:Privilege Grantee="public" Rights="RS">
    <orm:Privilege Grantee="Admin" Rights="CRUDS" />
    <orm:Privilege Grantee="Manager" Rights="RUS" />
    <orm:BuildRecordset Name="RecSet">
       <orm:WhereClause Clause="@ID = [ID]" />
       <orm:BuildTable Name="Projects">
          <orm:BuildColumn Name="ID" Column="ID" SQLType="UniqueIdentifier"
               NETType="System.Guid" IsPrimaryKey="true" />
          <!– Repeat for other columns –>
          <orm:AllColumns />
       </orm:BuildTable>
    </orm:BuildRecordset>
    <orm:BuildRecordset Name="RecSet1">
       <orm:WhereClause Clause="@ID = [Assignments].[ProjectID]" />
       <orm:BuildTable Name="Assignments">
          <orm:BuildColumn Name="ProjectID" Column="ProjectID"
               SQLType="UniqueIdentifier" NETType="System.Guid"
               IsPrimaryKey="true" />
          <!– Repeat for other columns –>
          <orm:AllColumns />
       </orm:BuildTable>
       <orm:BuildTable Name="Roles">
          <orm:BuildColumn Name="ID" Column="ID" SQLType="Int"
               NETType="System.Int32" IsPrimaryKey="true" />
          <orm:BuildColumn Name="Name" Column="Name" SQLType="VarChar"
               NETType="System.String" IsPrimaryKey="false" MaxLength="50" />
          <orm:AllColumns />
       </orm:BuildTable>
```

```
        <orm:Join Type="Inner">
            <orm:Left JoinTable="Assignments" />
            <orm:Right JoinTable="Roles" />
            <orm:JoinOn Clause="[Assignments].[Role]=[Roles].[ID]" />
        </orm:Join>
    </orm:BuildRecordset>
    <orm:RunSP Name="wx_appProject_Insert" Task="Create">
        <orm: RunSPParam Name="ID" Column="" SQLType="UniqueIdentifier"
                NETType="System.Guid" IsPrimaryKey="true" TableName="Projects" />
        <!— Repeat for other parameters —>
    </orm:RunSP>
    <orm:RunSP Name="wx_appProject_SelectOne" Task="Retrieve">
        <orm:RunSPParam Name="ID" Column="" SQLType="UniqueIdentifier"
                NETType="System.Guid" IsPrimaryKey="true" />
    </orm:RunSP>
    <orm:RunSP Name="wx_appProject_Update" Task="Update">
        <orm:RunSPParam Name="ID" Column="" SQLType="UniqueIdentifier"
                NETType="System.Guid" IsPrimaryKey="true" TableName="Projects" />
        <!— Repeat for other parameters —>
    </orm:RunSP>
    <orm:RunSP Name="wx_appProject_Delete" Task="Delete">
        <orm:RunSPParam Name="ID" Column="" SQLType="UniqueIdentifier"
                NETType="System.Guid" IsPrimaryKey="true" />
    </orm:RunSP>
    <orm:RunSP Name="wx_appProject_SelectAll" Task="SetSelect" />
</orm:SPSet>
```

Generating Retrieve Stored Procedures

The retrieve stored procedure selects information for a single logical business object. This generally means retrieving a single base-level record and perhaps retrieving its child and grandchild records. The stored procedure used to retrieve data illustrates many of the techniques used in stored procedure generation. I'll walk through this stylesheet in detail and then briefly discuss the other four stylesheets.

Looking at the Retrieve Target

Listing 7-2 shows the script for creating a retrieve stored procedure. This stored procedure sets the `QUOTED_IDENTIFIER` and `ANSI_NULLS` options and drops the existing stored procedure if it exists. The stored procedure returns two recordsets.

One contains a record from the Projects file. The other containing the corresponding child records forms the Assignments table joined to the Roles table.

NOTE Brackets are used around table and column names because I don't know whether the names in the database you'll run against are legal names without being escaped with parentheses. Specifically, I don't know if you've used spaces or keywords in your table and column naming.

After creating the stored procedure, the script resets the environment values and sets privileges on the stored procedure.

Listing 7-2. Script for Creating a Retrieve Stored Procedure

```
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_NULLS ON
GO

if exists (
       select * from dbo.sysobjects
       where id = object_id(N'[dbo].[wx_appProject_SelectOne]')
         and OBJECTPROPERTY(id, N'IsProcedure') = 1)
drop procedure [dbo].[wx_appProject_SelectOne]
GO

CREATE PROCEDURE wx_appProject_SelectOne
(
@ID UniqueIdentifier
)
AS

   SELECT
      [Projects].[ID],
      [Projects].[Name],
      [Projects].[Started],
      [Projects].[Ended],
      [Projects].[Description],
      [Projects].[DepartmentID]
   FROM [Projects]
   WHERE @ID = [ID]
```

```
   SELECT
      [Assignments].[ProjectID],
      [Assignments].[ResourceID],
      [Assignments].[Assigned],
      [Assignments].[Role],
      [Roles].[ID],
      [Roles].[Name]
   FROM [Assignments] Inner JOIN [Roles]
      ON [Assignments].[Role]=[Roles].[ID]
   WHERE @ID = [Assignments].[ProjectID]

   RETURN

GO
SET QUOTED_IDENTIFIER OFF
GO
SET ANSI_NULLS ON
GO

GRANT EXECUTE ON wx_appProject_SelectOne TO public
GRANT EXECUTE ON wx_appProject_SelectOne TO Admin
GRANT EXECUTE ON wx_appProject_SelectOne TO Manager
```

Examining the Retrieve Stored Procedure Stylesheet

The XSLT stylesheet to create retrieve stored procedures opens with the standard header and root `xsl:stylesheet` element. The root element includes two namespaces—`dbs` and `orm`—used in the SPInput.xml input metadata:

```
<xsl:stylesheet version="1.0"
         xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
         xmlns:xs="http://www.w3.org/2001/XMLSchema"
         xmlns:dbs="http://kadgen/DatabaseStructure"
         xmlns:orm="http://kadgen.com/KADORM.xsd" >
<xsl:import href="SPSupport.xslt"/>
<xsl:output method="text" encoding="UTF-8" indent="yes"/>
<xsl:preserve-space elements="*" />
```

NOTE Your XSLT stylesheets must contain namespaces corresponding to any namespaces used in your XML input files, as well as any namespaces used in your XSLT stylesheet.

The stylesheet uses a supporting utility stylesheet named SPSupport.xslt that contains important utility templates that I'll discuss in the section "Supporting Stored Procedure Templates." The `xsl:preserve-space` element helps you manage whitespace, as discussed in Appendix A and Chapter 3.

NOTE The stylesheets I use for code generation use `text` as the `xsl:output` type. You can get some variations in behavior by using other output modes, but they'll cause additional difficulties with whitespace management. I suggest you stick with standard output types—`xml` for all variants of XML output except HTIML, `html` for HTML, and `text` for all other types of files.

The stylesheet includes the four stylesheet parameters generally used with the code generation templates included in this book. Stylesheet parameters and variables are available from any templates in the stylesheet or in any imported or included stylesheet. You create stylesheet variables by declaring them at the top of the stylesheet, outside any templates:

```
<xsl:param name="Name"/>
<xsl:param name="filename"/>
<xsl:param name="database"/>
<xsl:param name="gendatetime"/>

<xsl:variable name="spname" >
  <xsl:call-template name="StripPathAndExtension">
     <xsl:with-param name="fname" select="$filename" />
  </xsl:call-template>
</xsl:variable>
```

You can use complex criteria to define stylesheet variables—as I did with the `spname` variable. The `spname` variable provides the simple version of the stored procedure filename with the extension and path removed. By declaring this variable at the top of the file, you can access the `spname` variable from any template in the main stylesheet or SPSuport.xslt. SPSupport.xslt contains the `StripPathAndExtension` template, which is quite similar to the `StripPath` recursive template discussed in Chapter 3.

Chapter 3 discusses the value of an entry-level template. The purpose of this entry-level template is to establish context and initiate processing. In this case, it calls a high-level processing template for the `RunSP` element that defines the retrieve stored procedure being created:

```
<xsl:template match="/">
   <xsl:apply-templates
            select="//orm:SPSet[@Name=$Name]/orm:RunSP[@Task='Retrieve']"
            mode="RunSP" />
</xsl:template>
```

NOTE I anticipate the `orm:SPSet` element being a child of an `orm:Build` element, but the stylesheet doesn't require this layout.

The high-level processing template calls the `OpenSP` template located in `Support.xslt`. This template outputs the standard stored procedure opening, which includes dropping any existing stored procedure with the same name. It also includes the stored procedure declaration. The `OpenSP` template outputs the first 12 lines of the stored procedure script shown in Listing 7-2. I'll discuss the `OpenSP` template and other utility templates in the section "Supporting Stored Procedure Templates." You can call `OpenSP` with this:

```
<xsl:template match="orm:RunSP" mode="RunSP">
   <xsl:call-template name="OpenSP" />
```

Like other stored procedure types, the `RunSP` element contains `RunSPParam` child elements that define the parameters for the stored procedure. Outputting the parameter list involves looping through the `RunSPParam` child elements. The effect of `xsl:apply-templates` is to combine an `xsl:for-each loop` with an `xsl:call-template`, which outputs a parameter declaration for each parameter:

```
(<xsl:apply-templates select="orm:RunSPParam" mode="SPParameters"/>
)
AS
```

NOTE Literals such as the parentheses and the word `AS` are output by XSLT exactly as written.

Once you've output the opening of the stored procedure and the parameters, it's time to create the `SELECT` statements that output SQL recordsets. A retrieve stored procedure may output many recordsets, one for the base object and one for each child, grandchild, and so on collection. If multiple recordsets are needed, each will have a `BuildRecordset` element as a child of the `SPSet` element (see Listing 7-1). The template needs to create a `SELECT` statement for each `BuildRecordset` element. Each of these `SELECT` statements may include a column list, tables, and join criteria. I use the `ON` clause of the join definition for the join criteria, so the `WHERE` clause just needs to handle the restrictive criteria. In Listing 7-2, the `ON` clause manages the `Assignments.Role=Roles.ID` relation, so the `WHERE` clause just needs to restrict the output based on the `ProjectID`.

To provide this output, the template loops through the `BuildRecordset` elements. For each one, it outputs the `SELECT` and `WHERE` literals that form the framework of the familiar statement. Interleaved with this are calls to utility templates that supply the key information such as the column list and join list. An `xsl:value-of` directive outputs the `WhereClause` that's stored in the input metadata file. The `NewLine` templates output a few blank lines, so multiple `SELECT` statements in the generated output are visually distinct:

```
<xsl:for-each select="orm:BuildRecordset">
   SELECT <xsl:apply-templates select=".//orm:BuildColumn"
                mode="ColumnSPList" />
   <xsl:call-template name="ORMJoinList" />
   WHERE <xsl:value-of  select="orm:WhereClause/@Clause"/>
   <xsl:call-template name="NewLine" />
   <xsl:call-template name="NewLine" />
</xsl:for-each>
```

TIP It's easier for SQL Server to optimize your stored procedures if you explicitly list columns rather than using the asterisk (*) to indicate all columns.

Once you've output the recordsets, the only step remaining is to close up the stored procedure. The `RETURN` statement ends the stored procedure, and the `CloseSP` template outputs the housekeeping code for the environment values and assigns permissions:

```
   RETURN

<xsl:call-template name="CloseSP">
   <xsl:with-param name="privileges"
```

```
            select="ancestor::orm:SPSet//orm:Privilege[contains(@Rights,'R')]" />
</xsl:call-template>

</xsl:template>

</xsl:stylesheet>
```

Because the current context is the `RunSP` element and privileges are defined on the `SPSet` element, the XPath to retrieve the privileges finds the ancestor `SPSet` and then checks for retrieve privileges via the `R` in the `Rights` attribute.

TIP Judicious use of axes, such as `ancestor::<element name>` rather than `..` (two periods) for the parent, will make your templates more flexible to structural changes. For example, the `orm:Privilege` may or may not be included in an `orm: Privileges` collection. If you use the parent reference, you count on the value being in the immediate parent. If you use the ancestor axes, the value can be in the parent or any other ancestor. The closest ancestor is used.

The retrieve stored procedure illustrates how you build the structure of your stored procedure, and the next section shows how the utility templates work to complete the stylesheet.

Supporting Stored Procedure Templates

You'll use supporting or utility stored procedures to perform a variety of tasks. They provide code reuse and make it much easier to read your main stylesheets—the same reasons you use subroutines in other languages. The only significant difference is that you have two distinct flavors of XSLT templates. XSLT named templates, called using the `xsl:call-template` directive, are similar to subroutines in other languages. Match templates are activated using the `xsl:apply-templates` directive and run for each member of a specified node set. Match templates effectively merge a loop and subroutine call. Appendix A discusses match templates with a broader perspective, but at this point the simplistic view of merging a loop and a template is adequate.

Opening the Stored Procedure with the OpenSP Template

The `OpenSP` utility template outputs stored procedure code that turns quoted identifiers and ANSI nulls on, and it drops any stored procedure that already

exists with the same name. Finally, it outputs the name of the stored procedure. The `spname` variable used in the `OpenSP` template is a stylesheet variable declared in the calling stylesheet and discussed in the section "Examining the Retrieve Stored Procedure Template." The code to set the environment values, drop an existing stored procedure, and create the new stored procedure is as follows:

```
<xsl:template name="OpenSP">
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_NULLS ON
GO

if exists (
       select * from dbo.sysobjects
       where id = object_id(N'[dbo].[<xsl:value-of select="$spname"/>]')
         and OBJECTPROPERTY(id, N'IsProcedure') = 1)
drop procedure [dbo].[<xsl:value-of select="$spname"/>]
GO

CREATE PROCEDURE <xsl:value-of select="$spname"/>
</xsl:template>
```

NOTE There's a significant difference in working with stylesheet parameters and variables vs. passing template parameters. If you reference a stylesheet variable or parameter and it's missing, you'll receive an error. If you accept a parameter in the called template but it isn't passed by the calling template, you'll just output an empty string.

Closing the Stored Procedure with the CloseSP Template

The corresponding `CloseSP` finishes up the script by outputting the code to reset quoted identifiers and ANSI nulls, as well as statements to grant stored procedure permissions. To grant the correct permissions, the calling code passes the list of permissions to set as the `privileges` parameter. The `CloseSP` template loops through this list and creates a separate `Grant` statement for each element:

```
<xsl:template name="CloseSP">
<xsl:param name="privileges"/>
```

```
GO
SET QUOTED_IDENTIFIER OFF
GO
SET ANSI_NULLS ON
GO

<xsl:for-each select="$privileges">
GRANT EXECUTE ON <xsl:value-of select="$spname"/>
      TO <xsl:value-of select="@Grantee"/>
</xsl:for-each>
</xsl:template>
```

Outputting Parameters with the SPParameters Template

The `SPParameters` utility template outputs a list of stored procedure parameters. This is a match template so it's called once for each element in the node-set created by the `select` clause of the calling `xsl:apply-templates` directive:

```
<xsl:template match="dbs:RunSPParam" mode="SPParameters">
   @<xsl:value-of select="@Name" /><xsl:text> </xsl:text>
   <xsl:value-of select="@SQLType" />
   <xsl:if test="string-length(@MaxLength)>0">
      <xsl:text/> (<xsl:value-of select="@MaxLength"/>)<xsl:text/>
   </xsl:if>
   <xsl:if test="@IsAutoIncrement='true'">
      <xsl:if test="ancestor::orm:RunSP[@Task='Create']"> OUTPUT </xsl:if>
   </xsl:if>
   <xsl:if test="position()!=last()">, </xsl:if>
</xsl:template>
```

TIP `<xsl:if test="position()!=last()">, </xsl:if>` is a quick, easy way to include a comma only at the end of a list.

NOTE A space is required between the name and the type, and the `<xsl:text>` element ensures this will be output as significant whitespace.

NOTE The SPParameters template contains no literals at the end of lines and a literal only at the beginning of the first line. Thus, all the output from the template occurs on the same line.

You could read this XSLT in English as, "For each RunSPParam element, output the value of the Name and SQLType attributes. Unless MaxLength is an empty string, output the MaxLength attribute within parentheses. If IsAutoIncrement is true, and this is a create stored procedure, generate the word OUTPUT. Except for the last element, output a comma after each element."

Outputting Columns with the ColumnSPList Template

The ORMColumnSPList template outputs the columns used by a SELECT statement. It's used for both the BuildColumn child elements of the BuildTable and the SetSelectColumn child elements of SetSelect. You'll see this template applied in the context of SetSelectColumn elements in the section "Examining the SetSelect Stored Procedure Template." Because the ORM transformation process did the hard work of determining the columns for each BuildTable, this template isn't complex:

```
<xsl:template match="orm:BuildColumn | orm:SetSelectColumn" mode="ColumnSPList">
   <xsl:variable name="tablename">
      <xsl:choose>
         <xsl:when test="name()='orm:SetSelectColumn'">
            <xsl:value-of select="ancestor::orm:SetSelect/@TableName"/>
         </xsl:when>
         <xsl:otherwise>
            <xsl:value-of select="ancestor::orm:BuildTable/@Name"/>
         </xsl:otherwise>
      </xsl:choose>
   </xsl:variable>
   [<xsl:value-of select="$tablename"/>].[<xsl:text/>
   <xsl:value-of select="@Column"/>]<xsl:text/>
   <xsl:if test="@Name!=@Column">
   <xsl:text/> AS <xsl:value-of select="@Name"/>
   </xsl:if>
   <xsl:if test="position()!=last()">, </xsl:if>
</xsl:template>
```

NOTE The vertical bar (|) indicates the union of two XPath expressions. In this case, it means the template will be processed for any `orm:BuildColumn` or `orm:SetSelectColumn` element.

NOTE The XPath `.//orm:BuildColumn` indicates that you want all descendant `orm:BuildColumn` elements of the current context, which is the `orm:BuildRecordset` element. This outputs the column information for each `BuildColumn` regardless of which `BuildTable` it resides in.

The table and column names are always enclosed in brackets to handle nonstandard naming such as names with spaces and columns or names that are keywords. If the column name is nonstandard or there are redundant column names, the resulting stored procedure has to specify the column name for the output. If the problem is spaces in the column name, earlier metadata processing created a valid name as the `Name` attribute. If there are redundant columns, you may need to specify some column names in the ORM file. In any case where the `Name` isn't the same as the `Column` attribute, an `AS` clause instructs SQL Server to use a valid alternate name in the output recordset.

Looking at the Three Templates to Join Tables

Three templates contribute to joining tables. The `ORMJoinList` template is called to output the tables, regardless of whether there's a single table or joined tables. The `ORMJoinList` also tests for a valid number of tables. The `BuildJoin` template calls the `BuildSingleJoin` template to output the left and right sides of the join and outputs the join type.

Examining the ORMJoinList Template

Creating a join list lets you spread your wings a bit in generating stored procedure code via XSLT templates. Outputting the literal `FROM` gives you a reference point if there's a problem. If `FROM` doesn't appear in your output, the template wasn't called. If `FROM` exists in the output but nothing follows, the problem is in the call to the `BuildJoin` template.

The template then performs two checks and outputs error text directly into the template if there's a problem. These two tests ensure that there's at least one `BuildTable`, and if there's more than one, `BuildTable` ensures that there's a join

element. These tests are one step in avoiding Cartesian product joins[3] that might return millions of records.

If the `BuildRecordset` contains only one `BuildTable`, its name is output as the `FROM` clause and the rest of the template isn't processed. The code to output this part of the script is as follows:

```
<xsl:template name="ORMJoinList">
   FROM <xsl:text/>
   <xsl:choose>
      <xsl:when test="count(orm:BuildTable)= 0" >
         ERROR - Where are the tables?
      </xsl:when>
      <xsl:when test="count(orm:BuildTable)=1" >
         [<xsl:call-template name="GetTableName"/>]
      </xsl:when>
      <xsl:when test="count(orm:Join) = 0">
         ERROR - You need a join if you have multiple tables
      </xsl:when>
```

NOTE Outputting error messages directly into the script output causes it to fail when you execute the script against SQL Server, forcing you to handle these errors if they occur. This is particularly important because you input the join details of the ORM file by hand and could mistype something.

TIP Outputting a literal at the start of an XSLT template (when it's appropriate) gives you a quick confirmation that the template was processed.

The `GetTableName` template returns the original table name in the database. If there are multiple `BuildTable` elements and a `Join` element exists, `ORMJoinList` shifts the context to the first `Join` element and calls another template. Breaking up processing into multiple templates helps logically organize the template. In this case, it also supports recursion. The remainder of the `ORMJoinList` template, including the call to `BuildJoin`, is as follows:

```
      <xsl:otherwise>
         <xsl:for-each select="orm:Join[1]"   >
            <xsl:call-template name="BuildJoin"/>
```

3. A Cartesian product occurs when you fail to enter criteria for join criteria. Without join criteria, most relational databases attempt to combine every record from each table with every record in the other. This quickly becomes a ridiculous number of records.

```
            </xsl:for-each>
         </xsl:otherwise>
      </xsl:choose>
   </xsl:template>
```

The example of the SPSet shown in Listing 7-1 includes the specification for a two-table join. The example of the target output shown in Listing 7-2 shows the syntax I used for this join. Because there are a couple of ways to specify a join, you might want to look back at Listing 7-2. I think using the explicit JOIN...ON syntax is both the most readable way to specify a join, as well as being the most straightforward for code generation.

Examining the BuildJoin Template

The BuildJoin template outputs the left side of the join, the join type, and the right side of the join. The left and right portions of the join are created via the BuildSingleJoin template discussed next. The first part of the BuildJoin template, including calling the BuildSingleJoin template to output the left and right tables, is as follows:

```
<xsl:template name="BuildJoin">
   <xsl:for-each select="orm:Left">
      <xsl:call-template name="BuildSingleJoin" />
   </xsl:for-each>
   <xsl:variable name="type" >
      <xsl:choose>
         <xsl:when test="@Type=''">Inner</xsl:when>
         <xsl:when test="@Type"><xsl:value-of select="@Type"/></xsl:when>
         <xsl:otherwise>Inner</xsl:otherwise>
      </xsl:choose>
   </xsl:variable>
   <xsl:text> </xsl:text><xsl:value-of select="$type"/> JOIN <xsl:text/>
   <xsl:for-each select="orm:Right">
      <xsl:call-template name="BuildSingleJoin" />
   </xsl:for-each>
```

The ORM file specifies the criteria used to join the tables, which is manually entered prior to the ORM transformation process. The BuildJoin template outputs the criteria manually entered in the ORM file; if you messed up the syntax, you'll get an error when you execute the script to create the stored procedure. The template builds a variable to contain the JOIN clause to make it easier to check its length. If the length of this clause is zero, the template outputs an error about a Cartesian product. Again, this is output directly into the stored procedure script,

so it causes an error when you execute the script. The code that outputs the clause or the error message is as follows:

```
   <xsl:variable name="clause">
      <xsl:value-of select="orm:JoinOn/@Clause"/>
   </xsl:variable>
   <xsl:if test="$clause=''">
      ERROR - CARTISIAN PRODUCT
   </xsl:if>
      ON <xsl:value-of select="$clause" />

</xsl:template>
```

Examining the BuildSingleJoin Template

The third template used for creating joined clauses returns either the left or right portion of the join. In this case, using the separate templates supplies code reuse because the `BuildJoin` template calls the `BuildSingleJoin` template twice. If the clause specifies a table via the `JoinTable` attribute, the `BuildSingleJoin` template outputs the attribute value (the table name). If the clause specifies another join via the `Join` attribute, that `Join` element has to be named, and the template recursively calls the `BuildJoin` template to create that portion of the output. This XSLT code allows you to create `SELECT` statements with `JOIN` clauses far more complex than SQL Server would be happy with:

```
<xsl:template name="BuildSingleJoin">
   <xsl:choose>
      <xsl:when test="@Join">
         <xsl:variable name="joinname" select="@Join" />
         <xsl:for-each select="ancestor::orm:Joins/orm:Join[@Name=$joinname]">
         (<xsl:call-template name="BuildJoin" />)<xsl:text/>
         </xsl:for-each>
      </xsl:when>
      <xsl:otherwise>
         <xsl:value-of select="@JoinTable"/>
      </xsl:otherwise>
   </xsl:choose>
</xsl:template>
```

Now that you've walked through the utility templates and the retrieve stored procedure in some detail, I'll cover just the high points of the other four stored procedure types.

Generating Create Stored Procedures

The create stored procedure contains an INSERT statement that adds a single record to your database. Comparing the create stored procedure stylesheet with the retrieve stored procedure stylesheet will illustrate some of the similarities and differences between stylesheets. This stored procedure also shows a slightly different style that uses internal loops, rather than calling utility templates. This is appropriate where the pattern is simple and you won't be reusing the pattern.

Looking at the Create Stored Procedure Target

The target for the create stored procedure is similar to the retrieve stored procedure. It sets the environment identifiers and drops any existing stored procedure with the same name. It then outputs a stored procedure with a SQL INSERT statement that loads data into the database.

Unlike the retrieve stored procedure, the create stored procedure accesses only one database table. The middle tier calls the correct stored procedure for each business object. If there's an identity (auto-increment) primary key in the table, the corresponding parameter would be an output parameter set prior to the stored procedure return. Although identity keys aren't illustrated in the sample shown in Listing 7-3, you'll see how it's handled in the stored procedure stylesheet.

Listing 7-3. Target for the Create Stored Procedure Template

```
SET QUOTED_IDENTIFIER ON
GO
SET ANSI_NULLS ON
GO

if exists (
       select * from dbo.sysobjects
       where id = object_id(N'[dbo].[wx_appProject_Insert]')
         and OBJECTPROPERTY(id, N'IsProcedure') = 1)
drop procedure [dbo].[wx_appProject_Insert]
GO

CREATE PROCEDURE wx_appProject_Insert
   (
   @ID UniqueIdentifier,
   @Name VarChar (50),
```

```
    @Started DateTime,
    @Ended DateTime,
    @Description Text,
    @DepartmentID Int
    )
    AS

    INSERT INTO [Projects]
    (
              [ID],
              [Name],
              [Started],
              [Ended],
              [Description],
              [DepartmentID])
    VALUES
    (
              @ID,
              @Name,
              @Started,
              @Ended,
              @Description,
              @DepartmentID)

    RETURN (@@ERROR)

GO
SET QUOTED_IDENTIFIER OFF
GO
SET ANSI_NULLS ON
GO

GRANT EXECUTE ON wx_appProject_Insert TO Admin
```

Examining the Create Stored Procedure Stylesheet

The stylesheet opening, stylesheet parameters, and stylesheet variables are identical to the opening of the retrieve stored procedure in the section "Examining the Retrieve Stored Procedure Stylesheet." The entry-level template is nearly identical to the retrieve stored procedure entry-level template, but it calls the

high-level processing template for elements with a `Task` attribute equal to `Create` instead of `Retrieve`:

```
<xsl:template match="/">
   <xsl:apply-templates
         select="//orm:SPSet[@Name=$Name]/orm:RunSP[@Task='Create']"
         mode="RunSP" />
</xsl:template>
```

The create stored procedure template needs the original name of the database table—which will differ from the `TableName` attribute of the `SPSet` if the original name contained a space. The `origtablename` variable contains the `OriginalName` attribute from the input metadata file:

```
<xsl:template match="orm:RunSP" mode="RunSP">
   <xsl:variable name="tablename" select="ancestor::orm:SPSet/@TableName"/>
   <xsl:variable name="origtablename"
               select="//dbs:Table[@Name=$tablename]/@OriginalName"/>
```

Opening the stored procedure, including the parameter declarations, is identical to the retrieve stored procedure stylesheet. Although the parameters are quite different, the differences are held in the metadata list of parameters for each stored procedure. Extra work earlier in creating metadata pays off in this sort of template simplification. The code that starts the template and outputs the parameters is as follows:

```
<xsl:call-template name="OpenSP"/>
(<xsl:apply-templates select="orm:RunSPParam" mode="SPParameters"/>
)
AS
```

An `INSERT` SQL statement consists of two similar blocks—one containing the fields used for the new values and the other contains the values to insert—contained in passed parameters. In both cases, you need to exclude identity (auto-increment) fields contained in the `INSERT`. To do this, I loop through the parameters, skipping any where the `IsAutoIncrement` column is set to `true`. This portion of the resulting template is as follows:

```
INSERT INTO [<xsl:value-of select="$origtablename"/>]
(<xsl:for-each select="orm:RunSPParam">
   <xsl:choose>
      <xsl:when test="@IsAutoIncrement='true'"/>
      <xsl:otherwise>
         [<xsl:value-of select="@Column"/>]<xsl:text/>
```

```
            <xsl:if test="position()!=last()">, </xsl:if>
         </xsl:otherwise>
      </xsl:choose>
   </xsl:for-each>)
VALUES
(<xsl:for-each select="orm:RunSPParam">
   <xsl:if test="not(@IsAutoIncrement='true')"/>
      @<xsl:value-of select="@Name"/><xsl:text/>
      <xsl:if test="position()!=last()">, </xsl:if>
   </xsl:if>
</xsl:for-each>)
```

TIP You can't use the XPath `"@IsAutoIncrement!='true'"` in the previous code if the attribute might not exist. That's because the lack of the attribute will return `false`. You can use `"not(@IsAutoIncrement='true')"` as the conditional expression.

Once the stored procedure has inserted the record, any of the stored procedure parameters that are identity columns need to be set before returning from the stored procedure:

```
<xsl:for-each select=".//dbs:TableColum[@AutoIncrement='true']">
   SELECT @<xsl:value-of select="@Name"/> = @@IDENTITY
</xsl:for-each>
```

NOTE Per SQL Server rules, there should be only one identity (also called *auto-increment*) column.

The close of the stored procedure is identical to the retrieve stored procedure close except that you use the privileges containing create rights:

```
   RETURN (@@ERROR)

   <xsl:call-template name="CloseSP">
      <xsl:with-param name="privileges"
          select="ancestor::orm:SPSet//orm:Privilege[contains(@Rights,'C')]" />
   </xsl:call-template>
</xsl:template>
</xsl:stylesheet>
```

Generating Delete Stored Procedures

The delete stored procedure stylesheet illustrates how you work with a specific record in the database. You'll reuse this simple technique in the section "Generating the Update Stored Procedure Stylesheet."

There are at least two ways to handle deletions in your database tables, each having applicability in specific scenarios. You can perform physical deletions from your database via the `DELETE` statement—in which case SQL immediately and permanently removes the record from your database with no trace outside log files. Alternatively, you can maintain a status field that indicates whether you've deleted the record. This logical deletion allows you to reinstate records and track deleted records. You can periodically remove logically deleted records from the database. The benefit of the physical delete is that you don't waste database space with meaningless records. The benefit of the logical delete is the simplicity of re-creating an accidentally deleted record. I'll show how to do the physical deletion in the next section. If you'd like to perform logical deletions, you can change the approach to use the `UPDATE` statement to modify your status field instead of using the `DELETE` statement.

Looking at the Delete Stored Procedure Target

The open and close of the target script for the delete stored procedure is identical to the stylesheets for the retrieve and create stored procedures. Because of this similarity, I'll skip including these portions of the script and just include the core portion that defines the stored procedure. The stored procedure parameters are the primary key columns for the table. Listing 7-4 shows the target for the template.

Listing 7-4. Target for the Delete Stored Procedure Template

```
CREATE PROCEDURE wx_appProject_Delete
   (
   @ID UniqueIdentifier
   )
   AS

   BEGIN
   SET NOCOUNT ON

      DELETE FROM [Projects]
      WHERE [ID] = @ID
   END
```

NOTE This and other stored procedures in this chapter support multicolumn primary keys.

Examining the Delete Stored Procedure Template

Because the opening and closing portions of the template are just like the retrieve stored procedure, I haven't included them so that you can focus on the high-level template. Like the create stored procedure template, this template determines the original table name and assigns this value to a variable for later use. The `OpenSP` template outputs the start of the stored procedure, and the `SPParameters` template outputs the parameters:

```
<xsl:template match="orm:RunSP" mode="RunSP">
   <xsl:variable name="tablename" select="ancestor::orm:SPSet/@TableName"/>
   <xsl:variable name="origtablename"
                select="//dbs:Table[@Name=$tablename]/@OriginalName"/>

   <xsl:call-template name="OpenSP"/>
   (<xsl:apply-templates select="orm:RunSPParam" mode="SPParameters"/>
   )
   AS

   BEGIN
    SET NOCOUNT ON
```

The `DELETE` statement uses the `origtablename` variable to identify the table to delete records from. The table name will differ from the `SPSet` element's `TableName` attribute when there are spaces in the SQL Server table name. The `WhereClause` uniquely identifies the record to delete. The template code that outputs the `DELETE` SQL statement is as follows:

```
   DELETE FROM [<xsl:value-of select="$origtablename"/>]
   WHERE <xsl:value-of
            select="ancestor::orm:SPSet/orm:WhereClause/@Clause" />

END
```

The remainder of the template is the same as the retrieve and create stored procedure templates:

```
   <xsl:call-template name="CloseSP">
      <xsl:with-param name="privileges"
          select="ancestor::orm:SPSet//orm:Privilege[contains(@Rights,'U')]" />
   </xsl:call-template>

</xsl:template>
```

Generating Update Stored Procedures

The update stored procedure stylesheet combines the techniques of the create and delete stored procedure stylesheets. You need to identify a specific record and then address the individual fields of that record.

Looking at the Update Target

As in the delete stored procedure script, I won't present the opening and closing parts of the script here, just the definition of the stored procedure itself. Listing 7-5 shows the stored procedure the template creates.

Listing 7-5. Target for the Update Stored Procedure Template

```
CREATE PROCEDURE wx_appProject_Update
   (
   @ID UniqueIdentifier,
   @Name VarChar (50),
   @Started DateTime,
   @Ended DateTime,
   @Description Text,
   @DepartmentID Int
   )
   AS

BEGIN
SET NOCOUNT ON
```

```
   UPDATE [Projects]
      SET
         [ID] = @ID,
         [Name] = @Name,
         [Started] = @Started,
         [Ended] = @Ended,
         [Description] = @Description,
         [DepartmentID] = @DepartmentID
      WHERE
         [ID] = @ID

END
```

Examining the Update Stored Procedure Template

I'll skip the opening and closing portions of the stylesheet to focus on the template that creates the script. Like the create and delete stored procedures, this template stores the SQL Server table name into the `orgtablename` variable. The `OpenSP` template again outputs the start of the script, and the `SPParameters` template outputs the parameter list:

```
<xsl:template match="orm:RunSP" mode="RunSP">
   <xsl:variable name="tablename" select="ancestor::orm:SPSet/@TableName"/>
   <xsl:variable name="origtablename"
                select="//dbs:Table[@Name=$tablename]/@OriginalName"/>

   <xsl:call-template name="OpenSP"/>
   (<xsl:apply-templates select="orm:RunSPParam" mode="SPParameters"/>
   )
   AS

BEGIN
SET NOCOUNT ON
```

The `UPDATE` statement uses the `origtablename` variable to determine the table to update. The template loops through the `RunSPParam` to set values. Looping through the parameters might be a little surprising when you're setting column values. But you can't set column values unless you've passed in a value, so this is an appropriate approach. The SPInputMetadata.xml file includes the original column name for each parameter to facilitate this assignment. Like the create stored procedure template, you've got to skip any identity (auto-increment)

columns. Like the delete stored procedure template, the `WhereClause` element uniquely identifies the record to update:

```
UPDATE [<xsl:value-of select="$origtablename"/>]
   SET
   <xsl:for-each select="orm:RunSPParam">
      <xsl:choose>
         <xsl:when test="@IsAutoIncrement='true'"/>
         <xsl:otherwise>
      [<xsl:value-of select="@Column"/>] = <xsl:text/>
            <xsl:text />@<xsl:value-of select="@Name"/><xsl:text/>
            <xsl:if test="position()!=last()">, </xsl:if>
         </xsl:otherwise>
      </xsl:choose>
   </xsl:for-each>
   WHERE <xsl:value-of
            select="ancestor::orm:SPSet/orm:WhereClause/@Clause" />
```

The remainder of the template is the same as the earlier stored procedure templates:

```
END

   <xsl:call-template name="CloseSP">
      <xsl:with-param name="privileges"
          select="ancestor::orm:SPSet//orm:Privilege[contains(@Rights,'U')]" />
   </xsl:call-template>

</xsl:template>
```

Generating SetSelect Stored Procedures

The stored procedure that selects a set of records is a little different than the other four stored procedures. The output is quite simple—just a select of specified columns for all records in the table.

Your select and set select stored procedures are the area of stored procedure creation that you're most likely to modify to fit your application. You may want to provide templates that create set select stored procedures for each parent table, or you may decide to handcraft stored procedures that select sets of records.

If you just want to add columns retrieved by the set select stored procedures, you add the `UseForDesc` attribute to the `TableColumn` element prior to the ORM

transformation. You can do this either by incorporating the `IncorporateTableInfo` stylesheet after the merge in the transformation process described in Chapter 6 or by adding `UseForDesc` as a SQL Server extended property on the columns you'd like to include.

Looking at the SetSelect Target

The set select stored procedure is simple. It just outputs the primary key column(s) and any `BuildColumn` elements with the `UseForDesc` attribute set to `true`. These could be manually included in the ORM file or included as an extended property in the database. The `SetSelect` stored procedure would also include a `WHERE` clause if there was a `SetSelectWhereClause` element. Listing 7-6 shows an example of the stored procedure output by the template.

Listing 7-6. Target for the Stored Procedure That Selects a Set of Records

```
CREATE PROCEDURE wx_appProject_SelectAll
AS
   SELECT
      [Projects].[ID],
      [Projects].[Name]
   FROM Projects
RETURN
```

Examining the SetSelect Stored Procedure Template

The set select stored procedure template uses techniques you've already seen. Again I'll skip other stylesheet elements to focus on the template that creates the stored procedure script. The opening of this template is similar to the other templates in this chapter. Unlike other templates, the `SetSelect` stored procedure template might not include parameters, so the parameter block is enclosed in an `xsl:if` directive. You need this check because empty parentheses are illegal in this context when SQL Server executes the script. The resulting XSLT code is as follows:

```
<xsl:template match="orm:SetSelect" mode="SetSelect">
<xsl:call-template name="OpenSP"/>
<xsl:if test="orm:RunSPParam">
   (<xsl:apply-templates select="orm:RunSPParam" mode="SPParameters"/>
   )
</xsl:if>
AS
```

The SELECT portion of this template uses the ColumnSPList match template to output whatever SetSelectColumn elements are included. You saw this utility template earlier in the section "Examining the Retrieve Stored Procedure Template." The FROM clause is simply the SetSelect element's TableName attribute. You may or may not have a WHERE clause, so an xsl:if conditional determines whether to output the literal and the WHERE clause. Without this conditional, a missing SetSelectWhereClause (the normal case) would result in a syntax error when you ran the output script because of the hanging WHERE without a corresponding clause:

```
SELECT <xsl:apply-templates select="orm:SetSelectColumn"
    mode="ColumnSPList" />
FROM <xsl:value-of select="@TableName" />
<xsl:if test="orm:SetSelectWhereClause">
WHERE <xsl:value-of select="orm:SetSelectWhereClause/@Clause" />
</xsl:if>
```

The remainder of the stored procedure template is identical to earlier stored procedures:

```
RETURN

   <xsl:call-template name="CloseSP">
      <xsl:with-param name="privileges"
           select="orm:Privilege[contains(@Rights,'S')]" />
   </xsl:call-template>

</xsl:template>
```

Summary

In this chapter, you learned the following:

- Stored procedure templates are the simplest type of template to create and maintain, and they have a very quick payback for code generation.
- The five stored procedure template patterns are quite similar, reusing many techniques.
- Many of the techniques used in the more complex middle tier templates are illustrated in stored procedure templates, including using supporting utility templates for common functionality, using branching and looping structures, and using variables.

Additional Reading

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

- *Advanced Transact-SQL for SQL Server* by Itzik Ben-Gan and Tom Moreau (Apress, 2000)
- *Inside Microsoft SQL Server 2000* by Kalen Delaney (Microsoft Press, 2000)
- *The Guru's Guide to Transact-SQL* by Ken Henderson (Addison-Wesley, 2000)

EXHEAP

EXHEART

CHAPTER 8

Generating Middle Tier Templates

Templates let you generate code to work with an available framework.

ONCE YOU'VE ESTABLISHED THE ARCHITECTURE and metadata you'll use for your middle tier, you're ready to generate the code. This chapter shows you how to build a middle tier that tightly integrates into an existing framework, and the next two chapters cover how to use that middle tier to create a User Interface (UI).

In writing this chapter, I faced a dilemma. Code-generated middle tier classes and your handcrafted code work closely with a framework. This framework is important to the success of your application, but I'm not writing a book about frameworks. There are many issues to consider in building full-scale architectures and frameworks, which is why there are full-length books about those subjects. This book focuses on code generation.

At the same time, I want to show code generation with real templates—templates you can take into your own projects and use this afternoon that aren't so oversimplified that you'll be splicing onto them until they become difficult to maintain. If you or your organization already has strong ideas about architecture, you can use any set of templates for reference; they just need to reflect at least the complexity you'll encounter. On the other hand, you may be a bit lost in the architectural realm—reading all those white papers from Microsoft and trying to figure out which ones to implement. In that case, I don't want to give you just *any* architecture. I want to give you a *very good* architecture you can comfortably use in real-world applications. But my dilemma is that such architectures are inherently too complex to describe in a few pages.

To solve this, I'll use the architecture Rockford (Rocky) Lhotka built in his book *Expert One-on-One Visual Basic .NET Business Objects* (Apress, 2003). Rocky's done a lot with this book. The first part describes how architectural pieces fit together as building blocks for different deployment scenarios. He articulates why we have many variations on the basic n-tier architectures.

Throughout the rest of his book Rocky builds on this understanding to create an architecture that adapts between deployment scenarios based on configuration file settings. I'm excited about this architecture because it supplies such a high level of scalability.

Rocky calls this architecture Component-based, Scalable, Logical Architecture (CSLA). Using the CSLA architecture allowed me to keep this chapter focused on the real issues of middle tier code generation. At the same time, it does a service for anyone implementing CSLA and looking for a faster way to build middle tier classes. If you plan to use CSLA, you'll want to check `http://www.lhotka.net` for recent updates because Rocky is continuing to fine-tune the architecture and has numerous article and updates.

NOTE To accommodate code generation, you need to make or download a change to CSLA as described in the sidebar "Changing CSLA to Accommodate Code Generation."

Expert One-on-One Visual Basic .NET Business Objects discusses the CSLA framework itself and gives examples to show how your objects implement this framework. This chapter is about templates that generate classes that *use* the CSLA framework. The code these templates generate parallels the examples in *Expert One-on-One Visual Basic .NET Business Objects.*

I made several changes to the example business classes to make generating them easier. The section "Changing the Base Architecture" discusses the issues that led me to make these changes. Whatever architecture you're working with, you'll probably encounter some of these same issues. To make it easer to follow these changes if you've read Rocky's book, I'm using his PTracker database in my samples. (Figure 6-2 in Chapter 6 shows the structure of this database.)

NOTE I'm only explaining CSLA at a very high level, and if you use this approach in your applications, I suggest you read Rocky's book for a full understanding.

NOTE As you'll read in Chapter 11, one of the most exciting aspects of code generation is using it as a mechanism for encapsulating and passing on the best work of the best minds in the industry. My leveraging Rocky's work demonstrates this. You don't need to be an architectural expert; you can reuse his work through these templates that implement his framework.

I'm using CSLA because I think it's quite good, but the point of this chapter isn't these specific CSLA templates. Instead, these templates illustrate how you'll solve the challenges you'll encounter in generating middle tiers from existing architectures. I've tried to cover common complexities, but you'll be happy to hear that it's unlikely you'll run into all of these issues.

Differentiating Code Generation and Serialization to Code

When you're working with Visual Studio, it generates one particular style of code for you. This is the code serialization Visual Studio uses to store and retrieve values that appear in the Properties window. Serialization means that you're storing an object's data and retrieving it in a way that re-creates a copy of the original object. The way you store and retrieve doesn't matter, except to the tool's serializing and deserializing.

If you can serialize and deserialize (or reconstitute) to just about anything, that includes serializing and deserializing to code. A tool can serialize by generating code that sets particular properties. It can deserialize by executing this code. Visual Studio uses this technique for serializing property values of objects such as controls in the region generated by Windows.Forms. The `CodeDOMSerializer` writes out code that Visual Studio later executes to rebuild the object or parses for use with the property grid.

Serialization to code does generate code, but it's not the same as application code generation. The following aspects differentiate code serialization and code generation:

- First, code serialization stores object values, where application code generation generally creates a full range of classes for your application. Application code generation generally doesn't define objects that can be instantiated at either design or runtime.
- Second, the purpose of code serialization is to store the full information about the object, and code generation rarely uses the full information in the metadata for any single file. For example, middle tier code wouldn't normally contain the caption for each property.
- Finally, code serialization is done in bits and pieces as you need it during design time, and application code generation is a single one-click process.

You can extend your code generation to also serialize your metadata. But it's best to think of code serialization and code generation as independent actions that you might choose to combine in some situations.

Introducing Component-based, Scalable, Logical Architecture (CSLA)

CSLA is a solid architecture that offers the ability to use different deployment scenarios based on values in a config file. That's huge. The supported deployment models extend all the way from single user to server farm. CSLA includes features, such as its security model, that adapt as you move between these deployment models. It's difficult to build a high degree of scalability into an architecture, but CSLA does it.

The second-best thing about CSLA is the laundry list of small issues it solves. Among other things, this includes the ability to switch between ADO.NET transactions and Enterprise Services transactions (also known as COM+). It also addresses multiple-level undos within a business hierarchy, solving a nontrivial problem.

For all the things I like about it, CSLA isn't perfect for all scenarios. For example, you'll need to extend it to retrieve grandchild objects.[1] Normal business hierarchies often benefit from either using only a two-level hierarchy or closely controlling the grandchild objects' retrieval through dynamic loading. You don't want to retrieve the entire timesheet history of a large project just to see a list of people who have worked on it. Also, the sidebar "Isolating Data Structure Details in CSLA" discusses a security aspect that'll be problematic in a few scenarios. CSLA isn't designed for seamless use with multiple backends. You could alter the templates so different business objects could use different backend servers, but you'd have to alter the architecture or generate redundant classes[2] if you simultaneously deliver your product to clients using different backends such as SQL Server and Oracle. If you don't need simultaneous deployment to multiple backends, code generation does make it relatively easy to later switch your backend server type.

None of these problems are significant issues in mainstream scenarios, and they emphasize that no one has yet built the perfect architecture applicable in all scenarios—which I guess is why Microsoft produces so darn many architectural white papers. CSLA works in a large number of scenarios, with a solid core ready for you to extend to solve specific issues if they arise in your environment.

CSLA isn't a simple architecture. I suggest you understand it (or any architecture you implement) prior to using it in your development. I'm just skimming over the top of CSLA, so check out Rocky's book for the full explanation.

---

1. You can find an article on loading grandchild data in CSLA on Rocky's site at `http://www.lhotka.net`.

2. Similar redundant classes are a realistic solution when you're using code generation. The trick to appropriate implementation is maintaining a single copy of handcrafted code.

Isolating Data Structure Details in CSLA

A key detail of any n-tier architecture is how strictly you isolate details about the data structure from the UI programmers. These details include the name and parameters for stored procedures as well as connection strings. You don't want UI programmers directly using database details, so you want some sort of logical wall between them. But the appropriate strength of this wall varies greatly with your particular scenario. For example, in some scenarios, you can create a set of `IDoNotUseInUI_<Whatever>` interfaces and train UI programmers not to use them. The business objects can still access database details through these interfaces. At the other end of the spectrum, you may need to release business objects to other organizations or over the Internet in a Web Service deployment where a malicious programmer could potentially use database details to compromise your system.

The solution Rocky presents in his release of CSLA (that I use here) is a compromise between these two extremes. The database details are included in the business objects, but they're private, and you can only access them through .NET `Reflection`. This makes it quite difficult for a well-meaning UI programmer to accidentally use the database details in their code. But it also means that if you release business objects beyond your control, a malicious programmer can use Microsoft Intermediate Language (MSIL) or reflection to retrieve the database details.

Refection is a solid technology, but there are two other downsides to working with it. First, it's a bit more complicated than direct calls, and you'll have to do a bit more work catching and interpreting exceptions.[3] Second, there's a slight performance hit when using reflection. CSLA limits the use of reflection to instantiating objects and invoking methods for database access. Because reflection use is optimized, the performance hit is trivial.[4]

NOTE If you really hate reflection and you're a solo or small group developer where the same people do both framework and business object development and it's easy to trust UI programmers to follow rules, you can simply replace reflection with interfaces that include a prefix that identify them for exclusive middle tier use. To test the purity of your UI code, you can simply search the code for your interface prefix.

3. When exceptions are thrown within code accessed via reflection, the exception thrown is wrapped in a `System.TargetInvocationException` (surprisingly derived from `System.ApplicationException`). You have to explore nested exceptions or call `<exception>.GetBaseException` to get the original exception thrown. This not only takes an extra step in querying exception, it prohibits you from easily catching the specific inner exception. During development you can use a variation of the exception dialogue described in Chapter 5, but you'll have to do something less obtrusive when you deploy to users.

4. You can do your own benchmarks. The method invocation via reflection is around 100 times slower than a direct method call, but this is the difference between about $1 * 10^{-6}$ and 1 and $1 * 10^{-8}$. Even in a high production server, occasionally invoking a method via reflection will be dwarfed by other operations.

Although reflection performance isn't significant, in some scenarios putting any database information into MSIL on the client is a real security concern. If you have security concerns about business objects on your client containing database details, it's a bit more complex to modify the framework. You need to either remove the data information from the basic plumbing of the business object or provide different business objects for the UI and middle tier. You can accomplish the first approach by placing the data access code (the code in the `Data Access` region of CSLA-based objects) into another object. For the second, you can use code generation to create two different objects—one designed for the client and the other for the application server. You can serialize between these different classes. The server objects could be identical to the objects discussed in this chapter, and the client objects either have no data access methods or have them stubbed out and raise an exception if client code calls them.

NOTE The mechanism for isolating database details is internal to the framework and templates. With code generation in place, this means you can modify the framework as easily later as now and adapt your templates as necessary. Because you can change it later, you don't need to alter the design first because of the small chance you'll need to make a change later.

TIP If you make changes, be considerate of other programmers. .NET insulates you from Dynamic Link Library (DLL) problems, so incompatible DLLs sitting in different directories isn't going to hurt your end users—unless they have the same version number and are deployed to the Global Access Cache (GAC). But can you imagine maintaining several copies of CSLA with identical names but slight variations? If other programmers will use your modified code, either ensure backward compatibility or modify the name and namespace names.

Walking Through the CSLA Process

CSLA consists of a framework and business objects. I'll generally focus on business object generation and not attempt to fully explain the CSLA framework. However, I'll start with an overview to help you understand why the business objects behave the way they do. The majority of the magic of CSLA is how the framework, including base classes, simplifies the business objects. The framework

and base classes do most of the work. The key components of the framework are as follows:

- `BusinessBase`
- `BusinessCollectionBase`
- `DataPortal`
- Server `DataPortal`

The `DataPortal` and server `DataPortal` are separate classes to facilitate scaling to different deployment scenarios. The server `DataPortal` can be deployed on an application server while the `DataPortal` is its client or Web server counterpart. CSLA provides two different server `DataPortal` classes—a normal one and a serviced portal designed for COM+. The serviced portal is used automatically when you specify COM+ transactions. For simplicity, I'll ignore the two different server `DataPortal` classes in the rest of this discussion.

Figure 8-1 shows how methods of the `BusinessBase`, `DataPortal`, and server `DataPortal` interact to perform a fetch operation. The process consists of several steps because of the multiple deployment scenarios supported. The UI calls the shared `Get<BusinessObjectName>` (such as `GetProject`) of the business object. This method accepts information to build a criteria object and passes this criteria object to the *client* `DataPortal` class's `Fetch` method. The *client* `DataPortal.Fetch` method calls either the normal or the serviced *server* `DataPortal.Fetch`. The *server* `DataPortal.Fetch` uses reflection to instantiate an object of the appropriate business class and calls the `DataPortal_Fetch` method of the newly created business object. The `DataPortal_Fetch` of the business object instantiates the ADO.NET objects needed to retrieve data and performs this retrieval.

If you haven't seen CSLA before, you'll probably need to walk through this process a couple of times because of the many occurrences of the word *DataPortal*. The two sets of `DataPortal` classes are closely linked—two pieces that communicate across a boundary in many deployment scenarios. The `DataPortal_Fetch` method name reflects that it's used exclusively by the `DataPortal` objects (through reflection). Other data access operations follow a similar pattern. This walkthrough clarifies why these business objects templates have multiple methods for each data access step.

NOTE If you implement a different architecture that uses a separate Data Access Layer (DAL), you'll have at least two methods for each data access operation; however, these multiple methods will reside in at least two different classes. The basic pattern used in these templates remains relevant.

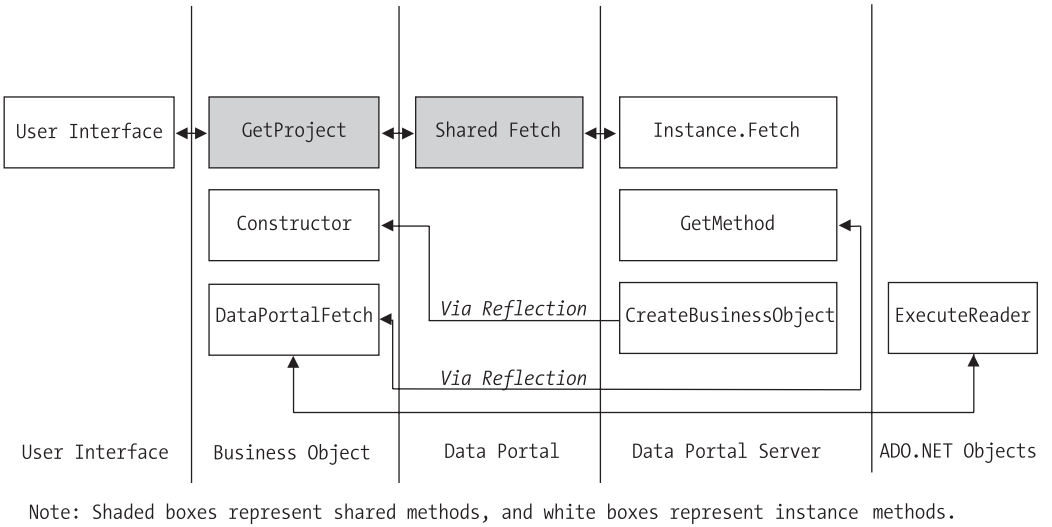


Figure 8-1. The interaction between different classes during a fetch operation

Exploring the Four Base Models

In evaluating any code architecture for translation into templates, the first step is to establish the basic models, or different types, of classes. Architectures often consist of the same four basic models. The contents of these archetypes differ somewhat between different architectures, but their presence is nearly universal:

- **Business object**: Represents an individual business object along with its data. This often corresponds to a single record in a database.
- **Business object collection**: Represents a set of business objects that have a logical connection. A group of business objects often represents the children of a different business object.
- **List**: Collections of simplified information generally used for search results.
- **Lookups**: Collections containing pairs of keys and corresponding text. Lookups are very important in normalization, localization, and so on. These are handled as name-value pairs in CSLA.

NOTE Rocky presents individual CSLA business objects as eight different patterns. This is convenient if you're creating them from scratch, but each of these eight patterns are variations of the four underlying and more or less universal patterns.

Business object collections, lists, and lookups all contain multiple elements. They differ in both complexity and purpose. Business object collections contain a set of fully viable business objects generally used for child collections of other business objects. Lists are simplified sets of entity information. Lists don't contain fully viable business objects but a simplified object (CSLA uses structures), and they're generally used to display available objects, such as the result of a search. Almost all applications also contain lookup values. Whether you enforce relations for lookup values, they differ from logical parents in providing a limited set of information such as a key and description or key, description, and abbreviation. The information provided by lookups may be a bit more complex, but all lookups in your application generally have the same data associated with them, regardless of the context. Lookup data is relatively static compared to other application data.

Providing Lookup Values

There are a variety of ways to store lookup values including what information is available, whether each set of lookups is in a different table, and whether you use a multipart key if you combine them in a single table. The most extensive lookup table system I've seen included a two-layer abstraction allowing the same lookup values to be combined in multiple lookup sets and a date range across which the lookup value was valid. Although that complexity was appropriate for the application, most applications are fine with a much simpler system. CSLA allows you to retrieve these values directly from the database, without a stored procedure.

All of these approaches are valid for some scenarios. Look at your individual requirements, and plan to tweak the templates to fit your lookup strategy.

TIP Use the same lookup strategy for all lookups in your application.

TIP You can improve performance, especially for ASP.NET applications, if you cache lookup values on the application server. One way to do this is to store the lookup values as Extensible Markup Language (XML) files with a mechanism for updating them when the corresponding database values change.

Business object and business object collection classes can exist as root classes, child classes, or both. The distinction between a root and a child class is whether it exists within a parent. If a class is inherently contained within another class, you won't run a stored procedure to retrieve its data (in many architectures) but will instead expect the parent to retrieve the data. A class may have only a child implementation, only a root implementation, or both.

When you're planning code generation, you want to limit the number of redundant templates. Because of this, I use one template for each of the major models, with conditional segments for root or child classes within each template as needed.

NOTE CSLA calls a class that can be a child or a root a *switchable* class.

Table 8-1 describes the four base models along with the set of templates and base class used for each one. Each set contains two templates—one for the generated class and one for the initial generation of the derived handcrafted (editable) class. This table also shows which of the eight patterns in *Expert One-on-One Visual Basic .NET Business Objects* map to each template. The CSLA base class is important because the code you generate from the corresponding template derives from this class.

Table 8-1. The Four Middle Tier Base Models

MODEL	TEMPLATES	CSLA BASE CLASS	*EXPERT ONE-ON-ONE...* PATTERN
Business objects	genCSLAEditSwitchable.xslt, CSLAEditSwitchable.xslt	`BusinessBase`	EditableRoot, EditableChild, EditableSwitchable
Business object collections	genCSLAEditCollection.xslt, CSLAEditCollection.xslt	`BusinessCollectionBase`	EditableRootCollection, EditableChildCollection
Lists	genCSLAList.xslt, CSLAList.xslt	`ReadOnlyCollectionBase`	ReadOnlyCollection
Lookups	genCSLANameValueList.xslt, CSLANameValueList.xslt	`NameValueList`	Name-value pair

Changing the Base Architecture

No matter what architecture you're using, you'll probably encounter the same types of issues when you implement templates. Discussing the changes I made to the CSLA business objects to make code generation easier serves two purposes. If you plan to use CSLA, it'll explain the changes I made to the business objects so they work smoothly with code generation. If you're using a different architecture, it'll help guide you through some key decision points you're likely to encounter. Many of these issues result from adding the derived handcrafted class.

Deriving Business Objects

The most significant change you'll make in adapting most architectures to code generation is adding derived classes to contain handcrafted code. This change is important, as discussed in Chapter 4. Adding the derived class may require only simple scoping changes. On the other hand, you may encounter challenges, particularly if you use reflection, or derive business objects from other business objects (a common case).

NOTE Whenever you look at the work of two authors, you risk having the same word mean different things. From the perspective of code generation, the word *editable* means a derived handcrafted class a programmer can edit. In the context of CSLA, *editable* means a class whose objects can be edited by the end user.

NOTE See the side bar "Changing CSLA to Accommodate Code Generation" for a discussion of changing the framework to support using derived classes for handcrafted code in this book's templates.

You'll generally want to allow business objects to derive from other business objects. For example, in the samples for Chapters 8–10,[5] the `ProjectAssignment` and `ResourceAssignment` classes derive from the `Assignment` class. You need six classes for this to work in conjunction with the derived handcrafted classes. Figure 8-2 illustrates how each of three handcrafted classes is associated with a

5. Thanks to Rocky Lhotka for the sample database design. This provides continuity if you're using the CSLA architecture.

generated class. The handcrafted `Assignment` class derives from `genAssignment`, and `genAssignment` provides the bulk of the plumbing. Any handcrafted code relevant to both logical children resides in the `Assignment` class. The `genProjectAssignment` and `genResourceAssignment` classes each derive from `Assignment` and finish out the plumbing for these classes. The `ProjectAssignment` and `ResourceAssignment` classes contain additional business class–specific handcrafted code.

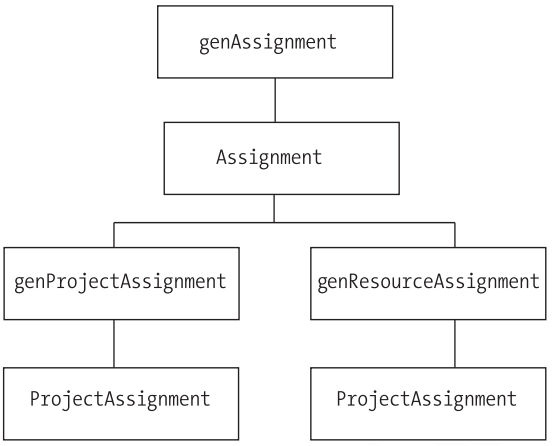


Figure 8-2. Combining derived handcrafted classes and derived business objects

TIP To support business object derivation, code generation templates must recognize child classes so they don't repeat the base class code.

Providing Parameterless Constructors

One of the little housekeeping changes you'll need to make to accommodate derived handcrafted class is to have a parameterless constructor with nonprivate scope in the generated base class. Whenever you can avoid it, you don't want any important code in your derived handcrafted classes. You can't keep later programmers from modifying code in the handcrafted classes, and you can't update edited classes as your architecture evolves with your project's life cycle. This presents problems with constructors. Constructors must generally be defined in the

class being instantiated. The only exception is that the compiler provides a public parameterless default constructor that calls the base class parameterless constructor if you don't supply any other constructor. You can take advantage of this feature to keep your derived classes empty—assuming you can use public parameterless constructors. This parameterless constructor of the base class must be visible in the derived class, so it can't be private or you'll get a compiler error.

NOTE You can't override shared methods or constructors.

CSLA presents two conflicting demands on the constructor in the derived handcrafted class. The CSLA design requires that business objects are instantiated in shared methods of the business object, not by external code. To block external instantiation, the constructor needs to be nonpublic. But the default constructor created by the compilers is public. The best available solution is to include the friend constructor in the derived handcrafted class. This isn't an ideal solution because a programmer who doesn't understand the architecture can simply change the scope of the constructor. That's certainly nothing worse than the situation without code generation, but the programmer is probably working in a scenario where they believe they can edit the derived handcrafted class without consequence. As discussed in Chapter 11, the next version of Visual Studio will include partial classes that allow you to split the noneditable portions without inheritance rules.

Changing Private Scope to Protected Scope

Because of the addition of the derived objects, it's reasonable to change the scope of many private properties and methods to protected. There are two scenarios to balance here. If you're deriving business objects from other business objects, you may need access to the fields that back the individual business properties (such as `ID` or `Name`). On the other hand, it's safer to force your handcrafted classes to use the properties and avoid direct field access. You can evaluate this in your own scenario and either make the backing fields protected or provide protected set methods. The second approach gives you more flexibility in the future because you can add code, such as validation checks, to the set method.

Several factors may drive the decision of what to make protected or private. An often-overlooked factor is the complexity of the displayed IntelliSense for the derived class. Limiting the scope of properties and methods you *don't* anticipate a programmer of the derived class using allows them to quickly find the methods you *do* expect them to use.

TIP In transferring an architecture into templates, you'll have to evaluate the scope of most private methods and properties. You need to do this because your business logic is now in a derived class and your plumbing is in the generated class.

TIP .NET languages don't currently provide property access with mixed visibility (for example, a public get accessor with a protected set accessor). To provide mixed visibility, use a read-only property with the visibility you want for the get accessor and create a separate method named something such as `Set<PropertyName>` with the visibility you want for the set accessor.

Including Parent Primary Key in Child Objects

If you have a containment hierarchy, should the child repeat the primary keys of the parent? For example, if the `Project` object contains a collection of `ProjectAssignment` objects, will the `ProjectAssignment` class include the `ProjectID` of the parent? There are at least three approaches. You can include the parent key, such as `ProjectID` in the child. Or you could not include it and pass the `Project` object to methods that need it such as an update method. Finally, you could include a reference to the parent object in the child (in addition to a collection of children in the parent) rather than include the parent key (`ProjectID`). This is largely a matter of style, and I prefer the first approach. The second approach is fragile if you need project information at unexpected times. The third is slightly more complicated, but it has the benefit of providing access to the entire object rather than just the primary key.

TIP If you have an existing architecture, you've probably already solved how you'll provide access to parent information from the child, and you can transfer this approach to templates if it's worked well for you in the past.

Tracking Whether the Object Is Loaded

It's helpful to know whether data has been loaded into an object. Unless you're doing dynamic loading (also called *lazy loading*) of objects, you'll rarely need this information outside the object because the object shouldn't be exposed until it's in a valid loaded state. On the other hand, code within the class may want to know whether the data is loaded.

Dynamic loading means you load additional information, generally additional information about children, only when you need it. This avoids preloading information and in some scenarios can have a significant positive impact on performance. However, if not carefully managed, dynamic loading can adversely impact performance, and there are other issues with it. For example, how will you cascade primary key changes to child records that aren't yet loaded? Ideally, only the business objects know what's loaded, and the first step to providing dynamic loading is to track the current loaded state. The templates in this book implement the `IsLoaded` state property, but I don't implement dynamic loading.[6]

TIP Tracking whether an object or collection is loaded with data is the first step to providing dynamic loading. The next step is to test whether an object is loaded when it's accessed and to automatically load it if needed.

CAUTION Dynamic loading can increase network traffic and database load. Don't assume dynamic loading will solve performance issues. Dynamic loading is especially valuable when users dive into a large database in a haphazard manner. It's least helpful when the amount of data retrieved is small or the path the user travels through the data is predictable (making preloading practical).

Handling Invalid Values

There are several approaches to ensuring that you only place valid values in your database and provide meaningful feedback for your users. You'll at least want to validate the data before saving. CSLA also supports an ongoing report of incorrect values with a list updated when you assign a value.

6. You can find an article on dynamic/lazy loading in CSLA on Rocky's site at `http://www.lhotka.net`.

The validation approach you use with code generation needs to work seamlessly with validation supplied via metadata (generally from database constraints and extended properties) and handcrafted validation. Validation requires flexibility. Although you may be able to automate the handling of the majority of your validation, you'll probably encounter some special validation you need to handcraft. Even if the initial release of your application doesn't include any custom validation, you may need to add handcrafted validation during maintenance. You could allow programmers to override the appropriate business object's *property* to add validation. A better approach is to supply an overridable validation method for each property named `Validate<PropertyName>`, such as `ValidateName`. The property set accessor can call this validation method rather than directly validating. The benefit of this approach is it gives the follow-on programmer one less place to mess things up and potentially offers the UI access to the validation routines.

CSLA includes a BrokenRules subsystem that supports an object-wide `IsValid` property, a collection of broken rules, and a string representation of the broken rules. This is good midlevel complexity, but you may want to modify the system further to supply validation information for individual properties[7] to support features such as ToolTips or use the Windows.Forms or Web Forms validators.

The templates in this book use overridable validation for each property and automate creation of two important aspects of validation—maximum length and whether a string field is required.[8] You'll probably expand this to accommodate the types of validation you need to perform. You can do this either by interpreting constraint information from your metadata or by overriding the `Validate<PropertyName>` method.

TIP Plan a validation strategy that supports easy overriding of validation to add handcrafted validation because you'll almost certainly need it.

Providing CanAdd, CanRemove, and CanEdit Properties

You'll need a security system to support your application. There are different ways you can handle security, and I won't discuss these variations because frameworks such as CSLA generally handle most aspects of security. Instead, I'll show how your templates can best support any of these security approaches. Your template's support of security will be in supplying the business logic behind who can do what—authorization.

7. You can find an article discussing validation enhancements in CSLA at Rocky's site at `http://www.lhotka.net`.

8. `IsRequired` and `AllowsNull` are different. If `IsRequired` is true, empty strings aren't allowed.

For example, you need to know the current user's rights within the business object when you're saving data in the `Save` method. You also need to know this in your UI. To supply this information, use the public shared `CanCreate`, `CanRetrieve`, `CanUpdate`, and `CanDelete` methods. These methods can use whatever security strategy you have in place to determine whether the user has the rights to perform the specific task and provide consistent information on the user's rights both to the business object and to the UI. In CSLA, these methods use the current principal and compare with an explicit list built during code generation (based on metadata information).

TIP Use the public `CanCreate`, `CanRetrieve`, `CanUpdate`, and `CanDelete` methods to hide or disable UI features not available for the current user.

Overriding ToString, Equals, and GetHashCode

Objects often override the implementation of the `ToString`, `Equals`, and `GetHashCode` methods. You need an algorithm that provides a good default, but a programmer may further derive from your methods. `ToString` should ideally provide a user-friendly string that identifies the specific object. `GetHashCode` must supply a contextually unique value for internal use. The primary keys offer a good default for `GetHashCode`, but you may want to modify `ToString`, perhaps to concatenate values from fields marked as `UseForDesc` if they are present.

It's tempting to use this:

```
Public Overrides Function GetHashCode() As Integer
  Return Me.ToString.GetHashCode
End Function
```

to return a hash code for the object. This isn't a good idea because you have no idea whether an overriding `ToString` method will return unique values—and many sensible overrides won't. Of course, the programmers could override the `GetHashCode` to return nonunique values, but it'll be easier for them to find the problem.

TIP Ensure that the `GetHashCode` values are unique within the context you're using them.

Creating Predictable Names

When you're creating classes by hand, you can be creative in your naming. It's often natural for naming to reflect the domain operation. For example, with a many-to-many table named `Assignment` that relates project to resources, you might name the business object `ProjectResource` because this defines the resources assigned to the project. You might also name the method that creates a `ProjectResource` record `Assign` because it represents the domain action of assigning a resource. Unfortunately, code generation can't predict this sort of naming, and to maximize automation in your code generation, you have to automate naming. To support automated naming, you want your business classes to reflect table names and relations where practical. For example, you could automatically create the name `ProjectAssignment` in this example because it's the concatenation of the parent and child table names. The method to create new records could be named `NewItem` based on the generic operation performed. How much adjustment you can make to your naming depends on your team and how far along you are in your design and development process.

TIP Where practical, use generic, operation-specific (instead of domain-specific) naming.

Supporting Multiple Criteria

The examples in this book retrieve the entire set of records in each table, such as all projects or resources. This is often an unworkable approach in the real world. Instead, you'll often retrieve a subset of data based on criteria. The same business object collection may have multiple criteria for retrieving different subsets of data.

Because there are several potential solutions to this problem (including stored procedures that take partial criteria, multiple collections, or multiple criteria per collection), I haven't implemented a solution. To help you solve it, I've moved responsibility for filling the command parameters for data retrieval into the criteria itself. This `AddParameters` method of the criteria class allows multiple criteria by removing knowledge of the specific criteria from the business object class. While I was modifying the `Criteria` class, I also switched from fields to properties for the criteria data. This change allows you to data bind these properties in your UI.

TIP If you're using different stored procedures for different criteria, define the stored procedure name in the `Criteria` object, rather than defining it in the business object. This allows you to use different stored procedures to retrieve data based on different criteria.

Using Internal Methods for Assignments

You'll frequently have a series of assignments setting or using the values of your business object's properties. These assignments move data between the business objects and UI, between the DataReader and business objects, or between the business objects and ADO.NET parameters. When you're doing code generation, you'll create these in a looping mechanism within your template (regardless of your template mechanism). You can leave these assignments in the method where they're used, or you can call a method containing them.

Although it's partly a matter of personal tastes, I find it more convenient to move these assignments into specialized methods. This allows the core method to be more consistent between classes, making it easier to find problems. At the same time, having the assignments from different contexts in close proximity makes it easier to compare these series when problems occur.

TIP Applying consistency to your task-based method often allows you to move the task-based methods into your framework. Although code generation of redundant code segments isn't a problem, it's easier to maintain code than templates. CSLA is already well factored with most repeating patterns moved into the framework.

Providing a Single Location for State Changes

When you're building classes by hand, it's easy to wind up with key state-changing mechanisms located in different locations in different classes. When you're doing code generation, it's easier to debug and easier to reuse template code when you place such state-changing operations in one location. A key state-changing operation in the middle tier occurs when you're loading data and the `Read` method of the DataReader changes the current location to the next record. The samples in *Expert One-on-One Visual Basic .NET Business Objects* call the `Read` method in the `DataPortal_Fetch` method of the root `Project` object and the

Fetch method of the ProjectResources collection. To provide a consistent location, I moved the Read statement into the AssignFromReader method of each business object, adjacent to the assignments.

Handling Special Types

Chapter 6 covered defining special types in some detail. Special types are a mechanism to allow you to define wrapper classes to provide special handling, such as null handling of dates. When creating the templates that implement the architecture, you'll need to use alternate methods for these special types. You should solve this entirely within the business objects, and the UI should contain few artifacts of whether you're using special wrapper classes. This allows your use of wrapper classes to evolve over the project life cycle. The example of special types in this chapter is the SmartDate of CSLA.

TIP To support special types, wrap the SQLDataReader (or OLEDBDataReader) in a custom class. This allows you to add a specialized method to correctly retrieve data of the special type data. In CSLA, this wrapped DataReader is called the SafeDataReader.

Changing CSLA to Accommodate Code Generation

CSLA, as presented in *Expert One-on-One Visual Basic.NET Business Objects,* has one basic incompatibility with code generation. This lies in how the business objects are instantiated. The mechanism CSLA used to determine the class to instantiate doesn't well work with derived handcrafted classes. Rocky has updated the code at his site http://www.lhotka.net (the November 1, 2003, update) so you can download his code that enhances CSLA to support code generators. Looking at the problem itself illustrates how adding derived classes may cause unexpected problems.

The issue arises from the following code in the Server.DataPortal class. This code always creates the class that contains the nested Criteria class—which is the generated class, not the derived handcrafted class:

```
Private Function CreateBusinessObject(ByVal Criteria As Object) As Object
    ' get the type of the actual business object
    Dim businessType As Type = Criteria.GetType.DeclaringType
    ' create an instance of the business object
    Return Activator.CreateInstance(businessType, True)
End Function
```

You could solve this by declaring the criteria in the derived handcrafted class. But that class should be as empty as possible to facilitate maintenance. Instead, changing how objects are instantiated in the framework lets you instantiate the derived class without moving the `Criteria` class. There are four parts of the change—three in the framework and one in the business object. The templates in this book reflect the required business object change, but these changes are discussed in this section to show how you use the framework changes.

CAUTION You must make the following changes to your CSLA framework code to run the templates in this book! You can download the updated CSLA framework that includes this change from `http://www.lhotka.net`.

To instantiate business objects of the correct class, you have to change how the type of business object is retrieved. Instead of looking at the type of the class where the criterion is nested, create a new base class for a criterion that contains a field for the object type. For an existing architecture such as CSLA, this base class serves two purposes. In addition to specifying the type of object to create, it lets you identify `Criteria` objects that are part of the new design because they inherit from `CriteriaBase`. This allows you to avoid backward-compatibility issues. The entire code of this class is as follows:

```
<Serializable()> _
Public MustInherit Class CriteriaBase
   Public ObjectType As Type
   Protected Sub New(ByVal ObjectType As Type)
      Me.ObjectType = ObjectType
   End Sub
End Class
```

The second change is to modify the `CreateBusinessObject` method of the `CSLAServer.DataPortal` class to use the `ObjectType` property of the `CriteriaBase` in determining which type to create. This happens only when the `Criteria` object derives from `CriteriaBase` to ensure backward compatibility. Once you've got the correct type, you can instantiate it. The new code is as follows:

```
Private Function CreateBusinessObject(ByVal Criteria As Object) As Object
  ' get the type of the actual business object
  Dim businessType As Type
  If TypeOf Criteria Is CriteriaBase Then
     businessType = Criteria.ObjectType
  Else
     businessType = Criteria.GetType.DeclaringType
  End If
```

```
  Return Activator.CreateInstance(businessType, True)
End Function
```

The third change is similar but made to the `GetObjectType` of the client `DataPortal` class. The new code is as follows:

```
Private Shared Function GetObjectType(ByVal Criteria As Object) As Type
     ' get the type of the actual business object
   If TypeOf Criteria Is Server.CriteriaBase Then
      Return CType(criteria, Server.CriteriaBase).ObjectType
  Else
       Return criteria.GetType.DeclaringType
  End If
End Function
```

The templates in this book already contain the third change. You need to modify the `Criteria` object of the generated business classes to derive from the `CriteriaBase` and pass the correct object type to the base class constructor. The resulting code is similar to this:

```
<Serializable()> _
Protected Class Criteria
   Inherits CSLA.Server.CriteriaBase
   <<declarations>>

   Public Sub New(ByVal ID As System.Guid)
      MyBase.New(GetType(Project))
      mID = ID
   End Sub
   <<remainder of class>>
End Class
```

Note that the `Criteria` class is still nested in the generated class (`genProject` in this case), but it specifies the derived handcrafted class, which is what you want to instantiate.

TIP At the same time you make these changes, you may also want to change the type used when you pass `Criteria` objects as parameters. I didn't make this change in the templates because I wanted to minimize my framework changes. If you'd like additional strong typing here, change the type in methods such as `DataPortal_Fetch` in both the template and the framework base class from `Object` to `CriteriaBase`.

Generating the Middle Tier

The templates that create business objects are based on the four models discussed in the section "Exploring the Four Base Models" and consist of several thousand lines of template code. It'd be very tedious to review all of that code and would offer very little benefit. Instead, I'll walk through one portion of code that illustrates what's involved in building middle tier business objects. As background to this, I'll show part of the input metadata in the section "Looking at the GenInput.xml Structure," and I'll show the target output for the portions of the templates in the section "Looking at the Business Object Target: genProject."

NOTE The metadata input for generating the middle tier is the output from the ORM process described in Chapter 6.

Looking at the GenInput.xml Structure

The input metadata for each object is extensive to provide flexibility in building the middle tier. Listing 8-1 shows the input, with redundant sections replaced with markers such as `[[privileges]]`, `[[more columns]]`, or `[[more properties]]`. You can use Listing 8-1 as a reference while walking through segments of the templates in later sections.

Listing 8-1. A Fragment of the GenInputMetadata.xml file That Drives the Business Object Templates

```
<orm:Object Name="Project" TableName="Projects" CollectionName="Projects"
          Caption="Project" Namespace="cslaMiddleTierTest" Generate="CRUDS"
          MapDataStructure="PTracker" TransactionForRetrieve="false"
          RetrievePattern="app[Object]_SelectOne" RetrieveSetPattern=""
          CreatePattern="app[Object]_Insert"
          UpdatePattern="app[Object]_Update"
          DeletePattern="app[Object]_Delete"
          Root="true" Collection="false" Child="false" CollectionRoot="true"
          CollectionChild="false" ObjectChild="false" ReadOnly="false">
  <orm:UseSPSet Name="Projects" TableName="Projects">
  </orm:UseSPSet>
  <orm:ChildCollection Name="ProjectAssignments"
          ObjectName="ProjectAssignment" ChildTableName="Assignments"
          Ordinal="1" Namespace="cslaMiddleTierTest">
     <orm:ChildKeyField Name="ProjectID" Ordinal="1"/>
```

```
    </orm:ChildCollection>
<orm:AllProperties/>
<orm:UseSPSet Name="Projects" TableName="Projects"/>
<orm:CreateSP Name="appProject_Insert" TransactionType="ADONET">
  <orm:Parameter Name="ID" Type="uniqueidentifier"/>
  [[ more parameters ]]
  [[ privileges ]]
</orm:CreateSP>
<orm:RetrieveSP Name="appProject_SelectOne" TransactionType="None">
  <orm:Parameter Name="ID" Type="uniqueidentifier"/>
  [[ privileges ]]
  <orm:SPRecordSet Name="Table" Test1="Here">
    <orm:Column Name="ID" Column="ID" SQLType="UniqueIdentifier"
        BaseNETType="System.Guid" NETType="System.Guid"
        Default="System.Guid.NewGuid" Empty="System.Guid.Empty"
        IsPrimaryKey="true" OriginalName="ID" Caption="ID" Ordinal="0"
        AllowNulls="false" MaxLength="-1" AllowDBNull="true"
        AutoIncrement="false" Unique="false" Test2="Projects" Test3="ID"/>
    [[ more columns ]]
  </orm:SPRecordSet>
  <orm:SPRecordSet Name="Table1" Test1="Here">
    <orm:Column Name="ProjectID" NETType="System.Guid" Caption="ProjectID"
        AllowDBNull="true" AutoIncrement="false" MaxLength="-1"
        Unique="false" Ordinal="0" Test2="Projects" Test3="ProjectID">
    </orm:Column>
    [[ more columns ]]
  </orm:SPRecordSet>
</orm:RetrieveSP>
<orm:UpdateSP Name="appProject_Update" TransactionType="ADONET">
  [[ parameters ]]
  [[ privileges ]]
</orm:UpdateSP>
<orm:DeleteSP Name="appProject_Delete" TransactionType="ADONET">
  [[ parameters ]]
  [[ privileges ]]
</orm:DeleteSP>
<orm:SetSelectSP Name="appProject_SelectAll" TransactionType="None">
  [[ privileges ]]
  <orm:SPRecordSet Name="Table" Test1="Here">
    <orm:Column Name="ID" Column="ID" SQLType="UniqueIdentifier"
        BaseNETType="System.Guid" NETType="System.Guid"
        Default="System.Guid.NewGuid" Empty="System.Guid.Empty"
        IsPrimaryKey="true" OriginalName="ID" Caption="ID" Ordinal="0"
        AllowNulls="false" MaxLength="-1" AllowDBNull="true"
```

```
          AutoIncrement="false" Unique="false" Test2="Projects" Test3="ID"/>
      </orm:Column>
      [[ more columns ]]
      </orm:Column>
    </orm:SPRecordSet>
  </orm:SetSelectSP>
  <orm:Property Name="ID" OriginalName="ID" Caption="ID" Ordinal="0"
          AllowNulls="false" SQLType="UniqueIdentifier" NETType="System.Guid"
          IsPrimaryKey="true" Column="ID" BaseNETType="System.Guid"
          Default="System.Guid.NewGuid" Empty="System.Guid.Empty"
          AllowDBNull="true" AutoIncrement="false" Unique="false"/>
  </orm:Property>
  [[ more properties ]]
</orm:Object>
```

Looking at the Business Object Target: genProject

I'll use the opening of the business object stylesheet's genCSLAEditSwitchable.xslt template to illustrate middle tier templates. The remainder of this template and the other templates repeat the same techniques. Listing 8-2 shows the opening portion of the target file for this template.

Listing 8-2. The Start of the genProject.vb-Generated Business Object File

```
Option Explicit On
Option Strict On

Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports CSLA.Data
Imports CSLA

<Serializable()> _
Public Mustinherit Class genProject
   Inherits BusinessBase
   Implements IBusinessObject

#Region " Class Declarations "
   Protected mIsLoaded As Boolean

   Private mID As System.Guid
```

```
    Private mName As System.String
    Private mStarted As SmartDate
    Private mEnded As SmartDate
    Private mDescription As System.String
    Private mDepartmentID As System.Int32

    Protected mProjectAssignments As ProjectAssignments = _
          ProjectAssignments.NewProjectAssignmentsChild()
#End Region

#Region " Business Properties and Methods "

    Public ReadOnly Property IsLoaded() As Boolean
       Get
          Return mIsLoaded
       End Get
    End Property

    Public ReadOnly Property ID() As System.Guid
       Get
          Return mID
       End Get
    End Property
    Protected Sub SetID(Value As System.Guid)
       mID = Value
    End Sub

    Public Property Name() As System.String
       Get
          Return mName
       End Get
       Set(ByVal Value As System.String)
          If (mName Is Nothing Or Value Is Nothing) OrElse _
                   mName <> Value Then
             mName = Value

             ValidateName()
             MarkDirty()
          End If
       End Set
    End Property
    Protected Overridable Sub ValidateName()
       BrokenRules.Assert("NameLen", "Project Name too long", _
                   mName.Length >50)
```

```
    BrokenRules.Assert("NameRequired", _
                "Project Name is required", mName.Length = 0)
End Sub

<< additional properties snipped >>

Public ReadOnly Property ProjectAssignments() _
               As ProjectAssignments
    Get
        Return mProjectAssignments
    End Get
End Property

Public Shared Function CanRetrieve() _
          As Boolean
    If Threading.Thread.CurrentPrincipal.IsInRole("Admin") _
          OrElse _
          Threading.Thread.CurrentPrincipal.IsInRole("Manager") _
          OrElse _
          Threading.Thread.CurrentPrincipal.IsInRole("public") _
          Then
        Return True
    Else
        Return False
    End If
End Function
```

Exploring the Business Object Stylesheet: genCSLAEditSwitchable

The opening portions of the genCSLAEditSwitchable stylesheet illustrate how I structure stylesheets, a few simple token replacements, and a number of complex output segments, including several supporting templates. Your templates won't all be complex, but I'm guessing that by now you've got a good handle on simple token replacement and that you're interested in seeing how to handle complex scenarios.

TIP If you're writing your own templates using brute-force or CodeDOM generation, you can use this walkthrough to understand the issues involved. Creating a parallel template with the other techniques will give you great experience for creating templates, even if you're using a different architecture.

Opening Declarations and Entry-Level Template

This template opens like ones you've seen earlier in this book with a standard header, a bunch of namespaces, and the import of the supporting stylesheet named CSLASupport.xslt:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
          xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:dbs="http://kadgen/DatabaseStructure"
          xmlns:orm="http://kadgen.com/KADORM.xsd"
          xmlns:msxsl="urn:schemas-microsoft-com:xslt"
          xmlns:net="http://kadgen.com/StandardNETSupport.xsd" >
<xsl:import href="CSLASupport.xslt"/>
<xsl:output method="text" encoding="UTF-8" indent="yes"/>
<xsl:preserve-space elements="*" />
```

NOTE The namespace for the Microsoft extensions to XSL Transformations (XSLT)—the `msxsl` prefix—is included to support creating explicit node-sets as described in the Chapter 6 sidebar "Variables Containing Sets of Nodes." The last namespace (`net` prefix) supports callbacks to utility methods of the generation harness such as `GetPlural` and `GetSingular`.

TIP You can add your own methods to the `Utility/Tools` class of the `Utility` assembly used by the generation harness and call them via XSLT callbacks.

The harness passes six parameters to the stylesheet. In addition to the name and standard harness parameters, the harness passes the `Root` and `Child` attributes of the selected node. These are Boolean values indicating the roles for the business object the template will create. The full list of parameters is as follows:

```
<xsl:param name="Name"/>
<xsl:param name="filename"/>
<xsl:param name="database"/>
<xsl:param name="gendatetime"/>
<xsl:param name="Root"/>
<xsl:param name="Child"/>
```

A stylesheet variable contains a list of properties for the object. Because this is a stylesheet variable, it's visible in all templates of this stylesheet and the imported stylesheet, as are the stylesheet parameter values. This avoids retrieving the node-set at multiple points in the stylesheet and guarantees a consistent property list. You set the stylesheet variable using this:

```
<xsl:variable name="properties"
              select="//orm:Object[@Name=$Name]//orm:Property"/>
```

The entry-level template sets the context for the remainder of the stylesheet and calls the high-level processing template:

```
<xsl:template match="/">
   <xsl:apply-templates select="//orm:Object[@Name=$Name]"
                        mode="Object"/>

</xsl:template>
```

Processing the High-Level Template

The high-level processing template shows the overall structure of the stylesheet and of the output file because each template corresponds to a region in the code output. I include a full import .NET list when I'm doing code generation because it reduces the resulting code's dependency on the project file specifics. The opening of the template and `Imports` list is as follows:

```
<xsl:template match="orm:Object" mode="Object">
Option Explicit On
Option Strict On

Imports System
Imports System.Data
Imports System.Data.SqlClient
Imports CSLA.Data
Imports CSLA
```

The generated class is serializable and can't be instantiated because it's marked as `MustInherit` (`abstract` in C#). If the metadata input contains an explicit base class in the `Inherits` attribute, the business object inherits from that class. If the `Inherits` attribute doesn't exist, the base class is `BusinessBase`. The code generation architecture assumes that if alternate base class is used, it

ultimately inherits from BusinessBase. The portion of the template that declares the class is as follows:

```
&lt;Serializable()> _
Public Mustinherit Class gen<xsl:value-of select="$Name"/>
<xsl:choose>
   <xsl:when test="string-length(@Inherits) > 0">
   Inherits <xsl:value-of select="@Inherits"/>
   </xsl:when>
   <xsl:otherwise>
   Inherits BusinessBase
   </xsl:otherwise>
</xsl:choose>
<xsl:if test="string-length(@Inherits)=0">
   Implements IBusinessObject
</xsl:if>
```

Once the class declaration is complete, the high-level processing template calls a series of templates, each of which creates a region in the output code. I'll walk through the ClassDeclarations and BusinessPropertiesAndMethods templates in the next sections. The templates called to complete the class are as follows:

```
      <xsl:call-template name="ClassDeclarations"/>
      <xsl:call-template name="BusinessPropertiesAndMethods"/>
      <xsl:call-template name="SystemObjectOverrides"/>
      <xsl:call-template name="SharedMethods"/>
      <xsl:call-template name="Constructors"/>
      <xsl:call-template name="Criteria"/>
      <xsl:call-template name="DataAccess"/>
      <xsl:call-template name="InternalMethodsAndProperties"/>
End Class
</xsl:template>
```

Declaring the Class

The ClassDeclarations template creates class-level variables, or *fields*. If the business object derives directly from BusinessBase, the template creates protected or private fields for IsLoaded, each property, and each child object. If the output class derives from another business class, the template doesn't create a field if the base business class also contains the field. In that case, the base class property is used:

```
<xsl:template name="ClassDeclarations">
<xsl:if test="string-length(@Inherits)=0">
   Protected mIsLoaded As Boolean
<xsl:variable name="inherits" select="@Inherits" />
<xsl:for-each select="$properties">
   <xsl:variable name="propname" select="@Name"/>
   <xsl:if test=
         "count(//orm:Object[@Name=$inherits]//orm:Property[@Name=$propname])=0">
   Private m<xsl:value-of select="@Name"/> As <xsl:value-of select="@NETType"/>
   </xsl:if>
</xsl:for-each>
<xsl:text>&#x0d;&#x0a;</xsl:text>

<xsl:if test="string-length($inherits)=0">
<xsl:for-each select="orm:ChildCollection ">
   Protected m<xsl:value-of select="@Name"/> As <xsl:text/>
      <xsl:value-of select="@Name"/> = _
      <xsl:value-of select="@Name"/>.New<xsl:text/>
      <xsl:value-of select="@Name"/>Child()<xsl:text/>
</xsl:for-each>
</xsl:if>
</xsl:if>
</xsl:template>
```

TIP Comparing the string length of an attribute with zero returns `false` if the attribute is missing, or it contains an empty string.

NOTE `<xsl:text>
</xsl:text>` inserts a line break to provide an empty line between the properties and child collection declarations.

TIP `<xsl:text/>` at the start or end of a line suppresses the line feed if it sandwiches the whitespace characters between XSLT elements.

The EmptyForColumn template supplies a default value if one is available and is simply this:

```
<xsl:template name="EmptyForColumn">
   <xsl:if test="string-length(@Empty)>0"> = <xsl:text/>
               <xsl:value-of select="@Empty"/></xsl:if>
</xsl:template>
```

I included this as a separate template because I suspected I may add more complexity to the concept of empty in the future and will want to reuse the template.

Outputting the Template for Business Properties and Methods

The BusinessPropertiesAndMethods template outputs the Business Properties and Methods region of the output file. This region starts by providing the properties for items with backing fields: IsLoaded, each property, and each child collection. Outputting this section is conditional, and it isn't output if the business object inherits from another business object that contains the property. The following section "Outputting Property Procedure" discusses the internal working of the Property template. The template that creates the Business Properties and Methods region is as follows:

```
<xsl:template name="BusinessPropertiesAndMethods">
#Region " Business Properties and Methods "
<xsl:if test="string-length(@Inherits)=0">
   Public ReadOnly Property IsLoaded() As Boolean
      Get
         Return mIsLoaded
      End Get
   End Property
</xsl:if>

<xsl:variable name="inherits" select="@Inherits" />
<xsl:for-each select="$properties">
   <xsl:variable name="propname" select="@Name"/>
   <xsl:if test=
         "count(//orm:Object[@Name=$inherits]//orm:Property[@Name=$propname])=0">
      <xsl:call-template name="Property"/>
   </xsl:if>
</xsl:for-each>
```

```
<xsl:if test="string-length(@Inherits)=0">
<xsl:for-each select="orm:ChildCollection">
   Public ReadOnly Property <xsl:value-of select="@Name"/>() As <xsl:text/>
           <xsl:value-of select="@Name"/>
      Get
         Return m<xsl:value-of select="@Name"/>
      End Get
   End Property
</xsl:for-each>
</xsl:if>
```

The `IsValid` and `IsDirty` flags return values dependent on the current object and its children. The `IsValid` flag is `true` only when the `IsValid` flag of the current object and all child objects are `true`. The `IsDirty` flag is `true` if the `IsDirty` flag of the current object or any of its child objects are `true`. The portion of the template that creates the `IsValid` and `IsDirty` properties is as follows:

```
Public Overrides ReadOnly Property IsValid() As Boolean
   Get
      Return MyBase.IsValid <xsl:text/>
<xsl:for-each select="orm:ChildCollection">
   <xsl:text/> AndAlso m<xsl:value-of select="@Name"/>.IsValid<xsl:text/>
</xsl:for-each>
   End Get
End Property

Public Overrides ReadOnly Property IsDirty() As Boolean
   Get
      Return MyBase.IsDirty<xsl:text/>
<xsl:for-each select="orm:ChildCollection">
   <xsl:text/> OrElse m<xsl:value-of select="@Name"/>.IsDirty<xsl:text/>
</xsl:for-each>
   End Get
End Property
```

TIP The bitwise and operator (`And` in VB .NET) is faster than the Boolean and operator (`AndAlso` in Visual Basic .NET) for simple values. However, Boolean operators are short-circuited—if the result can be determined from the first value, the second value isn't evaluated. When the second operation isn't cheap, as in this code, the Boolean operator is faster. The bitwise `Or` and the Boolean `OrElse` behave similarly.

The remainder of this region contains four shared methods that return whether the current user has a specific permission. These methods are only available if the class inherits directly from `BusinessBase` and aren't generated when it inherits from another business object because it assumes these methods are already defined in the base class and don't need to be regenerated. The section "Checking Privileges" discusses the `CanPrivileges` template. The template code to create one of the functions that determines whether the current user has privileges valid for retrieving data is as follows:

```
<xsl:if test="string-length(@Inherits)=0">
   Public Shared Function CanRetrieve() _
                As Boolean<xsl:text/>
   <xsl:for-each select="orm:RetrieveSP">
      <xsl:call-template name="CanPrivileges">
         <xsl:with-param name="privilegetype" select="'R'"/>
      </xsl:call-template>
   </xsl:for-each>
   End Function
```

NOTE The code generated by the `CanPrivileges` template wraps the `IsInRole` method of the .NET `Principal` object. The .NET `Principal` object is further enhanced in CSLA.

TIP In working with XSLT, you'll frequently use an `xsl:for-each` directive to shift the current context, even if you expect exactly one occurrence to be returned by the XPath expression.

The remainder of this template just contains similar blocks for `orm:CreateSP`, `orm:UpdateSP`, and `orm:DeleteSP`.

Outputting Property Procedures

Off-loading complex features into their own templates clarifies the structure of your stylesheet and makes it easier to work with them. In this case, I moved the code that creates a property into its own template. I put this template in the supporting stylesheet CSLASupport.xslt to facilitate reuse. As a result, I'm using it in

additional scenarios, such as for `Criteria` objects, which makes it a little more complex. The `incriteria` parameter indicates whether you're calling the template while you're building a `Criteria` class, and the `xsl:param` directive supplies a default of `false`.

As a general pattern for complex templates, it works well to assign variables at the start of a template and then use these variables for output later in the template. I use this pattern here, which lets me concentrate on correct creation of each variable and test that they're correct.

TIP XSLT variables can't be changed once they're created, so you can test the values after they're created and count on these values throughout the rest of the template or stylesheet.

In this template, a property is read-only if it's marked with a `ReadOnly` attribute or it's a primary key, and I store this value in the `readonly` variable. I declare the `type` variable and then use it in the XPath expressions within the declaration of the `usetype` variable. I might want this variable for other purposes, so I didn't create it within the `usetype` variable declaration. The `msgstart` variable is used later when the template creates `Validate<PropertyName>` method, such as `ValidateName`. The creation of these XSLT variables is as follows:

```
<xsl:template name="Property">
   <xsl:param name="incriteria" select="'false'" />
   <xsl:variable name="readonly">
      <xsl:if test="@ReadOnly='true' or @IsPrimaryKey='true'">ReadOnly </xsl:if>
   </xsl:variable>
   <xsl:variable name="type">
      <xsl:value-of select="@NETType"/>
   </xsl:variable>
   <xsl:variable name="usetype">
      <xsl:choose>
         <xsl:when test="//orm:SpecialType[@Name=$type and
                  orm:Property/@AccessVia]" >
            <xsl:value-of select=
                  "//orm:SpecialType[@Name=$type]/orm:Property/@AccessVia"/>
         </xsl:when>
         <xsl:otherwise><xsl:value-of select="@NETType"/></xsl:otherwise>
      </xsl:choose>
   </xsl:variable>
   <xsl:variable name="msgstart" select="concat(../@Name,' ',@Caption)"/>
```

NOTE The `readonly` variable contains the word *ReadOnly* or an empty string, allowing it to be output directly later in the template.

The `propget` and `propset` variables are a little tricky. These variables contain an empty string if the property isn't a special type. If the property is a special type, you access the real data via properties of the wrapping object contained in the class's field. Thus, these variables contain the dot separator (.) and the accessing property. If that's unclear, you might want to review the Chapter 6 section "Defining Special Types." Creation of the `propget` and `propset` XSLT variables uses named templates that grab the `Set` or `Retrieve` attribute of the special type's `Property` element. The template code that sets these variables is as follows:

```
<xsl:variable name="propget">
   <xsl:call-template name="GetMethod" />
</xsl:variable>
<xsl:variable name="propset">
   <xsl:call-template name="SetMethod" />
</xsl:variable>
```

Presetting the variables, especially for complex values, makes it easier to follow what's going on when you output the code. Because the `readonly` variable contains either the word `ReadOnly` (and a space) or an empty string, you can output it directly. The template that declares the property is as follows:

```
Public <xsl:value-of select="$readonly" />Property <xsl:text/>
      <xsl:value-of select="@Name"/>() As <xsl:value-of select="$usetype"/>
```

Outputting the `Get` portion of the property is simple token replacement:

```
Get
   Return m<xsl:value-of select="@Name"/>
           <xsl:value-of select="$propget" />
End Get<xsl:text/>
```

If the property is read-only, the property can't contain a set accessor portion. Testing for read-only requires that you test the length of the string because it contains the word `ReadOnly` rather than `true/false`.[9] Listing 8-2 shows that the set

9. XSLT returns `true` for a string that's nonzero length, but it's a subtlety that even experienced XSLT programmers might not have be thinking about as they work with your code. So it's simpler to compare with the expected value or check the length of the string rather than rely on the implicit conversion.

accessor of the property compares the value, assigns the value, tests validation, and marks it as dirty. Comparisons are problematic, as discussed in the next section "Comparing Values," so they're handled by a separate template. The template that creates the set accessor is as follows:

```
    <xsl:if test="string-length($readonly)=0" >
    Set(ByVal Value As <xsl:value-of select="$usetype"/>)
       If <xsl:call-template name="Comparison">
          <xsl:with-param name="first" select="concat('m',@Name, $propget)" />
          <xsl:with-param name="second" select="'Value'" />
          <xsl:with-param name="operand" select="'&lt;>'" />
          <xsl:with-param name="type" select="$usetype" />
       </xsl:call-template> Then
          m<xsl:value-of select="@Name"/>
                <xsl:value-of select="$propset"/> = Value
          <xsl:if test="$incriteria!='true'">
          Validate<xsl:value-of select="@Name"/>()<xsl:text/>
          </xsl:if>
          MarkDirty()<xsl:text/>
       End If
    End Set<xsl:text/>
</xsl:if>
End Property<xsl:text/>
```

NOTE Booleans in XPath have some rules that you may not be expecting. For example, if you convert a string to a Boolean, it's `true` if the length is greater than zero and `false` if it's zero. This means a string `"false"` converted to a Boolean will be `true`. To avoid surprising yourself or maintenance programmers, explicitly compare values in XPath expressions or use a function such as `string-length`.

If the property isn't read-only, output includes the `Validate<propertyname>` method for the property. This method automatically includes a test for maximum length and whether the property is required. You can add more BrokenRules checks by including them in your metadata (perhaps through extended properties on the data column in SQL Server) and changing this portion of the template to handle new conditions such as ranges. The `BrokenRules.Assert` method takes three parameters—the name of the condition, the text to display to the user if the condition is broken, and an expression that will indicate whether the value

is valid. The portion of the template that creates the `Validate<propertyname>` method is as follows:

```
<xsl:if test="$incriteria!='true' and string-length($readonly)=0">
   Protected Overridable Sub Validate<xsl:value-of select="@Name"/>()<xsl:text/>
      <xsl:if test="@MaxLength &gt; 0">
         <xsl:if test="string-length(@MaxLength) &lt; 6">
      BrokenRules.Assert("<xsl:value-of select="@Name"/>Len", "<xsl:text/>
                <xsl:value-of select="$msgstart"/> too long", m<xsl:text/>
                <xsl:value-of select="concat(@Name,'.Length >', @MaxLength)"/>
                <xsl:text/>)<xsl:text/>
         </xsl:if>
      </xsl:if>
      <xsl:if test="@IsRequired='true' and @NETType='System.String'" >
      BrokenRules.Assert("<xsl:value-of select="@Name"/>Required", "<xsl:text/>
                <xsl:value-of select="$msgstart"/> is required", m<xsl:text/>
                <xsl:value-of select="concat(@Name,'.Length > 0')"/>)<xsl:text/>
      </xsl:if>
   End Sub
   </xsl:if>
```

NOTE Validation in this architecture allows the field to be set to an incorrect value and doesn't raise an exception. It just sets the BrokenRules system to reflect any problems. This approach gives your UI flexibility in how it communicates problems to the users and supports recovery in scenarios where one field depends on another.

NOTE This design relies on programmers that override the validate method in a derived handcrafted class to call the base class `Validate<propertyname>` method or perform all required validation in the derived handcrafted class.

If the property is read-only, you don't want outside code to have access to the methods and properties, even code in derived classes. However, code in the derived classes needs to be able to set these values. You could provide this mixed mode access either by making the backing field protected or by providing a separate protected method that sets the value. The benefit of the separate set method

is that you can provide additional code, such as validation checks, in the future if the need arises. The part of the template that creates the `Set<propertyname>` method is as follows:

```
  <xsl:if test="string-length($readonly) != 0">
   Protected Sub Set<xsl:value-of select="@Name"/>(Value As <xsl:text/>
                  <xsl:value-of select="$usetype"/>)
      m<xsl:value-of select="@Name"/> = Value
   End Sub
  </xsl:if>
</xsl:template>
```

TIP Use a separate protected `Set<PropertyName>` method rather than making the backing field protected to provide more long-term flexibility.

Comparing Values

Comparing values isn't always simple in an automated scenario. The equals operator (=) doesn't work in many scenarios in Visual Basic because operators, such as the comparison operators, aren't overloaded. You could use the `Equals` method, but do you want to rely on it being implemented properly by all classes and structures?

The `Comparison` template demonstrates how to provide special comparison behaviors for unusual types you encounter. For the example database, the problem arises with Global Unique Identifier (GUID) fields because you can't compare these values with an equals sign in Visual Basic. The appropriate comparison methods are `op_equality` and `op_inequality`. This template checks for special comparison conditions and outputs a straight comparison if special conditions don't exist:

```
<xsl:template name="Comparison">
   <xsl:param name="first"  />
   <xsl:param name="second"  />
   <xsl:param name="operand"  />
   <xsl:param name="type"  />
   <xsl:choose>
      <xsl:when test="$type='System.Guid' and $operand='&lt;>'">
```

```
            <xsl:text/>System.GUID.op_Inequality(<xsl:text/>
           <xsl:value-of select="$first"/>, <xsl:value-of
                                select="$second"/>)<xsl:text/>
        </xsl:when>
        <xsl:when test="$type='System.Guid' and $operand='='">
            <xsl:text/>System.GUID.op_Equality(<xsl:text/>
            <xsl:value-of select="$first"/>, <xsl:value-of
                                select="$second"/>)<xsl:text/>
        </xsl:when>
        <xsl:otherwise>
            <xsl:value-of select="concat($first,' ', $operand,' ', $second)"/>
        </xsl:otherwise>
    </xsl:choose>
</xsl:template>
```

Because less than and greater than aren't meaningful for GUIDs, if they're passed, the normal less than and greater than operators are output leading to compiler errors.

TIP Add new conditions to this template if you encounter types that can't be compared using the normal = and <> (!= in C#) operators. Adding them to this template should result in them being handled correctly throughout your middle tier code generation.

Checking Privileges

The `CanPrivileges` template provides a block of code that determines whether the user has rights to perform a specific task. The privileges are available through the generated `Can<Operation>` methods of the business object such as `CanCreate`, `CanSelect`, and so on.

This template checks for four conditions. If the `privilegetype` isn't passed or is empty, the template can't produce meaningful results, so it outputs a message and code that always returns `false`. If the privilege is granted to public in the input metadata file, the template outputs code that always returns `true`. If there are no privileges defined, the template outputs code that always returns `false`. If none of these conditions are met, the template outputs code that tests whether the current principal is in one of the roles with privileges for the action. The template that outputs the four functions that check privileges is as follows:

```
<xsl:template name="CanPrivileges">
  <xsl:param name="privilegetype"/>
  <xsl:choose>
  <xsl:when test="string-length($privilegetype)=0">
      ' HARNESS: There's an error setting privileges in the code generation template
      ' NOTE: No security defined for action. No users have access.
      Return False
  </xsl:when>
  <xsl:when test="orm:Privilege[@Grantee='Public']">
      <!– Just a quick and dirty way to ignore these –>
      ' Everyone (Public) is allowed to perform this action
      Return True
  </xsl:when>
  <xsl:when test="count(orm:Privilege)=0">
      ' NOTE: No security defined for action. No users have access.
      Return False
  </xsl:when>
  <xsl:otherwise>
      If <xsl:text/>
    <xsl:for-each select="orm:Privilege">
      <xsl:text/>Threading.Thread.CurrentPrincipal.IsInRole("<xsl:text/>
                  <xsl:value-of select="@Grantee" />") <xsl:text/>
         <xsl:if test="position() != last()"> OrElse _
         </xsl:if>
    </xsl:for-each>
         <xsl:text/> Then
         Return True
      Else
         Return False
      End If
  </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Although there's some natural variation in how the remainder of this template and other templates are constructed, this section demonstrates many of the most important techniques. Much of the remainder of the templates are more straightforward token replacement. I selected this segment because it stretches the simple case of token replacement to use more complex criteria, often contained in separate templates.

Summary

Middle tier business objects generally follow four base models:

- Business objects represent individual business objects along with their data, which often corresponds to a single record in a database.
- Business object collections represent sets of business objects, often the children of a different business object.
- Lists present a simplified set of information, often used for search results.
- Lookups, also called *name-value pairs*, contain pairs of keys and the corresponding data, generally a description.

In converting a middle tier architecture to code generation, you're likely to consider the following:

- Adding derived classes to contain handcrafted code
- Operation-specific naming that you can automate
- Support for multiple criteria
- Locations for state changes

Adding a new derived class for handcrafted code impacts how you handle aspects of the generated plumbing class. This impacts how you handle the following:

- Constructors
- Scope of fields
- Location of parent primary keys
- Dynamic loading
- Validation
- Security

Samples in this section and in the downloadable code implement business objects for the CSLA architecture presented by Rocky Lhotka in his book *Expert One-on-One Visual Basic .NET Business Objects*. Benefits of the CSLA architecture include the following:

- A fully object-oriented model
- High performance
- Remarkable scalability to diverse deployment scenarios
- n-level per-object undo
- Tracking of validation problems with broken business rules
- Choice of ADO.NET or COM+ transaction model
- Multiple UI approaches (such as Windows and Web) from one set of business objects

Additional Reading

To find more information about the topics covered, try the following resource:

- *Expert One-on-One Visual Basic .NET Business Objects* by Rockford Lhotka (Apress, 2003)

EXHEART

CHAPTER 9

Generating a WinForms Application

Maximize the role of code generation by generating your Windows user interfaces.

WinForms (the Windows.Forms namespace) applications present additional challenges for code generation because the output code is tightly integrated into an extensive design-time and runtime library. The code you'd like to generate is initially interspersed with the code you need to handcraft. .NET contains features that allow code separation, but the approach may not be intuitive. This chapter covers WinForms fundamentals with designs that make user interface generation practical, provides solutions to specific issues, and walks through a portion of the downloadable templates. You'll also see how an additional task-specific transformation can simplify your templates.

Generating user interface code for .NET is difficult as follows:

- Layout details are interspersed with code you'd like to generate.
- Visual Studio locks controls in base forms and doesn't let you move them in derived forms using visual designer tools.
- You can't override event handling.
- Visual Studio .NET also generates code—overwriting portions of your code.
- Visual Studio .NET gives doesn't always give good error messages when your code fails to run at design time.
- More variability is likely between forms for different applications or organizations than infrastructure code.

User interfaces present two distinct types of customization. You may want to change the visual layout, or you may want to customize the behavior. This chapter shows you how to separate layout from other types of form customization so you can handcraft either one or both without affecting generations of the rest of the application. This supports a smooth transition in your project from fully autogenerated prototypes to a deployed system with complex forms handcrafted and the remainder generated. Understanding what you can do with code generation in WinForms gives you a basis for knowing when to pull complex forms out of code generation. To provide a smooth transition from prototypes to final code, you can initially generate all of your User Interface (UI) and then handcraft individual forms when they become too complex. There are so many UI variations that you may not be able to use the form layout I use as your base form. So you'll see user controls that lay out the data portions of your screens and can drop these onto a variety of form layouts.

In the narrow sense, this chapter shows specific techniques that widen the range where it's practical to use code generation in your UI. In a broader sense, this chapter is about generating useful code in a complex runtime and design-time environment.

NOTE There's an underlying theme in this chapter. To maximize the code you can generate, you need to recognize factors that drive code from generated code to handcrafted code. These factors are richer and more complex in user interfaces than in business objects, which is why there's a perception you can only generate simple forms. Recognizing driving factors allows you to isolate code dependent on different factors and handcraft smaller amounts of code in many scenarios. Driving factors tend to be subtle, so I've used the relatively nonabstract arena of user interfaces to illustrate them.

Reviewing WinForms Architecture

Successful code generation depends on working with a handful of nuances of WinForms. This isn't a tutorial about WinForms development, and if you haven't built WinForms applications, you can explore the tutorials in the Visual Studio help.

Forms in WinForms are classes and behave like any other class in .NET. Their special features (both at runtime and design time) result from the base classes they derive from and the interfaces they implement. For example, features of WinForms support using controls on the designer canvas (drawing

surface) and Properties window. Visual Support is so helpful that there sometimes seems to be magic behind the curtain, but it's actually just complex design and runtime use of normal .NET Object-Oriented Programming (OOP) behavior.

Looking at the Inheritance Hierarchy

All WinForms forms ultimately derive from `Windows.Forms.Form`, which itself derives from `Windows.Forms.Control`. You can have any number of intervening classes between `Windows.Forms.Form` and the form displayed at runtime. These intervening classes can each provide the following:

- A constructor that creates controls
- Methods and properties that override `Windows.Forms.Form` behavior
- Additional properties and methods
- Event Handlers
- Additional Events
- Shared methods and properties

None of these are required. You can have an empty intervening class that only serves to define a form as part of a set of forms. By deriving a set of forms from a particular base class, you can identify it as part of the set using the `TypeOf..Is` operator (C#'s `is`).

NOTE You can't make WinForms `MustInherit`. This is because Visual Studio needs to instantiate the base class forms at design time, and even Visual Studio can't instantiate objects from `MustInherit` class.

Constructors in WinForms create controls and set their initial values in the `InitializeComponent` method. The constructor and `InitializeComponent` method are located in the `Windows Form Designer generated code` region, also called the *spit code region*. This region contains a warning not to modify the contained code. When you're doing WinForms code generation, you'll create the spit code region as well as the rest of the class.

Exploring Design-Time Support

At design time, Visual Studio .NET deserializes information contained in your form layout and presents it in the Properties window you can edit. Deserialization is parsing the code to display object properties in the Properties window. Visual Studio needs both read/write access to the current form and read-only access to the properties and methods of the form class's base classes. As you make changes, Visual Studio writes these changes into code—also called *code serialization*. This method of serialization and deserialization is entirely generic and works with newly designed controls as well as those in the .NET Framework. The serialized information is one of the things contained in the spit code region of the WinForms class.

NOTE Visual Studio uses the `CodeDOMSerializer` to create the spit code region. In unusual circumstances, you may need to write your own serializer using the CodeDOM. Most of the CodeDOM techniques in Chapter 3 also apply to CodeDOM serialization.

You have to be cautious when altering the spit code region both because Visual Studio may overwrite your changes and because Visual Studio will be unable to display your form if there's a bug in this part of the class. Visual Studio also can't display the form when there's a bug in the base form because the base forms are instantiated at design time. This means base class form constructors, including everything in the `InitializeComponent` method, have to run successfully at design time.

Looking at Events and Methods in Derived Forms

Events behave differently than methods when you derive classes. You're probably familiar with derived classes overriding or shadowing base class methods. This is also called *hiding* and specified using the `Overloads` in Visual Basic .NET (VB .NET) or `overload` in C# where it's also the default behavior. Events don't provide this level of control. Each time you hook up a method to handle a specific event, the .NET runtime adds a delegate pointing to that method to a collection of *listeners* for that event. The collection of listeners is also called an *invocation list*. When something triggers the event, the .NET runtime invokes each delegate in the collection of listeners. They don't override or hide the previously attached events.

NOTE A delegate is just a reference to a function. Strictly speaking, that's a function pointer, but .NET delegates are a special type of function pointer that's *typesafe,* meaning you can't point it to an inappropriate function or invalid memory location. This protection is critical in making your application robust and providing adequate security.

NOTE In VB .NET, you generally hook up event handlers declaratively using the `Handles` clause of the method that handles the event. You can also use the `AddHandler` statement. In C# you hook up events by adding a new event handler object to the event using an overload of the += operator.

Because the .NET runtime invokes every delegate, if you specify a handler in a base class and a handler in the derived class, they'll both fire. This is the same behavior events have in any other .NET class.[1] You can't alter this behavior, but you can use a pattern that offers additional control. Your event handler can call a normal overridable method that corresponds to events on the form or class. Rather than reattaching events in derived classes, override these methods. The .NET Framework demonstrates this pattern in many classes, including the WinForms classes. You may create your own events, either within derived controls or for communicating between classes. If you create your own events, offer an overridable protected `On<EventName>` method for derived classes to use—unless it's totally illogical for a derived class to provide behavior for the event.

NOTE If you attach multiple handlers to the same event, it's important not to rely on the order they fire in. Although the order events fire in is currently predictable, the language's specifications[2] are clear that the order that events fire in is indeterminate and may change in the future. Using methods allow you to specify the order base class methods are called in, as well as avoid the redundant handler problem.

1. There's nothing wrong with deliberately using multiple event handlers, and I occasionally use multiple handlers, each accomplishing a different task. The problem arises when different event handlers have overlapping purpose and redundantly do the same subtasks.

2. Although Microsoft submitted only C# to a standards committee, both C# and Visual Basic .NET have a formalized design specification.

CAUTION To ensure events fire in addition to the methods, call the corresponding base class method, such as `MyBase.OnLoad`.

The second way to apply this pattern is subtly different and most easily seen with forms, but the same thing could happen in another type of class. A contained item (or subunit) of the form can kick off an action, rather than the form itself. Forms contain controls, and often when an event of one of these *controls* on a base form fires, it logically affects the *form*. An example is clicking a cancel button. You can add new overridable methods to the form without creating new events on the form. These new methods let you respond to this action without redundantly attaching events and managing the associated problems. These methods are little different because they don't parallel a corresponding form event, and you make up the names and parameters:

```
Private Sub btnCancel_Click( _
                    ByVal sender As System.Object, _
                    ByVal e As System.EventArgs) _
                    Handles btnCancel.Click
    OnBtnCancelClick(sender, e)
 End Sub
 Protected Overridable Sub OnBtnCancelClick( _
                    ByVal sender As System.Object, _
                    ByVal e As System.EventArgs)
    ' Do whatever
 End Sub
```

You can name these methods based on a specific control event that calls them as I did with the `OnBtnCancelClick` method, or you can name them for an abstraction of what caused the event. In this case, you could name the method `OnCancel`. The abstraction makes sense when several underlying actions should trigger the same even handler response by calling the same abstracted method.

NOTE If an exception occurs in one of the methods of the invocation list, the .NET runtime can't intelligently recover so it falls out of the invocation list. You can see this if you raise an exception in a `Format` or `Parse` event. Because this can cause weird problems, be careful raising exceptions within event handlers.

Implementing User Controls

A user control is a visual portion of a form that's contained within its own class. User controls have design-time and runtime support. They're excellent tools for generating the portion of each form specific to one business object as an independent unit. You can use user controls to define form fragments that you can put together in various ways depending on the parent form.

User Controls vs. Deriving Controls

There are several mechanisms for creating new controls in WinForms. The two most accessible techniques are deriving from existing controls and creating user controls. Each technique solves a different problem, and both can be helpful in code generation. A user control lets you combine many other controls to create a portion of a form. User controls generally have the same design-time support as WinForms, and it's fairly easy to morph user controls into forms and vice versa. User controls derive from `Windows.Forms.UserControl` and inherit general behavior from this class.

You can derive from existing controls when you want to supply additional behavior that's closely related to an existing control. For example, you could create a textbox that only accepted numeric input. You don't have a design time canvas that's designed for easily combining multiple controls. One of the benefits of deriving from existing controls is that you get all of the existing design-time and runtime behavior of the base control in addition to your enhancements.

During code generation, you'll generate user controls and may specify derived controls for specific scenarios, such as a specific control for all data of a particular data type.

User controls help you apply generic templates—such as those at the Web site—to your specific applications goals. You may not like the way the containing forms in the samples work, but the individual user controls may be adequate for many of your purposes. Either way, you can tweak the templates and generic forms.

CAUTION Visual Studio instantiates base classes of WinForms forms from the compiled version. If you change a base class you must recompile for these changes to be available.

Working with Data: Bind vs. Scatter

There are two ways to get your data into and out of forms. You can bind the data, or you can explicitly assign data to each control and explicitly retrieve it. Explicitly assigning data to user controls is called *scattering*, and explicitly retrieving it is called *gathering*.

Historically, data binding has been a poor solution for real business applications. Although there were numerous problems, the core problem was flexibility. .NET changed the picture by offering a viable data-binding scenario, and the sample templates use data binding. Benefits of data binding in .NET include the following:

- Simple definition of binding to any property at either design time or runtime
- Ability to bind to any object that supports the `IList` interface
- Easy combo and listbox manipulation
- Concurrency with in-memory data
- Automatic synchronization of controls linked to the same data source
- An easy way to display data for multiple rows using a data-bound grid

On the downside, data binding remains a little slower than explicit assignments, weird errors can happen if you fall out of the invocation list because of an exception, and the code isn't as intuitive as explicit assignments. For balance, I tend to use data binding when concurrency with the data source or synchronization of controls is valuable, and I try to use the same binding strategy throughout an application. Data binding code looks like this:

```
Me.txtCompany.DataBindings.Add(New System.WinForms.Binding( _
                "Text", Me.dsCustomers, "Customers.CompanyName"))
'etc
```

Scatter/gather code looks like this:

```
Protected Overridable Sub Scatter()
   txtCompany.Text = dsCustomers.Customers.CompanyName
   ' etc
End Sub
```

```
Protected Overridable Sub Gather()
   dsCustomers.Customers.CompanyName = txtCompany.Text
   ' etc
End Sub
```

It's extremely easy to generate both these styles of exchanging data between business objects and controls. So the actual effort required to generate data binding and Scatter/Gather is almost identical, and you can make this architectural decision on technical benefits of each approach.

Interacting with Visual Tools

The programmer customizing the application is likely to have to move controls around on a form. Visual Studio generates code on behalf of the programmer that contains the position and other properties of each control. Because Visual Studio generates this code *in response to the programmer's actions*, the code containing the positioning and other property information is handcrafted from the application generation perspective. That Visual Studio generates the code, rather than the programmer typing it in, doesn't change that this code belongs to the programmer. So you have to protect the spit code region across regeneration. Because the property information lies within a section of code you'd like to generate, you might consider overriding the form class the way you did for business objects and placing the control positioning in the derived class. But this doesn't work because the controls are declared in the class that defines their position, and you need the controls available to the generated code.

Developing Code Generation Strategies for WinForms

You could just re-create your current WinForms with code generation. The problem is that you're likely to have to pull your code out of code generation and maintain it by hand much more often than necessary. By slightly altering the strategies you learned earlier in this book, you'll be able to generate a much higher percentage of your code.

Many applications contain a handful of forms that are quite complex. This chapter is about maximizing what you can generate, but there will be a point of complexity where you'll find it easier to drop out of code generation. This complexity is often related to the visual layout of screens, including tasks such as containing user controls on different tab pages. You'll need to extend the techniques in this chapter to accommodate more complex controls, possibly combining generated user controls with a handcrafted containing form for some of these complex layouts.

Building a New Derivation Strategy

You can maximize code generation by recognizing and isolating driving factors that push you to handcraft class-specific code. User interfaces have two distinct driving factors—the visual layout and properties may need customization, or the behavior may need customization. Altering the derivation strategy lets you individually address each of these driving factors.

It's convenient to think of code generation in terms of responsibility. The programmer is responsible for writing handcrafted code, and code generation is responsible for the rest of it.[3] For business objects, there's one driving factor—you need to provide special behavior to the business object. Because there's one driving factor, you need one handcrafted class. In user interfaces, you have two driving factors. Supplying two handcrafted classes—corresponding to the two driving factors—lets you modify the behavior while still generating the layout, and vice versa. If you combine your code that handles these two issues into one file, you'll have to pull the entire file out of code generation to customize either aspect. The code that holds the layout isn't an empty class; it contains significant generated code by default (although the handcrafted class for behavior customization may initially be a nearly empty class).

NOTE In Chapter 8, you encountered a set of classes with multiple driving factors—and saw it solved using multiple classes. That's the derived business object architecture described in Figure 8-2. In that case, the driving factors were logical changes to the business behavior of the `Assignment`, `Project Assignment`, or `Resource Assignment` classes.

In building business objects, I used a derivation strategy incorporating two entity-specific classes as shown on the left side of Figure 9-1.This corresponds to one driving factor for handcrafting most business objects.[4] For WinForms forms and user controls, a three-class strategy with two handcrafted classes supports the two driving factors, as shown on the right side of Figure 9-1. This provides a separate class file for the layout changes and specialization.

3. That's a simplification because the programmer is always responsible—directly by programming or indirectly through templates.

4. If you create business objects that have more than one driving factor for handcrafting code, consider providing more than one handcrafted derived class.

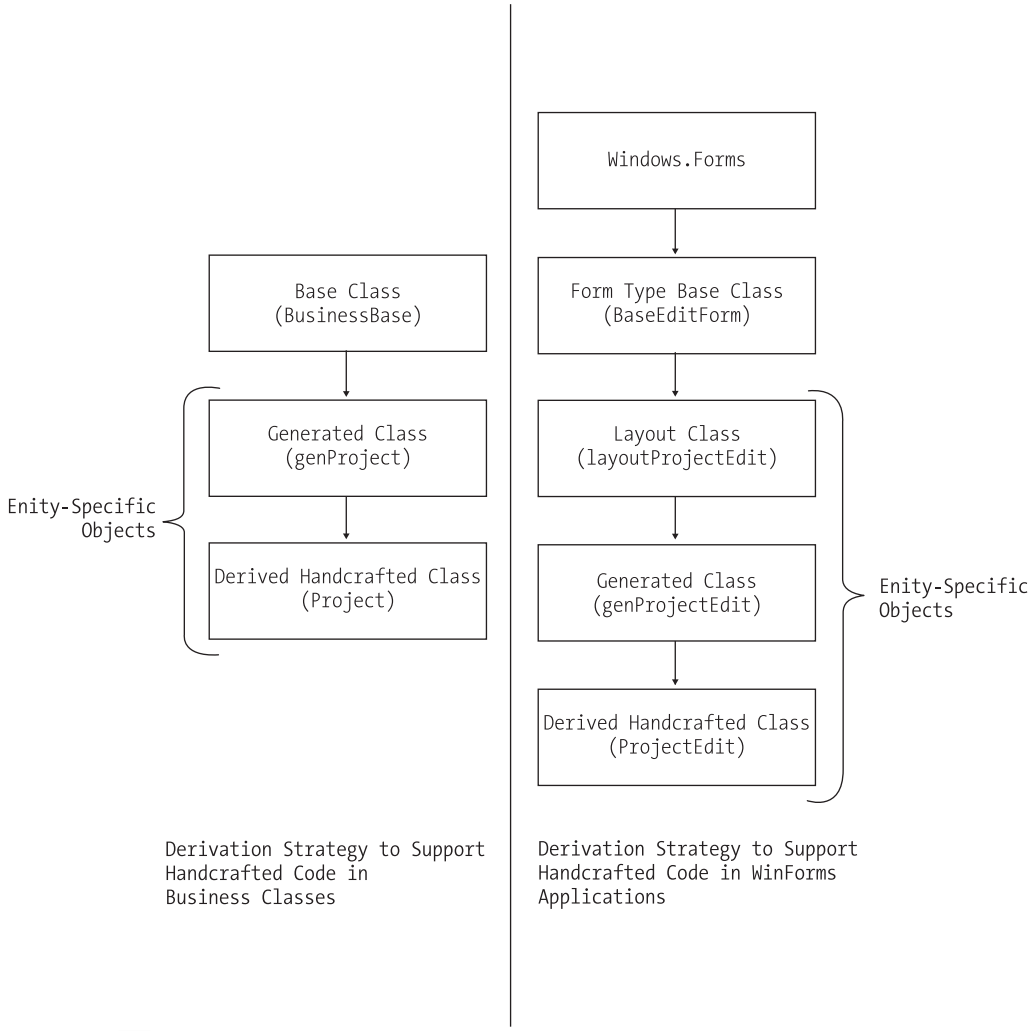


Figure 9-1. Comparing the derivation strategy used for business objects with the derivation strategy for WinForms components

The three-class architecture laid out on the right side of Figure 9-1 dedicates a class in the hierarchy to each of these tasks. The tasks performed by each of these classes are as follows:

- Layout
- Standard UI behavior, including plumbing
- Specialized UI behavior

You can generate the layout class and edit it when the default form layout is insufficient. The generated class handles all the plumbing and standard behavior, so you only need to work with the derived handcrafted class when you need specialized behavior—such as additional properties and methods that allow user controls and forms to interact with each other.

This strategy turns out to be important for another reason. You have to include the declaration of your controls in the class closest to the root in the derivation hierarchy that uses them. This is the only class where you can reposition them visually or supply values via the property dialog. If you generate binding, scatter/gather, or other plumbing behavior, you must have the controls declared in that class or higher, and you can't reposition these controls visually in the child class. Thus, you lose visual and property dialog support during customization in a two-class strategy. You have to pull the entire form out of code generation to make any visual changes. By having the potentially handcrafted layout class closer to the root, you solve this problem.

Like all compromises, this design isn't perfect. Although there are benefits, having distinct things in two different places can be a pain. You can minimize confusion from splitting the handcrafted code by following a set of guidelines for where to place specific code. My guidelines are as follows:

- Code all handcrafted *behavior* in the handcrafted derived class.
- Define all handcrafted *layout* in the base layout class.
- Define event handling methods in the base layout class.
- Have the event handling methods call parallel, protected, overridable methods.
- Break the third and fourth rules if the only change you're making is adding new events (don't pull base layout classes out of code generation just for these issues).

My goal in these guidelines is to maximize code generation and ensure only one event handler for each event and to hook up all these events in the same location whenever possible.

Preparing Metadata

The skill you use in managing metadata is a key factor of your success with code generation. As you saw in Chapter 2, part of managing metadata well is having it friendly to the code generation you're doing. The metadata you saw in Chapter 8 isn't particularly friendly to generating user interfaces. It's got three problems:

- It doesn't contain application-specific layout values such as margins.
- It doesn't know what type of control to use for each property type.
- It doesn't reflect handcrafted enhancements of your business objects, such as new properties.

Inserting the first two items earlier in the metadata stream wasn't appropriate because it's specific to WinForms and often to a specific application you're developing. The handcrafted enhancements aren't available until you've built the business objects. You can't ignore the handcrafted properties and methods you've added because you probably added them for use in the UI. Modifying the metadata prior to generating the UI solves these issues.

The section "Transforming Metadata with WinInputTransform.xslt" discusses how the book's templates use a metadata preprocessing step.

Merging Additional Metadata

If you enhance business objects by adding methods and properties in the handcrafted business class, your metadata is incomplete because it lacks these new features. UI code needs access to these handcrafted extensions. You added those methods and properties for a reason, and it was probably to display extra information to the end user. There are two approaches to solving this problem. You can re-create your metadata via reflection as discussed in the sidebar "Reflection-Based Metadata for UIs." You can also manually create a metadata file that includes the updated information. Because the metadata is the same for either approach, there's no difference in how you work with the metadata—only how you create it. I use a manual file in the examples on the Web site to clarify the technique.

The metadata is the same for the two approaches because you'll merge this new metadata into your existing metadata, so it matches that structure. In the example later in this chapter, a `DataGrid` includes a lookup value. Users don't want to see the numeric value; they want to see the text. Because I don't want to include a tutorial about solving `DataGrid` issues, the example uses a read-only grid that's edited in a popup form, so I don't need full combo box behavior—just the text.

NOTE To clarify what happens here, the handcrafted `RoleName` appears as a normal property on the object instantiated at runtime because it's instantiated from the derived handcrafted class that includes the property. But prior to incorporating this manual metadata, the metadata has no clue that this property exists.

To provide information about the handcrafted property, I add a new property element to the business object's element for the read-only `RoleName` property:

```
<?xml version="1.0" encoding="utf-8" ?>
<orm:MappingRoot xmlns:orm="http://kadgen.com/KADORM.xsd" FreeForm="true" >
   <orm:Assembly Name="BusinessLayer">
      <orm:Object Name="ProjectAssignment" >
         <orm:Property Name="RoleName" NETType="System.String"
                       Caption="Role Name"/>
      </orm:Object>
   </orm:Assembly>
</orm:MappingRoot>
```

To incorporate this property into the metadata so it's available for UI generation, I merge this manual metadata file into GenInputMetadata.xml. The `FreeForm` attribute of the root specifies how the intimate metadata merge treats the data. The contained data matches the organization of the metadata into which you're merging. The key addition is the `orm:Property` element. If you create a reflection-based metadata extraction, you'll want to match this format so you can smoothly incorporate it into your metadata stream.

Reflection-Based Metadata for UIs

You might want to automate inclusion of business object extensions into your metadata stream. The tool you'll use for this is reflection-based metadata extraction. Reflection is a .NET Framework feature that allows you to query for information about your compiled application.[5] You'll retrieve information via features such as the `System.Type` class's `GetMembers` method.

You'll want to keep a couple of things in mind when you design your reflection-based metadata. Reflection-based metadata doesn't natively include all the information you have available for existing properties and methods. You don't have as much information available in the class you're reflecting as you did in its original metadata. For example, metadata includes information about lookup fields that generally isn't included in the business object's generated code. To avoid a problem, use your reflected metadata as additional information to enhance your existing metadata, rather than a wholesale replacement.

The reflection information you have on new properties and methods isn't as rich as what you have for generated properties. For example, you don't have a caption or a maximum length for strings. If you need this information, you could combine reflection with manual UI metadata that includes these details,

5. This information is called *metadata* within the .NET Framework—which is confusing in this context. Metadata is data about data, which applies to both scenarios.

but if you're using reflection-based metadata, it's probably easier to use code attributes. Code attributes are custom .NET attributes you can place on any .NET item, such as a class or method. Include code attributes for information you want automatically incorporated into the metadata for the new methods and properties. You can then use reflection-based metadata extraction to create an Extensible Markup Language (XML) file with the new properties and their attributes.

Code attributes are distinct from the XML attributes contained in your metadata, so the semantics can be confusing, particularly because code attributes become XML attributes during reflection-based metadata extraction. For more information about code attributes, see the sidebar "Using Code Attributes to Specialize Behavior" later in this chapter.

Because you'll be merging reflection-based metadata on properties into existing metadata, the structure's already defined and you just match that format. If you're also merging methods—such as processes you want included as buttons in the UI—you'll have to develop a structure for this new type of information. Identify key information such as descriptive name, required parameters, any output, and how the process should appear in the UI.

The final thing to consider is that not all methods and properties in your derived handcrafted class should appear in your UI. The easiest way to accommodate this is probably a .NET code attribute such as `NoUI` or `NoWinUI` you can apply within the derived handcrafted class.

Transforming Metadata for Easy UI Generation

Business object metadata isn't friendly to generating UI code. For example, it doesn't include the control type to use for each property. Forms depend on a set of layout values, such as the vertical margin between controls that are generally consistent throughout an application but not consistent between application types (Windows and Web). Ideally, you could define the defaults once for WinForms applications in your organization and override them as needed in XML.

Calculations within your template set the positions of controls, and you may need to customize the input values for these calculations. These are attributes of the generation, not the underlying business objects. For example, you already use a specific default height for controls and margin between controls. You may not be aware of this because these defaults are normally supplied via Visual Studio's design tools. During code generation, you'll need to provide defaults, and you may want to customize them for all applications in your organization, for this application, or for just the form. These values are retrieved in a manner

analogous to a Cascading Style Sheet (CSS) in Hypertext Markup Language (HTML). For each item, the value on the most specific element is used.

Working with WinForms, you'll have a unique set of information associated with each property such as the corresponding control type and name. This isn't generalized information but information specific to how WinForms uses the underlying metadata. Finally, your application may have business objects that you don't want to display to the user. In the PTracker example, the `User` business object is used by other forms but shouldn't be displayed to users. You can solve all three of these issues by transforming the merged metadata.

NOTE A benefit of the script-based generation harness is that it's nearly free to add an extra metadata preparation step.

The metadata target for this transformation is as follows:

```
<orm:Property Name="Role" OriginalName="Role" Caption="Role"
              Ordinal="4" AllowNulls="false" SQLType="Int"
              NETType="System.Int32" IsPrimaryKey="false" Column="Role"
              BaseNETType="System.Int32" Empty="0" Parent="Role" IsLookup="true"
               LookupCollection="Roles" ControlType="WinForms.ComboBox"
              ControlPrefix="cbo" ControlName="cboRole" ReadOnly="false" />
```

Bolded items in this list are particularly important to UI generation. The WinInputTransform.xslt adds the `ControlType`, `ControlPrefix`, and `ControlName`. You could go on to remove attributes such as `SQLType` not appropriate for UI programming.

NOTE Be careful in how you remove attributes because you want new attributes entered through the manual Object Relational Mapping (ORM) metadata file (see Chapter 6) or SQL Server extended properties (see Chapter 2) passed through to the metadata UI programmer's use. You'll have to alter this transformation every time you add a new attribute if you pass only explicit attributes the transformation process. It's more flexible to explicitly remove unwanted attributes.

This WinInputTransform.xslt stylesheet includes the `ControlPropertyInfo` template, which determines the control type for each property. You'll probably further enhance this template to specify additional controls. The guidelines I used for establishing the control type are as follows:

- If the property is a foreign (child) key and the corresponding parent table element has an `IsLookup` attribute, the control type is ComboBox and the prefix is cbo.

- If the property `NETType` of the property is `DateTime`, the control type is DateTimePicker.

- Otherwise, the control type is TextBox.

Adding Manual Metadata for UI Generation

Now that you know how you'll use the manually entered UI metadata, what does it look like? These XML files are usually fairly simple; they may contain layout values on the root element. You saw the portion defining the handcrafted `RoleName` property earlier. I repeated it so you could see how this manual metadata file fits together. To include the `property` attribute, you have to define it in terms of its object and assembly as shown in the following sample:

```
<?xml version="1.0" encoding="utf-8" ?>
<orm:MappingRoot xmlns:orm="http://kadgen.com/KADORM.xsd" FreeForm="true" >
   <ui:UIRoot xmlns:ui="http://kadgen.com/UserInterface.xsd"
               VerticalMargin="1" >
      <ui:Forms >
         <ui:Form Name="UserEdit" NoGen="true"  />
         <ui:AllForms/>
      </ui:Forms>
   </ui:UIRoot>
   <orm:Assembly Name="BusinessLayer">
      <orm:Object Name="ProjectAssignment" >
         <orm:Property Name="RoleName" NETType="System.String"
               Caption="Role Name"/>
      </orm:Object>
   </orm:Assembly>
</orm:MappingRoot>
```

The first step in using this file is performing the standard intimate merge of this UIInfo.xml file into the GenInputMetadata.xml. Like any intimate merge, values of the UInfo.xml file overwrite any existing values in the file, so the `Caption` and `NETType` attributes specified override the values in GenInputMetadata.xml or add new elements where they don't already exist.

The second step is processing the WinInputTransform.xslt stylesheet to create the input metadata for UI generation. The WinInputTransform.xslt

stylesheet contains the `ControlTypeInfo` template discussed in the previous section and also contains the `UIInfo` template that incorporates layout values. The rules for the `UIInfo` template are as follows:

- Any attributes that appear on `ui:Forms` or `ui:Root` elements will appear as attributes on the resulting `ui:Form` elements.
- Default values are supplied for a set of required UI elements:
  - **Height**: The height of individual controls
  - **VerticalMargin**: The vertical margin between controls and the margin used at the top and bottom of forms
  - **HorizontalMargin**: The horizontal margin between controls and the margin used at the left and right of forms
  - **LabelWidth**: The default width of the column containing labels
  - **ButtonWidth**: The default width for buttons
  - **ButtonHeight**: The default height for buttons
  - **FormWidth**: The default width for forms
  - **FormHeight**: The default height for forms
- Attribute values cascade; values on a form element override any on a `Forms` element, which overrides attributes on the `UIRoot`, which overrides any defaults.
- There are no restrictions on the XML organization of the `ui:UIRoot`, `ui:Forms`, and `ui:Form` elements. None of them have to exist, and if they exist, they may appear in any order or nesting.
- If any `ui:Form` elements appear in the file, `ui:Form` elements won't be added, unless the `ui:AllForms` element also appears in the file.
- If there are no `ui:Form` elements, or the `ui:AllForms` element is included, new `ui:Form` elements are added. The forms list is determined by the XPath expression `//orm:Object[@Root='true' and string(@IsLookup)!='true']`. This means the transform will create `ui:Form` elements for any root objects that aren't marked as lookups.

The time spent preparing metadata with an extra transformation makes it easier to write and maintain templates. By pushing evaluations into a metadata transformation step, the same results are used in all templates, and they all respond to any updates you later make. Sure, this means more XSLT code to maintain in the transformation, but it simplifies later templates, such as UI templates.

Putting User Controls to Work

Overall form appearance and form functionality are likely to evolve in response to different pressures than your specific UI structure corresponding to each business object. You'd like to provide maximum flexibility, simplify organization, and minimize changes to your templates as the appearance of your forms evolves during your application life cycle. It's an easy goal to articulate, but it can be hard to implement unless you split your forms into a consistent framework of menus, buttons, status bars, and so on and the controls that represent the specific business object. Some layout features, such as menus and buttons, remain more or less unchanged between different forms in your application. These features give your application its look and feel. Splitting the superstructure of your forms into portions driven by overall look and feel from those specific to the data you're displaying requires two sets of techniques. Dynamic techniques are discussed in the next section, and user controls are discussed here.

The Nature of a Form

When I use the term *form* loosely, it's because the phrase brings to mind an image of your application's individual displays. Looking at a printout for any of your forms, you can draw lines around the portions representing the look and feel of your application, which generally repeat across different forms (at least within a category of forms). You can also highlight portions of each form that represent what's specific to the type of data (or business object) you're displaying. You can isolate the portion specific to each specific business object into user controls. You might have a user control for each object, collection, and list. The object user control might be a detail layout with individual controls for each property; the collection user control contains a grid or listview with full information, and the list contains limited information for selection.

You can respond to these independent factors by building a generic form, containing menus and form behavior, and use this form to house user controls that represent individual business objects. The generic form has a clear interface that

passes duties such as loading and saving onto the user control and its associated business object. The user control both displays data to the user and provides interfaces with the business object for important behavior such as validation. Once you've built individual user controls, building forms is just a matter of pasting these to create each form. Sometimes you'll need to manually combine these to build complex forms. In other cases, the combination is predictable, and you can combine them using dynamic techniques without manual intervention.

Using Code Attributes to Provide Specialized Behavior

Working with dynamic data, you often need more information about a class or method than normally provided. The easiest way to provide additional information is by using *code attributes*. You can place a code attribute on a class and retrieve the attribute by querying the class using reflection. An example of using code attributes is indicating the root classes in the Component-based, Scalable, Logical Architecture (CSLA) architecture. Code in the Multiple Document Interface (MDI) form can dynamically create an Action menu corresponding to each of these objects.

TIP I call these *code attributes* to differentiate them from XML attributes used in many other parts of the book.

Each available attribute is a class. Each time you apply the attribute to an item (usually a class or method) in code you create a new instance of this class, execute its constructor, and attach it to the item. When an object of the class is instantiated, the constructor of the specific attribute class is called. These constructors often accept parameters that are available later through properties of the attribute object. When you're working with code generation, you'll generally attach attributes to methods and classes in generated code and handcraft the corresponding attribute classes in your application's framework code.

The syntax for declaring the simplest attribute class is as follows:

```
<AttributeUsage(AttributeTargets.Class)> _
Public Class RootAttribute
   Inherits System.Attribute
End Class
```

NOTE This definition of a custom attribute class uses the `AttributeUsage` attribute.

This declares a custom attribute class applicable to classes. Although this class is simple, it's sufficient to flag root classes. By convention, attribute class names end with `Attribute`, but you don't include `Attribute` when you apply the attribute. For example, you specify this attribute (the `RootAttribute` class) as simply `Root`. You apply attributes by including them in angle brackets before the class or method name in VB .NET or in square brackets in C#. You can apply multiple attributes as a comma-delimited list. An example of applying attribute is as follows:

```
<Serializable(), Root()> _
Public Class Project
  Inherits genProject
```

TIP Including attributes on the line before the class declaration makes it easier to read your code—specifically making it easier to find individual classes and methods.

NOTE When you apply an attribute, you instantiate an object of the attribute class. Like any other object you instantiate, you'll need access to the underlying class, either as part of the current namespace or by referencing that namespace.

A template to generate a class with two attributes is as follows:

```
&lt;Serializable()<xsl:if test="@Root='true'">, Root()</xsl:if>> _
Public Class <xsl:value-of select="$Name"/>
  Inherits gen<xsl:value-of select="$Name"/>
```

NOTE You have to *escape* the opening angle bracket (the less-than symbol) so it's output rather than being interpreted as part of the template and creating an error.

You can retrieve code attributes by reflection. The `BuildMenus` method shown in Listing 9-1 later in this chapter adds any objects in the business object assembly that have a `Root` attribute to the Action menu. Because the business object assembly is already loaded, the method just needs to retrieve a reference to it, which it does via the `IBusinessObject` interface it contains. The `types` array contains all of the types within the assembly. To determine those marked with

the `Root` attribute, you've got to loop through the types and loop through the custom attributes of each type.

TIP When developing naming standards for attributes, use the `Is` prefix (such as `IsRoot`) only for attributes that take a Boolean value. Use the unprefixed form (such as `Root`) when the presence or absence of the template indicates the attribute. This is consistent with .NET Framework conventions such as the `Serializable` attribute.

Using Dynamic Techniques

There are two ways to approach user interface creation—establishing all details statically at design time or determining some portion of the details at runtime. If you've been building forms using the designers and property dialog boxes, you've been doing static designs. Adding controls at runtime is an example of dynamic user interface creation, but there are many other ways to make your form generic—dynamic menus, dynamic control positioning, and so on.

NOTE Adding controls at runtime for individual properties is rarely needed in a code generation application. However, the generic `EditForm` discussed in "Working with the Generic EditForm" adds the user control generated to edit the specific business object at runtime.

The downside of dynamic user interface creation is losing your design-time tools and getting slightly slower performance. Minor performance differences such as those caused by dynamic techniques are rarely an issue with WinForms because code runs on the client's computer, which is probably wasting cycles waiting for the user anyway. Losing design-time tools is a more serious problem. I solve this by generating a user control (which has full design-time support) and placing this user control onto a static form at runtime. The static form contains form-level behavior, such as menus and buttons.

The next sections show how to build dynamic menus based on the available root business objects and instantiate a generic form containing the appropriate generated user control using for editing.

Implementing Dynamic Menus

MDI menus and the methods that handle menu events are the only nongeneric code in most MDI forms. You can predict these menu options based on metadata. There are at least four ways you can supply this menu's details as follows:

- Handcraft the MDI form.
- Generate the MDI form including menu entries and menu handlers.
- Generate an XML file defining menu behavior and use reflection to fire handlers.
- Use reflection to generate the menu directly from available user interface elements.

Although all four approaches are valid, I prefer the last because it's very responsive to changes you make to your application while minimizing the amount of work. In the context of this book, it lets me demonstrate how you can combine code generation and dynamic techniques.

NOTE If you decide to generate your MDI form, you can use the derivation strategies such as those described in the section "Building a new Derivation Strategy" to maximize the code you can generate and limit the scenarios where you have to handcraft this class.

Using the dynamic menus approach lets the MDI form be generic—part of the infrastructure of the application. You might add additional features—perhaps Options or Help menu items—and you can include these changes throughout all of the applications that use your infrastructure. I derive the `Main` MDI form in the WinProject application from the more generic `BaseMain` MDI form located in the WinSupport project. I don't have any customizations to make, so the `Main` MDI form contains just Visual Studio-generated code. Because the form's already in place, I can customize it at any point in the future. These handcrafted classes will evolve because of different pressures—the base form evolving when the overall look and feel used by the organization changes and the derived form changing when the look and feel of the specific application changes or new features are needed.

NOTE Reusing user interface elements, including common menu elements of the MDI forms, is especially valuable to organizations that want all applications to present a common UI.

The base form supplies the status bar and the File menu containing Login and Exit options. By placing code in the base form, you can reuse it for other projects and more importantly have an avenue to easily update the code throughout your organization. For example, you could update all `Login` methods in your organization should security constraints change. The base form contains the method that builds the menu. Figure 9-2 illustrates this relationship using Unified Modeling Language (UML) diagrams. At runtime, the `OnLoad` method of the `BaseMain` MDI form calls the `BuildMenus` method when the user successfully logs in. Listing 9-1 shows the `BuildMenus` method.

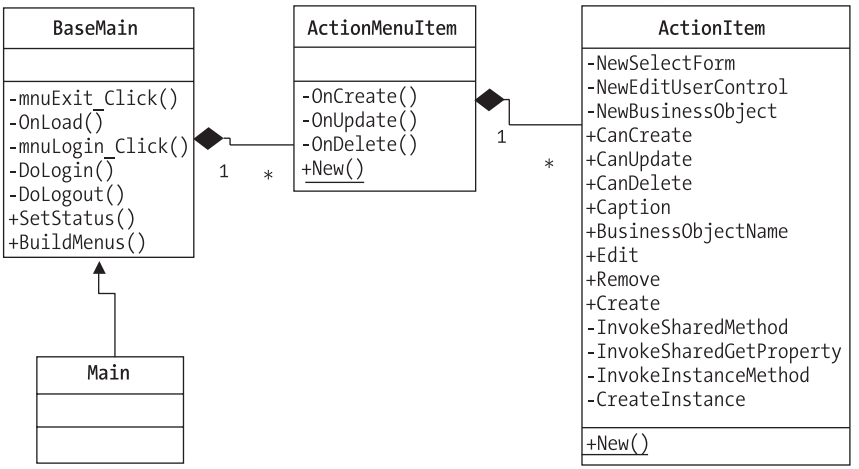


Figure 9-2. The relations between the MDI parent form and its supporting classes

In looking at the code in Listing 9-1, the `MainBase` MDI form requests the business object assembly's type from the application-specific `Main` form. It does this by calling the `BusinessObjectAssemblyList` method, which you should override in the `Main` form. Because you might have multiple business object assemblies, this method returns an array. Once you've got the assembly reference, you can

traverse the assembly's business objects to add items to the menu. A loop in the `BuildMenus` method traverses the types in the assembly looking for types with the `IsRoot` code attribute on the class. To determine whether a code element contains a particular custom attribute, you've got to loop through the custom attribute collection testing each attribute. It makes no difference whether you created the type through code generation or handcrafted it.

Listing 9-1. The `BuildMenus` *Method*

```
Protected Sub BuildMenus()
   Dim type As System.Type
   Dim asms As Reflection.Assembly() = BusinessObjectAssemblyList()
   Dim asm As Reflection.Assembly
   Dim types() As System.Type
   Dim attrs() As System.Attribute
   Dim bo As IBusinessObject
   Dim mnu As WinSupport.ActionMenuItem
   Dim frmSelect As System.Type
   Dim ucEdit As System.Type
   Dim selectFormType As System.Type
   Dim editUCType As System.Type
   Dim meType As System.Type = Me.GetType

   mnuAction.MenuItems.Clear()
   For i As Int32 = 0 To asms.GetUpperBound(0)
      asm = asms(i)
      types = asm.GetTypes
      For Each type In types
         If Not type.IsAbstract Then
            attrs = System.Attribute.GetCustomAttributes(type)
            For Each attr As System.Attribute In attrs
               If TypeOf attr Is RootAttribute Then
                  ' type belongs in menu
                  selectFormType = meType.Assembly.GetType( _
                              meType.Namespace & "." & type.Name & "Select")
                  editUCType = meType.Assembly.GetType( _
                              meType.Namespace & "." & type.Name & "Edit")
                  If Not selectFormType Is Nothing And _
                              Not editUCType Is Nothing Then
                     mnu = New WinSupport.ActionMenuItem(type, _
                              editUCType, selectFormType)
                     mnuAction.MenuItems.Add(mnu)
                  End If
               End If
```

```
                Next
            End If
        Next
    Next
    mnuAction.Enabled = True

End Sub
```

NOTE This code relies on careful naming to determine the forms to display for each business object. Alternatively, you could provide code attributes on the forms that specified the information needed to open them and code attributes indicating that they belonged in the Action menu. This approach is a bit more complex, but it's beneficial if you want forms in your menus that don't have corresponding business objects.

Once you've got the right business objects, you can create the menu item. In Listing 9-1, the `mnu` variable is an object of the `ActionMenuItem` class. The `ActionMenuItem` shown in Listing 9-2 derives from menu item and extends it to provide dynamic menu handling. The `ActionMenuItem` class largely delegates its duties to the `ActionItem` class. This delegation allows `ActionItem` objects to respond to any user action—for example, a menu click or a button click. The constructor uses properties such as the `CanCreate` property of the `ActionItem`. `ActionItem` properties in turn call the business object's `CanCreate` property to customize the menu for the specific user logged in.

Listing 9-2. The `ActionMenuItem` *Class*

```
Public Class ActionMenuItem
    Inherits Windows.forms.MenuItem
    Private mActionItem As ActionItem
    Private Const vbcrlf As String = Microsoft.VisualBasic.vbCrLf

    Public Sub New( _
                ByVal BusinessObjectType As System.Type, _
                ByVal EditUCType As System.Type, _
                ByVal SelectFormType As System.Type)
        mActionItem = New ActionItem(BusinessObjectType, _
                EditUCType, SelectFormType)
        Me.Text = mActionItem.Caption
        If mActionItem.CanCreate Then
```

```
            Me.MenuItems.Add("New " & mActionItem.Caption, AddressOf OnCreate)
        End If
        If mActionItem.CanUpdate Then
            Me.MenuItems.Add("Update " & mActionItem.Caption, AddressOf OnEdit)
        End If
        If mActionItem.CanDelete Then
            Me.MenuItems.Add("Delete " & mActionItem.Caption, AddressOf OnRemove)
        End If
    End Sub

    Private Sub OnCreate(ByVal sender As System.Object, _
                 ByVal e As System.EventArgs)
        mActionItem.Create(sender, e)
    End Sub

    Private Sub OnRemove(ByVal sender As System.Object, _
                 ByVal e As System.EventArgs)
        mActionItem.Remove(sender, e)
    End Sub

    Private Sub OnEdit(ByVal sender As System.Object, _
                 ByVal e As System.EventArgs)
        mActionItem.Edit(sender, e)
    End Sub
End Class
```

The `ActionItem` class is rather long, so Listing 9-3 contains just the key portion of it. The portion displayed is the `Edit` routine that lets the user select the record they want to edit and displays the edit form. All the code in this section is generic; objects instantiated from the `mEditUCType`, `mSelectFormType`, and `mBusinessObjectType` types provide all the specific behavior. Values for these class fields originate in the `BuildMenus` event shown in Listing 9-1 and are passed to the constructor, which I don't show here. The UML diagram in Figure 9-2 shows the full structure of the `ActionItem` class. The `NewSelectForm` method uses reflection to instantiate the form for selecting the item (such as the project) to edit. This entity-specific form derives from `BaseSelectForm`, and the base class `BaseSelectForm` includes the read-only `Result` property. `Result` is of type `Object` to accommodate the different types of primary keys that you might use for different classes. Passing the `Result` to the shared `Get<BusinessObject>` (such as `GetProject`) method of the business object class retrieves the business object to edit. The `Get<BusinessObject>` method is called via reflection. The generic `EditForm` provides editing for the corresponding business object by instantiating the correct user control based on `mEditUCType`.

CAUTION The current design of `Result` doesn't support compound primary keys made up of multiple columns.

Listing 9-3. The `Edit` *Method of* `ActionItem`

```
' Class level declarations
Protected mBusinessObjectType As System.Type
Protected mEditUCType As System.Type
Protected mSelectFormType As System.Type
Private mCaption As String
Private mName As String
Private Const vbcrlf As String = Microsoft.VisualBasic.vbCrLf

<snipped other code of the class including constructor that sets field values>

Friend Sub Edit(ByVal sender As System.Object, _
          ByVal e As System.EventArgs)
   Dim dlg As BaseSelectForm = Me.NewSelectForm
   Dim caption As String = Me.Caption
   Dim obj As cslaMiddleTierTest.IBusinessObject
   Dim frm As New WinSupport.EditForm
   Dim result As Object

   dlg.Text = "Edit " & caption
   dlg.ShowDialog(Windows.Forms.Form.ActiveForm)

   result = dlg.Result
   If Not Utility.IsEmpty(result) Then
      Try
         Windows.Forms.Cursor.Current = Windows.Forms.Cursors.WaitCursor
         obj = CType(Me.InvokeSharedMethod( _
                   Me.mBusinessObjectType, _
                   "Get" & Me.BusinessObjectName, result), _
                   cslaMiddleTierTest.IBusinessObject)
         frm.Show(Me.NewEditUserControl, _
                   Windows.Forms.Form.ActiveForm, _
                   obj)
         Windows.Forms.Cursor.Current = Windows.Forms.Cursors.Default
      Catch ex As Exception
         Windows.Forms.Cursor.Current = Windows.Forms.Cursors.Default
```

```
        Windows.Forms.MessageBox.Show("Error loading " & _
                caption & vbcrlf & ex.ToString)
    End Try
  End If
End Sub
```

NOTE You could modify the dynamic menus to use business object–specific forms, rather than the generic form with the business object–specific user control.

Working with the Generic EditForm

Once you can dynamically create a menu, the next step in simplifying your templates is providing a generic form to contain your user controls. Figure 9-3 illustrates the UML diagram of the generic `EditForm`. `EditForm` provides UI features as well as fill and save behavior, but only the user control interacts with the business object. Thanks to the strong framework (CSLA) the user controls work in, they have little to do and are generally just responsible for data-binding behavior.

`EditForm` works with common features of the `BaseUserControl` class so it doesn't need to use reflection. Listing 9-4 shows the portion of the `EditForm` code that implements its save behavior. The `btnSave_Click` method handles the event and merely passes the task onto the `OnBtnSaveClick` method, as I outlined in the section "Looking at Events and Methods in Derived Forms." This approach allows you to override the response to the `OnBtnSaveClick` event in a derived class, without redundantly handing the `btnSave_Click` event. The `OnBtnSaveClick` method calls `EndCurrentEdit`—necessary in some cases for correct WinForms behavior. The button's text might be either *OK* or *Save*, indicating to both the user and the program what happens when the button is clicked. The caption text is *OK* during the pop-up form display for editing children within the context of another form. In that case, the child form saves its data into the business object, but it waits for the parent business object to initiate the database save. If the caption is *Save,* the form instructs the user control to save its data back to the database.

Because the controls are bound, once the form calls `EndCurrentEdit`, the business object's data matches the user control data and `EditForm` can bypass the user control. Instead, it calls the `ApplyEdit` and `Save` methods of the business object.

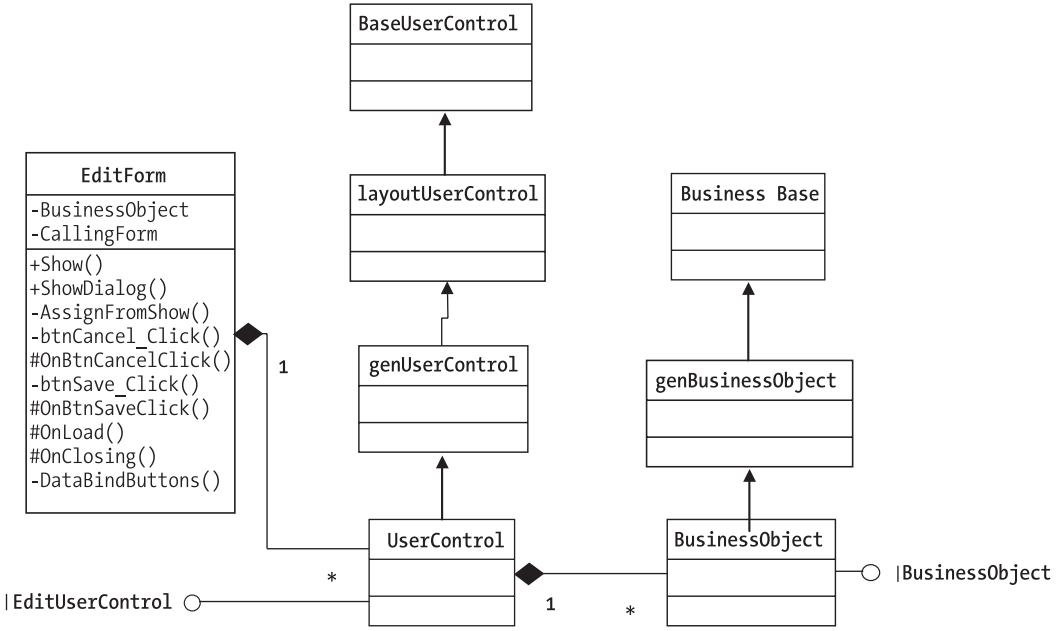


Figure 9-3. The relationship between `EditForm` *and the business-specific user controls*

Listing 9-4. Methods Implementing Save Behavior

```
Private Sub btnSave_Click( _
                    ByVal sender As System.Object, _
                    ByVal e As System.EventArgs) _
                    Handles btnSave.Click
    OnBtnSaveClick(sender, e)
 End Sub
 Protected Overridable Sub OnBtnSaveClick( _
                    ByVal sender As System.Object, _
                    ByVal e As System.EventArgs)
    Me.BindingContext(mObject).EndCurrentEdit()
    If btnSave.Text = "Save" Then
       Try
```

```
            Cursor.Current = Cursors.WaitCursor
            mObject.ApplyEdit()
            mObject = CType(mObject.Save, cslaMiddleTierTest.IBusinessObject)
            Cursor.Current = Cursors.Default
        Catch ex As Exception
            Cursor.Current = Cursors.Default
            MessageBox.Show(ex.ToString)
        End Try
    End If
    Hide()
End Sub
```

NOTE After the `Save` method, `mObject` points to a new business object, and the controls within the user control are no longer linked to the data in `mObject`. This isn't a problem because the form is immediately closed, but if you're manipulating them further, you'll want to rebind them or work exclusively with the `mObject` data.

Using Flow Layout in WinForms

You currently lay out your WinForms at design time with the Visual Studio designer using absolute positioning. That's great until you change the size of your font. Changing the font size on the form flows through to defaulted controls, which is good but not good enough because the controls aren't repositioned. I don't want programmers to have to pull their layout classes out of code generation just because they resize a form's font. There are two solutions—you can change the initial spacing by changing the `Height` property in UIInfo.xml, and you can apply flow layout within WinForms. I like the second approach because more applications should let the user select the font size.

NOTE I'm one of many people who struggle with my eyesight when looking at a computer monitor, and it isn't going to get better as I get more gray hair. I gravitate to applications that allow me to easily adjust the font size. Adjustable font sizes is a way you can help people enjoy working with your application.

The benefits of design-time layout include that you can work with the great visual designers in .NET. The benefits of runtime layout is that your form can be responsive. For example, you can respond to significant font size changes and can even vary label widths to accommodate differences in caption lengths when you localize to different cultures. The drawback of runtime layout is that you have to write some code to do it. However, with code generation, generation will write this code for you.

The flow approach I use depends on the following:

- The controls are in a two-column format within the container, usually labels and textboxes, and the left of the textbox should be at the right edge of the widest item in the first column.

- The controls are sorted from top to bottom and then left to right.

You can sort the controls by adding them to the control collection in the right order, or you can sort them in the panel layout event. Once the controls are sorted, you can just explicitly layout the controls. The code for this control is in the download.

Handling Primary Keys

You don't generally want your users to edit primary keys because if you allow edits, you'll have to support cascading key values in your middle tier and/or in your database. If you don't allow editing, you'll need to disable these keys, or hide them, during editing. I find enforcing this important enough that I set a `ReadOnly` attribute on the `Property` element in the WinForms metadata file for all primary key properties. This lets me use the generation templates to set the `ReadOnly` property of the TextBox control:

```
<xsl:if test="@ReadOnly='true'">
Me.<xsl:value-of select="@ControlName"/>.ReadOnly = True
</xsl:if>
```

I could alternatively hide the primary key with code such as this:

```
<xsl:if test="@IsPrimaryKey='true'">
Me.<xsl:value-of select="@ControlName"/>.Visiable = False
</xsl:if>
```

Exploring User Interface Templates

The UI sample at the Apress Web site roughly parallels the UI sample that Rockford (Rocky) Lhotka uses in his book on the CSLA architecture *Expert One-on-One Visual Basic .NET Business Objects* (Apress, 2003).[6] If you're familiar with that architecture, this UI allows you to see how code generation affects the user interface. However, I significantly expanded on this code to build a user interface shell that illustrates the code generation techniques covered earlier in this chapter.

The templates in this chapter rely heavily on the CSLA architecture. Chapter 8 talks about building the templates that interact with the CSLA framework. The benefit of the architecture is simplifying the work of the user interface. The more work you can push upstream in your development, the less overall work you'll do. That's because you can reuse upstream classes—stored procedures and business objects. You can rarely reuse code that's in line in your UI.

There are three different types of forms in the PTracker application. The application uses an MDI form, select forms, and edit forms. Generation of the select and edit form classes relies on a transformed version of the metadata specifically created for WinForms code generation. The section "Preparing Metadata" discusses value of this initial step.

Transforming Metadata with WinInputTransform.xslt

Two steps morph GenInputMetadata.xml into a format friendly for code generation. Chapter 6 covers the process that creates GenInputMetadata.xml. To prepare the metadata for UI code generation, you'll first merge it with a manual metadata class that contains UI information. The section "Adding Manual Metadata for UI Generation" covers building this metadata.

The second step is processing the WinInputTransform.xslt stylesheet that creates WinInputMetadata.xml. The WinInputTransform.xslt stylesheet starts with the standard opening goo that isn't shown here. This is followed by creating a series of stylesheet variables holding default values for the form layout. A value specified for `Forms` takes precedent over one specified in the `ui:UIRoot` element, and if neither is specified, the default value is used. The following stylesheet declarations include only the height variables declaration because the others are similar:

```
<xsl:variable name="defaultheight" select="20" />
<xsl:variable name="defaultvmargin" select="8" />
```

6. The full reference is included in Chapter 8's "Additional Reading" section.

```
<xsl:variable name="defaulthmargin" select="8" />
<xsl:variable name="defaultlabelwidth" select="80" />
<xsl:variable name="defaultbtnwidth" select="75" />
<xsl:variable name="defaultbtnheight" select="24" />
<xsl:variable name="defaultformwidth" select="750" />
<xsl:variable name="defaultformheight" select="450" />
<xsl:variable name="height">
   <xsl:choose>
      <xsl:when test="//ui:Forms/@Height">
         <xsl:value-of select="//ui:Forms/@Height" />
      </xsl:when>
      <xsl:when test="//ui:UIRoot/@Height">
         <xsl:value-of select="//ui:UIRoot/@Height" />
      </xsl:when>
      <xsl:otherwise><xsl:value-of select="$defaultheight"/></xsl:otherwise>
   </xsl:choose>
</xsl:variable>
```

The entry-level template calls the `PropertyControlInfo` and `UIInfo` templates and looks like this:

```
<xsl:template match="/ | @* | node()">
   <xsl:if test="mame()='ui:UIRoot' and count(//ui:UIRoot)=0">
      <xsl:element name="UIRoot">
         <xsl:call-template name="UIInfo"/>
      </xsl:element>
   </xsl:if>
   <xsl:copy>
      <xsl:apply-templates select="@*"  />
      <xsl:choose>
         <xsl:when test="name()='orm:Property'">
            <xsl:call-template name="PropertyControlInfo" />
            <xsl:apply-templates select="node()"/>
         </xsl:when>
         <xsl:when test="name()='ui:UIRoot'">
            <xsl:call-template name="UIInfo" />
         </xsl:when>
         <xsl:otherwise>
            <xsl:apply-templates select="node()"/>
         </xsl:otherwise>
      </xsl:choose>
   </xsl:copy>
</xsl:template>
```

The `PropertyControlInfo` template is straightforward, so I'll skip walking through it. Once these variables are set, creating the element follows the pattern used in Chapter 6 to output the default values only if there are no manually set values. The `runall` variable indicates whether to create `ui:Form` elements for the root `orm:Object` elements in GenInputMetadata.xml:

```
<xsl:template name="UIInfo">
   <xsl:variable name="runall" select="count(//ui:Form)=0 or //ui:AllForms"/>
```

Then, each `ui:Form` element in the manual UIInfo.xml file is copied to the output:

```
<xsl:for-each select="//ui:Form">
   <xsl:copy>
      <xsl:attribute name="Name"/>
```

TIP Explicitly include the `Name` attribute to place it first in the list of attributes. You can add the value later.

As each element is copied, all attributes from the `ui:UIRoot` and `ui:Form` elements are copied into it within the `CopyRootAttributes` template. This lets you add any values you like to the UIInfo.xml file and have them included in your code generation input. The `FormInfo` template creates, or overwrites the values of, attributes corresponding to the required UI layout values, such as `VerticalMargin`. Finally, the template copies any attributes and child nodes of the `ui:FormInfo` element you're copying. This design relies on overwriting existing values of an attribute with any new values. Thus, the cascade works because the more specific elements—`ui:Form`—overwrite the more general elements—`ui:UIRoot`:

```
<xsl:call-template name="CopyRootAttributes"/>
<xsl:call-template name="FormInfo" >
   <xsl:with-param name="name" select="@Name" />
</xsl:call-template>
<xsl:apply-templates select="@*"  />
<xsl:variable name="formtype" select="@FormType"/>
```

The `businessobject` variable is the form name unless the name contains the form type, such as `Edit`. Thus, if the form name were `ProjectEdit`, the corresponding business object would be `Project`. You can use the `NoGen` attribute in

filters in the harness script to avoid creating unnecessary forms. This code creates the `BusinessObject` and `NoGen` attributes:

```
      <xsl:variable name="businessobject">
          <xsl:choose>
             <xsl:when test="string-length($formtype)>0">
                <xsl:choose>
                   <xsl:when test="contains(@Name,$formtype)" >
                     <xsl:value-of
                               select="substring-before(@Name,$formtype)" />
                   </xsl:when>
                   <xsl:otherwise>
                      <xsl:value-of select="@Name" />
                   </xsl:otherwise>
                </xsl:choose>
             </xsl:when>
             <xsl:otherwise>
                <xsl:value-of select="@Name" />
             </xsl:otherwise>
          </xsl:choose>
       </xsl:variable>
       <xsl:attribute name="BusinessObject">
          <xsl:value-of select="$businessobject" />
       </xsl:attribute>
       <xsl:if test="//orm:Object[@Name=$businessobject and
                               @HideInUI='true']">
          <xsl:attribute name="NoGen">true</xsl:attribute>
       </xsl:if>
    </xsl:copy>
</xsl:for-each>
```

If the `runall` variable indicates you should add `ui:Form` elements for each object element, they're added in the `xsl:for-each` loop. If the `ui:Form` element doesn't yet exist, it's added. Like the previous element copy, I add the `Name` attribute first because I want it first in the resulting attribute list. The template then copies the root attributes and calls the `FormInfo` template to incorporate the required values:

```
   <xsl:if test="$runall='true'">
      <xsl:apply-templates select="//orm:Object[@Root='true' and
                                  string(@IsLookup)!='true']" mode="AddUIList">
         <xsl:with-param name="formtype" select="'Edit'" />
```

```
      </xsl:apply-templates>
      <xsl:apply-templates select="//orm:Object[@Root='true' and
                                  string(@IsLookup)!='true']" mode="AddUIList">
         <xsl:with-param name="formtype" select="'Select'" />
      </xsl:apply-templates>
   </xsl:if>
</xsl:template>
```

If you use the Harness9.xml harness script at the Web site, the output of this transformation is WinInputMetadata.xml. You'll use the output of this transformation in WinForms code generation. You can refer to the following sample element of the input metadata as you read the later section "Building the Edit Form Layout Template:"

```
<ui:Form Name="ProjectEdit" VerticalMargin="1" Height="20" HorizontalMargin="8"
                LabelWidth="80" ButtonWidth="75" ButtonHeight="24" FormWidth="750"
                FormHeight="450" />
```

Generating the MDI Form

The MDI form is generic and displays forms that derive from the `BaseEdit` and `BaseSelect` classes. One of the ways I modified the PTracker application was to make the MDI form more generic within the CSLA architecture. My goal was simple; it's hard to generate forms that have interspersed menu definitions. I could've solved this by generating menus in derived forms, but instead I switched to an approach based on reflection. The most significant benefit of this approach is that you don't need to repeat the security information already contained in the business object in the code that builds the menu. The section "Implementing Dynamic Menus" discusses building the action menu. Listing 9-1 shows the code for the dynamic menus.

Generating Select Forms

The select forms are fairly simple, and I don't anticipate you needing to change their layouts except by specifying the columns that appear in the grid. You'll change that via SQL Server extended properties or XML attributes in the ORM file by adding `UseForDesc` to the table columns you'd like to include. I anticipate you using this form only to select specific records, without any editing.

Because I didn't expect handcrafted layout changes, I used a two-class approach—with one template creating the `gen<objectName>Select` class that contains all of the code for displaying and plumbing the form. The other `<objectName>Select` class is ready for any customizations you need to make. Because of the two-class design, if you want to alter the layout of the form using visual tools, you'll need to pull the whole class out of code generation or alter the templates.

Generating Edit Forms

The edit form is inherently more complex, and I anticipate you needing to change the layout. Users like their forms arranged in a specific way that may not match my default. To accommodate this, I used the three-class strategy discussed in the section "Building a New Derivation Strategy." The three stylesheets are layoutWinEdit.xslt, genWinEdit.xslt, and WinEdit.xslt. The WinEdit.xslt and genWinEdit.xslt stylesheets use many of the techniques covered in Chapter 8.

Building the Edit Layout Template

The layoutWinEdit.xslt stylesheet faces new problems—particularly in generating the WinForms spit code region. So I'll walk through parts of this stylesheet. Because it's about 400 lines, I'll skip the repetitive parts in this walkthrough.

Declaring Parameters and Variables

After the opening goo of the template, a number of parameters and variables are declared. The template is called for each `ui:Form` element in the input XML. The `Name` parameter holds the name of the user control being created. The `BusinessObject` parameter contains the name of the business object in the input metadata. You'll need both of these values to build the user control. The stylesheet also uses the layout variables set in WinInputTransform.xslt:

```
<xsl:param name="Name"/>
<xsl:param name="filename"/>
<xsl:param name="database"/>
<xsl:param name="gendatetime"/>
<xsl:param name="Root"/>
<xsl:param name="Child"/>
```

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

<xsl:variable name="baseformname" select="'WinSupport.BaseEditUserControl'" />
<xsl:variable name="formname" select="$Name"/>
<xsl:variable name="objectname" select="$BusinessObject"/>

<xsl:variable name="height" select="//ui:Form[@Name=$Name]/@Height" />
<xsl:variable name="vmargin" select="//ui:Form[@Name=$Name]/@VerticalMargin" />
<xsl:variable name="hmargin" select="//ui:Form[@Name=$Name]/@HorizontalMargin" />
<xsl:variable name="labelwidth" select="//ui:Form[@Name=$Name]/@LabelWidth" />
<xsl:variable name="btnwidth" select="//ui:Form[@Name=$Name]/@ButtonWidth" />
<xsl:variable name="btnheight" select="//ui:Form[@Name=$Name]/@ButtonHeight" />
<xsl:variable name="formwidth" select="//ui:Form[@Name=$Name]/@FormWidth" />
<xsl:variable name="formheight" select="//ui:Form[@Name=$Name]/@FormHeight" />

<xsl:variable name="ctlleft" select="$hmargin + $labelwidth"/>
<xsl:variable name="ctlwidth" select="$formwidth - 1 * $hmargin - $ctlleft"/>
<xsl:variable name="childwidth" select="$formwidth - 2 * $hmargin"/>
<xsl:variable name="grptopmargin" select="$height"/>
<xsl:variable name="lasttop"
               select="count(//orm:Object[@Name=$objectname]/orm:Property)*
                       ($height+$vmargin) + 2* $vmargin"/>
<xsl:variable name="formminheight"
             select="$lasttop +
                     (count(//orm:Object[@Name=$objectname]/orm:ChildCollection) *
                     ($grptopmargin + (3 * $btnheight) + (7 * $vmargin)))"/>
<xsl:variable name="formminwidth"
             select="$labelwidth + 2 * $btnwidth" />
```

Using the Entry Templates

The entry-level template calls the `Object` mode template for the corresponding object:

```
<xsl:template match="/">
   <xsl:apply-templates select="//orm:Object[@Name=$objectname]"
                        mode="Object"/>
</xsl:template>
```

The `Object` mode template outputs the file header, delcares the class, and calls the `ClassDeclaration`, `WindowsFormsDesigner`, and `EventHandlers` templates to output the three regions of the form:

```
<xsl:template match="orm:Object" mode="Object">
Option Explicit On
Option Strict On

Imports System
Imports System.Windows.Forms
Imports System.Threading
Imports cslaMiddleTierTest

Public Class layout<xsl:value-of select="$formname"/><xsl:text/>
   Inherits <xsl:value-of select="$baseformname"/>
  <xsl:call-template name="ClassDeclarations"/>
  <xsl:call-template name="WindowsFormDesigner"/>
  <xsl:call-template name="EventHandlers"/>
End Class
</xsl:template>
```

Outputting Class Declarations

The `ClassDeclarations` template is very simple, just creating the minimum size field you can later access:

```
<xsl:template name="ClassDeclarations">
#Region " Class Declarations "
   Protected mMinimumSize As New System.Drawing.Size(<xsl:text/>
             <xsl:value-of select="$formminwidth" />, <xsl:text/>
             <xsl:value-of select="$formminheight" />)
#End Region
</xsl:template>
```

Generating the Spit Code Region

When you're doing visual development, Visual Studio outputs code into the spit code region for you. During code generation, you have to reproduce this region. If you haven't looked at before, it may seem a bit overwhelming because it's such a long block within your applications. Most of this code is simple assignments,

and generating this section isn't particularly difficult, particularly if you start from a working target form. In this section, you use the variables defined earlier to calculate features such as control positions and tab order. I want to output this region as much like the Visual Studio output as possible, including the comments. Because it's fairly long, I'll break it up with explanations of each portion of the template.

You output the user control's constructor and dispose event as plain text:

```
<xsl:template name="WindowsFormDesigner">
#Region " Windows Form Designer generated code "
   Public Sub New()
      MyBase.New()
      'This call is required by the Windows Form Designer.
      InitializeComponent()
      'Add any initialization after the InitializeComponent() call
   End Sub
   'Form overrides dispose to clean up the component list.
   Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
      If disposing Then
         If Not (components Is Nothing) Then
            components.Dispose()
         End If
      End If
      MyBase.Dispose(disposing)
   End Sub
```

The next section of the template declares each control. This takes two `xsl:for-each` loops—one looping through the properties of the business object and the other looping through the child objects. Each property requires two controls—a label and a textbox:

```
 'Required by the Windows Form Designer
Private components As System.ComponentModel.IContainer

<xsl:for-each select="orm:Property">
Friend WithEvents lbl<xsl:value-of select="@Name"/>
         <xsl:text/> As System.Windows.Forms.Label
Friend WithEvents <xsl:value-of select="@ControlName"/>
         <xsl:text/> As <xsl:value-of select="@ControlType"/>
</xsl:for-each>
```

Each child collection requires a GroupBox, DataGrid, and three buttons for adding, removing, and editing child elements. Finally, all forms include two additional controls—lstRules and chkIsDirty controls:

```
<xsl:for-each select="orm:ChildCollection">
Friend WithEvents grp<xsl:value-of select="@Name"/>
          <xsl:text/> As System.Windows.Forms.GroupBox
Friend WithEvents btnAdd<xsl:value-of select="@Name"/>
          <xsl:text/> As System.Windows.Forms.Button
Friend WithEvents btnRemove<xsl:value-of select="@Name"/>
          <xsl:text/> As System.Windows.Forms.Button
Friend WithEvents btnEdit<xsl:value-of select="@Name"/>
          <xsl:text/> As System.Windows.Forms.Button
Friend WithEvents dg<xsl:value-of select="@Name"/>
          <xsl:text/> As System.Windows.Forms.DataGrid
</xsl:for-each>
Friend WithEvents lstRules As System.Windows.Forms.ListBox
Friend WithEvents chkIsDirty As System.Windows.Forms.CheckBox
```

Building InitializeComponent

The InitializeComponent method is the tough part of creating your user control. The issues revolve around positioning controls. The XSLT code that outputs the control properties has to position the controls in relation to each other on the form—not an easy task for XSLT because of its restrictions on variables. This messy positioning code appears a little later in the template. The opening is straightforward—initializing the class-level variables that were just declared:

```
'NOTE: The following procedure is required by the Windows Form Designer
'It can be modified using the Windows Form Designer.
'Do not modify it using the code editor.
&lt;System.Diagnostics.DebuggerStepThrough()><xsl:text/>
            <xsl:text/> Private Sub InitializeComponent()
   <xsl:for-each select="orm:Property">
   Me.lbl<xsl:value-of select="@Name"/> = New System.Windows.Forms.Label
   Me.<xsl:value-of select="@ControlName"/> = New <xsl:text/>
             <xsl:value-of select="@ControlType"/>
   </xsl:for-each>
   <xsl:for-each select="orm:ChildCollection">
   Me.grp<xsl:value-of select="@Name"/> = New System.Windows.Forms.GroupBox
   Me.dg<xsl:value-of select="@Name"/> = New System.Windows.Forms.DataGrid
```

```
Me.mnuRoles = New System.Windows.Forms.ContextMenu
Me.btnAdd<xsl:value-of select="@Name"/> = New System.Windows.Forms.Button
Me.btnRemove<xsl:value-of select="@Name"/> = <xsl:text/>
          <xsl:text/>New System.Windows.Forms.Button
Me.btnEdit<xsl:value-of select="@Name"/> = New System.Windows.Forms.Button
Me.grp<xsl:value-of select="@Name"/>.SuspendLayout()
</xsl:for-each>
Me.lstRules = New System.Windows.Forms.ListBox
Me.chkIsDirty = New System.Windows.Forms.CheckBox
Me.SuspendLayout()
```

NOTE Suspending layout of the form and the group boxes speeds up the form definition and improves initial display performance.

Laying out the form starts with the properties. Remember, you can't reassign variables in XSLT, but you can create variables in any scope. The `top` variable in the loop that processes `orm:Property` elements goes out of scope and can be re-created each time you pass through the loop. The `position` XSLT function returns an index number or position of the current element in the current node-set. In this case, the node-set is the set of the child `orm:Property` elements of the current `orm:Object` element. The `tabindex` variable, used later to set the tab index, is the `position` times two because each property outputs two controls—a label and an edit control:

```
<xsl:for-each select="orm:Property">
<xsl:variable name="top" select="$vmargin+(position()-1)*
          ($height+$vmargin)"/>
<xsl:variable name="tabindex" select="(position()-1)*2"/>
'
'lbl<xsl:value-of select="@Name"/>
'
```

The `Location`, `Name`, `Size`, `TabIndex`, and `Text` are set for each label in the next block of code. Note the code that specifies the `Name` property. A double-quoted string in the output contains the literal `lbl` and `@Name` attribute of the property:

```
Me.lbl<xsl:value-of select="@Name"/>.Location = <xsl:text/>
          <xsl:text/>New System.Drawing.Point(<xsl:text/>
          <xsl:value-of select="$hmargin"/>, <xsl:text/>
          <xsl:value-of select="$top"/>)
```

```
Me.lbl<xsl:value-of select="@Name"/>.Name = <xsl:text/>
          "lbl<xsl:value-of select="@Name"/>"
Me.lbl<xsl:value-of select="@Name"/>.Size = <xsl:text/>
          <xsl:text/>New System.Drawing.Size(<xsl:text/>
          <xsl:value-of select="$labelwidth"/>, <xsl:text/>
          <xsl:value-of select="$height"/>)
Me.lbl<xsl:value-of select="@Name"/>.TabIndex = <xsl:text/>
          <xsl:value-of select="$tabindex"/>
Me.lbl<xsl:value-of select="@Name"/>.Text = <xsl:text/>
          <xsl:text/>"<xsl:value-of select="@Caption"/>"
```

NOTE Blocks similar to this appear for every control on the form, but I won't include each block.

NOTE Extra `<xsl:text/>` are to provide output that would be legal if you typed what appears, while still fitting in the margins of this book. If you work with a longer line length, you can remove some of these distracting `<xsl:text/>` elements for readability.

Creating the control is similar to the label, except the `AnchorStyle` property, which is similar to setting the `AnchorStyle` of the child `GroupBox` shown next.

Each child collection consists of five controls—three buttons, a `DataGrid`, and a `GroupBox`. Before outputting the code to set properties for these controls, I set the `lasttabindex` variable to the current `tabindex`. This just simplifies the inline calculations. The `Name` attribute within this loop is the attribute of the `ChildCollection` element. Setting the height of the group box (`grpheight`) is a bit complicated. If there are no child collection objects, the height doesn't matter, and I set it to zero. If there are child collections, the height of each group box depends on how many of children there are. You use the number of controls in each group box to determine the `TabIndex` and position for each control:

```
<xsl:variable name="lasttabindex" select="(count(orm:Property))* 2"/>
<xsl:variable name="grpheight">
   <xsl:choose>
      <xsl:when test="count(orm:ChildCollection) = 0">0</xsl:when>
      <xsl:otherwise>
```

```
        <xsl:value-of select=" round(($formheight - $lasttop) div
                count(orm:ChildCollection))" />
      </xsl:otherwise>
   </xsl:choose>
</xsl:variable>
<xsl:variable name="grpctlcount" select="5"/>
```

NOTE The slash (/) isn't a division symbol in XSLT; you have to use `div` for division.

With these variables set, you're ready to loop through the child collections. If you anchor all of the group boxes to all sides of the user control, things don't work when the user resizes the form. This template handles that by anchoring only the last `GroupBox` to the bottom of the user control:

```
<xsl:for-each select="orm:ChildCollection">
'
'grp<xsl:value-of select="@Name"/>
'
Me.grp<xsl:value-of select="@Name"/>.Anchor = <xsl:text/>
          <xsl:text/>CType(<xsl:text/>
          <xsl:if test="position()=last()">
             <xsl:text/>System.Windows.Forms.AnchorStyles.Bottom Or _
          </xsl:if>
          <xsl:text/>System.Windows.Forms.AnchorStyles.Top Or _
          System.Windows.Forms.AnchorStyles.Left Or _
          System.Windows.Forms.AnchorStyles.Right, <xsl:text/>
          <xsl:text/>System.Windows.Forms.AnchorStyles)
```

TIP If you have a variable number of controls, you can't use the techniques shown in these templates for setting the tab index. You can use .NET callbacks to create stateful values, but that's tricky. Instead, you can set the next `TabIndex` values based on the maximum number of controls you might have and have a large break in the `TabIndex` sequence.

Each of the other four controls for the child collection are added to the GroupBox's Controls collection, starting with the DataGrid. The Me prefix identifies the control in this XSLT, which might otherwise be difficult to read:

```
Me.grp<xsl:value-of select="@Name"/>.Controls.<xsl:text/>
          <xsl:text/>Add(Me.dg<xsl:value-of select="@Name"/>)
Me.grp<xsl:value-of select="@Name"/>.Controls.<xsl:text/>
          <xsl:text/>Add(Me.btnAdd<xsl:value-of select="@Name"/>)
Me.grp<xsl:value-of select="@Name"/>.Controls.<xsl:text/>
          <xsl:text/>Add(Me.btnRemove<xsl:value-of select="@Name"/>)
Me.grp<xsl:value-of select="@Name"/>.Controls.<xsl:text/>
          <xsl:text/>Add(Me.btnEdit<xsl:value-of select="@Name"/>)
```

Setting the Name, Location, Size, TabIndex, and Text properties of the group box are similar to setting these properties on other controls, such as the label shown previously, so I've snipped that portion.

Defining the DataGrid is similar to previous controls but also includes the DataSource definition. Because I set the actual DataSource in the DataBind method created by the genEdit stylesheet, it's just set to Nothing in the spit code region:

```
Me.dg<xsl:value-of select="@Name"/>.DataSource = Nothing
```

TIP It's already Nothing, so this isn't required for correct functioning—but it does help the spit code region this template generates match the one Visual Studio generates—making it easier for later programmers.

Calculating the TabIndex for controls within the GroupBox is a bit tricky. The position XPath function returns the index of the child collection within the Object element. Because this is one based, subtracting one gives zero. That means the last part of the equation is zero for the first element, and the TabIndex is one plus the lastindex:

```
Me.dg<xsl:value-of select="@Name"/>.TabIndex = <xsl:text/>
         <xsl:value-of
              select="$lasttabindex+1+(position()-1)*$grpctlcount"/>
```

Setting properties for the remainder of the controls uses similar techniques. You can explore this template if you need to change the layout of all of your forms. By convention, the last thing you set property values for is the user control itself, including ClientSize. You can see how precalculating the values simplifies the property assignment. In addition to setting these properties,

you've got to attach other controls to the form's `Controls` collection. All controls have to be attached to a container to be displayed at runtime. Controls within the `GroupBox` are already attached, so this final code in the `InitializeComponent` method just needs to attach the common controls (`chkIsDirty` and `lstRules`), the `GroupBox` controls, and the controls for the individual properties. I loop through the properties twice—once for the labels and once for the data controls. That lists them in the output code the same way Visual Studio listed them in my target sample:

```
'
' <xsl:value-of select="@Name"/>Edit
'
Me.ClientSize = New System.Drawing.Size(<xsl:text/>
          <xsl:value-of select="$formwidth"/>, <xsl:text/>
          <xsl:value-of select="$formheight"/> )
Me.Controls.Add(Me.chkIsDirty)
Me.Controls.Add(Me.lstRules)
<xsl:for-each select="orm:ChildCollection">
Me.Controls.Add(Me.grp<xsl:value-of select="@Name"/>)
</xsl:for-each>
<xsl:for-each select="orm:Property">
Me.Controls.Add(<xsl:text/>
          <xsl:text/>Me.<xsl:value-of select="@ControlName"/>)<xsl:text/>
</xsl:for-each>
<xsl:for-each select="orm:Property">
Me.Controls.Add(<xsl:text/>
          <xsl:text/>Me.lbl<xsl:value-of select="@Name"/>)<xsl:text/>
</xsl:for-each>
Me.Name = "gen<xsl:value-of select="@Name"/>Edit"
Me.Text = "<xsl:value-of select="@Name"/> Edit"
```

At this point, the form is ready to go. Earlier code in the method suspended the layout of the user control and any contained `GroupBox` controls. Before leaving the routine, you need to call `ResumeLayout` for the form display to be correct (try resizing a form without resetting these):

```
       <xsl:for-each select="orm:ChildCollection">
       Me.grp<xsl:value-of select="@Name"/>.ResumeLayout(False)
       </xsl:for-each>
       Me.ResumeLayout(False)
   End Sub
#End Region
</xsl:template>
```

NOTE You may want to restructure the `InitializeComponent` method to `ResumeLayout` in a `Finally` block. I didn't do that because I was matching the Visual Studio spit code region. If you put a `Finally` block in the spit code region, you'll want to experiment with how Visual Studio overwrites this region when you change the layout of the form via the visual designers.

Creating Event Handlers

The `EventHandler` template defines event handling for the child collection's `DataGrid` and the three associated buttons (add, remove, and edit). These handlers are nearly identical, so I'll show only the add button. Because these are associated with the child collections, they're created within an `xsl:for-each` loop that traverses the `ChildCollection` elements of the `Object` element. The `childobject` variable contains the name of the object, and the `childnamespace` contains the associated namespace:

```
<xsl:template name="EventHandlers" >
#Region " Event Handlers "
   <xsl:for-each select="orm:ChildCollection">
      <xsl:variable name="childname" select="@Name" />
      <xsl:variable name="childobject"
                select="//orm:Object[@CollectionName=$childname]/@Name" />
      <xsl:variable name="childnamespace"
                select="//orm:Object[@CollectionName=$childname]/@Namespace" />
```

The `btnAdd<childName>_Click` method handles the click event, meaning it runs when the user clicks the button. All it does is call the `overridable OnBtnAdd<childName>` method. The "Looking at Events and Methods in Derived Forms" section earlier in this chapter discusses this pattern. The `Overridable OnBtnAdd<childName>` doesn't do anything; it just establishes the method that the `gen<objectName>Edit` class will override. The `gen<objectName>Edit` class handles the plumbing and has access to the appropriate variables to handle these events correctly:

```
   Private Sub btnAdd<xsl:value-of select="@Name"/>_Click( _
                     ByVal sender As System.Object, _
                     ByVal e As System.EventArgs) _
                     Handles btnAdd<xsl:value-of select="@Name"/>.Click
      OnBtnAdd<xsl:value-of select="@Name"/>Click(sender, e)
   End Sub
```

```
   Protected Overridable Sub OnBtnAdd<xsl:value-of select="@Name"/>Click( _
                        ByVal sender As System.Object, _
                        ByVal e As System.EventArgs)
   End Sub
  </xsl:for-each>
#End Region
</xsl:template>
```

These sections represented the most interesting parts of the layoutWinEdit.xslt stylesheet. The remainder of the template repeats these techniques to create parallel elements, such as similar event handling blocks for the other buttons that allow you to add, edit, and delete child items in your form.

Building the Generated UserControl Template

The generated user control classes perform plumbing duties for the form hierarchies they're a part of. Much of this template is straightforward and uses techniques previously introduced. The `Databind<ChildName>` method for the child groups might be a little confusing. A `Databind<ChildName>` method is output for each `ChildCollection` element of the `Object` element. The output routine creates a `DataGridTableStyle` object and maps the tables and columns to the grid:

```
<xsl:for-each select="orm:ChildCollection">
<xsl:variable name="childcollection" select="@Name" />
Protected Sub Databind<xsl:value-of select="@Name"/>Grid()
   Dim dgtbs As New DataGridTableStyle
   Dim dgcs As DataGridColumnStyle
   With dg<xsl:value-of select="@Name"/>
      .SuspendLayout()
      dgtbs.MappingName = "<xsl:value-of select="$childcollection"/>"
```

To output the individual columns, the template loops across a subset of properties of the `Object` element corresponding to the `childcollection` variable, which contains the name of the current child collection. This technique lets you retrieve any information about the object corresponding to a child collection from the child collection name. Within this loop, the `Name` attribute is the property name:

```
      <xsl:for-each select=
            "//orm:Object[@CollectionName=$childcollection]//orm:Property
                  [not(@IsLookup='true')]">
      dgcs = New DataGridTextBoxColumn
      dgcs.MappingName = "<xsl:value-of select="@Name"/>"
```

```
        dgcs.HeaderText = "<xsl:value-of select="@Caption"/>"
        dgtbs.GridColumnStyles.Add(dgcs)
        </xsl:for-each>
        .TableStyles.Add(dgtbs)
        .SetDataBinding(m<xsl:value-of select="$objectname"/>.<xsl:text/>
                    <xsl:value-of select="@Name"/>, "")
        .ResumeLayout()
    End With
End Sub
</xsl:for-each>
```

Summary

Windows UIs present extra challenges for code generation. You're interacting with complex runtime and design-time libraries, and later programmers will probably need to edit the layout and provide special properties to the generated UI.

You can use four core strategies to maximize the role code generation plays in your Windows UI creation as follows:

- Use user controls to separate individual UI fragments—each corresponding to a business object—from a generic container form that defines how the form operates.
- Create metadata that's specific to your WinForms code generation. This metadata allows you to incorporate handcrafted changes to your business objects, specify control types, and precalculate values such as form layout margins.
- Use dynamic techniques to allow your UIs to respond at runtime to provide features such as menu and control positioning.
- Use a three-class strategy consisting of the following:
    - A base class declaring controls and containing layout code. This class derives from a WinForms class or a specialized class in your framework.
    - A generated class that derives from the layout class and manages the plumbing for the form or user control.
    - A derived handcrafted class for additional behavior or properties, including code that allows interaction between forms.

CHAPTER 10

Generating a Web User Interface

Extend the use of code generation into your WebForms user interface.

I'VE COVERED A LOT OF GROUND. You've seen how to extract metadata from sources such as your database, morph this metadata to describe business objects in an Object Relational Mapping process, build business objects, manage details such as project files, merge Windows-specific User Interface (UI) information into your metadata, and create a Windows UI. Creating WebForms leverages all of these techniques, with a few new twists. I'll focus on the twists, and not repeat previous techniques you've already seen, which means Chapter 9 will still contain useful information, even if you don't plan to do smart client development.

You can approach WebForms several different ways. For example, you could take a strict ASP.NET perspective, or you could do the opposite and rely heavily on .NET classes and the code-behind model. There are also many ways to solve details such as page transitions, state management, and so on. Because there are so many variations in what you might do in ASP.NET development, I'm expecting you to extend these templates, regardless of how hard I work on them. To give you a clean starting point, I've kept the templates simple.

The new things in this chapter include a new derivation approach and templates for a different language because ASP.NET provides new challenges for code generation. These techniques allow you to extend the UI techniques introduced for WinForms in Chapter 9 and let you handle a couple of wrenches Visual Studio throws into the works. I'm not going to cover core ASP.NET concepts best learned from a book focusing on ASP.NET, except a couple of sidebars on fundamental concepts such as state. Although I'll focus on ASP.NET, you can use similar templates for other browser output, such as directly generating HTML. The templates for Chapter 6 show many examples of complex Extensible Markup Language (XML) transformations, which you can use if you need to preprocess XML for output through the browser. If you're outputting XML directly to the browser using stylesheets, you can use static XSLT stylesheets or *generate* XSLT stylesheets based

on the user's requests. You'd use the XML outputting techniques described in Chapter 6 because XSLT is just a specialized version of XML. If you're working with Web Services, you might build a smart client as shown in Chapter 9 and add a wrapper to your business objects.

Reviewing ASP.NET

The WebForms side of ASP.NET offers a tool for building Hypertext Markup Language (HTML). HTML is the lingua franca[1] of Web browsers. Browsers accept the HTML descriptions of output, and the browser uses this to build the display. You or your users can look at the HTML that the browser is displaying by right-clicking and selecting View Source. HTML is similar to XML, but it isn't XML because it doesn't need to be well-formed. The browser will try to figure out what you meant, even if you break rules such as not closing elements.[2] Many HTML programmers routinely leave elements such as `<br>` unclosed.

WebForms run on your server to create the HTML output the client browser can display. WebForms consist of two parts—an ASPX page that describes your page in a markup language and the .NET code-behind files. Code-behind files are normal .NET classes that participate in an inheritance hierarchy with `Web.UI.Page` at the root and a class built from the ASPX markup as the most derived class. The code-behind class is the .NET class adjacent to the ASPX page in the hierarchy, the class that the ASPX is immediately derived from. Within the ASP.NET markup, you have to include some tags, such as the `<% Page...>` tag that identifies key features of the page to ASP.NET, including defining the name of the code-behind file. You can use code generation to build the ASPX markup file and the code-behind classes.

NOTE Behind the scenes, the compiler builds a class from the ASPX page's markup, and this class participates in the runtime hierarchy.

NOTE For user controls, discussed in the section "Utilizing User Controls," the root class is `Web.UI.UserControl`, rather than `Web.UI.Page`.

1. *Lingua franca* means the common language.

2. The XHTML standard that you are likely to see in the near future provides XML rules to HTML documents. This may take some getting used to, but tools will be able to catch mistakes in HTML pages. Happily, many tools are already adopting a well-formed approach to HTML.

Looking at the Differences Between Windows and Web

There are five major areas of differences between Windows and Web development as follows:

- State
- Postbacks
- Performance
- Navigation
- Security

I'm limiting my coverage of these key items, but these are things to consider in designing the architecture of your templates. I've provided several references in "Additional Reading" if you want more depth on ASP.NET. Forces such as the ease of creating WebForms, the small amount of information available on fundamentals, and pressure from management that only nominally understands how the Web works make it difficult to incorporate these fundamentals. My intent in this section is to shine a light on these key areas briefly to review the types of issues you'll settle in designing code generation templates for Web development. I discuss state in the sidebar "State, Stateful, and Stateless." I don't cover security because it's a subject that deserves focused attention and "Additional Reading" at the end of this chapter has several references for you to explore security further. The following sections discuss the other three issues—postbacks, performance, and navigation.

State, Stateful, and Stateless

State is data held in an object—the data values held in fields. Each object in your application can hold state, avoid using any state by having all required values passed, or reset state each time it's created.

ASP.NET clears the state from page objects when you send information to the browser. This state-clearing behavior is important to allow your Web server to handle a large number of users. For most applications, you need state, and for most Web applications you need to reset the state on every postback from the user. There are three ways to reset state: restore it from the postback itself, restore it from cache, or reload it from the database. You restore state from the postback when the objects are re-created. The state contained in the postback might be limited to values contained in the `HTTPRequest` such as the value of textboxes or more extensive information if you use ViewState. ASP.NET offers several important tools for server caching state on the server.

All methods of reconstituting state have an associated cost, so it makes sense to limit state to what you really need for your application. This is a consideration in designing your business objects, as well as how your WebForms pages manage state. If you express a large linked object model, you'll reconstitute a lot of state information for even a small change. On the other hand, a more atomic model may not represent your business practices. A third approach is to design your application to minimize state. This requires special handling of the database and business object. The user interface only needs to ensure that the correct information is round-tripped.

Generally, an update stored procedure expects you to pass data for all columns. To minimize state, you can use a bitwise flag to indicate what data the user changed.[3] Each column is a different position in the flagged enum such as the following:

```
Public Enum ColumnFlags
   ID = 2
   Name = 4
   ' Additional enum values
End Enum
```

NOTE You must explicitly state the values of a flagged enum so they are unique bits: 1, 2, 4, and so on. To generate these values in XSL Transformations (XSLT) templates, you can use a script, a recursive template, or .NET callbacks. In a recursive XSLT solution, it's easier to skip the value one, which is why it's missing in this code.

The business object containing the data has a class-level variable that reflects which property values have been changed based on the flagged enum. The set accessor for each property updates this value. A portion of the altered code of the business object class looks like this:

```
Private UpdateFlag As Int64
Private mName As String

Public Property Name() As String
   Get
      Return mName
   End Get
```

3. Thanks to Bill McCarthy for showing me this technique.

```
    Set(ByVal Value As String)
        mName = Value
        UpdateFlag = UpdateFlag Or ColumnFlags.Name
    End Set
End Property
```

NOTE You'll also need to reset the `UpdateFlag` to zero anytime data is loaded, saved, or cleared.

The stored procedure includes an extra parameter for the `UpdateFlag` containing the bitwise value that indicates which values have changed. This extra stored procedure parameter is used in the `SET` clause in the SQL `UPDATE` command as follows:

```
UPDATE Project
SET
     ID = CASE WHEN ((@UpdateFlag & 2) <> 0) THEN @ID ELSE ID END,
     Name = CASE WHEN ((@UpdateFlag & 4) <> 0) @Name ELSE Name END,
WHERE  ID = @ID
```

NOTE Because this isn't dynamic SQL, it executes quickly on the SQL Server and is more secure.

This technique allows you to update only data that you round-tripped to the client. You may not know which values the user actually changed, and that isn't significant. The advantage is updating the subset of data the user saw in the client browser and thus the only data they could have possibly updated. The client code—code outside the business object—is unaffected by this mechanism, except in reducing the state demands on the user interface design.

Because this code is generated, the enum is easy to maintain, but having to use hard-coded values in the stored procedure is a bummer. If you're also generating the stored procedure, you don't have to manage these values yourself. To maintain consistency between the stored procedure and the business object's enum, you'll want to include the flagged values early in the metadata generation step and include it in the XML metadata input used for generating both the stored procedures and the business object. This ensures that the same values are used in both locations.

By using this approach, you can generally limit state information to what's contained in the `HTTPRequest` and a couple of extra values. The extra values include primary key information, information to manage concurrency, and potentially parent key information. This technique works well with time stamp concurrency checking because you can round-trip the time stamp as a small item in the page's ViewState. It also works well if you're only interested in concurrency on the fields already round-tripped—those that the user can edit. It doesn't work as well if you're using value-based concurrency based on all fields because you'll have to pass all fields to the UI to reconstitute them during the postback.

State management is one of the inherent problems of the browser[4] and underscores that the Internet was designed for the delivery of information, and all round-tripping application implementations are hacked onto this. Smart client solutions avoid this problem by letting you maintain state on the client. But smart clients are still working toward the same ease of deployment browser-based solutions automatically enjoy. It'll also be a long time before smart client implementations achieve the phenomenal reach of browser-based solutions. So there are still great reasons to do WebForms development—you'll just have to match your environment with the best solution for managing state.

Looking at the Anatomy of a Postback

Closely linked to state, discussed in the sidebar "State, Stateful, and Stateless," is the concept of postbacks that's core to the request/response model of the Internet. The user, through the client machine's browser, makes a request to the Web server. Through the infrastructure of the Internet, that request arrives at your Web server—which is running Internet Information Services (IIS) if you're using ASP.NET. The IIS Web server determines whether a program it knows about can handle the extension associated with the request—.aspx for WebForms pages. These requests are routed to the ASP.NET engine.

Requests can arrive in two different contexts. They can be the initial request for the page or the page posting information back to itself. The `IsPostBack` property of the page lets the programmer know which scenario they're working in. The job of the ASPX and code-behind you write is responding appropriately to the user's request. If it's the initial display of the page, you fully build the page in HTML and send it out as the response to the user.

If it's a postback, you'll respond to the user's request using information in the reconstituted page and information you explicitly retrieve from application state.

4. Other problems include UI sophistication, performance effects of round-tripping, and server load because the server builds the user interface.

As examples of the kind of actions performed, you might load or save data. After completing any actions appropriate for the postback, you'll send a response to the user. This response might redisplay the page, display a different page (through a server transfer), or redirect the user to another page by suggesting the next page the browser request via `Response.Redirect`.

Between the initial building of the page and each postback, the server discards all information associated with that particular page. To reuse information across postbacks, you need a specific plan for how you'll store that data as state.

Improving Performance

You appropriately measure Web performance in terms of requests handled per second. The actual turnaround time for the user when your server isn't running at maximum capacity is less important than the point at which it maxes out and performance for all users plummets. This is not new. I know of a non-Web-based distributed timesheet application that failed to scale when it went into production because the designers failed to recognize that 90 percent of the work needed to be done between 11 a.m. and noon on Friday. Maximum load scenarios are the key to performance in many applications, but your Web server is creating UI as well as supplying data, so it has to work hard. Microsoft's scale-out vision can help, but you've got to consider server farm implications in your application design to later implement this architecture.

As you design your Web application and its templates, keep an eye out for potential performance bottlenecks. Yes, you can tune your application later in response to real issues, but if you design your application around an inefficient data retrieval model or a nonexistent caching strategy, reworking the application will be a difficult task. Performance is rarely a problem with code running on a smart client these days, but it remains a central concern to applications running on a server—and all the code of your ASP.NET application (except client scripting) runs on your servers.

Allowing Navigation

A WinForm application can have many forms simultaneously open, and the user can just flip between the many loaded forms. The browser supports only a single page at a time.[5] When the user navigates between pages, the browser sends a request to the server, which issues the new page as a response. The

5. You can fake multiple pages using frames.

page is reloaded *every time* the user interacts with the server.[6] In addition to being able to jump to other pages via hyperlinks you provide, users expect to be able to jump to other pages via browser features such as Back, History, and Favorites. Thus, navigation in a Web application requires attention because you don't easily know where the user is coming from.

You'll often want to control the page the user goes to next. You can do this transfer in three different ways. To specify where the user goes in response to hyperlinks, you'll use the `NavigateURL` property of the hyperlink control.

To send users to another page through code, you can use either the `Response.Redirect` or `Server.Transfer` methods. `Response.Redirect` sends a message back to the client browser that you're rerouting the user to another page. Assuming the browser is well behaved, it'll issue a request back to your server to load the new page. It's a bit inefficient to round-trip this message to the client, but it lets the client browser know what's going on so it can do things such as maintain history. `Server.Transfer` immediately sends the client to the new page server side without notifying the browser[7] and loads that new page in the client's browser. Other than skipping the round-trip, the big benefit of `Server.Transfer` is sharing information between the two pages, including any state you've reconstituted.

Traditionally, you handle the Uniform Resource Locator (URL) of the target page you're navigating to as a string. I hate strings, particularly for things such as navigation because I make frequent typos. I've also seen a project cling to a terrible directory structure simply because fixing the links was an overwhelming task. The section "Allowing String-Free Navigation" later in this chapter offers an alternative for managing navigation links. Code generation makes this approach practical because it can easily generate the required code.

Laying Out ASP.NET Pages

You lay out your ASPX pages in a variation of a markup language. This code can intersperse client-side script, HTML, and ASP.NET-specific directives. I won't cover scripting, except to remind you that you can either generate script directly in the ASPX page or build it as a string output at runtime. The second option can offer extra flexibility in generation when you need it.

---

6. It's more correct to say "anytime the user interacts with the server via an `HTTPRequest` for an HTML-based resource." Applications can interact with the server via JScript calls to Web Services, which doesn't reload the page.

7. Although it's more efficient than a redirect, a `Server.Transfer` has implications for proper loading of pages, particular when you're transferring to pages in a different directory within your application. Be sure you understand those implications before using it.

The markup of your ASPX page starts with a declaration indicating that it's an ASPX page. Listing 10-1 shows a sample ASPX page.[8] The later section of this chapter "Looking at WebSelect.xslt" shows the template that generates this code. The page declaration also specifies attributes that define the `Codebehind` file and the `Inherits` class. You'll often need to prefix class names with their .NET namespace in ASPX pages—such as in the `Inherits` attribute of this `Page` directive. If you're using any tag prefixes, you'll define them in a separate `Register` directive. In this case, the page includes a user control (`Header`) with the `uc1` tag prefix. Much of the rest of the file is HTML markup with WebForms controls prefixed with `asp:` controls.

Listing 10-1. A Sample ASPX Page for Selecting Records to Edit

```
<%@ Page Language="vb" AutoEventWireup="false"
                        Codebehind="ProjectSelect.aspx.vb"
                        Inherits="WebPTracker.ProjectSelect"
                        enableViewState="False"%>
<%@ Register TagPrefix="uc1" TagName="Header" Src="..\..\Header.ascx" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
   <HEAD>
      <title>Projects</title>
      <meta name="GENERATOR" content="Microsoft Visual Studio .NET 7.1"/>
      <meta name="CODE_LANGUAGE" content="Visual Basic .NET 7.1"/>
      <meta name="vs_defaultClientScript" content="JavaScript"/>
      <meta name="vs_targetSchema"
            content="http://schemas.microsoft.com/intellisense/ie5"/>
   </HEAD>
   <body>
      <form id="Form1" method="post" runat="server">
         <P>
            <uc1:Header id="Header1" runat="server"/>
         </P>
         <P>
            <asp:Label id="Label2" runat="server" Font-Names="Bauhaus 93"
                  Font-Size="XX-Large">Projects
            </asp:Label>
         </P>
         <P>
```

8. I'm including the full listing so you can refer back to it while reading "Looking at WebSelect.xslt." Details of this listing aren't important at this point.

```
            <asp:HyperLink id="hlHome" runat="server"
                  NavigateUrl="<%#
                        WebPTracker.NavigateURL.Home %>">
               Home
            </asp:HyperLink>
               
            <asp:LinkButton id="btnNew" runat="server" >
                     Add new Project
            </asp:LinkButton>
            <asp:DataGrid id="dg" runat="server" AutoGenerateColumns="False"
                     BorderColor="#CC9966" BorderStyle="None"
                     BorderWidth="1px" BackColor="White" CellPadding="4">
               <SelectedItemStyle Font-Bold="True" ForeColor="#663399"
                        BackColor="#FFCC66"/>
               <ItemStyle ForeColor="#330099" BackColor="White"/>
               <HeaderStyle Font-Bold="True" ForeColor="#FFFFCC"
                        BackColor="#990000"/>
               <FooterStyle ForeColor="#330099" BackColor="#FFFFCC"/>
               <Columns>
                  <asp:BoundColumn Visible="False"
                        DataField="ID"
                        HeaderText="ID"/>

                  <asp:ButtonColumn
                        DataTextField="Name"
                        HeaderText="Name"
                        CommandName="Select"/>
                  <asp:ButtonColumn Text="Remove" CommandName="Delete"/>
               </Columns>
               <PagerStyle HorizontalAlign="Center" ForeColor="#330099"
                     BackColor="#FFFFCC"/>
            </asp:DataGrid>
         </P>
      </form>
   </body>
</HTML>
```

I created this file with Visual Studio tools and modified it via the visual editor, with some changes made manually in the ASPX file. Once it was complete, I used it to create the template discussed in the section “Looking at WebSelect.xslt” and generated the code you see in this listing. Figure 10-1 shows the resulting page.

Figure 10-1. The Projects page

Utilizing User Controls

WebForms user controls let you encapsulate both the visual design of a portion of a page and its functionality in a single component, letting you reuse it throughout an application. WebForms user controls and pages are similar. Unlike pages, user controls must be located on a WebForms page, and the events raised by user controls and pages are slightly different. But they both aggregate a set of controls to present a logical unit of user interface behavior. In addition to making development easier, user controls let you segment the application display in a logical way. You can easily cache any part of the display that stays the same for a segment of your users. When this saves you from repeating expensive operations such as data retrieval, the performance improvement can be dramatic.

The trick to using code generation with user controls is how to specify the segmentation of the page. Often this segmentation follows the data retrieval patterns because the expensive operation is retrieving the data. When they don't, you'll need to specify how you're splitting the data in the UIInfo.xml file discussed in the section "Specifying UI Details in UIInfo.XML."

To keep the templates in this chapter as simple as possible and minimize infrastructure, I generate pages rather than user controls. You can easily alter the template to create user controls and create a generic page in your infrastructure to display them.

NOTE The "Putting User Controls to Work" section about user controls discusses other issues involving user controls and code generation.

Considering Server Controls

Server controls allow you to define how a control is rendered (output as HTML) and reconstituted from a later HTML response. You'll generally do this to provide extended behavior. You can also derive directly from an existing ASP.NET control—but you get a bit more flexibility if you use the server control route. In the past, I've even used server controls to specify the type of HTML control used at runtime—allowing the layout designers to use a generic control and the final controls to reflect database details such as the type and size of the field.

Exploring Code Generation Strategies for ASP.NET

Specific code generation strategies can increase the effectiveness of your ASP.NET code generation. The two key ASP.NET strategies I'll discuss in depth are a strategy for deriving classes and an approach to navigation that minimizes the use of strings. I'll also discuss a UIInfo.XML input file approach that's quite similar to the one in Chapter 9 and briefly cover strategies that take advantage of user controls.

Establishing a Derivation Strategy

Features of Visual Studio's handling of ASP.NET WebForms creates challenges for code generation. Responding to these challenges forces you to create a rather ugly derivation scheme to manage control declarations. This strategy is effective in overcoming the problems described in "Avoiding Redundant Declarations" and "Locating and Naming Code-Behind Files."

Avoiding Redundant Declarations

Visual Studio's code generation throws two problems at you. The biggest problem is that when you open an ASPX file, Visual Studio expects the controls within the ASPX page to be declared in the code-behind file—the .NET class referenced in the `Page` directive. It doesn't just want it somewhere up the inheritance tree, it wants it right there in the closest file—the code-behind file. And if these declarations aren't there, Visual Studio adds them. Because they're redundant with declarations you made in the autogenerated plumbing class (such as `genProject`), the declarations conflict, and you get compiler errors.

If you follow the derivation strategy I described for WinForms, the closest class is the derived handcrafted class—and that class should be empty. It certainly

shouldn't have a bunch of declarations that are difficult to maintain by hand. The nonempty class is just part of the problem. The plumbing in the generated class (the class's base) needs access to variables defining the controls, and if they're defined in the derived handcrafted class, they aren't available.

The solution to this is effective but not pretty. Figure 10-2 shows this derivation strategy. The base classes are similar to what you've seen for Windows.Forms. But derived from the handcrafted base class is an extra generated shim class (`codebehndProjectEdit` in Figure 10-2) containing the declarations. Note the generated and handcrafted classes in this diagram. You can generate the shim class to contain the declarations that Visual Studio is so adamant about having available.

NOTE If you're *never* using Visual Studio to edit ASPX files, you can remove the extra class in this hierarchy. If there's any chance someone will open your files in the Visual Studio ASP.NET editor, Figure 10-2 shows a valid derivation strategy.

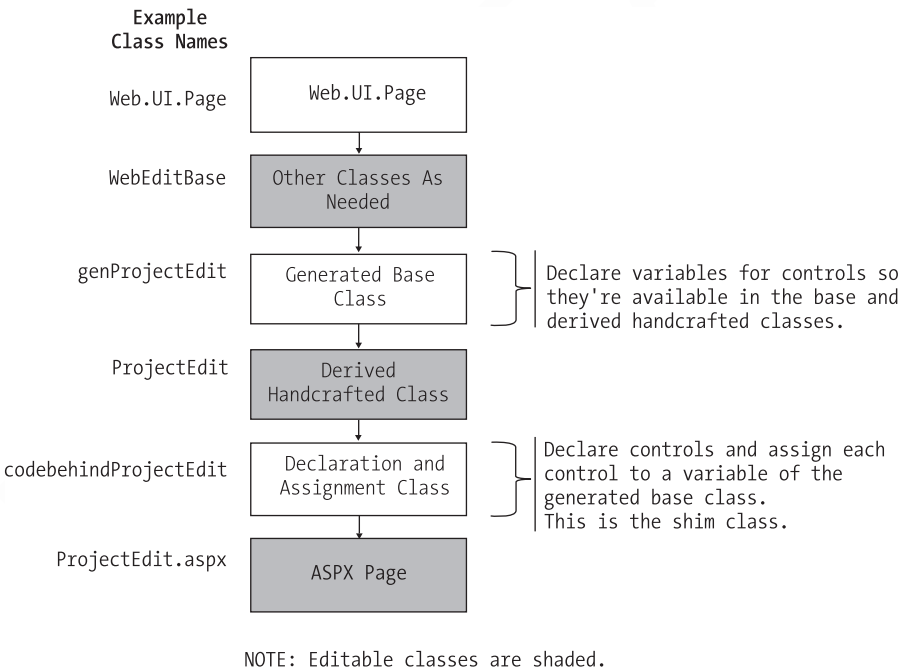


Figure 10-2. An inheritance strategy that works with Visual Studio

To make this derivation strategy work, you need to pair declarations in the shim class with similar declarations in the autogenerated class. You'll have to munge the field names in the base class in a recognizable way to avoid naming conflicts—I prefixed the plumbing class field names with an *m*. Finally, the shim class needs to include an override of the `OnLoad` event that contains a series of assignments. These assignments set the variables declared in the autogenerated class to the controls created in the shim class. An example of the code in the shim class is as follows:

```
#Region "Autogenerated code - DO NOT EDIT"
   Protected WithEvents btnNew As System.Web.UI.WebControls.LinkButton
   Protected WithEvents hplHome As System.Web.UI.WebControls.HyperLink
   Protected WithEvents txtID As System.Web.UI.WebControls.TextBox
   Protected WithEvents txtName As System.Web.UI.WebControls.TextBox
   Protected WithEvents txtStarted As System.Web.UI.WebControls.TextBox
   Protected WithEvents txtEnded As System.Web.UI.WebControls.TextBox
   Protected WithEvents txtDescription As System.Web.UI.WebControls.TextBox
   Protected WithEvents txtDepartmentID As System.Web.UI.WebControls.TextBox
   Protected WithEvents hplList As System.Web.UI.WebControls.HyperLink
   Protected WithEvents btnSave As System.Web.UI.WebControls.Button
   Protected WithEvents btnCancel As System.Web.UI.WebControls.Button

   Protected Overrides Sub OnLoad(ByVal e As System.EventArgs)
      mbtnNew = btnNew
      mhplHome = hplHome
      mtxtID = txtID
      mtxtName = txtName
      mtxtStarted = txtStarted
      mtxtEnded = txtEnded
      mtxtDescription = txtDescription
      mtxtDepartmentID = txtDepartmentID
      mhplList = hplList
      mbtnSave = btnSave
      mbtnCancel = btnCancel
      Mybase.OnLoad(e)
   End Sub
```

Once you build this class, you can interchangeably use the generated and shim class variables in .NET. You'll generally use the class variables in the plumbing and handcrafted classes. You'll use shim class variables in the ASPX file.

Locating and Naming Code-Behind Files

I said Visual Studio threw two problems at you. Visual Studio's expectations for code-behind files lie at the core of the second problem as well. Visual Studio requires that the code-behind file be physically located in the same directory as the ASPX file. For smooth operations, you also need to give the file the name of the ASPX file with an additional .vb extension. If you name the ASPX file Projects.aspx, Visual Studio wants the corresponding code-behind file to be named Projects.aspx.vb.

Within the ASPX file (AssignmentSelect.aspx), you define the page with a `Page` directive similar to the following:

```
<%@ Page Language="vb" AutoEventWireup="false"
          Codebehind="AssignmentSelect.aspx.vb"
          Inherits="WebPTracker.AssignmentSelect" %>
```

The `Codebehind` attribute allows you to name the code-behind file, but Visual Studio doesn't support a relative directory path. This means you have to put the code-behind file—which in this case is the generated shim file—in the same directory as the ASPX file. Although you'll initially generate the ASPX file as well, there's a good chance you'll be editing your ASPX file as your application evolves to fine-tune the user interface. As you've seen in other chapters, it's advantageous to keep the generated and handcrafted code in separate directories—ideally in a directory that corresponds to the template. You can't do that with ASPX files and their corresponding code-behind files.

Visual Studio has a final quirk that makes going along with Visual Studio's expected naming scheme worthwhile. When you add the ASPX and the shim code-behind file to the project, Visual Studio needs to hook them up. If you name your shim file anything other than the name of the ASPX file with the additional .vb extension, Visual Studio displays a dialog box asking you about connecting each code-behind file. It displays this dialog box for every single file, which is annoying if you have a large number of files. Because of this quirk, it's probably best to name your files so they play nice with Visual Studio. As a result, you'll have to watch filenames carefully because after you generate, you'll have a set of closely related filenames such as ProjectEdit.aspx, ProjectEdit.aspx.vb, and ProjectEdit.vb. You'll also have a mismatch between the name of the shim file (ProjectEdit.aspx.vb) and the contained class name (`codebehindProjectEdit`). This allows the simple class name (`ProjectEdit`) to remain the derived handcrafted class—consistent with the rest of code generation.

Allowing String-Free Navigation

Many Web sites rely on the use of strings for navigation to different URLs. Strings are bad because the compiler can't check them. Thus, a typo in a string can wind up in your code, and you'll only find it if you hit every line of code in testing. Although that sort of rigorous testing before deployment and after every change may be a goal in your organization, few applications achieve this level of testing. So it's preferable to use a set of best practices that maximizes the compiler's effectiveness in catching simple bugs such as typos. One of the most important of these best practices is avoiding using strings in code—such as using strongly typed business objects and not specifying navigation URLs as strings.

In addition to pushing common mistakes into easy-to-find compiler errors, avoiding strings for URLs also lets you abstract your directory structure so you can make any changes in a single location. This is much easier than trying to use search and replace if you decide to modify the naming of your files or your directory structure. The code generation techniques I've shown make the ability to manage directory changes even more important because the layout of the site includes multiple subdirectories and the layout depends on the template that creates each source code file. Thus, if you changed the template you used to create a specific page,[9] you'd change its location to retain consistency in directory layout—which would create havoc in your page navigation if you hard-coded navigation URLs in scattered places.

Abstracting URLs to a Common Class

You can avoid this problem if you isolate the strings used to call each page. There are several ways to do this—all easy when you're using code generation. You could use constants, pass an enum to a function, or call individual functions to go to each page. A combination of the first and last approaches is the most robust. With this approach, you'll have three generated base classes containing the bulk of the redirection information as follows:

- **NavigateURL**: Contains the single location where each URL is defined as strong typing relative to the project root
- **Navigate**: Contains methods that supply a `Response.Redirect`
- **ServerTransfer**: Contains methods that provide a `Server.Transfer` operation

9. This isn't a common change, but it may happen in the lifetime of your application.

NOTE There are still strings defining the individual URLs in a single location. I call this style *string-free navigation* because navigation is strongly typed and doesn't use strings.

Implementation of these classes follows the generated base class and derived handcrafted class pattern you've seen throughout this book. A generated class provides most of the code, and a derived handcrafted class is available for customization. The generated classes are `genNavigateURL`, `genNavigate`, and `genServerTransfer`. The corresponding derived handcrafted classes are `NavigateURL`, `Navigate`, and `ServerTransfer`.

Code generation allows you to create these classes, which you can keep together in a single generated file and a single handcrafted file for simplicity. The `genNavigateURL` class contains a series of constant declarations that specify the page location and name. You'll use these declarations both in the `genNavigate` and `genServerTransfer` classes. You can also use them in ASPX pages for the `NavigateURL` property values of controls that support redirecting (such as the hyperlink control). Each of the declarations in the `genNavigateURL` class looks similar to:

```
Public Class genNavigateURL
   Public Const ProjectEdit As String = "~/ASPX/WebEdit/ProjectEdit.aspx"
```

NOTE The tilde (~) sets the path relative to the project root and avoids hard-coding server names or absolute directory locations where your application resides on the Web server. I'm still hard-coding the relative references within the project directory, but this is generated in response to actual file locations.

The `genNavigate` class contains a series of methods such as the following:

```
Public Class genNavigate
   Public Shared Sub ProjectEdit(page as Web.UI.Page)
      page.Response.Redirect(NavigateURL ProjectEdit)
   End Sub
```

The `genServerTransfer` class contains methods such as this:

```
Public Class genServerTransfer
   Public Shared Sub ProjectEdit(page as Web.UI.Page)
```

```
        page.Server.Transfer(NavigateURL.ProjectEdit)
    End Sub
```

The section "Building the URL Abstraction Classes" shows how you build these classes. The next sections show how to use them.

Using the Abstracted URLs

With these classes in place, abstracting URLs in the code-behind file is easy. You just call the corresponding shared method of the `Navigate` or `ServerTransfer` class. Currently you have calls in your calling code that look like this:

```
page.Response.Redirect("ASPX\WebEdit\ProjectEdit.aspx")
```

or like this:

```
page.Server.Transfer("ASPX\WebEdit\ProjectEdit.aspx")
```

To provide a level of strong typing in the main body of code for your application, you can replace this with code such as this:

```
Navigate.ProjectEdit(Me)
```

or this:

```
ServerTransfer.ProjectEdit(Me)
```

TIP Strongly typed paths require that the navigation methods be available before you can use them in code. If you want to navigate to pages you haven't yet created, add temporary methods that redirect to an `UnderConstruction` page during development to avoid compiler errors. You can place these temporary methods in the handcrafted derived class discussed in "Building the URL Abstraction Classes." Because the interim class won't be marked `Overrides`, when you add the page and regenerate the navigation file to include the base class method, you'll get a compile error (in VB .NET, not C#). This will remind you to remove the temporary method.

Using Abstract URLs in ASPX Markup

You can access the constants of the `NavigateURL` class directly in an ASP.NET directive. For example:

```
<asp:hyperlink id="hplList"
        runat="server"
        NavigateUrl="<%# WebPTracker.NavigateURL.ProjectSelect %>">
   Project List
</asp:hyperlink>
```

This allows you to navigate to the correct URL via the same string used in the `Navigate` and `ServerTransfer` classes. If you always use this method, you can change the names and locations of files without adversely affecting your site. Unfortunately, the compiler doesn't catch URL errors in your ASPX pages as quickly as the code-behind errors. That's because it doesn't compile the page until the first time it's displayed.

NOTE The `<%# ... %>` syntax indicates ASP.NET data binding. Data binding is reflection based, so it's slightly slower than directly including the URL. You can avoid this small hit by setting the property in the .NET code of the derived handcrafted (or generated) classes, but this might be confusing if programmers are used to seeing all properties set at the ASP.NET declaration of the control. Overall, the trade-off between this small performance hit and the benefits of fully abstracting URL locations is justified.

Using the Navigate and ServerTransfer Classes from Global.asax

The ASP.NET application file—Global.asax—includes the `AcquireRequestState` event. This event fires as the IIS server rebuilds the state for the postback. You might want to use this event to check that the user is authenticated or perform other duties that might include redirecting the user—perhaps to a login page. But, you can't pass Global.asax as a parameter to the current methods of the `Navigate` and `ServerTransfer` classes. That's because the methods of these classes expect you to pass a `Page` object. Global.asax doesn't derive from `Page` but rather derives from `HTTPApplication` and `System.Object`. Without a common base class or interface, you can't write a single method that accepts either class as a parameter. Using `Object` as the parameter type would solve this problem but lose

strong typing benefits and result in ugly code with extra casts as you prepared to retrieve the `Server` or `Response` properties. Extending generation to build overloaded methods accepting `HTTPApplication` objects is a good solution if you'll frequently access paths from code that isn't in a `Page` object. But if you only see a reason to transfer to the login page from Global.asax, you can just add an overloaded method in the derived `Navigation` class. The code for the method is as follows:

```
Public Class ServerTransfer
   Inherits genServerTransfer

   Public Overloads Shared Sub Login(ByVal app As HttpApplication)
      app.Server.Transfer(NavigateURL.Login)
   End Sub

End Class
```

Building the URL Abstraction Classes

The first step in building the URL abstraction classes is to create a file list as shown in Chapter 5 (shown there as part of altering the .vbproj and .csproj project files). Once you've got the file list, you can run an XSLT template to create the `genNavigateURL`, `genNavigate`, and `genServerTransfer` classes. You want to create constants and methods only for the file list elements that correspond to ASPX files. You can do this with the following XPath expression:

```
<xsl:apply-templates select="//fil:File[contains(@Name,'.aspx') and
                              not(contains(@Name,'.vb')) and
                              not(contains(@Name,'.cs')) and
                              not(contains(@Name,'.resx')) ]"
               mode="NavigateURL"/>
```

This XPath expression grabs the files that have the .aspx extension but don't also have the .vb, .cs, or .resx extensions—excluding Visual Basic .NET (VB .NET) and C# code-behind files as well as resource files).

Creating the constants in `genNavigateURL` requires a recursive template (`GetPageName`) that works toward the root of the file tree until it reaches nodes whose grandparent node isn't a `fil:Dir` element. Looking at the grandparent is

necessary because the file list includes the project root, and the string URLs don't. The `NavigateURL` template is as follows:

```
<xsl:template match="fil:File" mode="NavigateURL">
   <xsl:variable name="pagename">
      <xsl:call-template name="GetPageName" />
   </xsl:variable>
      Public Const <xsl:value-of select="$pagename"/> As String = <xsl:text/>
                   <xsl:text/>"~/<xsl:value-of select="$filename"/>"
</xsl:template>
```

The templates that create the methods of the `genNavigate` and `genServerTransfer` classes are similar, and the `Navigate` template is as follows:

```
<xsl:template match="fil:File" mode="Navigate">
   <xsl:variable name="pagename">
      <xsl:call-template name="GetPageName" />
   </xsl:variable>
      Public Shared Sub <xsl:value-of select="$pagename"/>(page as Web.UI.Page)
         page.Response.Redirect(genNavigateURL.<xsl:value-of
                           select="$pagename"/>)
      End Sub
</xsl:template>
```

The `GetPageName` template retrieves the nonextension portion of the filename and replaces illegal method names. The only one I've run across is for Default.aspx. `Default` isn't a valid method name in Visual Basic, so I replaced it with the method name `Home`. Avoiding keyword conflicts in either language will make it easier for other programmers to use your classes. If you also have a Home.aspx page, you'll have to modify this naming. The `GetPageName` template is as follows:

```
<xsl:template name="GetPageName">
   <xsl:choose>
      <xsl:when test="substring-before(@Name,'.')='Default'">Home</xsl:when>
      <xsl:otherwise>
         <xsl:value-of select="substring-before(@Name,'.')"/>
      </xsl:otherwise>
   </xsl:choose>
</xsl:template>
```

Specifying UI Details in UIInfo.XML

In Chapter 9, you saw how to use a manually created XML file to specify customizations to UI creation. This will often involve specifying that a UI shouldn't be created for certain entities. For example, the following UIInfo.XML excludes a UI for the `UserEdit` page:

```
<?xml version="1.0" encoding="utf-8" ?>
<orm:MappingRoot xmlns:orm="http://kadgen.com/KADORM.xsd" FreeForm="true" >
   <ui:UIRoot xmlns:ui="http://kadgen.com/UserInterface.xsd"
                  ProjectNamespace="WebPTracker" >
      <ui:Forms Fred="4">
         <ui:Form Name="UserEdit" NoGen="true"  />
         <ui:AllForms/>
      </ui:Forms>
   </ui:UIRoot>
</orm:MappingRoot>
```

NOTE I'm currently using the same XML Schema Definition (XSD) schema for the metadata input files used to generate both Windows and Web applications and therefore using the attribute `Form` rather than `Page`. As you enhance your use of these files, you may want to diverge into two different schemas. The benefit of the shared file is that you can simultaneously exclude features from WinForms and WebForms. The drawback is that it's harder to specify a different set of rules for each UI.

In addition to handling the issues discussed in Chapter 9, this code specifies the .NET namespace of the UI assembly. When calling between different classes of the assembly in .NET code, you rarely need to know the project namespace. Within ASPX pages, you'll often need the project namespace to resolve class names. Including this value as an attribute of `ui:UIRoot` in the UIInfo.XML file pulls it into the WebInputMetadata file and allows you to access it, as shown in the section "Looking at WebSelect.xslt."

Putting User Controls to Work

User controls can serve the same purpose in Web applications as they serve in Windows applications. They provide a mechanism to combine a subset of information related to a particular object. You can combine these subsets in a consistent manner between forms.

User controls in ASP.NET offer another advantage. If the data is unchanged, you can cache it on the server, rather than re-creating it when needed. This offers significantly improved performance.

You create user controls in files with an .ascx extension. The code-behind file for a user control is ultimately derived from `Web.UI.UserControl` while the code-behind file for the page is ultimately derived from `Web.UI.Page`.[10] The user control (the .ascx file) opens with a directive such as the following:

```
<%@ Control... %>
```

and the page file (the .aspx file) opens with a directive such as this:

```
<%@ Page... %>
```

To create the .ascx files via code generation, you need to incorporate these three differences into your templates.

Although the same generated plumbing and derived handcrafted classes can support several different page layouts in different user controls *or* pages, the derivation strategy means you can't share the generated plumbing and derived handcrafted classes between user controls *and* pages. If you anticipate a design that puts data on both user controls and pages, first look at your design to see if the data pages could be pages containing a single user control. If you take this route, you'll display all of your data in user controls.

Walking Through ASP.NET Templates

Generating the plumbing and code-behind files for WebForms uses templates that are similar to templates created for other types of .NET classes. Although the architecture behind the templates is significantly different, the techniques used in creating the actual templates match those shown in earlier chapters of this book. Rather than repeating those details, I'll focus on the templates that create the ASPX pages. From the perspective of code generation templates, ASPX pages present an entirely new language to output. The issues and techniques for ASPX pages apply equally to ASCX user controls that you create as part of your application code generation.

This section looks at two XSLT stylesheets. The first—WebSelect.xslt—generates classes such as the one shown in Listing 10-1. Figure 10-1 shows sample output for a page generated from this template. The generated class provides abbreviated information on a list of records retrieved from the database and allows the user to select the record they want to edit. The second—WebEdit.xslt—allows the user to edit details about a single record.

---

10. The similarity between pages and user controls is also shown by the fact that both `Web.UI.UserControl` and `Web.UI.Page` derive from `Web.UI.TemplateControl`.

Looking at these stylesheets will show XSLT code generation at work in a different language. A few extra issues arise, such as outputting special characters and different whitespace demands. Sidebars cover these issues, and discussion about the main flow of the stylesheet is interspersed with the stylesheet code.

Whitespace in ASPX Templates

ASPX templates don't have the stringent whitespace requirements of text output or .NET code. Both the .NET and ASP.NET compilers ignore whitespace, and the humans looking at ASPX pages are far more used to variations in whitespace management. Thus, when you're working with your ASPX templates, if it's significantly easier to cut a corner on the whitespace, you don't need to feel guilty. On the other hand, ignoring whitespace issues will result in ASPX pages that are difficult to read.

Looking at WebSelect.xslt

WebSelect.xslt outputs the `<entity>Select` classes such as `ProjectSelect`. These classes allow you to select a single record from the corresponding `SetSelect` stored procedure. This stylesheet opens with the standard goo. The last five namespaces are for the UIInfo.xml file, the Microsoft XSLT extensions, the user control, ASP.NET, and calling back into .NET code (via XSLT extension objects). You need the `uc1` and `asp` prefixed namespaces because they're used in the output file. The stylesheet uses a supporting template and outputs to `text` as described in the sidebar "Outputting Less-Than Signs." The opening goo looks like the following:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
          xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:dbs="http://kadgen/DatabaseStructure"
          xmlns:orm="http://kadgen.com/KADORM.xsd"
          xmlns:ui="http://kadgen.com/UserInterface.xsd"
          xmlns:msxsl="urn:schemas-microsoft-com:xslt"
          xmlns:uc1="Header.ascx"
          xmlns:asp="asp"
          xmlns:net="http://kadgen.com/StandardNETSupport.xsd" >
<xsl:import href="webSupport.xslt"/>
<xsl:import href="../../Chapter 8/BusinessObjects/CSLASupport.xslt"/>
```

```
<xsl:output method="text" encoding="UTF-8" indent="yes"/>
<xsl:preserve-space elements="*" />
```

The stylesheet accepts the standard parameters plus a `BusinessObject` parameter. Like the WinForm templates, this stylesheet runs against the `ui:Form` elements rather than the `orm:Object` elements. So the `BusinessObject` parameter maps to the `ui:Form` element's `BusinessObject` attribute and is later used to retrieve the correct `orm:Object` element. The parameter declarations are as follows:

```
<xsl:param name="Name"/>
<xsl:param name="filename"/>
<xsl:param name="database"/>
<xsl:param name="gendatetime"/>
<xsl:param name="BusinessObject"/>
```

Declaring these variables as stylesheet variables allows you to define them once for use throughout all the stylesheet's templates. These variables include a .NET project namespace defined in the `ui:UIRoot` element. You need this namespace because ASP.NET often requires a namespace to reference classes even within the same project. The `objectname` variable provides a variable with a name and casing consistent with other stylesheet variables, but it has the same contents as the `BusinessObject` parameter that matches an XML attribute in the input metadata file. Creating the `columns` variable provides a node-list of column elements guaranteed to be consistent throughout different templates called by the stylesheet. This increases the robustness of the template and allows you to make any changes in the node-set's selection criteria in one place. The stylesheet variable declarations are as follows:

```
<xsl:variable name="projnamespace" select="//ui:UIRoot/@ProjectNamespace"/>
<xsl:variable name="formname" select="$objectname"/>
<xsl:variable name="objectname" select="$BusinessObject"/>
<xsl:variable name="columns"
      select="//orm:Object[@Name=$objectname]
      //orm:SetSelectSP//orm:SPRecordSet//orm:Column
      [@IsPrimaryKey='true' or @UseForDesc='true']"/>
```

The entry-level template is similar to ones you've seen elsewhere in this book. The criteria for the processing template uses the `objectname` variable (ultimately the `BusinessObject` parameter) to select the correct `orm:Object` element. The entry-level template is as follows:

```
<xsl:template match="/">
   <xsl:apply-templates select="//orm:Object[@Name=$objectname]"
                        mode="Object"/>
</xsl:template>
```

Outputting Less-Than Signs

The less-than (<) sign is an XML symbol, so it's treated as an open element tag in XSLT. This makes it nontrivial to output the less-than signs required as part of your HTML output. How you output the less-than sign depends on the type of output you're creating. XSLT supports different types of output, and these present three different approaches to outputting HTML elements as follows:

- Use `xml` or `html` as the output mechanism, and output as literal elements:

```
<html>..stuff...</html>
```

- Use `xml` or `html` as the output method, and use the `xsl:element` directive to output elements:

```
<xsl:element name="html"> ... stuff ....</xsl:element>
```

- Use `text` as the output method. You have to escape less-than characters or they'll cause errors:

```
&lt;html>...stuff...&lt;html>
```

Unfortunately, each mechanism has drawbacks. The first and second mechanisms supply output as XML or HTML. The problem with these mechanisms is that you can't easily output the <% delimiters for ASP directives or nest XSLT directives in angle-delimited text such as HTML tags. When these are output, you get `<%` in the output, which unfortunately isn't recognized by Visual Studio. You could perform a postprocessing step that cleaned up the `<`, but XSLT doesn't allow you to output <% directly with these output formats.

The third mechanism is the most reliable, and it's what I'll use in the templates but not happily. The problems are that it's ugly, and it requires that you escape every less-than sign in the entire file that isn't part of an XSLT element. Whew, that's a pain.

Because this template is fairly short and doesn't use regions, I kept it as a monolithic template. This lets you see the structure of the ASPX page in the template. As discussed in the sidebar "Outputting Less-Than Signs," each opening angle bracket (less-than sign) is escaped using `<`. This is effective, but it does make ugly templates. The `uc1` tag prefix has to be registered for the page, along with the corresponding file location. The following is the start of the processing template:

```
<xsl:template match="orm:Object" mode="Object">
&lt;%@ Page Language="vb" AutoEventWireup="false"
            Codebehind="<xsl:value-of select="@Name"/>Select.aspx.vb"
            Inherits="WebPTracker.<xsl:value-of select="@Name"/>Select"
            enableViewState="False"%>
&lt;%@ Register TagPrefix="uc1" TagName="Header" Src="..\..\Header.ascx" %>
&lt;!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
```

Outputting the HTML header of the file is straightforward and looks like this:

```
&lt;HTML>
   &lt;HEAD>
      &lt;title><xsl:value-of select="@CollectionName"/>&lt;/title>
      &lt;meta name="GENERATOR" content="Microsoft Visual Studio .NET 7.1"/>
      &lt;meta name="CODE_LANGUAGE" content="Visual Basic .NET 7.1"/>
      &lt;meta name="vs_defaultClientScript" content="JavaScript"/>
      &lt;meta name="vs_targetSchema"
            content="http://schemas.microsoft.com/intellisense/ie5"/>
   &lt;/HEAD>
```

Outputting the HTML body includes `xsl:value-of` token replacements that customize the page. The start of the body section of the output, along with an `xsl:value-of` replacement, is as follows:

```
&lt;body>
   &lt;form id="Form1" method="post" runat="server">
      &lt;P>
         &lt;uc1:Header id="Header1" runat="server"/>
      &lt;/P>
      &lt;P>
         &lt;asp:Label id="Label2" runat="server" Font-Names="Bauhaus 93"
               Font-Size="XX-Large"><xsl:value-of select="@CollectionName"/>
         &lt;/asp:Label>
      &lt;/P>
```

To output a data-binding expression, you output the `<%#` delimiter as `<%#` and include the expression. The expression here is the result of an `xsl:value-of` directive that uses the `concat` function. ASP.NET needs the results as a string so the entire expression is contained in double quotes:

```
&lt;P>
   &lt;asp:HyperLink id="hlHome" runat="server"
         NavigateUrl="&lt;%#
               <xsl:value-of select="concat($projnamespace,
                        '.NavigateURL.Home')"/> %>">
```

```
    Home
&lt;/asp:HyperLink>
```

Much of your template will just be outputting the same text that would appear on the page if you built it in Visual Studio. For your own application, you might make changes such as using imported utility stylesheets, using the same default approach to values such as colors that I used for WinForms in Chapter 9, or using Cascading Style Sheets (CSS). I kept these templates straightforward to focus on how templates are output, and you can change them in a way that fits your workgroup.

You can output a grid either by using the WebForms `DataGrid` or by building an HTML table. The `DataGrid` offers more functionality but puts more demands on your server. Here I build a `DataGrid` by defining overall properties and then cycling through each column. I don't want primary key columns displayed, but it can be handy to have the values, so I include the column but make it invisible. For this selection page, the user can select the record by clicking a displayed value, so there are `ButtonColumn` controls. See the sidebar "Outputting Nonbreaking Spaces (`&nbps;`)" for a discussion of how nonbreaking spaces such as those in the first line are handled. The portion of the template that outputs the remainder of the selection pages is as follows:

```
<xsl:text>&amp;nbsp;&amp;nbsp;&amp;nbsp;&amp;nbsp;</xsl:text>
&lt;asp:LinkButton id="btnNew" runat="server" >
          Add new <xsl:value-of select="$objectname"/>
&lt;/asp:LinkButton>
&lt;asp:DataGrid id="dg" runat="server" AutoGenerateColumns="False"
          BorderColor="#CC9966" BorderStyle="None"
          BorderWidth="1px" BackColor="White" CellPadding="4">
   &lt;SelectedItemStyle Font-Bold="True" ForeColor="#663399"
             BackColor="#FFCC66"/>
   &lt;ItemStyle ForeColor="#330099" BackColor="White"/>
   &lt;HeaderStyle Font-Bold="True" ForeColor="#FFFFCC"
             BackColor="#990000"/>
   &lt;FooterStyle ForeColor="#330099" BackColor="#FFFFCC"/>
   &lt;Columns><xsl:text/>
   <xsl:for-each select="$columns">
      <xsl:choose>
         <xsl:when test="@IsPrimaryKey='true'">
      &lt;asp:BoundColumn Visible="False"
             DataField="<xsl:value-of select="@Name"/>"
             HeaderText="<xsl:value-of select="@Caption"/>"/>
         </xsl:when>
```

```
                    <xsl:otherwise>
                  &lt;asp:ButtonColumn
                        DataTextField="<xsl:value-of select="@Name"/>"
                        HeaderText="<xsl:value-of select="@Caption"/>"
                        CommandName="Select"/><xsl:text/>
                    </xsl:otherwise>
                  </xsl:choose>
               </xsl:for-each>
                  &lt;asp:ButtonColumn Text="Remove" CommandName="Delete"/>
               &lt;/Columns>
               &lt;PagerStyle HorizontalAlign="Center" ForeColor="#330099"
                     BackColor="#FFFFCC"/>
            &lt;/asp:DataGrid>
         &lt;/P>
      &lt;/form>
   &lt;/body>
&lt;/HTML>
</xsl:template>

</xsl:stylesheet>
```

That's all there is for building a selection form. You can expand and fine-tune this template so it's consistent with your own style.

Outputting Nonbreaking Spaces (`&nbps;`)

HTML normally uses ` ` as the nonbreaking space character and outputs a space. In contrast, the normal space is considered source whitespace and ignored in HTML output. The nonbreaking space is just another ASCII character (character 160 decimal), but the `&nbps;` representation is required in ASP.NET. It doesn't recognize character 160 by itself and outputs a weird character if you use it. If you try to output ` ` directly in your XSLT-generated ASP.NET code, you'll get an error because XSLT treats the ampersand as a special character. You can't just output the escaped value ( ) because that places the 160 character in your output, which ASP.NET doesn't recognize. The easiest solution is to escape the ampersand, resulting in this:

```
&amp;nbsp;
```

I'll use this approach in this chapter's templates.

Looking at WebEdit.xslt

WebEdit.xslt outputs the `<entity>Edit` classes such as `ProjectEdit`. These classes allow the user to edit the data displayed by the form. The opening goo, the parameter declarations, and the entry-level template are the same as the WebSelect.xslt stylesheet discussed in the preceding section "Looking at WebSelect.xslt," and the variables are quite similar:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
          xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:dbs="http://kadgen/DatabaseStructure"
          xmlns:orm="http://kadgen.com/KADORM.xsd"
          xmlns:ui="http://kadgen.com/UserInterface.xsd"
          xmlns:msxsl="urn:schemas-microsoft-com:xslt"
          xmlns:net="http://kadgen.com/StandardNETSupport.xsd" >
<xsl:import href="webSupport.xslt"/>
<xsl:import href="../../Chapter 8/BusinessObjects/CSLASupport.xslt"/>
<xsl:output method="text" encoding="UTF-8" indent="yes"/>
<xsl:preserve-space elements="*" />

<xsl:param name="Name"/>
<xsl:param name="filename"/>
<xsl:param name="database"/>
<xsl:param name="gendatetime"/>
<xsl:param name="BusinessObject"/>

<xsl:variable name="projnamespace" select="//ui:UIRoot/@ProjectNamespace"/>
<xsl:variable name="objectname" select="$BusinessObject"/>
<xsl:variable name="objectsingular"
                   select="//orm:Object[@Name=$objectname]/@Name"/>

<xsl:template match="/">
   <xsl:apply-templates select="//orm:Object[@Name=$objectname]"
                           mode="Object"/>
</xsl:template>
```

This stylesheet's processing template is also monolithic. Although this demonstrates the structure of the template, you might want to refactor this into several templates—particularly if it provides reuse of template parts such as the header section of HTML. The opening of the template and the HTML header section are as follows:

```
<xsl:template match="orm:Object" mode="Object">
&lt;%@ Page Language="vb" AutoEventWireup="false"
          Codebehind="<xsl:value-of select="$objectsingular"/>Edit.aspx.vb"
          Inherits="<xsl:value-of select="concat($projnamespace,'.codebehind',
                            $objectsingular)"/>Edit"%>
&lt;%@ Register TagPrefix="uc1" TagName="Header" Src="..\..\Header.ascx" %>
&lt;!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
&lt;HTML>
   &lt;HEAD>
      &lt;title><xsl:value-of select="$objectsingular"/>Edit&lt;/title>
      &lt;meta content="Microsoft Visual Studio .NET 7.1" name="GENERATOR">
      &lt;meta content="Visual Basic .NET 7.1" name="CODE_LANGUAGE">
      &lt;meta content="JavaScript" name="vs_defaultClientScript">
      &lt;meta content="http://schemas.microsoft.com/intellisense/ie5"
                  name="vs_targetSchema">
   &lt;/HEAD>
```

The `uc1:header` user control allows you to provide the same header for all pages. Because it's abstracted into a separate user control, you could modify its contents or change it to reflect web.config settings, without affecting the active pages of your application. You can do this either to allow the look of the page to change over time or to provide customized solutions for different clients. Using data binding for the hyperlink and inserting nonbreaking spaces were already discussed. The beginning of the part of the template that outputs the HTML body is as follows:

```
&lt;body>
   &lt;form id="<xsl:value-of select="$objectsingular"/>Edit"
              method="post" runat="server">
      &lt;P>
         &lt;uc1:header id="Header1" runat="server">
         &lt;/uc1:header>
      &lt;/P>
      &lt;P>
         &lt;asp:label id="Label2"
               runat="server" Font-Names="Bauhaus 93" Font-Size="XX-Large">
            Edit <xsl:value-of select="$objectsingular"/>
         &lt;/asp:label>&lt;/P>
      &lt;P>
         &lt;asp:hyperlink id="hplHome"
               runat="server"
               NavigateUrl="&lt;%#
                     <xsl:value-of select="concat($projnamespace,
                              '.NavigateURL.Home')"/> %>">
```

```
            Home
         &lt;/asp:hyperlink>&amp;nbsp;&amp;nbsp;
         &lt;asp:hyperlink id="hplList"
               runat="server"
               NavigateUrl="&lt;%# <xsl:value-of
                     select="concat($projnamespace,'.NavigateURL.',
                              $objectsingular)"/>Select %>">
            <xsl:value-of select="$objectsingular"/> List
         &lt;/asp:hyperlink>&amp;nbsp;&amp;nbsp;&amp;nbsp;&amp;nbsp;
         &lt;asp:linkbutton id="btnNew" runat="server" >
            Add new <xsl:value-of select="$objectsingular"/>
         &lt;/asp:linkbutton>
      &lt;/P>
```

For a nice appearance, I used an HTML table to position the edit controls on the page. This table looks like an HTML table created using Visual Studio editors, and I actually created it by copying a table I built in Visual Studio. Differences include escaping the less-than sign as well as XSLT directives to add values such as the captions, control types, and data binding to the page.

To create the individual table rows and controls, the template loops through the `orm:Property` elements of the object. If the maximum length of the field is greater than 100, the template anticipates a multiline textbox and makes the table cells taller. Again, this is something you might want to handle via a CSS style or using the defaulting techniques described in Chapter 9. An XSLT conditional adds the `ReadOnly` attribute only when needed. The template code that builds these controls is as follows:

```
&lt;P>
   &lt;TABLE id="Table1" cellSpacing="0" cellPadding="0" border="0">
      <xsl:for-each select="orm:Property">
      &lt;TR>
         &lt;TD <xsl:if test="@MaxLength>=100">
                        <xsl:text/>style="HEIGHT: 39px"</xsl:if>>
            <xsl:value-of select="@Caption"/>
         &lt;/TD>
         &lt;TD <xsl:if test="@MaxLength>=100">
                        <xsl:text/>style="HEIGHT: 39px"</xsl:if>>
            &lt;<xsl:value-of select="@ASPControlType"/>
                        id=<xsl:value-of select="@ControlName"/>
                        runat="server"
                        Text="&lt;%# m<xsl:value-of
                              select="$objectsingular"/>.<xsl:value-of
```

```
                        select="@Name"/>.ToString %>"
                    <xsl:if test="@ReadOnly='true'">
                    ReadOnly="True"</xsl:if>
                    <xsl:if test="@MaxLength>=100">
                    TextMode="MultiLine"</xsl:if>>
        &lt;/<xsl:value-of select="@ASPControlType"/>>
```

If the value is required, the template adds a WebForms `RequiredFieldValidator` to the page and configures it for the control:

```
    <xsl:if test="@IsRequired='true'">
    &lt;asp:requiredfieldvalidator
            id="rqd<xsl:value-of select="@Name"/>"
            runat="server"
            ControlToValidate=
                "<xsl:value-of select="@ControlName"/>"
            ErrorMessage=
                "<xsl:value-of select="@Caption"/>
                <xsl:text/> is required">
    &lt;/asp:requiredfieldvalidator>&lt;/TD>
    </xsl:if>
  &lt;/TD>
&lt;/TR>
</xsl:for-each>
```

Once the property controls are added to the form, the template adds controls for each child collection. This includes a grid and controls to manipulate the child collection. The portion of the XSLT template that sets up the grid is as follows:

```
<xsl:for-each select="orm:ChildCollection">
&lt;TR>
  &lt;TD><xsl:value-of select="@ChildTableName"/>&lt;/TD>
  &lt;TD>
    &lt;asp:datagrid
            id="dg<xsl:value-of select="@Name"/>"
            runat="server" AutoGenerateColumns="False"
            EnableViewState="False">
      &lt;Columns>
```

The child collection contains the name of the `orm:Object` element. You need to switch the context to this corresponding `orm:Object` before looping through its

properties to build the columns of the grid. The column isn't displayed if it's a primary key:

```
<xsl:variable name="name" select="@Name"/>
<xsl:for-each select=
                "//orm:Object[@CollectionName=$name]">
   <xsl:for-each select="orm:Property">
   &lt;asp:BoundColumn<xsl:text/>
      <xsl:if test="@IsPrimaryKey='true' or
                @IsLookup='true'">
       Visible="False"</xsl:if>
       DataField="<xsl:value-of select="@Name"/>"
       HeaderText="<xsl:value-of select="@Caption"/>">
   &lt;/asp:BoundColumn>
   </xsl:for-each>
```

If you want to display and edit the other parent, you'll need to hide the current parent (it's already displayed and can't be changed) and display a meaningful value as the other primary key. You'll probably want to display this as a button column and redirect the user to a page where they can modify the value. For this implementation, I just hid the fields because editing of parents wasn't required. The portion of the template that manages these columns in the child grid is as follows:

```
<xsl:for-each select="orm:ParentObject">
&lt;asp:ButtonColumn
      Visible="false"
      DataTextField="<xsl:value-of
                select="orm:ChildKeyField/@Name"/>"
      HeaderText="<xsl:value-of
                select="@SingularName"/>"
      CommandName="Select<xsl:value-of
                select="@SingularName"/>">
&lt;/asp:ButtonColumn>
</xsl:for-each>
```

You can also add buttons to remove child rows and to display them in a detailed display so the user can edit the row values. Adding new rows requires a button outside the grid. The portion of the template that provides buttons for user interaction is as follows:

```
&lt;asp:ButtonColumn
      Text="Remove"
```

```
                    CommandName="Delete">
                &lt;/asp:ButtonColumn>
                &lt;asp:ButtonColumn
                    Text="Details"
                    CommandName="Select">
                &lt;/asp:ButtonColumn>
            </xsl:for-each>
            &lt;/Columns>
        &lt;/asp:datagrid>
        &lt;asp:button
            id="btnAdd<xsl:value-of select="@Name"/>"
            runat="server" Width="168px"
            Text="Add <xsl:value-of select="@Caption"/>">
        &lt;/asp:button>&lt;/TD>
    &lt;/TR>
    </xsl:for-each>
  &lt;/TABLE>
&lt;/P>
```

Buttons that allow the user to save the data and cancel editing appear outside the HTML table's boundaries. The XSLT code that defines these buttons is as follows:

```
    &lt;asp:button id="btnSave"
        runat="server" Text="Save">
    &lt;/asp:button>&amp;nbsp;&amp;nbsp;&amp;nbsp;
    &lt;asp:button id="btnCancel"
        runat="server" Text="Cancel">
    &lt;/asp:button>&lt;/form>
  &lt;/body>
&lt;/HTML>
</xsl:template>
</xsl:stylesheet>
```

NOTE The code in this ASPX markup doesn't perform any actions (except data binding). All actions occur in the code-behind file after the page is posted back to the server. You could use JavaScript (JScript) if your page has actions that could occur on the client side (such as additional validation), but actions such as retrieving and saving data require round-trips to the server.

Summary

ASP.NET WebForms present a new architecture that supports browser-based HTML dynamic data display. This new architecture differs from WinForms in how it does the following:

- Manages state
- Supports postbacks and a disconnected model
- Makes new performance demands
- Allows different navigation routes
- Demands special attention to security

If you edit your ASPX pages in Visual Studio (or anyone else will), you need to accommodate some quirks in Visual Studio as follows:

- Declarations for controls must be in the immediate code-behind file (not a base class), and Visual Studio will add declarations, even if these new declarations are redundant with base class declarations.
- Code-behind files must be in the same directory as the corresponding ASPX page files.
- If the code-behind file isn't named with the name of the ASPX page with the .vb (or .cs) extension added, Visual Studio will prompt you before associating each file.

You can work around these quirks by creating an extra shim class in your derivation hierarchy that includes the declarations and assigns the new variable to the corresponding variables in the generated base plumbing class to the shim class variables. This shim class has the anticipated name and resides in the same directory as the ASPX file, but this makes you break good habits used elsewhere in code generation.

User controls provide a good option for reusing logical segments of data, as well as improving performance in some cases.

ASPX presents a new language for code generation templates to generate. Templates that generate ASPX pages must successfully output the less-than sign (normally treated as an XSLT open element) and the nonbreaking space.

Additional Reading

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

- For an overview of designing and creating Web applications, try *Programming the Web with Visual Basic .NET* by Lynn Torkelson, Constance Petersen, and Zac Torkelson (Apress, 2002)

For general ASP.NET, try the following resources:

- *Mastering ASP.NET with VB.NET* by A. Russell Jones (Sybex, 2002)
- *Mastering ASP.NET with C#* by A. Russell Jones (Sybex, 2002)
- *Beginning ASP.NET Using VB .NET* by Chris Ullman, Ollie Cornes, Juan T. Libre, and Chris Goode (Wrox Press, 2001)
- *Beginning ASP.NET Using C#* by Rob Birdwell, et. al. (Wrox Press, 2001)
- *Professional ASP.NET 1.0, Special Edition* by Richard Anderson, et. al. (Wrox Press, 2002)
- *ASP.NET for Developers* by Michael Amundsen and Paul Litwin (Sams, 2001)

For security, try the following resources:

- Inside Microsoft, this book is required reading on security: *Writing Secure Code, Second Edition* by Michael Howard and David LeBlanc (Microsoft Press, 2002)
- You can find specific recommendations in the Patterns and Practices series' *Improving Web Application Security: Threats and Countermeasures* by J. D. Meir, Alex Mackman, Srinath Vasireddy, Michael Dunner, Ray Escamilla, Anandha Murukan (Microsoft Press, 2003)
- To see what several members of the .NET framework security team at Microsoft have to say on security, check out *.NET Framework Security* by Brian A. LaMacchia, et. al. (Addison Wesley, 2002)
- Another good bet is *.NET Security* by Jason Bock, Pete Stromquist, Tom Fischer, and Nathan Smith (Apress, 2002)
- For a good introduction that won't weigh down your knapsack, try *.NET Security Programming* by Donis Marshall (John Wiley & Sons, 2002)

EXHEART

CHAPTER 11

Establishing Perspective

You've hit a wall in productivity, and it's time to stop typing out code line by line in an editor.

THE HISTORY OF THE PROGRAMMING INDUSTRY over the past ten years troubles me. This is our industry, and I'm passionate about keeping it healthy. We've just passed through a very unhealthy cycle. A crash in our industry was the natural result of an astounding volume of wasted money. Many factors contributed to this waste—but a fundamental reason is that *we're writing programs the same way we did 20 years ago even though we're writing entirely different programs.* It's time for a change, and you're on the front lines of that change.

Development processes aren't resilient. You plan for one thing, and when something else happens, whether it's a requirements change or a personnel problem, the development process is unable to adapt. Not only has it become impossible to keep up writing out code line by line in an editor, but the tools have become more complex, users demand more features, and you face unfamiliar issues such as security and multiple delivery mechanisms. You just can't do your job the old way anymore—you need to focus your efforts and brainpower where it's needed and automate writing the semiredundant portions of your applications. Code generation represents such a shift, and this chapter puts this change, and your role in that change, in perspective.

The industry has changed, and those changes have been painful for many people. The only way to fix that, to turn the industry around and put people back to work, is to make development a cost-effective solution where management can trust its investment. Rather than putting programmers out of work, code generation and other new paradigms will bring more money, and therefore jobs, to developers. You have tools to build remarkable applications—the

benefit side of the cost/benefit equation is in place.[1] The *cost* side of that equation has to come under control for management to get excited about these new opportunities. You've got to rebuild trust through more reliable, effective budgets and time frames. I want the industry to stand proudly behind a new slogan of "No-Excuses Programming."

Code generation doesn't address how you interact with people, schedules, management, budgets, and so on, so it isn't likely to solve personnel problems or save the season for the company softball team. But it does address how to improve the speed, reusability, agility, and consistency in your own application development.

Code generation isn't too good to be true—not as long as it's in the context of an excellent process. It's your job to sell code generation to Information Technology (IT) management, and this chapter offers help in that process. Looking just around the corner at announced releases from Microsoft, I'll show how changes in the next versions of Visual Studio .NET and SQL Server affect code generation. Finally, I'll dust off my crystal ball and soothsay about code generation as the basis for building new paradigms that'll radically change the industry.

Selling Code Generation to Management

If I've done my job, you're ready to jump into code generation for your applications. But you've probably got to convince your management, or your client, that code generation is appropriate for your next project. You'll need their support both because they write the checks and because they'll observe a different pattern of delivery. If they aren't expecting this new delivery pattern, they may have unrealistic expectations. If you tell the stakeholders on your project that you'll be able to work 50 times faster with code generation, you may deserve the resulting unrealistic schedule. If instead you sell them on the long-term benefits of principled code generation, you can make these gains real. To help you sell code generation in the context of a slightly slower process with a higher long-term payback, I've included the following in this chapter:

- A letter to managers distilling important points I made earlier in the book
- Notes on estimating code-generated applications
- A checklist to help evaluate your own code generation processes or a vendor's offering

1. And there are even more exciting possibilities opening up in the next five years with upcoming improvements in software, operating systems, and hardware.

Use these tools if they help you establish good code generation processes in your organization.

Sending a Letter to Managers

Share this letter with your manager to explain the long-term benefits of a conscious, principled code generation process and to help convince her that code generation is a viable way to bring down project costs and make development saner within your organization.

Hello Manager,

I expect you're an IT manager working with a group of programmers. You're probably under a good deal of pressure to deliver quality applications with inadequate funding and unrealistic time frames. You don't need a litany of problems—you already know that development is risky and projects are failing or reaching marginal success even with skilled, well-intentioned programmers. A chasm of trust may stretch you between financial decision makers and the programmers who seem to control the real budget and schedule of applications. From a business perspective, seven things need to happen as follows:

- The risk in software development needs to drop to near zero.
- The development calendar needs to become faster and reliable.
- Software quality needs to improve—particularly in security, usability, accessibility,[2] and ease of deployment.
- The overall quality/cost/feature ratios of development need to improve.
- The life cycle needs to be revamped by merging development and maintenance and removing application rewrites.
- Application development needs to become a partner in ongoing business reengineering.
- IT development needs to morph into a service-based, non-capital-driven arm of the consuming organization.

2. *Accessibility* means the ability of programs to be used by individuals with a range of physical abilities and addresses the same types of issues as the American with Disabilities Act (ADA).

Various technical and process initiatives and tools attempt to address these issues. Some of the technical initiatives will incorporate code generation, and you can acquire code generation tools that range from very expensive to freeware. You can also develop in-house code generation tools relatively easily. However, these tools will probably come up short in your organization if your programmers don't know how to incorporate them in a clear process. You need to know what to demand from code generation solutions. All code generation solutions should meet the following five principles:

- **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.

A code generation tool is a long-term commitment. Your ability to control the templates that create the application is necessary to synchronize it with IT policies in your organization over the life of the project and to allow reasonable opportunities to morph the code for wider use.

Metadata is the information about the application you're building—a broad net for your business logic. Isolating metadata allows your business logic and your application architecture and technology to evolve independently—potentially making the concept of "rewrites" obsolete. Your templates and infrastructure encapsulate your experience so it moves forward to new applications—even those run by different teams. The commingling of metadata and architecture is why applications become nonmaintainable in too short a time.

Code generation changes the maintenance cycle. With the distinction between metadata, templates, and handcrafted code, maintenance efforts propagate correctly throughout your application. Programmers work with a formalized architecture, minimizing the background chaos introduced in most maintenance cycles. Key to this is a scripted generation harness that allows maintenance programmers to precisely reproduce code generation processes.

Of course, your application is unique, and you need to handcrafted code to customize it. In languages offering inheritance, derivation offers the primary tool for giving your application its personality and housing your custom algorithms. You also need your code-generated application to match or exceed nongenerated applications in every way—security, performance, scalability, reliability—integrating with available tools for source control, testing, design, and so on. This demands that code generation give you everything you need for development, not just a quick way to barf out code.

Today's commercial offerings don't stack up well against these principles. But that doesn't need to hold you back because creating in-house code generation is a practical alternative, particularly starting with the extensible tools in this book.

Two long-term paradigm changes emerge from this principled approach to code generation. The first is the ability to encapsulate knowledge and experience for direct use in your application. This may involve pulling information from an external expert, such as my sharing Rocky Lhotka's architecture in the samples in this book. I call the mature vision of this infant paradigm *collaboration-based programming*, and it means your group can rely on the best minds in the industry, without having to interpret books and white papers. Part of this vision is transitory business relationships and your ability to switch out trust partners—and their infrastructure—for initial evaluation and over the project life cycle. The other paradigm shift is to make the concept of a project life cycle obsolete. There's no fundamental reason why applications can't last nearly forever and remain up-to-date. You have to respond to the factors that make applications obsolete in a granular fashion within the framework of the existing application. Code generation opens up this possibility.

In the area of process initiatives, agile techniques get a lot of attention. Many of these techniques strive to reduce reliance on a requirements phase, and code generation can work with formal requirements or evolve in the response to rapid feedback cycles. What's lacking in many agile techniques is a passionate commitment to the best available architecture. Merging code generation's ability to supply high-quality interchangeable architecture with agile techniques (especially for requirements gathering and testing) can greatly decrease the risk inherent in rapid techniques and make these techniques viable in new venues.

The delivery cycle of a code-generated application is significantly different from a manually written application. Assuming architectural templates and infrastructure are available, the development team builds the initial requirements and designs the database and business objects. An initial version of the application springs almost magically from this design and database. This first version is likely to expose flaws in the templates and require a round of refinement that may take a couple of weeks. The first version of the application is the canvas for custom code. How much work remains varies greatly based on the amount of custom code you need.

Code generation will positively affect the bottom line of your organization if you focus on long-term benefits such as establishing a formal conduit to share knowledge between projects, improving maintenance cycles, and isolating metadata extraction. I challenge you to look for 20–30 percent improvement (easily lost in current underestimates) until you establish your own metrics. This lets you use some of the effort saved in code generation to increase the life cycle and sanity of your development. Once this principled process is established, you'll see much greater cost savings and phenomenally short project startup times.

The pace of change in this industry will increase in the next five years, and code generation offers a mechanism for embracing this change in your organization by improving the speed, reusability, agility, and consistency of your development.

Good luck and enjoy!

—Kathleen Dollard
Author, *Code Generation in Microsoft .NET*

Creating a New Development Pattern

Imagine if early in the 1900s we had been technically unable to produce an assembly line for automobiles and the automobile industry had grown to a multibillion-dollar industry based on people who each built significantly fewer than 50 vehicles in their lifetimes. Imagine organizations trying to keep adequate vehicles in their delivery fleets.

That's where we are with application development today. The answer isn't an assembly line for application development per se because each custom application is special and requires handcrafting. Custom applications are good because they reflect what differentiates your organization from your competitors.

Rather than an assembly line, automate the creation of the redundant parts of the application and combine that generated code with a proven infrastructure. With these in place, you can focus on handcrafting the custom parts of your application. Code generation empowers you to use handcrafted code where it's most effective.

Estimating Code Generation Projects

I'm sticking my neck out to comment on estimating code generation projects. Almost all projects are estimated way too low in terms of both calendar time and cost. So if I say you're likely to cut your project effort by 20–30 percent on your first code-generated application, I'm probably just suggesting a way to meet your current schedule.

Another factor may decrease the short-term bottom-line gains of code generation. Development tends to be a zero-sum game. If you need extra time for one aspect, such as implementing a difficult feature, you're likely to have to subtract this time from another aspect, such as testing. The reverse also tends to be true: If your schedule pressure suddenly eased, you'd probably address some of the two dozen things you shoved aside to solve the current development crises. Users tend to be very demanding until fed up with the whole process or the budget is completely gone. A code-generated application is likely to leave more time to respond to quality issues and the user's needs.

Letting developers do a better job (and live a saner life) is a huge benefit of code generation, but it means you're unlikely to reflect all your time-savings directly in cost savings. That's unfortunate because I'm confident you'll speed up your development with code generation, but I'm not certain you'll be able to show a huge gain to your manager—especially on your first projects as you're establishing your templates, strategies, and possibly architecture and infrastructure.

It takes longer to build templates than to build the corresponding classes. If you have a target file to work from, it takes about two to five times as long, depending on your skill with the underlying template techniques. The really big payback comes in projects where you're generating more than 20 classes and as you reuse the templates in different projects in your organization.

So in estimating code generation projects, be generous in the time you anticipate for building and fine-tuning your architecture and templates, assessing requirements, designing database/business objects, responding to user requests, and handcrafting code. Overall the degree of effort you save will depend on the application and organization, and it'll be years before you have reliable metrics. As an educated guess, I think you'll achieve about a 20–30 percent improvement in development cost for your early applications, increasing dramatically as you reuse your templates and infrastructure on future projects. That's compared with other .NET applications. If you're just moving to .NET, you'll achieve even greater efficiency as you combine code generation with other efficiencies in the .NET environment.[3]

---

3. .NET is a productive language once you get to know it. The initial introduction to .NET coding can be intimidating because of the sheer volume of what .NET can do for you.

Evaluating Code Generation Solutions

The mechanisms I'm using in this book aren't the only ways to meet the principles for code generation. I put together a checklist you can use as a starting point for building your own evaluation (see Table 11-1). You can use this to evaluate commercial tools, existing code generation processes you may have, and new code generation processes you put in place. My intention is to break down the concepts behind the code generation principles into the details you'll see in the real world.

This is a challenging checklist of features, and no tool I'm aware of meets all of these items. I look forward to seeing how this checklist evolves as more people think about these issues. The checklist also needs to evolve because the underlying goals might be met in a different way, making the way some questions are worded misleading.

In itself, a checklist can be a limiting tool for evaluation because it leaves open the questions of *how well* a tool implements each item. But it's a good tool for opening dialogue on specific issues and challenging vendors to raise the bar on what's available in commercial code generation. I look forward to enhancing and fine-tuning this checklist.

NOTE Because this is intended as a starting point for *your* list, Table 11-1 is written in the first person from your perspective. Check off the questions as you evaluate specific code generation tools.

Table 11-1. A Checklist for Evaluating Code Generation Tools

PRINCIPLE	QUESTIONS	YES	NO
Principle #1: I'm in charge of the process.	Can I modify the templates? If so, how? (Are they independent templates that can be modified, or do I need to modify source code within a monolithic harness file? What languages do I have to know to modify the templates?)		
	Can I exclude the generation of particular files?		
	Can I change what template generates a particular set of files?		
	Can I add new templates to generate additional files?		
	Can I use the tool to refactor my existing .NET code into templates?		

Table 11-1. A Checklist for Evaluating Code Generation Tools, continued

PRINCIPLE	QUESTIONS	YES	NO
Principle #2: Metadata extraction is a separate step.	Is metadata extraction automated?		
	Is metadata extracted as a separate step?		
	Do I have access to the metadata for uses unrelated to this product?		
	Do I have access to a schema?		
	Can I merge metadata from other sources, including manual metadata?		
	Can I modify metadata such as modifying default captions? If I modify metadata, are my changes protected when I regenerate metadata from database information?		
	Can I supply indirect mappings between my database and business objects?		
	Does the metadata structure match industry standards, or is a transformation to match industry standards provided?[4]		
Principle #3: Code is generated as part of a one-click process.	Can I perform code generation with a single button click?		
	Can I control what's run as part of this process?		
	Can I add arbitrary items that run as part of the button click?		
	Does the tool provide a log of what has been done?		
	Can I run processes that morph metadata for specific tier generation?		
	Can I copy files?		
Principle #4: Generation embraces handcrafted code.	Can I supply class-by-class (or file-by-file) customization of business objects?		
	Can I supply procedure-by-procedure customization of stored procedure code?		

4. Note that there are no industry standards today, so this will always be unchecked. The question is so important that I include it for the future. Current attempts at standards aren't yet mature.

Table 11-1. A Checklist for Evaluating Code Generation Tools, continued

PRINCIPLE	QUESTIONS	YES	NO
Principle #4, continued:	Can I supply form-by-form customization of WinForms?		
	Can I supply page-by-page customization of WebForms?		
	Can I generate WinForms consisting of either forms or user controls?		
	Are the initial handcrafted code classes empty?[a]		
	Does the tool recognize nonedited handcrafted code and allow me to regenerate if my patterns change?		
	Does the tool recognize generated files edited in by mistake, protect the file, and raise an error?		
Principle #5: Quality is at least as high as non-generated applications.	Does the performance of the generated application match or exceed applications built by hand?		
	Does the tool also automate testing?		
	Does the tool identify a set of classes as most important for triage testing?[b]		
	Is the code generation process integrated with Visual Studio?		
	Does the tool automatically check generated files in and out of source control?		
	Can I provide an alternate generation-only user ID for source control?		
	Does the tool build project files?		
	Does the tool build solution files?		
	Do the templates support nulls?		
	Can I control whether templates use stored procedures or dynamic SQL?		

[a] As you've seen in this book, prior to the release of .NET 2.0 (Whidbey), this is probably impossible without compromising architecture.

[b] It is nontrivial because you have to determine coverage in code that is conditionally output by the template.

Table 11-1. A Checklist for Evaluating Code Generation Tools, continued

PRINCIPLE	QUESTIONS	YES	NO
Principle #5, continued:	Does the tool match or exceed my security goals?		
	If the tool provides a user UI, does it address accessibility?		
	If the tool provides a UI for the programmer, does it address accessibility?		
	Is there a mechanism for moving my experience forward to a new project?		
Overall product capabilities.	Can I generate business objects?		
	Can I generate stored procedures?		
	Can I generate Smart Client (WinForms) UI?		
	Can I generate WebForm UI?		
	Can I generate ASP.NET Web Services?		
	Is there an infrastructure provided for the business object tier?		
	Is source code available for the business object infrastructure?		
	Is the infrastructure designed to allow handcrafted changes to the business object infrastructure via derivation?		
	Is there an infrastructure provided for WinForms UI?		
	Do I have source code for the WinForms UI?		
	Is the infrastructure designed to allow handcrafted changes to the WinForms infrastructure via derivation?		
	Is there an infrastructure provided for WebForms UI?		
	Do I have source code for the WebForms UI?		
	Is the infrastructure designed to allow handcrafted changes to the WebForms infrastructure via derivation?		
	Is there an infrastructure provided for ASP.NET Web Services?		

Table 11-1. A Checklist for Evaluating Code Generation Tools, continued

PRINCIPLE	QUESTIONS	YES	NO
Overall product capabilities, continued.	Do I have source code for ASP.NET Web Services?		
	Is the infrastructure designed to allow handcrafted changes to the ASP.NET Web Service infrastructure via derivation?		
Support for version 2.0 of the framework (Whidbey).	Does the tool support partial classes?		
	Does the tool generate generic collections for strong typing?		
	Can I control the generation of strongly typed collections for compatibility with existing code?		
	Are snippets provided for common code, such as validation code, that are likely to be performed in handcrafted classes?		
	Can the tool refactor my existing code snippets (a Whidbey feature) into templates?		

Investigating Upcoming Features

Simultaneous release of the next version of the .NET Framework, Visual Studio, and SQL Server will probably occur around the beginning of 2005. This release is exciting for .NET programmers. There will be major advances in language details, programming efficiency, and runtime features. I'll review major new features in terms of how they may affect code generation. You can visit `http://msdn.microsoft.com` for details of these features.

Exploring Key SQL Server Features in Yukon

The next version of SQL Server, code-named Yukon, throws three new features into the laps of programmers. Each has interesting possibilities, but you'll have to decide how to use them. The big three from the perspective of .NET application architecture and code generation are the ability to store .NET types, call .NET methods, and store data as XML.

NOTE The following sections about Yukon are based on beta code that might change before release.

Storing .NET Types

The next version of SQL Server allows you to store .NET types. The data for these types is binary, serialized for SQL Server storage. This triggers my imagination, but I wonder how practical such storage will be because it isn't clear how you'll index and access individual methods and properties. Assuming you need to reconstitute it to a .NET object for access, the appropriate usage is likely to be fairly limited to protect performance. However, it may provide a caching advantage, particularly when you're running code on SQL Server.

Storing .NET types on SQL Server isn't likely to affect your code generation processes, unless your templates are affected by an architectural change because of this feature.

Running .NET Code

In the next version of SQL Server you'll be able to run .NET code on your SQL Server. Why is this exciting when you can already run non–Transact SQL (T-SQL) code using extended stored procedures? The new design opens the world to programmers who don't use C++ and avoids the Component Object Model (COM) interface. What's far more important is that running managed .NET code is much safer with few issues caused by a misbehaving program. This is a much more robust model than extended stored procedures not only because .NET code is more robust than unmanaged code but because you can control what type of code runs—any code, only managed code, or no code. The option to run any code on the SQL Server provides backward compatibility with extended stored procedures.

Although you might hear that "you can write stored procedures in .NET code" and that's true, it's unlikely you'll be redesigning more than a small fraction of your stored procedures. T-SQL is a set-based language that's optimized by SQL Server, and those optimizations provide tremendous runtime performance advantages. You're likely to use .NET code to replace extended stored procedures if you're using them and add functions for use within T-SQL stored procedures. But, you aren't likely to replace standard stored procedures with .NET code running on SQL Server.

Running .NET code on SQL Server affects code generation because you may be generating .NET code that runs on SQL Server. In addition to writing this code, you'll have to install it on SQL Server in a manner analogous to installing a stored procedure via a stored procedure script.

Using XML Data and XQuery

The next version of SQL Server provides Extensible Markup Language (XML) as a full-fledged data type. You'll be able to include elements and attributes in indexes and retrieve data based on commingling traditional T-SQL and XQuery in the same `WHERE` clause. The implications of this aren't yet evident because details such as performance effects aren't yet clear. It's an interesting development because it may lead to a need for additions to the normalization rules in use for more than 30 years. The potential is the simplification of database designs.

XML data types affect your generated code because you may be generating XQuery statements in your stored procedures.[5] To support this, you'll have to expand the database metadata extraction step to include the schemas of contained XML data. You'll need a design for carrying this into your stored procedure definitions and templates. This same schema information is likely to alter your business objects because they're likely to expose a strongly typed representation of the data rather than the raw XML. The schema is likely to reach into the UI layer as well. Because of the extent of these changes, how you represent the XML schema for each column in your metadata will be critical.

Exploring Key .NET 2.0, or Whidbey, Features

The next version of .NET, code-named Whidbey, presents a variety of new features. Many of these features don't impact your application code generation, except changes in your architecture corresponding to changes in your templates. Some new features—such as snippets, refactoring, Edit and Continue (E&C) in Visual Basic .NET (VB .NET), and generics—will affect how you code but not the code generation process itself. Others, such as partial classes and flow control in WinForms will directly affect your code generation. Microsoft has also updated its own code generation, including significant improvements to the strongly typed DataSet addressing some of the issues described in the Chapter 1 sidebar "The Strongly Typed Dataset."

NOTE The sections about Whidbey are based on alpha code that could change before release.

5. You'll also be putting XQuery in your application code if you're using dynamic SQL techniques.

Exploring Code Snippets

Code snippets allow you to drop a fragment into your code, with tags indicating what you should replace. The good side of snippets is that they encapsulate intelligence so you can use it directly in your code. Rather than reading code in a book or article, retyping it, evaluating what you need to change, and changing it, you'll just insert the snippet and fill in the blanks. The intelligence you're capturing could be an author, a lead group in your organization, or the conclusion you came to while researching the issue at an earlier time.

On the other hand, snippets are bad because they're a sophisticated copy/paste operation. All copy/paste operations decrease code reuse and cause you to maintain similar code in numerous locations. Used for small-scale tasks that don't belong in subroutines for other reasons, snippets increase the efficiency of your project development. At the other extreme, using them to replace subroutines in even a fairly small percentage of your application would be disastrous from a maintenance perspective.

Snippets are also cool because they represent a way to abstract code into repeating and replaceable units. The concept of isolating repeating segments lies at the heart of code generation. The kinesthetic[6] nature of snippets is likely to help more programmers visualize their code as standard templates with replaceable segments and is likely to make adoption of application code generation easier. You'll also be able to use snippets for the structure of common derived handcrafted class tasks, such as providing validation.

Features such as snippets that affect the way you code may also affect the way you build templates, particularly brute-force and CodeDOM templates. They may even represent a mechanism for bringing forward aspects of your project that are used once in each application but reused between applications. But I'm skeptical you can automate snippet insertion through your generation harness script. So, overall, code snippets will not significantly change your code generation.

Refactoring

Refactoring is a set of techniques that allows you to automate code improvements. For example, you can mark a section of code to move into a method, and Visual Studio will move the code into a new method and call this method from the original location. To do this, Visual Studio manages parameters for both the new method and the method call. Refactoring support in the Whidbey release will be more advanced in C# than VB .NET.

The effect of the Whidbey version of refactoring on code-generated applications will be minimal. Refactoring might help you create a better harness

---

6. *Kinesthetic* means learning through action or doing.

application, but it'll have only a minimal effect on your generated application. That's because you can't just refactor your generated code and allow regeneration. You have to return to the template to make the change, and refactoring won't do that for you.

Now, if you ever have a tool that refactors a target code sample into a code generation *template*, then refactoring will become a really cool code generation tool. But that's a nontrivial task.

Looking at Edit and Continue

Edit and Continue (E&C) returns the Visual Basic (VB) 6 and C++ debugging technique of stopping an application, making a change, and continuing to run the application with the new fix installed. This speeds debugging, particularly when there are many steps to reproduce a particular issue. E&C will appear only in VB .NET in the Whidbey release.

Like many programming tools, you can abuse E&C. Making random changes more efficiently isn't a move toward more effective debugging. However, any process that speeds the turnaround of experiments or tests in the process of elimination based on the scientific method discussed in Chapter 5 makes it more effective. The key is whether you're making purposeful changes to disprove coherent hypotheses or random changes with your fingers crossed.

You'll have to use E&C carefully in the context of code generation. If you make a change in a generated file, you'll either lose those changes or have a generation error (depending on the setting of the `OutputGenType` if you're using this book's harness). Unless you maintain a read-only file attribute on your generated files, you'll have to pay careful attention to whether you're in generated code when you're debugging.

Exploring Generics

Generics are a compile-time template technique. Generics allow you to use type parameters for classes and methods. By defining types as a special kind of parameter, they remain unspecified as you write the code. The types are filled in at compile time or runtime, so you interact with the class or method in a strongly typed manner, even though you didn't know what type would be used when you wrote the code.[7] You can most easily see the value of generics by looking at collections. The same code can be used to create a strongly typed collection of integers, business objects, WinForms forms, and so on.

---

7. Although generics are often compared to C++ templates, they're implemented quite differently and won't result in the excessive code bloat that plagues C++ template usage.

Generics are quite different from the application code generation you've seen in this book. You can think of generics as small-scale compile-time or run-time generation. You can create both generic classes and generic methods that consume generic classes. The primary purpose is to provide strong typing in scenarios where it was previously difficult to provide strong typing—particularly collection classes.

When you're ready to use Whidbey, you'll want to consider upgrading your templates to generate strongly typed collections for your business objects. This will require changes in your calling code to remove casts and type checking that's unnecessary with strongly typed collections. Because these changes are likely to extend throughout your code, you'll have to plan carefully to retrofit existing applications.

For new code, however, strong typing offers improvements both in the robustness and performance of your application. For example, because untyped collections store their contents as `Object` types, a `For Each` loop has to cast each item from `Object` to the type defined in the `For Each` variable. For example, the following `For Each` casts from `Object` to `Invoice` on each iteration of the loop:

```
For Each invoice As Invoice In invoiceCollection
```

Not only is this an extra cast, but if the collection contains something other than invoices, the error isn't found until the cast occurs *at runtime*. The cast is particularly inefficient when you're working with value types where items must be boxed when their added to an untyped collection and unboxed during the narrowing cast.[8]

Generics aren't without problems. Generic collections don't derive from `Collection`, but from a different hierarchy, so you can't interchange them with classes derived from `Collection` (they aren't polymorphic). Although constraints are supported, it isn't easy to retrieve constraint information. For example, when you work with a generic `EmployeeCollection` or `ContactCollection`, they aren't polymorphic, and it isn't easy to determine that you're working with a collection of items that derive from `Person`. It will take a little while for patterns to emerge on how to work around the difficulties with generics. There's plenty of room for generics to become more sophisticated in future versions of the framework. For example, covariants (which may be implemented by one of the sibling languages in the .NET family, Eiffel.NET) would make it easier to work with parallel collections, such as `EmployeeCollection` and `ContactCollection`.

---

8. Objects are reference types, and structures are value types. Reference types and value types are stored differently in memory—reference types on the heap (far memory) and value types on the stack (near memory). Converting from a value type to a reference type involves *boxing* to copy the item for storage on the heap. Converting the item back to a value type involves *unboxing* to copy the item for storage on the stack.

Splitting Code with Partial Classes

Partial classes are a mechanism to combine the contents of two files into a single class at compile time. A preprocessing step just slaps the two files together before the compiler takes over. The main class doesn't indicate the existence of the partial class—the partial class indicates that it's part of the main class via a keyword at the top of the file. Because of this design, you'll almost always generate the partial class and allow the programmer to handcraft the main class.

Partial classes are a big gain for application code generation in Whidbey. They let you isolate handcrafted and autogenerated code into separate files that make up a single class. This lets you isolate code into separate files without using an inheritance hierarchy. This is an important new option, but in most cases it won't replace the derived class pattern discussed in Chapter 4.

A key difference between partial classes and inheritance is that inheritance provides strict guidelines about what classes expect of each other. If member (properties or methods) names conflict, you can explicitly control the resulting behavior. You don't have this control with partial classes—you'll get a compiler error if member names conflict. For a method such as `Validation` that you anticipate programmers wanting to use, you'll have to develop another strategy. You can place your validation in a method with another name and have it call the `Validation` method. If you do that, the programmer working on the main class must add this method for the code to successfully compile. As you saw in Chapter 4, the less required code in the handcrafted class, the better. In this scenario, overridable base classes remain a better option.

Although it's dangerous to call back into the handcrafted main class relying on the existence of a particular method or property, it's safe to raise and respond to events. Imagine that you have a task you want to generate for the `Load` event of a form. You could override the `OnLoad` method to provide your functionality in the partial class, but if a programmer later added an `OnLoad` method to the main class, it'd break the generated code. It's safer, if a teensy bit slower, to catch the `Load` event. There's no restriction on how many handlers can catch a particular event, so the programmer can also catch the `Load` event in the main class without breaking any code. Note that the order of execution of multiple handlers is currently undefined.

Partial classes don't work well in scenarios such as the autogenerated plumbing classes where your generated code orchestrates more complex operations or where later programmers will want to pick and choose what implementation they override.

NOTE It appears that in VB .NET you'll be able to use different `Option Strict` values for different partial class files that'll be combined into the same class. This will allow late binding restricted to one portion of a class.

Partial classes do have two important contributions to code generation design. They let you remove required code—generally all required code—from handcrafted classes. This is huge. For example, you could put the constructor that blocks direct instantiation of a business object into a generated partial class paired with the business object. This allows programmers working in derived handcrafted classes to have confidence that they can change anything in the file they're working with. They won't have to remember what code is required, and they won't be able to circumvent your architecture by changing this code.

Partial classes also allow you to remove a couple of ugly hacks. For example, partial classes let you remove the separate code behind shim class introduced in Chapter 10. Instead of supplying a shim class that declares the Web page's variables and sets them equal to generated plumbing class objects, you can generate this code into a partial class. It's still ugly, and you still have to set the values equal to the generated plumbing class objects, but you can put this code in a location that cleans up the inheritance hierarchy for WebForm applications.

If you had any solutions that required the use of regions for isolating generated code, partial classes offer the obvious benefit of cleaning this code out of the classes that programmers handcraft. This lets you better manage the overwriting of files and source control behavior, including indications when humans change your files separately from when the code generation changes them.

Some of the things you might contemplate doing with partial classes won't work. For example, you can't include the same partial class as part of multiple main classes as a hack around the lack of multiple inheritance (unless you copy it). This may be a good thing because partial classes lack the strict architectural rules of inheritance.

Using Flow Layout for WinForms

In Whidbey, WinForms will offer flow layout that's much like the flow layout in WebForms and Hypertext Markup Language (HTML). Although flow layout for WinForms is a small feature compared to partial classes and generics, it lets you exclude a whole bunch of ugly code for positioning controls in WinForms and makes font size changes much easier to manage. Other WinForms features such as a container-based splitter and the new grid will make programming WinForms, and therefore programming your templates, easier.

Envisioning the Future

To close this book, I want to put on the dreamer's hat and look at some of the directions the future may hold. Much of what we think about programming arises from habit and the semantics of how we communicate. If we strip this away, and look at the nature of development in broader terms, new possibilities

appear. Not all of possibilities come to pass, and certainly fantasizing about the future doesn't expose the years of hard work to open these new horizons. But there are big changes coming over the next five or ten years in this industry, and it's helpful to imagine some of the forms those changes may take.

Standardizing Code Generation Interactions

Today you can move forward into code generation using the templates at the Apress Web site, acquiring a commercial or shareware code generation tool, or creating your own templates. If you're in charge of your own templates, you can move them forward to reuse your architecture in different applications, but what if someone builds a great solution that better fits the projects in your organization? Can you easily drop that into your development cycle? Can you retrofit your deployed application with the new templates and infrastructure? Can you easily try out several code generation solutions to see which you like best? The answer is "no," and your ability to do these things in the future depends on standardizing key points in the code generation process.

Sharing Templates and Frameworks

If a few technical problems can be solved, sharing templates between organizations will be become a viable option. This is exciting because code generation templates allow you to access the great minds in the industry and distill the best available into your own templates. It's natural to interchange these templates as best practices emerge, new issues (such as changes in security) come up, and the fashion in your organization evolves. If you can switch out old templates with new as your needs evolve, you can update your application's architecture in place at any point in the project's life cycle. Ideally you can do this even with deployed applications (with testing).

It's not quite this easy because you also need to integrate handcrafted classes. To fully realize the potential of sharing templates, the generated classes need to work identically with your existing handcrafted classes. This isn't as tough as it may sound, however, because it primarily involves standard method signatures for validation methods and business object properties.

If you're doing your own code generation, you can establish a solid set of standards for metadata, harness scripts, and the interfaces between generated and handcrafted code within your organization. You're de facto creating your own standards as you go along and can at least interchange your own templates. Pay attention to the structures you're using and how they evolve.

Your templates may be commercial. The key to successful commercial solutions is for companies to stop selling individual versions of code generation

itself—which you no longer need when you understand code generation fundamentals—and begin selling competing tools and architectures based on a set of standards. A good architecture you can drop into your application has value, and I look forward to a day when there's a vibrant market for code generation template/infrastructure sets. That vibrant market won't emerge until consumers can interchange sets. An architecture you're stuck with for the project life cycle is a risky proposition.

Standardizing Metadata: The Easy Part

The technical hurdles to overcome in allowing widespread exchange of templates involve standardizing key pieces. This includes the structure of the metadata input. If all templates anticipated the same metadata structure, you could interchange the templates themselves. Anticipating the same structure doesn't mean different tools have to actually use the same structure. Transformations to and from common structures could allow template designer to use the structures they find effective. This common structure—an XML Schema Definition (XSD) schema—needs to include just about every aspect of metadata any company might use and needs to evolve as new metadata usage emerges. Yes, that means that the input structure isn't proprietary, and that isn't going to be an easy change for commercial vendors. But code generation won't significantly expand within the industry until templates are a replaceable tool, rather than a link welded for the life of the application. I talk about more about healthy business relationships with vendors in the section "Inventing Collaboration-Based Programming."

Standardizing Runtime Application Interfaces: The Hard Part

The broader vision of sharing templates includes the ability to pick and choose among templates that perform different tasks. Why expect experts in business object architecture to also produce great templates for UI development? UI technology tends to evolve faster, and you're far more likely to support (or want to support) multiple UIs than business objects. You'll need to interchange and update UI template/infrastructure sets more frequently than your middle tier architecture.

It isn't easy to provide the infrastructure to support interchangeable templates that support different aspects of your application development. This infrastructure requires clear and predictable interfaces between layers, and these aren't easy to develop. However, even a finite set of standard interfaces would allow intermixing the generated code from at least some templates.

This is also important to individual organizations. Ideally, you'd be able to gather templates for certain tiers while generating the templates for other

application tiers in-house. For example, you might select a middle tier architecture and handcraft or generate a company-specific UI.

Providing Tools for Each Step

The five steps of code generation correspond to five areas, each of which has significant potential for growth in available tools both within your organization and from commercial vendors. These opportunities include at least the following:

Architecture: Code generation templates provide the code, and they pair with the infrastructure that runs your application. With standardized metadata, you'll be able to leverage the best available template/infrastructure sets at any point in time, and there's room for growth in the available sets.

Metadata: Metadata extraction from SQL Server is straightforward. However, there are many other viable options for metadata extraction. The most important of these are Unified Modeling Language (UML), Web Services Description Language (WSDL), novel techniques for extracting from formalized requirements, and front ends for manual entry.

One-click generation: The harness in this book works. But it isn't a full-scale commercial-quality application. It could be improved in an number of ways, including better reporting, better handling of nested scripts, better response to script or template errors, and a more sophisticated overall feel.

Handcrafted code: Handcrafted code techniques are straightforward for most .NET code. Better solutions for JScript and other languages would extend the reach of code generation.

Supporting features: Many small facets make up the code-generated application's capacity to exceed the quality of a handcrafted application. The most exciting area for growth is automating the creation of test scenarios. Other areas include better source code integration, easier project file creation, and tools that allow you to formalize the process of starting a new application.

With consistent interfaces, many of these tools could be independent (although they might be part of suites). This would allow you to pick the best solution for a particular issue important to your organization.

Extracting Metadata from Requirements

Metadata represents requirements. Requirements are currently some sort of document that contains text explaining the application. Building metadata from this sort of document is difficult. A human has to interpret the requirements into a different format for use as metadata. Different humans are likely to interpret the same set of requirements differently. Some of this interpretation is part of the design process, but many aspects of requirements are left behind as the applications description is formalized in design tools such as UML. There's a huge opportunity in formalizing requirements for more direct use in later parts of development. This isn't limited to code generation but includes at least testing design and a less ambiguous path into database design.

Although deriving metadata from requirements is the direct benefit—and direct benefits drive the adoption of new paradigms—codification of requirements would have an even more important side benefit. In the current scenario, when you create a document and perform an expensive translation process, the original requirements become a dead document. Sure, you can commit to updating it, but it's no longer a primary source. Codifying requirements in a way that's directly used makes requirements a first-class member of the development cycle. The result is that requirements naturally become iterative, and you achieve rapid requirements processes and long-term requirements management.

This book talks about building metadata and pressing a key to create the initial version of your application—imagine developing requirements in a codified manner and pressing a button to create the initial version of your application. You still have work to do because the initial version of your application doesn't yet contain your handcrafted code. But imagine basing your handcrafted code on a clear list of requirements not met by code generation, rather than spelunking requirements documents to find what hasn't been included.

Developing a New Vision of Code

Implementing code generation may get you thinking about the nature of code. At the most fundamental level, what's code? Today code is a set of semantics. C#, VB .NET, and all other languages are a set of code expressed in specific semantics, or a language, with a corresponding compiler that translates it into another language—Microsoft Intermediate Language (MSIL). Although MSIL is common, the semantics of individual languages isn't. Translating VB .NET to C# or vice versa involves a semantic translation unrelated to the compiler. You could decompile code, but this isn't an absolute process of returning precisely what you started with.

Code may always be like this—a semantic language bridging humans and computers. It's also possible that code will move into a more symbolic realm. Using the example of VB .NET and C#, it's possible that code could become an abstract symbol set for storage, with each language a representation of that symbol set for the programmer. The symbolic storage and the semantic representation could be interchanged—allowing simple translation between languages. The benefits would include programmers working on the same code base with whatever languages and tools they were comfortable with. This would also allow new possibilities in refactoring tools because they'd work with a logical object graph of the code—more robust than but similar to the CodeDOM. Application code generation could create this logical object graph, rather than worrying about individual language semantics. The benefits include removing the need to create code in the order you want it to appear.

The drawbacks, and the reasons it'll take a long time to get to this approach, are that it forces languages to resolve significant differences. For example, C# and VB .NET have significant differences in how they handle events. The sidebar "Language Difference" discusses more differences between the main .NET languages.

Language Differences

.NET changed how you select languages, which led to a lot of confusion. Prior to .NET, you selected a language because it was the best tool for a particular job. C++ was more appropriate for high-performance programming, and VB was a better fit for most business applications. The task dictated the tool. Now that VB .NET, C#, and most other .NET languages are on equal footing, are they also equivalent?

They're not. They differ in how they relate to the human programmer. There have always been distinctions in how closely languages matched human thought patterns, but this is now the primary distinction. This different way of relating to the human programmer isn't a matter curly braces. Microsoft made a mistake in not publicizing the underlying differences between the languages. C# is deeply rooted in C++'s concise computer-like manner. A statement is delimited by a semicolon rather than new line character because it's crystal clear. Humans natively think in fuzzier terms. The end of a line is a sufficient visual break for a human, so it should be for the computer in the BASIC world. A bunch of extra new line characters doesn't confuse the human, so it shouldn't confuse the computer the way a bunch of extra semicolons might confuse the compiler.

This human-friendly approach is played out in the extra characters available when you read VB .NET. Constructs such as `End While` and `End Function`

differentiate closing constructs rather than the generic } used by C#.[9] Friendliness to humans has been one of the fundamentals since the original versions of BASIC. VB .NET builds on this by providing a degree of protection against computer quirks, such as what happens when you add 1 to 32,767 as a 16-bit signed integer.[10] The distinction between VB .NET and C# plays out in specific ways, including the result of `f = 2/3` where `f` is a double[11] and VB .NET being locale aware by default.

Microsoft will base the way each language evolves partially on pressure from the community. Microsoft's announcements that VB .NET will focus on development features that make programming faster and more efficient while C# focuses on language features is consistent with this distinction. Inclusion of E&C and speed-dialing (the `My` namespace) in VB .NET vs. iterators and more robust refactoring in C# is consistent with this distinction. Happily, the two languages will remain parallel, and if the C# community wants E&C or the VB .NET community wants refactoring, expect to see these features cross over in upcoming versions.

NOTE The underlying VB .NET and C# compilers are quite different. These differences go beyond VB .NET, providing a background compiler that offers immediate design-time feedback. These differences will make it difficult to incorporate E&C in C#.

Because programming groups tend to contain similar types of programmers, groups can generally make a decision on the same basis as an individual's decision. An organization-wide directive that diverse types of programming groups use one or the other language will probably lose productivity and quality in one group or the other.

Neither VB .NET nor C# is an inherently better language. Base your language decision on the dozens of differences between the languages, compilers, and design-time support, as well as the future direction of each language.

9. Because of statement completion, you actually type a little bit less in VB .NET than C# for most constructs.

10. You should know the basic behavior of your own language, but you might not know the behavior of other languages. By default, `32767 + 1` is `-32768` in C# and throws an exception in VB .NET. You can alter this default behavior by changing the setting of the Integer Overflow Checking option in the Project Properties dialog box.

11. The resulting value of `f` in `f = 2/3` (where `f` is a double) equals zero in C# and 0.666 (with the correct number of decimal places) in VB .NET. That's because the division symbol is overloaded in C# to perform a different operation with integers than floating-point numbers (integer division instead of floating-point division). VB .NET is more explicit, using a different operator for integer division (`\`).

Looking to the Longhorn Release of Visual Studio

Longhorn is the code name for the next generation of client operating system. It's still a few years away, but this operating system has incredible potential. I won't discuss its features because you can find more complete information at `http://msdn.microsoft.com`, except to say that I'm really excited about what's coming. Instead, I'll touch on specific features the corresponding Visual Studio release could include to make application code generation easier.

Marking Generated Code Files

Within Visual Studio, I'd like to see an indication of whether a programmer or code generation process created a particular file. Indicating whether code was handcrafted, generated by Visual Studio, or generated by external code generation mechanisms will help programmers keep their bearings, particularly during debugging. This might extend to infrastructure files that the programmer should be able to read but not edit. I'd like to see a visual indication—such as a light gray background—for read-only files both during designing and debugging. At the same time, it's important to retain all other capabilities, such as code coloring and collapsible regions. Source control offers some of this today, but Visual Studio could do more to indicate the source of various code segments.

Supporting a Standard Hash Mechanism

Standard hash mechanisms are important for generated code. Ideally the same hash mechanism would be used for application code generation, CodeDOM generation, and UI generation such as WinForms code generation. As part of this, the hash mechanism could correctly handle language-specific whitespace and perhaps semantic issues such as the order of assignments for code generated by code serialization. This is important to allow different code generation mechanisms to recognize handcrafted code, even if the file was created using a different harness. Today there's no reason to assume different harnesses would use the same hash algorithms, header marking characters, or whitespace handling. As part of this, I'd also like to see Visual Studio to include a hash marker on generated files and regions and offer specific warnings when programmers edit the associated code.

NOTE This could be implemented with a new method, perhaps `GetSourceCodeHash`, with parameters indicating whether insignificant whitespace, comments, and noncritical source code ordering was considered. Any hash code mechanism consistently used by everyone doing code generation would be vastly preferable to the current scenario where harnesses can't recognize whether files created by another harness have been edited. For a complete solution, we also need a mechanism for standard markers.

Providing an XSLT Jump to Template at Design Time

In XSLT, I'd like to be able to jump to a definition (such as `<xsl:template>`) in Visual Studio at design time, just like I can jump to a subroutine in .NET languages. Thus, if positioned on a `xsl:call-template` element, I'd like to be able to jump to the template I'm referencing, either in the same file or an imported file. Similarly, I'd like to be able to jump to a variable declaration.

Supplying a Template-Based Strongly Typed DataSet

The strongly typed DataSet is currently created in a black-box approach using the CodeDOM. You can't alter the template. I'd like to see the strongly typed DataSet, or its successor, revamped into a template-driven mechanism. Microsoft could provide several templates, including the derived strongly typed DataSet we have today, a wrapped DataSet that would block access through untyped methods, and a business object that provided similar behavior with a much lower overhead.

Enhancing the CodeDOM

The CodeDOM is a very interesting tool. Unfortunately, its limitations and the difficulties in working with it make it almost impossible to use today in real-world scenarios. A lack of clarity in its future direction further undermines its use. For example, will it eventually support the full Common Language Infrastructure (CLI) specification? Eventually the CodeDOM must be enhanced or allowed to die. Because the types of things you'd do with the CodeDOM represent long-term investments, clarity on the future role of the CodeDOM is important.

Officially, the future of the CodeDOM isn't in question. It's as much a part of the framework as reflection or any other aspect. However, the nature of the CodeDOM makes it different from most other aspects of the framework. Because it produces code, its value lies in the ability to create up-to-date code, and if it can't create code that matches the current capabilities of most languages, its value greatly diminishes.

Microsoft can partially address the issue by updating the CodeDOM to meet the full CLI specifications. But the CLI specification is likely to remain significantly behind the capacities of the mainstream languages. Many tools only need to address a subset of languages—such as VB .NET and C#. It needs to be easier to extend the Microsoft language providers. But that in itself will propagate dozens of versions of providers and derived versions of `CodeObject` classes. It'd be far better for Microsoft to provide a standard language provider and a set of derived versions of `CodeObject` classes (or, far less optimally, standard `UserData`) that support features common to VB .NET and C#.[12]

CodeDOM snippets are currently an inadequate hack to manage unsupported CodeDOM features. They don't work for many things, such as `For Each` loops. If Microsoft extended the CodeDOM, code snippets might not be needed. If they're required, it makes sense to expand the use of CodeDOM snippets with the concept of a snippet set. A snippet set would be a collection of snippets, each with a language identifier. When the language provider outputs the code, it'd select the correct language snippet from the set. A mechanism for nesting CodeDOM-generated code within a snippet would solve problems such as the `For Each` loop. However, I'd prefer that be fixed by better providers.

The majority of programmers will never use the CodeDOM, so it's easy to ignore. However, Microsoft's use of the CodeDOM extends that touch to a large percentage of programmers. Bringing the CodeDOM up to the standards set by the rest of .NET would allow all programmers to access more robust tools available from vendors and in-house groups. These fixes would make the CodeDOM workable, and later it might help fulfill the potential for a common representation of code, as described in "Developing a New Vision of Code."

Refactoring Code and/or Snippets to Templates

Programmers' capacity to learn and use the CodeDOM would be greatly enhanced by a tool that could reverse-engineer code into a code graph—currently hampered by the many code constructs not supported by the CodeDOM. Although it's an extremely difficult task, it could be done today by

12. These extensions need to support common directives, such as regions, in addition to code itself. They also need to support language-specific comments—such as the comments for XML documentation.

people outside Microsoft.[13] That would lead each person working on the problem to devise their own extensions to the CodeDOM objects to represent currently unsupported structures—which wouldn't be constructive.

Building New Paradigms

It's time for significant changes in this industry. However smart, dedicated, and ethical individuals are, the industry sucks money—blood—from the very people it sets out to help. Cost overruns are the norm, a frightening percentage of projects are never finished, and we have lost the trust of the very management that feeds us. Paradigm shifts are needed to change this picture, and many people are working toward that in different ways.

These paradigm shifts aren't likely to be just technical. We have spectacular tools and solid techniques such as object-oriented programming. The answers are likely to be new paradigms for the development cycle and how we work together. With the power of code generation in your hands, you may see enough to revolutionize your own development. That's a piece of the upcoming changes, but not enough to get the industry through its current phase of puberty. I've been in this industry long enough to be cynical about new visions, but I'm optimistic about the changes code generation will bring. In itself, code generation is an exciting new paradigm, and the principled five-step process of code generation also pushes us toward two other important paradigm shifts. These are using collaboration-based programming and establishing a new definition of the project life cycle.

Inventing Collaboration-Based Programming

New ways of thinking are sometimes hard to understand. I'll jump ahead, draw a picture of collaboration-based programming, and then jump back and show you how you can get there:

By 2009, programming is no longer a solo or small group endeavor. It's truly a team sport. The strange thing is that you've never met some key members of your team. You elected to build a trust relationship with someone you may have never met—a third-party vendor, a famous writer, Microsoft (or Sun), or a research group in your own organization that might itself have a trust relationship with an outside party.

---

13. My son has been building a tool to create a CodeDOM graph of VB .NET code as the first step in an automated translation tool. This would allow the translation process to move beyond search and replace of semantic elements.

You aren't married to this trust partner—it's a normal business relationship that can end without damaging you or your application. It's more like the relationship you have with your mechanic than the one you have with your spouse. And like your relationship with your mechanic, you elect the degree of involvement the mechanic has with your car or your trust partner has with your code.

This is collaboration-based programming, and it involves a standards-based application generation harness that extracts metadata, runs code generation, and incorporates your handcrafted code and infrastructure. Experts in architecture and development provide templates and infrastructure that work smoothly together, and you elect whether to use one or more of these.

To convince you to use their tools, these experts are competing at a high level, adding real features to your development such as improved documentation and automated testing. They're also working to provide a high level of security and runtime performance because you have the luxury of running your own application with multiple vendors' offerings and comparing the results in your own environment.

We have reached a point in application development where there's a formal conduit for the greatest minds in the industry to contribute almost directly to your application's success—not just through white papers and books, but also through infrastructure and templates.

Did you catch it? The new paradigm isn't about how you deal with the computer. It blows my socks off how good many people in this industry are at solving problems with computers. It knocks me down how good my 16-year-old son is at structures and algorithms. We know how to *do*, and we know how to *teach*. We don't know how to *share*.

The fundamental flaw isn't in how we deal with the computer or these spectacular languages but in how we deal with each other. We've had no mechanisms for true partnerships. Sure, you partner with Microsoft when you buy their language, but it doesn't yet offer you full solutions you can tie into. Sure, you can tie into a third-party vendor, but you have to tie yourself to them—tying to a sailboat to get through a storm with no way to extricate yourself later. There could be several architectures available to jump-start your application, with you able to try out different architectures on your own data. With this level of direct communication, you can implement an architecture that's right for your application and upgrade it as architectures evolve. You get help right where you need it most.

So, what does it take to get to this vision of being able to trade out vendors if they're no longer meeting your needs? And how could you develop a trust relationship that isn't an all-or-nothing deal? What are the mechanics of getting there? It's pretty simple—you need the following:

- A standardized XSD for input metadata
- A set of standard interfaces for the generation harness script and how it calls templates

- Standard interfaces for the interaction between handcrafted code and generated code
- A standard format for defining Object Relational Mapping (ORM)
- Standard interfaces for each logical and physical tier

Collaboration-based programming is good for everyone who creates applications and everyone who supports programmers. It allows you to work together at an unprecedented level and reduce the number of times people are each "reinventing the wheel" by building similar solutions from scratch.

Inventing a New Project Life Cycle

The second paradigm shift involves radically rethinking of the project life cycle. The current project life cycles are some variation of this as follows:

1. Gather requirements.
2. Design.
3. Code.
4. Test.
5. Deploy.
6. Maintain.
7. Watch the application become obsolete because of business or technology changes.
8. Start over at step 1.

There are a number of problems with this cycle. They include the high risk associated with a monolithic (rather than iterative) requirements gathering phase, quality risks associated with delayed testing, and cutover issues with a single deployment. Most of those issues can be at least partially addressed with existing models such as iterative development and agile methods such as eXtreme Programming (XP). Code generation will help you incorporate these methods and improve the nature of this cycle.

But that doesn't address the fundamental problem. The problem is step 8. The idea of rewriting an application is fundamentally absurd. Why do we do it, and why does it sound unrealistic to *never* rewrite your application again?

Applications evolve based on two factors—business demands and technology demands. When Microsoft introduced VB .NET, it made a lot of VB 6 software obsolete. Sure, it may run indefinitely, but it doesn't reflect modern methods, will become gradually more difficult to maintain, will fail to incorporate new best practices in key areas such as security, and will gradually look quite old. Within each application, business rules and technology are tightly commingled. Even three-tier applications commingle technological issues—such as how you access a database—with business logic. The business logic is even expressed in terms of technology because it's expressed *in a computer language*.

What if instead you encapsulated your business logic in an abstract form that was independent of technology? Even if this abstraction only represented 90 percent of your business logic, with key complex algorithms expressed in code in specific locations, this abstraction allows you to decouple your business logic from the technology you're currently using to express it. *And you already have the kernel of this abstraction.* It's the metadata you're extracting and using in code generation.

Once you've isolated this business information, the business logic evolves in response to business pressures. Sometimes these pressures will be extreme, such as a database redesign. But you can express these changes through existing templates, significantly decreasing the time to deliver and test changes and limiting disruption.

Templates and infrastructure evolve in response to technology changes. Even a change as great as switching to a new language can primarily be accommodated in templates and infrastructure changes. The remaining changes in proprietary algorithms are isolated where you can easily find them and rewrite them when needed.

Eventually the evolution of the application may result in almost all the code in the application being replaced with new code. That's progress. Avoiding rewriting the application means that a particularly section of code is rewritten only when that code is part of the issues being addressed. Today we have to rewrite our *entire* application when *any part of it* slips into obsolescence. Attempts to circumvent this reality like n-tier applications allow certain types of changes, but eventually fail because they aren't sufficiently granular and continue to commingle business logic and technology.

By ceasing to commingle business logic (metadata) and technology, and allowing them to independently evolve, the potential exists to *never* perform the disruptive rewrites that are the bane of this industry. You'll be able to adopt a true service-based model of long-term iterative development. This service-based cycle would avoid the need to budget big chunky costs and drastically lower the risk of cost overruns and project failures.

Going Forward

The future is wide open for code generation. It's up to all of us to figure out where code generation goes next. There's lots of work to do in establishing standards and improving tools, but most of the work involves putting code generation to work on projects in individual IT shops.

For myself, I have a number of small projects in mind, such as a code generation–specific XSL Transformations (XSLT) preprocessor to make it easier to create XSLT templates, a tool to refactor or reverse engineer code into templates, and a few dozen ideas for books. I have no clue which of those I'll get to first. But I look forward even more to putting the techniques from this book in place via training and consulting. I think code generation's ability to codify the brain of a .NET expert also skilled in code generation is sort of a "Vulcan mind meld" for an organization. I'd like to spend about two months in different organizations using a combination of mentoring, training, master coding, and follow-up troubleshooting to build teams fully competent in their template mechanism with a viable set of templates reflecting the team's chosen architecture and have a target project nearly complete. If you're interested, you can reach me at `Kathleen@gendotnet.com`.

I hope you'll stay up to date with templates on the Apress Web site and on `http://www.gendotnet.com` because I expect the templates and tools to evolve. Start working with code generation, and let me know what problems you encounter and how you work around those problems.

Summary

As you investigate code generation options, hold them up against the five principles of code generation by looking at specific details. The five principles of code generation are as follows:

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

Although code generation has the potential for significant savings, you may only save 20–30 percent on your first project. That's because time will go into

establishing procedures and learning new techniques and because programmers are more likely to solve noncritical issues.

Upcoming features in the next versions of Visual Studio and SQL Server offer new opportunities for code generation, as well as new code generation techniques. The partial classes feature will most directly affect your application code generation.

Beyond these next releases, establishing standards for key elements of code generation will move it into the mainstream because organizations can interchange infrastructure and templates when the need arises.

On the horizon, approaching code generation based on the five principles open up new paradigms. Providing a conduit for industry experts to share their expertise with you allows collaboration-based programming. The separation of metadata and templates allows the potential to evolve each on an ongoing basis and abandon the current approach of rewriting applications every few years.

EXCERPT

APPENDIX A

X-Survival

The X-Files[1] **of XSL Transformations (XSLT),** XPath, Extensible Markup Language (XML), and XML Schema Definition (XSD) can sometimes be as surrealistic as the TV show. You can go off and spend three years becoming an expert in the X technologies, or you can read this appendix, which is designed to let you get right to work with these technologies. I've taken my experience and pulled together what you need to get the job done—Carhartt's[2] for your body, along with duct tape, quarter-inch bolts, and cup hooks. You aren't going to become an expert here, but you'll be functional and able to use X technologies in support of code generation.

Although there's emphasis in this appendix on XSLT for code generation, you'll need much of this information for all three code generation techniques. All three techniques use XML, XPath, and XSD to help get the job done.

Starting the XML Basic Training

Ugly angle brackets.

Why on earth would you dirty up your data with verbose and ugly angle brackets? Adding the angle brackets is important because it allows you to add descriptive tags that make your data self-describing. The little phrase *self-describing data* has gigantic implications and is changing programming as we know it. Self-describing data doesn't need a separate Rosetta stone for either software or humans to be able to process it. When you get the data, you don't have a bunch of bits hanging out in space waiting for interpretation but real information you can use. Information isn't just bits; it's bits with sufficient context so that it makes sense. The characters "4" and "2" in the series of characters "42" might be your jacket size, the year your mother was born, a football play, what you owe your milkman, or the meaning of the universe. In XML, you know without question that it's the height of your youngest child.

---

1. I'll use the phrases X-Files and X-Stuff interchangeably to indicate XSLT, XPath, XML, and XSD. The points I make can often be extended to other X standards, such as XQuery, that aren't specifically covered here. Occasionally, the points can also be extended to Hypertext Markup Language (HTML), but HTML isn't XML compliant. The primary difference between HTML and the XHTML standard is that XHTML is XML compliant (well-formed). The discussion here generally applies to XHTML

2. Carhartt is a brand of popular ranch and work wear.

XML is the goo behind some of the most useful technologies today. As one example, Web Services is nothing more than a universal messaging system on steroids. We've been doing messaging in various forms for a long time. But now XML's self-describing data lets you look up a service you want; self-describing data describes precisely what the service does, and self-describing data is retrieved from and sent to the server. That's about all Web Services is—messaging on steroids using self-describing data.

XML data takes more bytes than a stripped-down version of your data. How much bigger an XML file is than a comma-delimited or fixed-width format depends entirely on the nature of the data and the length of the descriptive tags. Although bloated data is certainly not one of XML's most appealing characteristic, it's rarely a problem. Disk space is so cheap that's it's effectively free unless you have an amazing number of XML documents. Up to a point, the size of packets passed over the network has only a minor effect on performance, and XML compresses very well. Although XML data is larger than a stripped-down version of the same data—because of the self-describing tags—this increased size is rarely an issue.

Web Services and all XML technologies depend on one thing. We have to all speak the same language. If my self-describing language is comma-delimited and yours is angle-brackets delimited, we're going to need entirely separate parsers to shred the data into its component parts. When we all settle on a common structure, we can shift the focus from the data formatting to the data itself. That's what standards are all about. But standard committees tend to be an odd mix of ivory tower theoreticians and vendors.[3] They've done okay. But there are definitely quirks that can bite you in the back side.

One fundamental thing to realize when you're in the X-File world is that it's sometimes best to just follow the rules. You can spend a lot of time understanding the depths of the rules and why they work the way they do. But to skate across the surface of X-File technologies, you can ignore how deep the lake is and just take the rules at face value. It's a careful balance because if you don't know enough of the underlying basics, you'll struggle with things seeming strange and inconsistent. Key information in this appendix will give you a huge jump-start on the technologies and be enough for code generation. There are a lot of other resources, including the .NET Help and resources in the "Additional Reading" section when you need more depth. I've tried to include a lot of keywords you can use to find things in the .NET Help indexes. Once you get to the right topic in the .NET Help, the explanations are often good. But finding the appropriate entry can sometimes be a challenge.

---

3. The important standard committee for X-Stuff is a W3C committee that you can find at `http://w3c.org`.

Learning the Rules

XML includes angle brackets that hold information in elements and attributes, each of which also contains a descriptive tag (you hope). You use XML to hold your data, and many related technologies, such as XSD and XSLT, are themselves XML. A good place to start is with some XML basics, but there really isn't a whole lot to XML itself. A couple of constructs and a few rules—that's it. There are a few (occasionally nasty) nuances, and the supporting technologies each have their own quirks and personalities.

Introducing Elements

Elements are delimited by angle brackets. Each element has a name, which appears immediately after the first open angle bracket. The element name is core to the self-describing features of XML. The element must be closed with either a slash at the end, such as the following:

```
<elementName/>
```

or a separate bracketed entry starting with a slash:

```
<elementName>stuff</elementName>
<elementName></elementName>
```

Data appearing between the start and end directives are the element's text. The first line in the previous examples shows an element with the text "stuff," and the second shows an empty element. You can also place other elements between the open and close tag. The next example is an element containing child and grandchild elements nested with their parents:

```
<elementName>
   <childElementName/>
   <childElementName>
      <grandchildElementName/>
   </childElementName>
</elementName>
```

When you load or create an XML document as an `Xml.XmlDocument` object in .NET, elements can be accessed as `XMLElement` objects, which is one of about

a dozen classes derived directly or indirectly from `Xml.XmlNode`. Although other classes, including `XmlDocument`, are derived directly from `Xml.XmlNode`, `XmlElement` is derived directly from `XmlLinkedNode` (which is itself derived from `XmlNode`). `XmlLinkedNode` adds features for placing a node in a hierarchy, such as an understanding of the next and previous sibling in the list. Figure A-1 describes this relationship.

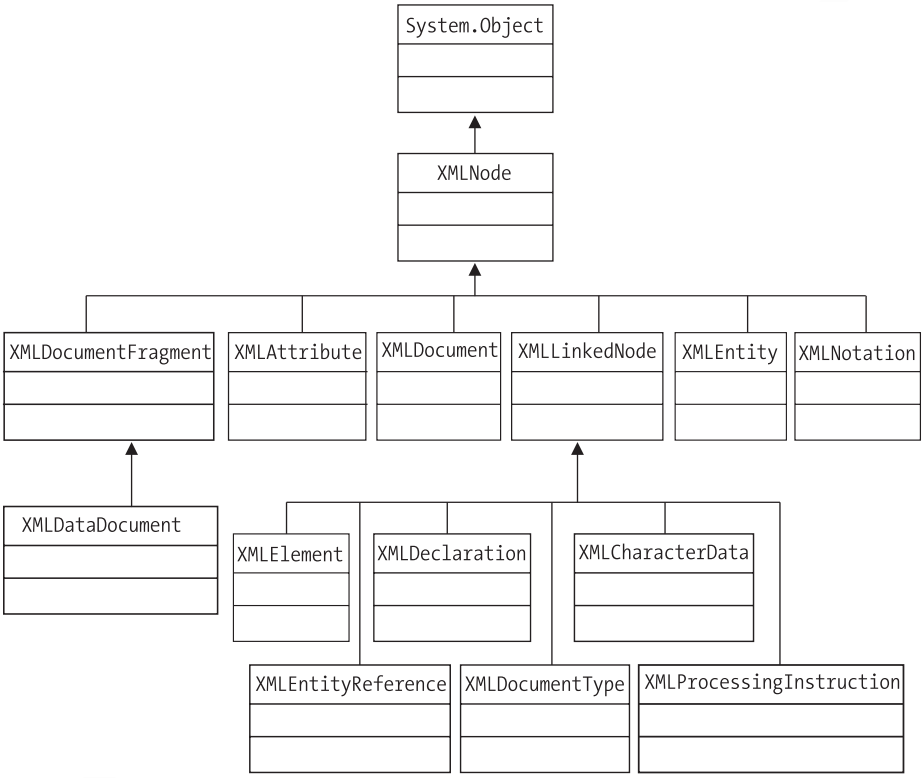


Figure A-1. The relationship between a few core objects of the XML namespace

Introducing Attributes

In addition to holding information in the text of elements, elements can contain additional information as *attributes*. The syntax of XML attributes is quite different from .NET Framework attributes, but the purpose is similar—to add additional information that can be retrieved if needed. The text of attributes is contained in single or double quotes following the attribute name. The entire

attribute name and value appears in the element's opening tag. Attributes can be placed on any element. They appear as name-value pairs such as this:

```
<dbs:TableColumn Name="CustomerID"
                 AllowNulls="false"
                 SQLType="nchar"
                 NETType="System.String"
                 MaxLength="5" />
```

If you load or create an `XmlDocument` in .NET, you can access attributes as `XMLAttribute` objects—another .NET Framework class derived from `XmlNode`. Although this derivation may seem odd, it's consistent with the World Wide Web Consortium (W3C) concept of attributes being a kind of node nested within the element with extra restrictions and displayed in a special way (within the parent's opening element tag). Because the `XmlAttribute` class is derived directly from `XmlNode`, it doesn't have a concept of order within the list because attributes are defined as a nonordered list.

A key difference between elements and attributes is that elements can have children but attributes can't. This is a rule you'll want to keep in mind when you're designing XML structures. If your data includes multiple instances of an item, or items contain child information, then you can't create it as an attribute.

Introducing XML Rules

XML works with a small number of rules. When a document follows these rules, it's legal XML, also referred to as *well-formed XML*.

- XML documents have a single root element.
- Syntax elements such as angle brackets, and quotes must be closed and cleanly nested.
- All open element tags must have matching close tags.
- Elements must be cleanly nested and not overlap.
- Each attribute can appear only once in any element.
- Element and attribute names must be legal XML identifiers—they can't contain spaces or punctuation (other than the underscore) and can't start with a number.

- Certain characters, primarily < and &, can't be used directly within the text of elements or attributes and must be escaped as `<` and `&`, respectively.

- XML files generally start with a standard header that includes the XML declaration. Although not required, this header is recommended.

NOTE There are other requirements, called *well-formedness constraints* in the XML specification, but these are the most important, and many of the others are esoteric.

At the top of most XML data is the *XML declaration*, which declares that what follows is XML data. It's not required, but it's strongly recommended. When you include it, it needs to include at least the following:

```
<?xml version="1.0" ?>
```

The question mark indicates that it's an XML declaration, not part of your data. This header indicates that it's an XML file and indicates which version of XML the file uses. Only 1.0 currently exists, so all XML declarations use this attribute. The declaration might also contain additional attributes. You can generally use defaults for the other attributes by leaving them out.

Clean nesting means that the XML is a simple hierarchy. Child elements must be closed before the parent element can be closed. The same thing goes for matched punctuation. Examples of invalid elements are as follows:

```
<elementName attributeName="fred>"
<elementName><childElement></elementName></childElement>
```

The first line is an error because it has no closing angle bracket because the less-than sign is contained within the quotes of the attribute `attributeName`. The second isn't cleanly nested because the closing tag of the child element occurs after that of `elementName`; the elements overlap.

Handling Special Characters

You've got to take special care with some characters, particularly the less-than (<) and ampersand (&). The less-than sign is otherwise interpreted as an

opening angle bracket on an element. If you use it in a way illegal as an opening bracket, it breaks the XML. The ampersand indicates that a special escape sequence follows, so it'd also be interpreted incorrectly in your XML. Replacing these characters with special codes is called *escaping*. Table A-1 shows several examples of escaping. Tools will often convert these characters for you when the data is coded and encoded.

Table A-1. Valid Escape Sequences for Displaying Different Characters

XML	EFFECT
`<xsl:text>
</xsl:text>`	Carriage return and line feed; contained within XSLT text elements. This is a normal end-of-line sequence.
`<xsl:text> </xsl:text>`	Tab; contained within XSLT text elements.
`<xsl:text>  </xsl:text>`	Three spaces; contained within XSLT text elements.
`\`	Backslash, demonstrating how to escape any character via its hexadecimal value.
`\`	Backslash, demonstrating how to escape any character via its decimal value.
`>`	Greater-than symbol (>) isn't required but is symmetric with `<`.
`<`	Used to output a less-than character (<).
`&`	Ampersand (&).

NOTE Sometimes you need to display an escaped sequence in the output. This frequently occurs with the common ` ` in HTML output. You can explicitly force the correct output with `&nbps`.

That's it. Those are the core rules for creating legal XML, also called *well-formed* XML. If your document doesn't contain legal XML, you won't be able to load it as an XML document in .NET; you'll get a runtime exception on the `XmlDocument.Load` method.

Introducing Whitespace

The layout of elements—the whitespace characters such as a space, carriage return, tab, or line feed, that appear between a closing angle bracket and the next opening bracket—is frequently not important; the whitespace is included just to make the XML easier for humans to read. This whitespace between elements is called *insignificant whitespace*. Whitespace before and after attributes in the element opening tag are also insignificant. Whitespace within the text of individual elements is called *significant* whitespace.

Another X-File fundamental is that insignificant whitespace isn't under your control. Many tools will rearrange the XML according to its own rules. You have to let go and not worry about how insignificant whitespace is handled within your XML documents.

Lions and Tigers and Bears, Oh My!

Perhaps when looking at .NET overall, it's fair to say, "You're not in Kansas anymore." Well, when working with X-Stuff, you aren't on the yellow brick road anymore either. Sort of like the forest, it's different and maybe even scary, but the real dangers are minimal. The biggest gotchas in the X-Files revolve around a lack of feedback and namespaces.

Working with Minimal Feedback

When working with XML, you need to suspend your normal .NET expectations. You should no longer assume that if you mess up, the tools will notice at either design time or runtime. For example, you're often working with XPath to find matches within your document. These matches are case sensitive, and if you incorrectly type your request, you'll rarely see an exception; instead, you'll simply get no results. To determine the source of the problem, you must determine whether the file had been loaded, whether it contained the nodes you're looking for, and whether your request was correct. We rarely realize how much we depend on the design time and runtime feedback until working without it.

This isn't Microsoft's fault. It's caught between your normal expectations and W3C standards. The standards assume you know what you're doing and you don't make dumb mistakes. Microsoft is more realistic. Its overall strategy in .NET is to help you avoid dumb mistakes and find them quickly when you do make them. Actually, it isn't fair to blame the standards committee either. You really can't have the same level of help and oversight in text processing.

Where it's consistent with W3C expectations, Microsoft does give you good feedback. For example, if you try to load a document with illegal XML, .NET will

raise an exception explaining the problem and the line in your XML file where it occurs.

There aren't any ruby slippers—you'll have to work with this. My prime suggestion is don't let yourself get stuck for more than an hour or so. If a problem is taking a really long time, you're probably looking at it wrong or looking past the answer. Verbalize the problem to a coworker, a newsgroup post, or maybe even your dog. X-Stuff presents new ways of thinking, but the most fundamental difference in how you work with it and debug is that X-Stuff expects you mean exactly what you say, even when it's not sensible.

NOTE Chapter 5 covers debugging XSLT.

Surviving Namespaces

A *namespace* is a unique identifier you use to sort out ambiguous naming. Yep, that's all it is. This is an important aspect of self-description. I tell you a piece of data is 42, and that's part of the meaning of the universe. You don't know if that is Douglas Adam's opinion, the mass of the universe in some physicist's units, or an encrypted character from the Golden Rule. You need context, and you need an entirely unique context. That's a namespace.

As the world expands and becomes more interactive, the names you create for elements can conflict with the names someone else creates. Even if you're doing work on your own, conflicts can arise with vendors, another department, or a company that hires you next week. It makes sense to use namespaces that make your names unique throughout the known universe.

NOTE It's hard to understand namespaces until you're using them. If the following sections put you to sleep, skip to the section "Dealing with the XSD Drill Sergeant" and come back to namespaces when you're using them and trying to understand their issues. The section "Making Namespaces Work" summarizes techniques to make namespaces work in your applications.

Looking at Namespaces

Because a namespace is nothing more than a unique identifier, it can be quite literally anything. Many people confuse this when using Uniform Resource

Locators (URLs) for namespaces. Technically a URL is a URI in a particular format that's used to identify resources on the Web. A URL looks something like this:

```
http://kadgen.com/appendixA/lookingatnamespaces
```

When you look at a namespace, you're supposed to think, "Ah, a URI, that's just an identifier." My fingers still sometimes reach for the mouse to click the URL or start looking for a file, and Visual Studio highlights it as a link. Although it may seem silly, it isn't going to go away because there's a really good idea behind the practice of using URLs as unique identifiers. If you think, "Hmmm, how do I find a namespace name that's unique throughout the known universe," a logical answer is, "I'll use my company's registered domain followed by some stuff I make up." There's no requirement that namespaces be URLs, and you'll see a lot of variation. For example, Microsoft uses a Uniform Resource Name (URN) for their XSD extensions. An example of a URN is as follows:

```
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
```

NOTE `kadgen.com` is one of my registered domain names. When using the code in this book, change the namespaces to ones unique to you and your organization.

Understanding Namespace Scope

Knowing why you use namespaces is great, but where do they appear in an XML file, and how do you use them? You specify a namespace using the `xmlns` attribute on any element. The namespace resolves the name for the element it's applied to and all child elements. Unless you can articulate the reason why you're using a namespace on any element other than the root, don't. It'll confuse people. There are some good reasons to use multiple namespaces, including validating different parts of your document against different schemas, but don't create extra namespaces without a reason. The most common use of namespaces is on the root element:

```
<?xml version="1.0" standalone="yes" ?>
<kg:GenerationScript xmlns:kg="http://kadgen.com/KADGenDriving.xsd">
```

You can prefix the namespace with an abbreviation. You include the prefix in the declaration immediately after the `xmlns` attribute—such as `xmlns:kg` to define a prefix of `kg`. The section "Introducing Namespace Prefixes" discusses prefixes in

more detail. In the next example, the stylesheet element (the root element of an XSLT stylesheet document) declares three namespaces. Two are W3C-defined namespaces, and the third (with the `dbs` prefix) is a namespace I added for processing this document:[4]

```
<xsl:stylesheet version="1.0"
             xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
             xmlns:xs="http://www.w3.org/2001/XMLSchema"
             xmlns:dbs="http://kadgen.com/DatabaseStructure" >
```

Explaining tempuri.org

I prefer sushi myself.

No, this isn't really a Japanese dinner dish. Instead, it's the standard way to handle a common situation. When you first create certain X-Files (especially the XSD schema) via a tool, the tool has to define a target namespace. Microsoft uses `tempuri.org`, which means "temporary URI." If you've followed the discussion so far, you understand that a namespace is a unique identifier or at least should be. Because all the Microsoft tools use `tempuri.org`, this is probably about the last identifier you'd want to use.

You should change this namespace on every schema you create, even strongly typed DataSets, even if other people aren't always doing this. You may want to create a departmental temporary URI and put it in the default templates introduced in the sidebar "Changing Visual Studio Templates," particularly if you create a lot of transient XSDs. Alternatively, you could search your projects for *tempuri.org* as your project moves into steady development.

Changing Visual Studio Templates

Visual Studio .NET uses templates when you create new items. It uses them when you add a new project or any type of file to a project. Unless your organization uses Enterprise Templates and has locked down access to the templates, you have control over them.

The first step to changing them is to find them. For Visual Basic .NET, most templates reside in a special template directory: [Program Files]\Microsoft Visual Studio .NET 2003\Vb7\VBWizards\[Item]\Templates\1033. `[Item]` is the name Visual Studio uses for the particular template. For example: [Program Files]\Microsoft Visual Studio .NET 2003\Vb7\VBWizards\XSLTFile\Templates\1033.

---

4. W3C happens to have pages at the locations described in the first two namespaces. But there's no reason these URIs need to point to a Web page.

For C#, most templates reside in this directory:

```
[Program Files]\Microsoft Visual Studio .NET 2003\VC#\CSharpProjectItems
```

You'll want to use a folder view of your computer and poke around until you find the template in which you're interested.

Visual Studio scripts use these templates to create new items. The script replaces key values within the template. For example, the default Visual Basic .NET class template is as follows:

```
Public Class [!output SAFE_ITEM_NAME]
End Class
```

You can change this in any way you'd like and save the result. For example, you could change this to the following:

```
Option Strict On
Option Explicit On

Imports System

Public Class [!output SAFE_ITEM_NAME]

End Class
```

The XSLT default provided for both C# and Visual Basic .NET is as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<stylesheet version="1.0" xmlns="http://www.w3.org/1999/XSL/Transform">
</stylesheet>
```

There are several problems with this. It doesn't use a prefix, and it doesn't include all the goo that I'd like to just stick into the template and forget about. The modified version I use is as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!— Summary:  !—>
<!— Created: !—>
<!— TODO: !—>
<xsl:stylesheet version="1.0"
```

```
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
          xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xsl:output method="text" encoding="UTF-8" indent="yes"/>
<xsl:preserve-space elements="*" />

<xsl:template match="/">
</xsl:template>

</xsl:stylesheet>
```

I suggest you take some time to upgrade the templates Visual Studio provides in order to start each project with a robust template that handles issues you'd otherwise have to deal with every time you create a new project or item.

Introducing Namespace Prefixes

Most of the samples you've seen use a namespace prefix such as `xmlns:xs` and `xmlns:msdata`. This is a good idea in many situations. The goal of prefixes is to easily refer to individual namespaces and resolve multiple namespaces within a single document. For example, in the following XSD fragment, four namespace/prefix combinations are defined (two happen to point to the same namespace and one prefix is blank—discussed in the section "Default Namespaces"):

```
<xs:schema id="GenerationScript"
          targetNamespace="http://kadgen.com/KADGenDriving.xsd"
          xmlns="http://kadgen.com/KADGenDrivingkd.xsd"
          xmlns:mstns="http://kadgen.com/KADGenDriving.xsd"
          xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
          attributeFormDefault="qualified"
          elementFormDefault="qualified">
```

Child elements can use any of the specified prefixes `mstns`, `xs`, and `msdata`. An important secondary benefit of namespaces is allowing you to see the general area where each attribute or element applies.

A prefix resolves conflicting names that occur within a single XML file. They're nothing more than an abbreviation valid in the current context. The healthiest

way to look at prefixes is to realize that they mean "put the specified namespace right here." The following document:

```
<?xml version="1.0" standalone="yes" ?>
<kg:GenerationScript xmlns:kg="http://kadgen.com/KADGenDrivingx.xsd">
   <kg:Section Name="PreProcessing" Checked="false" />
</kg:GenerationScript>
```

is actually processed as this:

```
<?xml version="1.0" standalone="yes" ?>
<http://kadgen.com/KADGenDrivingx.xsd:GenerationScript
                           xmlns:kg="http://kadgen.com/KADGenDrivingx.xsd">
   <http://kadgen.com/KADGenDrivingx.xsd:Section
                            Name="PreProcessing" Checked="false" />
</http://kadgen.com/KADGenDrivingx.xsd:GenerationScript>
```

Wow, the second one gives me a headache just looking at it! XML would be unreadable if you had to include namespaces in this manner—not to mention the nightmare that changing namespaces would be. So, it's a good idea to alias namespaces via prefixes. The scope of the prefix is (perhaps obviously) the same as the scope of the namespace within the same document. There's no technical reason you can't redefine the same prefix to a different namespace in different parts of the same document. If you do that, you have a sick sense of humor, and I never want to look at your XML.

You can assign the same prefix to different namespaces in different documents. Again, it's the humans who suffer if you do this. Work to establish a small set of standardized namespaces and prefixes and use them consistently within your organization. Table A-2 shows some frequently used namespaces and standard prefixes for them.

Table A-2. Standard Prefixes for Commonly Used Namespaces

NAMESPACE	DESCRIPTION	PREFIX
http://www.w3.org/2001/XMLSchema	XML schema definition	xs or xsd
http://www.w3.org/1999/XSL/Transform	XSLT directives	xsl
urn:schemas-microsoft-com:xml-msdata	Additional information added by Microsoft inference tools	msdata

The Default Namespace

I have a bias against default namespaces. Every time I use them, I seem to regret it. I'll explain them, along with the caveat that for nontrivial work with XML they can cause problems. How do you think the following `Section` element (ignoring the attributes) is processed?

```
<?xml version="1.0" standalone="yes" ?>
<GenerationScript xmlns ="http://kadgen.com/KADGenDrivingx.xsd">
   <Section Name="PreProcessing" Checked="false" />
</GenerationScript>
```

Did you guess that it's processed exactly like the sample in the previous section? Everything that you don't prefix is processed in the namespace `http://kadgen.com/KADGenDrivingx.xsd`. Using `xmlns` without a prefix specifies a *default namespace*. Remembering that the namespace prefix is just a shortcut for the namespace, this line effectively says, "The lack of a prefix in this or a descendant element is a shortcut for this namespace identifier." Using a namespace without a prefix creates what's called a *default namespace*.

If you were curious about the `attributeFormDefault` and `elementFormDefault` attributes in the earlier example, they'll make more sense now. Setting each of these to "qualified" indicates that all attributes and elements within the XSD should be resolved to a specific namespace. When you look at the XSD a little later in this chapter, you'll notice some of the attributes don't have prefixes. The default namespace is used for those attributes.

The default namespace is different from an empty namespace that's an unspecified default namespace. The `Section` element in the following similar sample:

```
<?xml version="1.0" standalone="yes" ?>
<GenerationScript>
   <Section Name="PreProcessing" Checked="false" />
</GenerationScript>
```

is processed exactly as written because there's no namespace present. The default namespace in this sample is an empty string because it's not specified. Thus, the presence of the default namespace changes your processing of the current and all child nodes.

Using XML Namespaces in .NET

XML namespaces become a bit more complex within .NET. They're very important because you use XPath extensively when you're working with an `XmlDocument`, and XPath has to resolve to the correct namespaces to get the job done. .NET provides an `XmlNamespaceManager` to specify what prefix matches up with what namespace. If you don't understand how this works under the covers, the behavior seems downright bizarre.

Namespaces within the document are effectively resolved when the document is loaded. Thus, the `Section` element in the following example:

```
<GenerationScript xmlns ="http://kadgen.com/KADGenDrivingx.xsd">
   <Section Name="PreProcessing" Checked="false" />
</GenerationScript>
```

is handled by .NET as this:[5]

```
<http://kadgen.com/KADGenDrivingx.xsd:Section
        http://kadgen.com/KADGenDrivingx.xsd:Name="PreProcessing"
        http://kadgen.com/KADGenDrivingx.xsd:Checked="false" />
```

You probably don't want to write XPath statements using this long name (and it'd make maintenance a nightmare), so .NET requires that you specify what prefix will indicate this namespace in your XPath statements. The original prefix of the document is completely unimportant to processing. You're assigning a prefix so that .NET can relate your XPath prefixes to the *namespace* defined for the XML element. I suggest it match the prefix used in the document, but this is only for the sanity of humans; .NET couldn't care less. You define and use namespaces like so:

```
Dim nsmgr As Xml.XmlNamespaceManager =
             New Xml.XmlNamespaceManager(mxmlDoc.NameTable)
nsmgr.AddNamespace("kg", "http://kadgen.com/KADGenDriving.xsd")
nsmgr.AddNamespace("xs", "http://www.w3.org/2001/XMLSchema")
```

When working with nodes, you use namespace prefixes like so:

```
Dim xNode As Xml.XmlNode
xNode.SelectNodes("//kg:Section", nsmgr)
```

5. This is a simplified explanation. .NET uses a name table for the resolution and doesn't actually treat the node as this long string.

When working with XPath expressions objects, you specify namespace prefixes like so:

```
Dim xpathExpr As Xml.XPath.XPathExpression = xpathNav.Compile("//kg:Section")
xpathExpr.SetContext(nsmgr)
```

There is one scenario where you can't use the same prefix in .NET that you're using in the XML document. The namespace manager of .NET doesn't cope with blank prefixes to indicate a default namespace. Because this is just a local abbreviation, you can use any abbreviation you want to assign to the appropriate namespace. You can use `tns` for "this namespace." I like `zzz`. It really doesn't matter, except that your life will be easier if you're consistent. .NET uses this prefix to point to a namespace, and you can assign this namespace to match the default namespace of your XML document via the `XmlNameSpaceManager.AddNamespace` method.

Making Namespaces Work

Clear as mud? You can continue to delve into this until you understand it, or you can use a few simple rules to practice safe XML as follows:

- Use prefixes for namespaces in XML documents you control.
- Use a finite number of namespaces and document a prefix for each one that you'll consistently use for that namespace in all locations, including .NET and XSLT.
- If you're dealing with a default namespace from an XML document you don't control, add a standard namespace such as `tns` or `zzz` to .NET and use it for processing in .NET.

Visual Studio doesn't know about these rules, so this differs from the default template behavior. You might want to make changes to the default templates as described in the sidebar "Changing Visual Studio Templates."

Dealing with the XSD Drill Sergeant

Earlier you saw the rules that make an XML document legal or well-formed. Very few rules have to be met to make an XML document well-formed. This offers no assurance that the structure of the document is what you expect or that the document will work with your code. To do this, you need a *schema*, or description of what elements and attributes are valid and how each should be used.

The value of a schema extends well beyond this core purpose. You can't just pick up an XML document and understand the structure of documents of that type. This is impossible because you can't know what optional elements happen to be missing from your sample document. As an example, XML handles null data values by leaving out the element. If you have an XML document representing a database table, you don't know what columns the table might contain that are null for all records in your sample. Because schemas define the full structure of the document, they even allow you to start with a blank document and add data to build an XML file valid for your scenario. An additional benefit of schemas is that user interfaces can be intelligent if they know the document to expect. You'll see this in .NET editing of XML files using IntelliSense features and in the code generation harness used in Chapter 3.

The structure of XSDs is kind of ugly and takes some getting used to. But the tools are good enough that you rarely have to read or edit them directly. Core tools are as follows:

XSD design surface of Visual Studio .NET: This is used when either a DataSet or XSD is added to your project. This allows you to visually edit many XSDs.

XSD.EXE: This infers schemas from XML documents either from the command line or within Visual Studio when you right-click an XML document.

Microsoft XSD Inference 1.0: Downloadable from `http://www.gotdotnet.com`, this contains an alternate inference engine that will infer schemas from many XML documents that XSD.EXE can't handle.

GetXMLSchema: This is a method of the ADO.NET DataSet class that can be used to create a schema for a DataSet programmatically.

You can use XSDs created through these tools when you edit XML in Visual Studio .NET. If the XSD is available in the same directory as the XML file, or in a special directory as described in the sidebar "How Visual Studio Finds Schemas," you'll get editing support including IntelliSense in XML based on your own schema. You associate an XML with an XSD when the `targetNamespace` of the XSD matches a namespace in the XML document. You can also use the `XMLValidatingReader` to validate XML documents against XSDs when you're opening the XML documents programmatically.

Looking at Other Schemas Definitions

In the relatively short history of XML, there have been many schema definitions. XSD is the latest and greatest in this series and the second to be a common

specification. The other common specification is Document Type Definition (DTD). Although part of the original XML specification, DTD isn't itself XML. External Data Representation (XDR) was Microsoft's interim `xsi:schema` introducing many of the concepts now expressed in XML. XSD is itself XML. This is valuable because you can use XML tools and access techniques such as loading an XSD into an `XmlDocument` object. There is minimal support for DTD and XDR in .NET, and XSD is the only schema definition discussed further in this appendix.

Exploring XSD Schemas

Although you rarely need to create XML schemas from scratch, it's convenient to be able to read and understand them. To help you become familiar with XSD techniques, I'll walk through an example. Chapter 3 describes using the harness XML that drives the code generation harness, and this is the XSD describing the XML for that script. This schema uses some advanced features such as named complex types. An XML document based on this XSD might look like this:

```
<?xml version="1.0" standalone="yes"?>
<kg:GenerationScript xmlns:kg="http://kadgen.com/KADGenDriving.xsd"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation=
          "http://kadgen.com/KADGenDriving.xsd ..\..\XFiles\KADGenDriving.xsd" >
  <kg:FilePaths>
    <kg:FilePath Name="BasePath" Path="..\..\" />
    <kg:FilePath Name="MetadataFileName"
                 Path="[BasePath]Metadata Files"
                 File="Northwind.xml" />
  </kg:FilePaths>
  <kg:Section>
    <kg:Standard Name="Prepare Metadata" Checked="false" />
    <kg:CreateMetadata Server="localhost" Database="WxManager"
                 SkipStoredProcs="true" SelectPatterns="" SetSelectPatterns=""
                 RemovePrefix="true" LookupPrefix="">
      <kg:Standard Name=" Northwind Extraction " Checked="true" Ordinal="0" />
      <kg:OutputRules HashOutput="true" OutputFileType="XML" _
                 OutputGenType="Overwrite" CheckOut="false" CheckOutProject="" />
      <kg:SinglePass OutputFile="[Metadata]" />
      <kg:AlternateProcess />
    </kg:CreateMetadata>
  </kg:Section>
</kg:GenerationScript>
```

You can refer to this document to see how aspects of the XSD relate to the XML it describes. Like other XML files, XSD schemas start with the XML header and a single root element. The root element of all XSDs is `schema`. This XSD defines a bunch of namespaces on the root element. You can tell that a Microsoft tool initially created this XSD because of the `msdata` prefixed elements. I'll show you the entire XSD with only repetitive segments clipped (see Listing A-1). Refer back to this XSD as you learn about the key element it uses in the sections that follow.

Listing A-1. The XSD Schema for the XML Harness Files Used with the Code Generation Tool

```
<?xml version="1.0" ?>
<xs:schema id="GenerationScript"
                  targetNamespace="http://kadgen.com/KADGenDriving.xsd"
            xmlns="http://kadgen.com/KADGenDriving.xsd"
            xmlns:kd="http://kadgen.com/KADGenDrivingx.xsd"
            xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
            attributeFormDefault="qualified" elementFormDefault="qualified">
   <xs:element name="GenerationScript">
      <xs:complexType>
         <xs:sequence>

[[ Filters definition snipped because it matches Files definition ]]

            <xs:element name="FilePaths" maxOccurs="1" minOccurs="0">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="FilePath" maxOccurs="unbounded">
                        <xs:complexType>
                           <xs:attribute name="Name" form="unqualified"
                                 type="xs:string" />
                           <xs:attribute name="Path" form="unqualified"
                                       type="xs:string"
                                 kd:Special="Directory" />
                           <xs:attribute name="File" form="unqualified"
                                       type="xs:string"
                                 kd:Special="File" />
                        </xs:complexType>
                     </xs:element>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
```

```
<xs:element name="Section" maxOccurs="unbounded" minOccurs="0"
                kd:CanCheck="true">
   <xs:complexType>
      <xs:choice maxOccurs="unbounded">
         <xs:element name="Standard" type="Standard"
                          kd:auto="true" />
         <xs:element name="CreateMetadata" minOccurs="0"
               maxOccurs="unbounded" kd:NoChildrenInTree="true"
               kd:CanCheck="true
                         kd:Desc="Extracts metadata from database">
            <xs:complexType>
               <xs:sequence>
                  <xs:element name="Standard"
                              type="Standard" />
                  <xs:element name="OutputRules"
                             type="OutputRules" />
                  <xs:element name="SinglePass"
                              type="SinglePass" />
                  <xs:element name="AlternateProcess"
                              type="Process"
                         minOccurs="0" />
               </xs:sequence>
               <xs:attribute name="Server" form="unqualified"
                         type="xs:string" />
               <xs:attribute name="Database" form="unqualified"
                         type="xs:string" />
               <xs:attribute name="SkipStoredProcs"
                            form="unqualified" type="xs:boolean" />
               <xs:attribute name="SelectPatterns" f
                             orm="unqualified" type="xs:string" />
               <xs:attribute name="SetSelectPatterns"
                         form="unqualified" type="xs:string" />
               <xs:attribute name="RemovePrefix"
                         form="unqualified" type="xs:boolean" />
               <xs:attribute name="LookupPrefix"
                         form="unqualified" type="xs:string" />
            </xs:complexType>
         </xs:element>

[[ Additional Section elements are snipped ]]

      </xs:choice>
   </xs:complexType>
</xs:element>
```

```
            </xs:sequence>
         </xs:complexType>
      </xs:element>
      <xs:complexType name="Standard"
                           kd:Desc="Attributes associated with all elements">
         <xs:attribute name="Name" form="unqualified" type="xs:string"
                     kd:Desc="The name of the processing step" />
         <xs:attribute name="Ordinal" form="unqualified" type="xs:int"
                     kd:Desc="Determines the order in which the process occurs" />
         <xs:attribute name="Checked" form="unqualified" type="xs:boolean"
                     kd:Desc="Whether the step is performed" />
      </xs:complexType>
      <xs:complexType name="OutputRules"
                     kd:Desc="Defines how comparison between and hash is done">
         <xs:attribute name="OutputFileType" form="unqualified"
                           type="OutputFileType" />
         <xs:attribute name="OutputGenType" form="unqualified"
                           type="OutputGenType" />
         <xs:attribute name="HashOutput" form="unqualified" type="xs:boolean" />
         <xs:attribute name="CheckOut" form="unqualified" type="xs:boolean" />
         <xs:attribute name="CheckOutProject" form="unqualified"
                           type="xs:string" />
      </xs:complexType>

   [[ Additional Complex Types are snipped ]]

      <xs:simpleType name="FileName">
         <xs:restriction base="xs:string" />
      </xs:simpleType>
      <xs:simpleType name="Directory">
         <xs:restriction base="xs:string" />
      </xs:simpleType>
      <xs:simpleType name="OutputFileType">
         <xs:restriction base="xs:string">
            <xs:enumeration value="XML" />
            <xs:enumeration value="VB" />
            <xs:enumeration value="CSharp" />
            <xs:enumeration value="StoredProc" />
         </xs:restriction>
      </xs:simpleType>

    [[ Additional Complex Types are snipped ]]

   </xs:schema>
```

NOTE You won't be able to display the XML created using this schema in the Visual Studio XML editor's Data tab. Instead you'll get a message "An error occurred while loading the schema with `targetNamespace` 'http://kadgen.com/KADGenDriving.xsd' referenced in this document. The same table (`OutputRules`) cannot be the child table in two nested relations." This is one of the scenarios where the XSD is legal, but the XML editor's Data view can't cope with it. The problem is that XML with the same element name under different parents can't be loaded into a DataSet, and a DataSet drives the DataGrid in the XML Data view.

Understanding Key XSD Nodes

The XSD describes the layout of your XML data using four basic constructs. Each of these constructs is an XML element in the XSD. It's a bit confusing to have an XML element describing an XML element, but that's exactly what happens. The four basic constructs are as follows:

xs:element: Defines the target element and associated information including the name and type.

xs:attribute: Defines attributes for an element along with associated information such as the name and type of the attribute. Attributes can only be contained in complex types.

xs:complexType: Defines a type that contains any combination of child elements and attributes. Complex types can appear either as a child of an `xs:element`, which is called an *unnamed type* or *anonymous type*, or as an independent entry under the root, in which case it's called a *named type*. You reference a named type with the `type` attribute.

xs:simpleType: Defines a type that contains no child elements or attributes. A simple type almost always occurs as an independent entry under the root and is referenced in the `type` attribute of other elements.

NOTE For clarity, I'll refer to elements within the XSD document as *nodes*.

Introducing Cardinality

The number of times an element can occur is its *cardinality*. This is the represented in the XSD by the `xs:minOccurs` and `xs:maxOccurs` attributes. These can appear on many different nodes including `xs:element`, `xs:choice`, `xs:sequence`, and so on. They can't appear on the root node `xs:schema` because exactly one root node is required in the target document. Both `xs:minOccurs` and `xs:maxOccurs` default to one, meaning that items described by nodes that don't specify these attributes must occur exactly once.

When `xs:minOccurs` is set to zero, the item doesn't have to appear in the target document. XML's concept of null is the item missing from the document. Thus, data items that are nullable appear in the XSD as `xs:minOccurs="0"`. The two attributes that control the number of times a specific element can occur in the XML document are as follows:

- **xs:minOccurs**: An attribute that indicates the minimum number of times the element can occur in the target node
- **xs:maxOccurs**: An attribute that indicates the maximum number of times the element can occur in the target node

Introducing Compositors

There are three logical ways you can combine child elements. These are called *compositors,* and valid values are `xs:sequence`, `xs:choice`, and `xs:all`. Table A-3 describes the three possible combinations and the XSD element that describes each one.

Table A-3. Possible Combinations of Child Elements

COMBINATION RULE	XSD COMPOSITOR
Require the elements to appear in the specified order.	`xs:sequence`
Allow any one, but only one, of the set of child elements to appear.	`xs:choice`
Allow elements to appear in any order, but each element appears zero or one times.	`xs:all`

Any combination of the `xs:sequence` and `xs:choice` compositors can be used to define a complex type. For example, you could have two `xs:choice` nodes that each forced only one of a set of possible child nodes and an `xs:sequence` that required there also be a specific sequence of child nodes. However, you can't

combine `xs:all` with other compositors. The `xs:all` directive has two extra requirements. The `xs:all` element itself can only have an `xs:minOccurs` of zero or one, and its `xs:maxOccurs` must be one. Also, its child elements can only have an `xs:minOccurs` of zero or one and must have an `xs:maxOccurs` of one.

NOTE Compositors are independent of cardinality. You can require elements appear in the specified order if they appear, but if `xs:minOccurs=0`, elements might be missing in the target document.

A single child node in the target document generally consists of three nodes in the XSD. This is the `xs:complexType`, one of these three compositors, and the `xs:element` node itself. A single element in the target document might be defined by only two nodes in the XSD if it contained only attributes and no child elements because it isn't appropriate to use the compositor unless you have child elements. To see an example, the FilePath `xs:element` node in the sample schema of Listing A-1 contains only attributes.

The resulting XSD may seem fairly verbose. The first series of three complex types, the first nine lines after the `xs:schema` node, defines three elements in the target document:

```
<kg:GenerationScript>
  <kg:FilePaths>
    <kg:FilePath />
```

However, this verbosity results in a clear and flexible schema definition.

xs:choice maxOccurs="unbounded"

Placing `maxOccurs="unbounded"` on the `xs:choice` element deserves special attention. This combination effectively says, "Each choice can include only one child element, but the parent complex type can contain as many of these choice elements as you'd like." In the following sample, each choice element can contain either a `Customers` or an `Orders` element (but not both), but you can have as many instances of the choice element as you'd like. The result of the following schema fragment allows any number of customers and orders to appear in any order:

```
<xs:element name="TempCustomersOrders" msdata:IsDataSet="true">
   <xs:complexType>
      <xs:choice maxOccurs="unbounded">
         <xs:element name="Customers"/>
```

```
            <xs:element name="Orders"/>
        </xs:choice>
    </xs:complexType>
</xs:element>
```

This style of element definition is especially important because XSDs created from an ADO.NET DataSet use this technique to define tables. The XML described by this sample contains a `TempCustomersOrders` DataSet with any combination of zero to many occurrences of `Customers` and `Orders` child elements in any order.

Validating XML in .NET

One of the purposes of an XSD is to validate whether an XML document matches a specified structure before you process it. If an XML document conforms to the rules of an XSD, you can say it's *validated* against that XSD or is *valid XML*. You can validate a single XML file against multiple schemas and of course use a single schema to validate multiple XML files.

Most of this book focuses on using the `XmlDocument` approach to XML processing (and XSLT, which is similar). This allows you to move around the document and retrieve particular nodes. However, there's another mechanism for accessing XML via streams that you'll need to know when validating XML. The `Xml.XmlValidatingReader` validates XML against a defined schema. `Xml.XmlValidatingReader` validates the XML as it reads it; it doesn't validate the whole file as soon as you attach it to the reader.

NOTE Just like ADO.NET offers the fully loaded disconnected model of the DataSet and the stream approach of the DataReader, XML processing in .NET offers the fully loaded `XmlDocument` class and several stream-based mechanisms for processing data.

You have a great deal of flexibility for how you manage streams during XML processing. Although you can work directly with the validating reader, you can also use it to load an `XmlDocument`. The next section explains how to do this.

Loading an XML Document via an XmlValidatingReader

The following example shows part of the code generation harness described in Chapter 3. After declaring variables, the method creates a new `XmlTextReader` that accesses the XML file and an `XmlValidatingReader` that reads from the `XmlTextReader` object:

```
Protected Overridable Sub LoadFile( _
                ByVal xmlFileName As String)
   Dim xmlDoc As New Xml.XmlDocument
   Dim textReader As Xml.XmlTextReader
   Dim validReader As Xml.XmlValidatingReader
   Try
      mXMLFileName = xmlFileName
      textReader = New Xml.XmlTextReader(xmlFileName)
      validReader = New Xml.XmlValidatingReader(textReader)
      validReader.ValidationType = Xml.ValidationType.Schema
```

Loading the `XmlDocument` from the validating reader causes it to be validated as it's loaded. The reader throws an exception on the line that calls the `xmlDoc.Load` method if the document fails to validate. Alternatively, you can use events to handle validation failures. In this case, if an exception occurs in this application during validation, it's simply thrown up to a higher method in the stack:

```
      xmlDoc.Load(validReader)
      LoadFile(xmlDoc, IO.Path.GetDirectoryName(xmlFileName))
   Catch ex As System.Exception
      Throw
   Finally
      textReader.Close()
   End Try
End Sub
```

Finding the Physical XSD File

If you're validating against XSDs, how do you determine the physical XSD files to use? There are two ways to do this in .NET. Often the .NET application itself knows the location of the validating schema. In this case, it can open the XSD, either from a file or from a cached schema, and load it into the `XmlSchemaCollection` held in the `Schemas` property of the `XmlValidatingReader`.

If the .NET application doesn't know the name of the physical file holding the XSD, the XML file may know. That's the case in the code generation harness. The `schemaLocation` attribute defines the file location, usually via a relative path. A different W3C namespace defines the `schemaLocation` attribute, so you have to add the new namespace, usually prefixed with `xsi`. The value of the `schemaLocation` attribute associates a namespace in the XML document with the location of the XSD that describes it. A space separates these two parts of the attribute string. For example:

```
<kg:GenerationScript
          xmlns:kg="http://kadgen.com/KADGenDriving.xsd"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation=
             "http://kadgen.com/KADGenDriving.xsd ..\..\XFiles\KADGenDriving.xsd"
          OutputDir=
            "D:\Documents and Settings\Kathleen\My Documents\Code Generation Book">
```

You can specify multiple namespace and physical XSD files.[6] To specify multiple physical files, space-delimit each entry. Entries at odd numbered positions (the first, third, fifth, and so on) are namespaces. Even numbered positions contain physical filenames.

Using the Target Namespace

There are two different contexts for namespaces in each XSD. Figure A-2 shows how this works. The `xmlns` attribute is a normal namespace for the XSD. Because the XSD is itself an XML file, the namespaces defined by the `xmlns` attributes resolve identifiers within the XSD and supports prefixes. These namespaces behave in the way discussed in the section "Namespaces" earlier in this appendix.

The `targetNamespace` has a different purpose. This tells what namespace the XSD defines. This is the namespace that the XSD will validate, and it corresponds to the `xmlns` attribute in the XML files the schema will validate. This namespace contains the complex types, elements, attributes, and so on that the XSD specifies. Each XSD file has only one `targetNamespace`.[7] The `targetNamespace` hooks up the namespace in the XML document you're validating with the XSD. Within Visual Studio, schemas are not hooked up via physical filename. Instead, the available schema with a `targetNamespace` matching the namespace of the XML node you're validating is used.

---

6. You can add the same physical files multiple times against different namespaces, but that requires using `xsl:import` or `xsl:include` within the XSD document.

7. When needed, `xs:imports` and `xs:include` can be used to include additional `targetNamespaces` in the XSD.

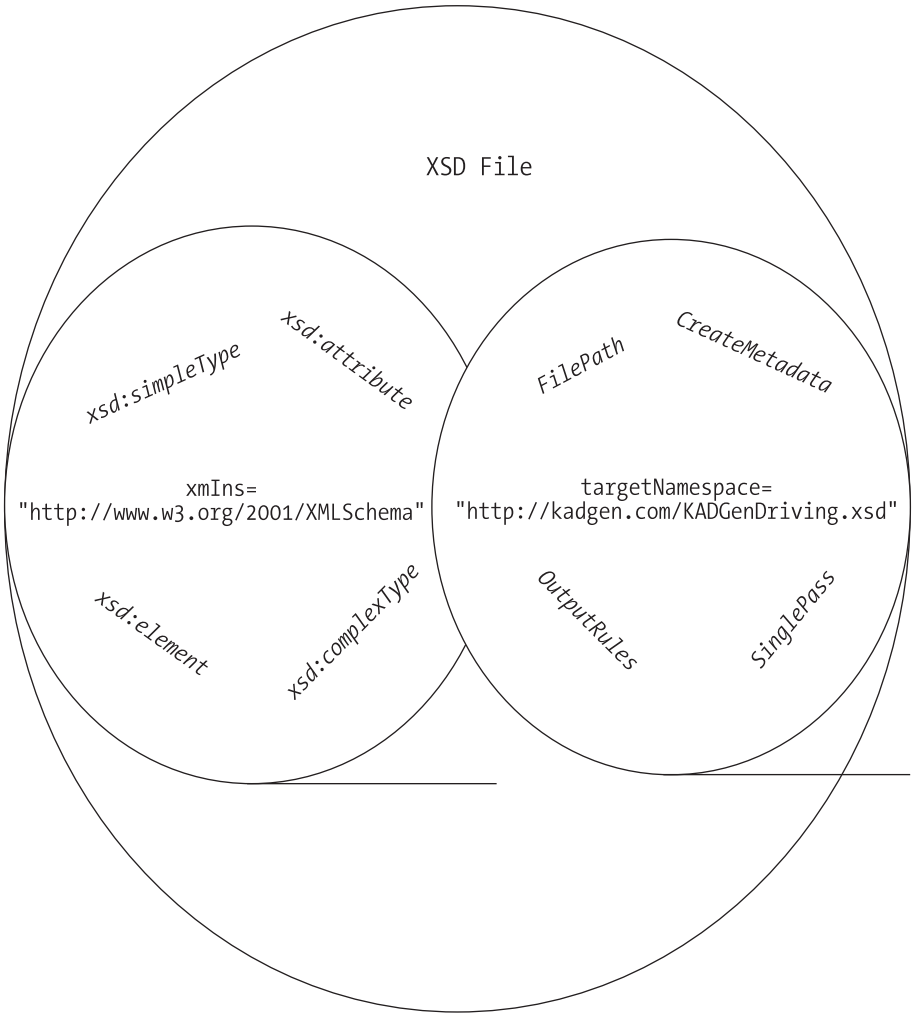


Figure A-2. The `xmlns` *and* `targetNamespace` *attributes define namespaces with different purposes.*

Editing X-Files in Visual Studio

Editing XML, XSLT, XSD, and so on in Visual Studio .NET isn't half bad. It's also not a lot more than half good. You may want to explore third-party tools because they offer a richer editing experience, but I do all of my X-File editing in Visual Studio. And believe me, I'm not one of those folks who can enter Hypertext Markup Language (HTML) or XML in Notepad. Visual Studio .NET has features that make editing these files much easier than a text editor. Visual Studio behavior differs depending on whether an XSD is available for the XML being edited.

Visual Studio Editing Without an XSD

Even without an XSD, Visual Studio offers several features that make it easier for you to edit XML files as follows:

- Automatic insertion of close directives
- Automatic indenting
- Tests for the XML being well-formed
- Data view of XML

It's pretty easy to experiment and see how Visual Studio .NET closes elements. Type the close bracket of an opening element, and .NET will slap in the corresponding close element and leave you positioned in the middle to add text or child elements. Flipping your view to the Data view of the XML and back updates both automatic indenting and the test for being well-formed.

The Data view may not be useful to you in itself, but these well-formed tests will certainly come in handy. In some cases, the Data view can be a helpful arrangement of your data elements. In other cases, you can't display legal XML within the Data view. One way this can happen is using the same element as a child of different parent elements. When it works with your schema, the Data view also offers a handy way to edit your XML data including adding and deleting nodes.

XSDs and Visual Studio X-File Editing

If you have an XSD available for your XML, Visual Studio .NET includes additional features. It offers XML validation checks using the XML menu's Validate XML option and many features related to IntelliSense.

All of the angle bracket editors in Visual Studio use XSDs to define IntelliSense, autocompletion, and so on. This allows you to select items from a list for both elements and attributes and minimizes the typing you have to do. The IntelliSense support is context sensitive, so you only see the elements valid as children of your current parent element and only attributes valid for the current element. The XSD is the source of IntelliSense information both when you use schemas provided by .NET and when you're using custom schemas for your own files. See the sidebar "How Visual Studio Finds Schemas" for an explanation of how .NET finds the XSD based on namespaces in the XML file.

How Visual Studio Finds Schemas

So how does Visual Studio come up with the correct XSD to support these cool features? It'll look for an XSD matching one of your document's namespaces in the current project directory. If there are multiple matches within your current project, which one it grabs appears to be indeterminate. If a matching schema isn't found in the project directory, Visual Studio looks in a "magic" directory. In a standard install, the magic directory is this: [Program Files]\Microsoft Visual Studio .NET 2003\Common7\Packages\schemas\xml.

This directory contains the XSDs for anticipated files, such as `xsdschema.xsd` and `asp.xsd`. If you add your own XSDs to this directory, they'll be used as well. I'm not sure whether it officially ranks as a bug, but it's certainly unexpected that Visual Studio doesn't match the `targetNamespace` but rather the default namespace.

The validation performed by the Validate XML option of the XML menu follows the same rules as IntelliSense. When you validate using `XMLValidatingReader`, the `xsi:schemaLocation` is used. Be careful with this because you could validate against different XSD's.

Considering the Issues

There are a few glitches and imperfections in the way this works. None of these are too bad, unless you're unsuspecting and they throw you way off track.

You might expect that Visual Studio .NET would use `schemaLocation` in determining what physical XSD file to use for a particular namespace. No, and that's unfortunate because both of the approaches it uses have problems. Placing your schemas in the magic directory is bad because commingling your files with vendor application files is almost always a bad idea. At the least, you'll have to remember to copy them into a new application install. Placing these schemas in the current project file is even worse because you'll often use the same schema across multiple projects, and you wind up with multiple copies of the XSD in separate directories because Visual Studio insists that everything in a project reside in the same directory. Pick your poison. I put them in the magic directory if used in multiple projects.

CAUTION Visual Studio doesn't use the `schemaLocation` attribute. Instead it looks in the local directory and in the magic directory [Program Files]\Microsoft Visual Studio .NET 2003\Common7\Packages\ schemas\xml. The .NET Framework itself supports the `schemaLocation` attribute.

You also might expect that XSLT would be one of the basic schemas Microsoft supplied as part of the Visual Studio installation. It isn't clear whether it's an oversight, but neither Visual Studio 2002 nor Visual Studio 2003 include a schema for XSLT. Happily, Joe Feser at Fesersoft worked this out and posted it on GotDotNet.com. There are directions with the download, but you basically just need to put the xslt.xsd file into the magic directory: [Program Files]\Microsoft Visual Studio .NET 2003\Common7\Packages\schemas\xml.

Visual Studio 2003 has a glitch in IntelliSense. It uses the default namespace rather than the target namespace. If you prefix the namespace in the current document and have a `targetNamespace` defined in the XSD (and you can't validate against the schema without the `targetNamespace`), Visual Studio adds extra non-prefixed items to the IntelliSense list. As a result, you'll have to pay attention to selecting only the prefixed versions.

Perhaps the most misleading glitch is the behavior of IntelliSense with `xs:sequence` and `xs:choice`. As you saw earlier in the "Introducing Compositors" section, the entries that are valid in each of these cases is restricted. For example, unless it has an `xs:minOccurs="0"` attribute, only the first child element of an `xs:sequence` is valid as the first child of the XML element. Similarly, once you select an item from an `xs:choice`, the other entries aren't valid. IntelliSense and the background compiler ignore these nuances, which can lead you to believe an invalid document is valid. You've got to select the Validate XML menu item to validate the document.

Finally, Visual Studio caches schemas. Although that may seem like a good thing while you're working with a stable schema, it can be frustrating if you're editing the XSD and IntelliSense doesn't reflect your changes. You'll have to close and reopen Visual Studio .NET for the new XSD schema to be used. The same patterns for XSD use applies to validation via the Validate XML menu item.

The Commando's Guide to XSLT

It'd be misleading to say that XSLT is easy. It isn't. There are some subtle nuances in XSLT and lots of reasons for the huge books on the subject. But the fundamentals—particularly what you need to get the job done for code generation—are limited and not individually difficult. The most difficult thing for you to understand may be how template processing differs from the procedural code you're familiar with. Although this aspect may seem odd, remember that set-based processing in SQL seems a bit odd at first, too.

TIP I understand from LISP programmers that the rule-based LISP mindset is similar to the XSLT mindset. If you're a LISP programmer, that analogy might help you get started with XSLT.

Processing XSLT

You run an application. You process an XSLT template. What's the difference?

When you get stuck on this, I'd like you to use SQL to shake your brain out. You know when you're working out (we all go to the gym six times a week, right?) or you've been at the keyboard too long, you sort of stretch your muscles to refocus. SQL will help you stretch your brain muscles to prepare for XSLT, but don't get stuck there because there are many differences between SQL and XSLT. A SQL statement such as the following is processed per data item, not sequentially:

```
UPDATE Customers SET CompanyName= "Something" WHERE CompanyID=12
```

You don't loop through the data setting the value for each row. Instead, you tell SQL what to do, what to do it with, and get out of its way. XSLT takes this idea even one step further because the templates are conditional. You aren't even in the template if it doesn't match the conditions. Also, XSLT runs templates against all sets of matching data. You can think of this as the processor picking up pieces of data (nodes in your XML document) and asking, "Is there anything I can do with that?" or "Do any available templates apply to this piece of data?" This approach is extremely effective in text processing. It also makes things a little different as follows:

- Templates may run that you didn't anticipate.
- Processing is rule based, and the order of rules generally doesn't matter.
- XSLT trusts you.
- You have no access to your output during processing.
- No side effects are allowed.

Okay, the first three are scary, you understand the next one, and what the heck is a side effect? Starting with the first item, template processing is completely predictable, and I'll show you how it works. But a common frustration starting with XSLT is figuring out why templates are running on elements you didn't want them to run on. Unfortunately, a lot of examples cause habits that result in this happening more often than needed. The approaches shown in Chapter 3 use a single entry-level template that processes the root and will largely avoid this problem, but this may look different from examples you see in other places.

XSLT trusts you. .NET and most other programming languages don't trust you, but XSLT does. I don't trust you, and I understand why .NET doesn't trust you. You're human, and you make mistakes. XSLT, in the W3C standard, ignores

this fact and assumes that whatever you ask for is what you want. This isn't arbitrary and is actually important in text processing. This means that if you mean `tableName` but type `TableName`, you won't get any warning or error. You'll simply not get the output you wanted—generally something missing from your output.

A side effect is anything that can be changed in the current scope and accessed outside the scope. You'll see the commitment to no side effects when you work with XSLT variables.

So, how does all this affect working with XSLT? Looking at template processing in the next section will show a new way of thinking. Let yourself think differently. Nothing in XSLT is arbitrary. It all works together in an effective way, just one that may feel strange at first. Just like SQL may have felt awkward when you started but natural now, XSLT may feel awkward in ways that you'll soon put behind you.

Introducing the Stylesheet

Each XSLT file is a stylesheet. Each stylesheet contains templates that do the actual work. The `xsl:stylesheet` is the root element, which means there can be only one `xsl:stylesheet` within the physical file (a single root element is one of the rules for valid XML). The `xsl:stylesheet` element is synonymous with the `xsl:transform` element.

Each stylesheet can contain other referenced XSLT files. During processing, these included files become part of the `xsl:stylesheet` root of the main XSLT file. Include files allow reuse and organization within and between complex stylesheets. Two directives cause files to be included: `xsl:import` and `xsl:include`. These directives provide different results when there are conflicts between templates, such as two templates with the same name. In general, you want to avoid these conflicts, and overall I think `xsl:import` has simpler behavior (although it has the additional requirement that it must appear first in the `xsl:stylesheet`).

Each XSLT transform outputs one stream that's often displayed or stored as a file. If nothing is output, an empty stream exists, and depending on the code that outputs the stream, a blank file may be output. You can't even output separate information to a log file. So all of the test stuff you use for debugging must appear in your output. I hope we'll see some logging features added in a future version of XSLT. In the meantime, you can get some logging behavior via extension objects, which are discussed in the section "Using Extension Objects."

For our purposes, each XSLT transform will access exactly one XML input file. The XPath `document` function allows additional XML files to be open if you find a need for that, but it significantly complicates the stylesheet.

An XSLT processor reads the XSLT file and creates the output. There are a few XSLT processors available to you. You can access an XSLT processor through

Internet Explorer by opening an XML file that has an XSLT stylesheet defined. You do this with an XML processing instruction:

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/xsl" href="BritishRoyalFamilyTests.xslt" ?>
```

In this case, Internet Explorer displays the output of the transform. It's a simple way to test XSLT, but the output is transformed to HTML, so its displayed whitespace is quite different from what's output to a text file. Also, the full XML file is loaded into the browser and is available using View Source in the context menu. You can also access an XSLT processor through .NET code using the `Xml.Xsl.XslTransform` object's `Transform` method. Chapter 1 contains a simple XSLT process example, Chapter 3 shows you how to process XSLT using the code generation harness, and Chapters 6–11 contain numerous sample templates.

Looking at the XLST File Header

The header is everything that appears before the first template. This is generally something such as this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!– Summary:  Creates friendly XML output for an XSD with a single DataSet !–>
<!– Created: Kathleen Dollard June 26, 2003 !–>
<!– TODO: !–>
<xsl:stylesheet version="1.0"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
          xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
   <xsl:import href="Support.xslt" />
   <xsl:output method="xml" encoding="UTF-8" indent="yes" />
   <xsl:preserve-space elements="*" />
```

The first line is a standard XML declaration. The declaration is optional but is sometimes necessary to include its attributes, such as an encoding other than the default. Here I explicitly include the default Unicode encoding scheme, UTF-8. This is followed by a few lines of comments. You indicate comments in XSLT with `<!– –>`, just like any other XML document.

TIP The `xsl:comment` directive creates a comment in XML output. Create comments within XSLT using standard XML commenting.

To meet the requirements for legal XML, there's a single root element, which for an XSLT stylesheet must be `xsl:stylesheet` or `xsl:transform` (synonymous) and must specify the version of XSLT. Today, only XLST 1.0 exists. This stylesheet element contains three namespace attributes. The earlier section "Surviving Namespaces" discusses namespaces. You'll need to include `xmlns` attributes for all namespaces used in your source document. In this case, I expect the source document to be an XSD file created with a Microsoft tool, so I included the `XMLSchema` and `xml-msdata` namespaces. You'll also need to include the `XSL/Transform` namespace.

TIP The default XSLT template provided with Visual Studio includes the `XSL/Transform` namespace as a default namespace. It's easier to read templates if you prefix the XSLT namespace. The common prefix for `XSL/Transform` is `xsl`. See the sidebar "Changing Visual Studio Templates."

The `xsl:imports` incorporates a supporting XSLT file containing utility templates. The output directive can specify `html`, `text`, or `xml`, which affects details of the output. Just use the right one and note that when called from .NET 1.0, the output stream type (whether it was an XML stream) takes precedence over this directive in determining processing. .NET 1.1 respects the output tag in the XSLT, and I recommend avoiding the `XMLTextWriter` for output because of problems when it's used for XSLT output of XML. The `xsl:preserve-space` element helps manage the output whitespace. You'll learn more about whitespace later in this appendix.

Looking at XSLT Templates

Templates perform the real work of XSLT. The processor looks at an element in the XML document, looks at the stylesheet, and runs all of the matching templates. A common beginner's problem is extra unwanted stuff appearing in the output because extra templates were run. Limiting the templates that are run is pretty easy to do. Just use an entry template that processes only the root. There's only one root, so this template will only be run once directly by the processor. If the template doesn't handle the root element, the child nodes of the root are processed. If the root is handled, other nodes won't be processed on entry, and you can clarify your intent by applying mode or name attributes to all other templates.

There are two types of templates—named templates and match templates. Other than the entry template, all templates can have an identifier. In the case of named templates, the name is the identifier. The `mode` attribute is the identifier for match templates. Let's sort this all out.

Using Named Templates

Named templates are very much like subroutines in procedural code. They're called using the `xsl:call-template` directive. When the processor encounters an `xsl:call-template` directive, it processes the named template once and puts all the output in the current location of the calling template. This generally means the output of the named template is immediately placed in the output file. But, it might be placed in the contents of a variable or a few other tasks in the calling template. For example, the following template calls a named template named `MakeUpperCase` and passes a parameter named `String`:

```
<xsl:template match="/">
   <xsl:call-template name="MakeUpperCase">
      <xsl:with-param name="String" select="'fred'"/>
   </xsl:call-template>
</xsl:template>
```

NOTE To pass literal parameters or assign a literal to a variable, include the value in single quotes within the double quotes.

The `String` parameter has the value `'fred'`. Because the `xsl:call-template` doesn't appear within a special directive such as a variable, the output of the named template is placed in the output file. The named template `MakeUpperCase` is as follows:

```
<xsl:template name="MakeUpperCase">
   <xsl:param name="String"/>
   <xsl:value-of select="translate($String,'abcdefghijklmnopqrstuvwxyz',
                 'ABCDEFGHIJKLMNOPQRSTUVWXYZ')"/>
</xsl:template>
```

This template accepts the parameter named `String` and outputs the value of the XPath `translate` function. The `translate` function looks at each character in the input string and looks for that character in the first string. If it isn't found, the original character is output. If the character is found, the character at the corresponding position in the second string is output. Thus, the output of this template is `FRED` regardless of the case of the input XML. This is a contrived example because the data is hard-coded into the named template. More often, parameter values are variables or are from the XML file.

Using Match Templates

Match templates differ significantly from named templates. Each match template specifies a match expression. A match template is only exercised when a matching node is found in the XML. If there are many matching elements, the template will be processed many times. These templates are always processed in the context of the matching node as defined in an XPath expression. Match templates are initiated in two ways—automatically as the stylesheet is processed and when the processor encounters `xsl:apply-templates` directives.

These are actually somewhat different operations. When the processor is doing the initial processing, it has a set of default *built-in templates* in its back pocket. It checks the stylesheet for a template that matches the root node, if it doesn't find it…well, you don't care because for code generation you're going to provide a root node, right?

NOTE In case you're curious, I'll explain. If the processor doesn't find a match for the root node, it goes down each branch of the hierarchy, processing the first template match it finds and then leaving that branch. It also outputs text nodes it finds at the tips of branches with no matching templates. Although I'm sure this is helpful in many contexts, it isn't helpful when processing metadata to generate code. Even if you understand and can predict the results, the next programmer may not, so stick to the simpler approach of providing a template that matches the root node. Be aware too that if you get unexpected results from XSLT processing, there may be some default processing going on that you're not considering.

Determining the Node-Set to Process

When an `xsl:apply-templates` directive is processed, it collects a set of matching nodes—called a *node-set*—and searches the stylesheet (including any included or imported files) for matching templates. The processor finds matching templates based on the template's `match` and `mode` attributes.

The `match` attribute for templates and the `select` attribute of the `xsl:apply-templates` element are XPath expressions. The "XPath Marks the Spot" section discusses XPath and includes a reference. The significant thing at this point is that the `select` attribute of the `xsl:apply-templates` directive results in a node-set being processed and the appropriate match template is called once for each node in the node-set. This node set can contain zero or more nodes. Thus, a match

template might not be processed, or it may be processed many times with different nodes, depending on the XML input document.

If there's no `mode` attribute on the `xsl:apply-templates` directive, the processor only tests templates without a `mode` attribute; otherwise, it only tests templates with a matching `mode` attribute. The processor then finds all templates that match the `select` expression. The match may occur several different ways, and there's no requirement of an exact match with the `select` clause. For example, all nodes match the * attribute, which can be used in a `match` clause to ensure a particular template is run for every processed node. A template with `match="*"` will be processed whether the `xsl:apply-templates select` clause is `"*"`, `"dbs:table"`, or any other XPath statement that returns nodes. Similar, you can also use * in a `select` clause to request that any template matching any node be processed.

NOTE It's the node, not the original `select` attribute, that's used to determine the match. If the mode attribute is specified in the `xsl:apply-templates` directive, the template will only be processed when it has the same mode and matches the node.

Write some experimental XSLT and practice predicting which templates will be processed. For example, in the following set of templates, the first template is processed for the root node. It has no mode, so it's available to the initial processing and is the entry-level template:

```
<xsl:template match="/">
   <xsl:apply-templates select="xs:schema" mode="DataSet" />
</xsl:template>

<xsl:template match="xs:schema" mode="DataSet">
   <xsl:element name="DataSet">
      <xsl:attribute name="Name">
         <xsl:value-of select="@name" />
      </xsl:attribute>
   </xsl:element>
</xsl:template>
```

The second template has a mode attribute so can't be directly accessed by the processor at entry. It can only be processed via an `xsl:apply-templates` directive of another template. In this case, the entry-level template calls it. The at (@) sign indicates the value of an attribute; in this case, the `name` attribute, should be output.

NOTE The only type of match you can guarantee is the match against a single root element because XML guarantees the presence of a single root node. This is why the root entry point is logical; you know it'll be called exactly once.

Sorting with xsl:sort

You can specify the order in which nodes are processed by match templates if you specify one or more `xsl:sort` child elements. For example:

```
<xsl:apply-templates
         select="xs:element/xs:complexType/xs:choice/xs:element"
         mode="Table" >
   <xsl:sort select="name" case-order="upper-first" />
</xsl:apply-templates>
```

TIP If you have the XSLT schema installed as described in the sidebar "How Visual Studio Finds Schemas," you'll get a list of available attribute values, making it much easier to write templates without excess time in the Help files.

Passing Parameters

You can pass parameters to both named and match templates. An example of a template declaration that accepts a parameter is as follows:

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

Within a template, arguments are accessed by prefixing them with a dollar sign ($):

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

This is the same syntax used to access variables, so variables and arguments are syntactically equivalent within the template. I've clipped the rest of the template because it isn't relevant.

An example of an XSLT directive that calls a parameterized template is as follows:

```
<xsl:call-template name="GetPKeyName">
   <xsl:with-param name="TableName" select="$tablename" />
</xsl:call-template>
```

CAUTION XSLT parameters are another scenario where XML trusts you. If you pass a parameter with the name misspelled or with incorrect casing, you won't get an error. The processor will assume that you aren't passing the parameter and simply use the default for the parameter in the template or an empty string if no default is supplied.

Understanding Flow Constructs

As much as possible, you want to think of XSLT as a rule-based processor and avoid thinking in procedural terms. There are scenarios where this will come up short. The most common is branching constructs. There are two branching constructs in XSLT. These are `xsl:if` and `xsl:choose`. The first is a simple conditional. There isn't even an `else` clause for the `xsl:if`. If you need to choose between two or more options, use `xsl:choose`. The `xsl:choose` directive is very much like a `Select Case` in Visual Basic or a `switch` in C# or C++ (but with no fall-through).

Executing Conditionally with xsl:if

The single branch construct `xsl:if` lets you selectively perform some processing. For example, the following template fragment outputs the programmer-friendly `AllowNulls` attribute only if the XML node contains the `minOccurs='0'` attribute:

```
<xsl:if test="@minOccurs='0'">
   <xsl:attribute name="AllowNulls">true</xsl:attribute>
</xsl:if>
```

That's pretty simple. The `xsl:if` directive has a `test` attribute that's interpreted as a Boolean value. The rules for converting to a null are the same as the `bool` XPath function. If `test` returns a node-set, it's considered `true` if any nodes exist. If the `minOccurs` attribute doesn't exist or isn't zero, the test fails, and the `AllowNulls` attribute isn't output.

CAUTION Rules for converting values to Booleans may result in some surprising results. A string is `true` if the length is greater than zero. Therefore, the string `"false"` is `true`. Try to avoid implicit Boolean conversions in XSLT.

Branching with xsl:choose

The multibranch construct `xsl:choose` lets you selectively perform any one of several different processing possibilities. You'll have to use `xsl:choose` even for a simple if/else/end logic. The following template fragment outputs `Int32` if the `Type` variable contains the string `'int'` and otherwise outputs the initial element's type. The `xsl:choose` directive must include at least an `xsl:when` directive and can optionally include an `xsl:otherwise` clause to pick up unhandled cases:

```
<xsl:choose>
   <xsl:when test="$Type='int'">Int32</xsl:when>
   <xsl:otherwise><xsl:value-of select="$Type"/></xsl:otherwise>
</xsl:choose>
```

NOTE The `xsl:choose` directive must contain at least one `when` directive.

Looping with xsl:for-each

You can loop in XSLT using `xsl:for-each`. In most cases where `xsl:for-each` will do the job, `xsl:apply-templates` will also do the job and vice versa, but there are both similarities and differences. The most obvious differences are that `xsl:for-each` is in line and templates called with `xsl:apply-templates` can be reused. In both cases, the looping code may not run or may run multiple times.

A more subtle difference is that `xsl:apply-templates` may run multiple different templates—either multiple templates for a single node or different templates for different nodes. Without explicit conditionals, the `xsl:for-each` directive always runs the same block of code. A discussed in the "`xsl:apply-templates` or `xsl:for-each`" sidebar, the decision between `xsl:for-each` and `xsl:apply-templates` has significant impact on your template organization.

xsl:apply-templates or xsl:for-each

There's a purity issue surrounding the use of `xsl:apply-templates` that has at least some merit. Experienced XSLT programmers are likely to tell you that you should avoid using `xsl:for-each` and instead use `xsl:apply-templates` for non-trivial XSLT. Heck, I'll tell you that. The `xsl:apply-templates` directive is a great way to organize your templates and will make you look like an XSLT jock. Here's the deal: XSLT is relatively hard to organize in a readable format. You've really only got a couple of good tools for organization: `xsl:call-templates` and `xsl:apply-templates`. Anytime you can use the `xsl:for-each` mechanism, you could also use `xsl:apply-templates` and modularize your templates in a more readable fashion. You can apply a `mode` attribute to the `xsl:apply-templates` directive to identify it and explain exactly what you're doing. This clarifies your code and avoids deeply nested `xsl:for-each` directives that are difficult to read. The exception to this is short fragments that would make a trivial template. If I can't name the template, I wonder if it's too small, particularly if it's just a few lines, and I don't reuse it.

On the other hand, you're in this to write maintainable templates. The `xslt:for-each` directive is very effective, and if you work on a team of procedural programmers, the easiest middle ground may be some use of `xsl:apply-templates` to avoid deep nesting but widespread use of `xsl:for-each` for shorter loops. If at all possible, come to a team consensus on how your templates should look, but focus on what your team finds easy to maintain, not an external vision of what the code *should* look like.

In the following example, I chose an `xsl:for-each` directive because this stylesheet has a series of potentially confusing steps to retrieve portions of an XSD. The nested `xsl:for-each` seemed the best way to maintain clarity. It also let me demonstrate that when appropriate it's entirely legal to nest `xsl:for-each` loops:[8]

```
<xsl:template name="GetPrimaryKey">
   <xsl:param name="pKeyName" />
   <xsl:for-each select="//xs:unique[@name=$pKeyName]">
      <xsl:element name="PrimaryKey">
         <xsl:for-each select="xs:field">
            <xsl:element name="Field">
               <xsl:attribute name="Name">
```

8. The element and attribute directives output XML elements and attributes. This is a portion of the `XSDToFriendlyXML.xslt`.

```
                <xsl:value-of select="substring-after(@xpath,'mstns:')" />
            </xsl:attribute>
        </xsl:element>
    </xsl:for-each>
  </xsl:element>
</xsl:for-each>
</xsl:template>
```

You can apply an `xsl:sort` directive to the `xsl:for-each` directive in the same way you add them to `xsl:apply-templates`.

Understanding Context

I'll come back to context again in relation to XPath. But while you're thinking about XSLT, it's important to note what context is and how XSLT works with context. Context is your current location in the document. Imagine that you have a printout of your XML and stick your finger someplace on the printout. That position is your context.

NOTE You may think of context as a kind of cursor. Be careful with that analogy because the context can move anywhere in the XML document, including the enclosing structural nodes, so in some ways differs from a database cursor.

The node being processed is the context within a match template. For example, the context within this template is a `dbs:Table` node:

```
<xsl:template match="dbs:Table" mode="BuildClasses">
```

Context changes as you process match templates. This occurs whether you're processing an entry template or processing a template via an `xsl:apply-templates` directive. Context also changes during processing of an `xsl:for-each` directive. In this example, the context when the `SingularName` attribute is output is the context of the calling template. When the `dbs:Name` and `dbs:NETType` variables are output within the `xsl:for-each`, the context is each `dbs:TableColumns` child element. If the context of the calling template doesn't contain any child elements of this type, the directives within the `xsl:for-each` are not processed:

```
<xsl:template name="ClassLevelDeclarations">
#Region "Class Level Declarations"
   Protected mCollection As <xsl:value-of select='@SingularName'/>Collection
```

```
   Private Shared mNextPrimaryKey As Int32 = -1
   <xsl:for-each select="dbs:TableColumns/*">
      Private m<xsl:value-of select="@Name" />
            <xsl:text/>As <xsl:value-of select="@NETType" />
   </xsl:for-each>
#End Region
</xsl:template>
```

Outputting Information

Values are output by using the `xsl:value-of` directive and by including literals in the template.

What does it mean that values are output? It's probably simplest to think of this as shoving the output values into whatever is currently going on. In normal circumstances, this means outputting the result to the file or stream created by the XSL transform. However, you can also put the output into a variable as it's being declared or pass it as a parameter to another template.

Outputting Literals

Any literals, other than whitespace, that appear in your XSLT stylesheet will appear in your output. This is convenient when you're doing code generation. You can output much of the code as literals, and these literals help you mentally synchronize your output code with your template and search within the template. For example, the following template outputs a complete region without requiring any additional XSLT directives:

```
<xsl:template name="Declarations">
#Region "Declarations"
   Private mDSet As Utility.BaseDSet
#End Region
</xsl:template>
```

Outputting Tokens with xsl:value-of

You output the value of a variable or an item in the XML input file using the `xsl:value-of` construct. You'll often use this technique for token replacement. The `select` attribute is an XPath expression. This XPath expression can output literals, variables, attributes, and element text.

You access literals by surrounding them with single quotes:

```
<xsl:value-of select="'Foo'"/>
```

NOTE This example is the same as including the literal `Foo` in the template.

You access variables and arguments via their name prefixed with a dollar sign:

```
<xsl:value-of select="$myVariableName" />
```

Attributes are prefixed with an @:

```
<xsl:value-of select="@Name" />
```

If no attribute is specified, `xsl:value-of` returns the text of the first matching child element. To output the text of the first child element named `Address`, you'd use this:

```
<xsl:value-of select="Address" />
```

The XPath expression used in the `select` clauses can be quite complex. You'll see more details on XPath expressions that retrieve information from your XML input file in the XPath section of this appendix.

Working with Variables

Variables in XSLT are a bit weird. You can assign a value to them when you create them, but you can't change this value once it's set. Variables work this way to remain consistent with the no side-effect rule of XML. They really act more like constants, but unlike .NET constants, they can come in and out of scope, and each time you create them, you can assign a different value. XSLT variables generally contain strings or node-sets.

There are two possible scopes for XSLT variables. Global variables appear at the top of the stylesheet outside any templates. They're initialized in the context of the root element of the input XML document when the stylesheet is called. Global variables are visible in all templates and any templates incorporated using `xsl:include` or `xsl:import`. Like other variables, you can't reassign their values.

The second scope is local scope. Variables with local scope are visible to the current block and any nested blocks within the same template. Local variables

aren't visible in child templates called within the block (using `xsl:apply-templates` or `xsl:call-template`).

The following example shows how to declare and assign a variable with local scope. The `xsl:variable` directive creates the variable and assigns it the current value each time the loop is processed:

```
<xsl:template match="/">
   <xsl:for-each select="root/*">
      <xsl:variable name="pos" select="position()"/>
      <xsl:value-of select="$pos"/>.
   </xsl:for-each>
</xsl:template>
```

NOTE This example is contrived, and it'd be just as easy to output the result of the `position()` function directly.

If the input XML has three child elements of a root named `root`, this template outputs the following:

```
1.
2.
3.
```

Sometimes the value of a variable depends on some specific condition. Rather than using a `select` attribute, you can supply any valid XSLT within the element itself. In this example, a second variable is added and its value assigned based on the current position. This also outputs a `<br/>` for display in Internet Explorer for testing:

```
<xsl:template match="/">
   <xsl:for-each select="root/*">
      <xsl:variable name="pos" select="position()"/>
      <xsl:variable name="var1">
         <xsl:choose>
            <xsl:when test="$pos='2'"> Is two </xsl:when>
            <xsl:otherwise> Is NOT two </xsl:otherwise>
         </xsl:choose>
      </xsl:variable>
      <xsl:value-of select="$pos"/> -
      <xsl:value-of select="$var1"/>&lt;br/>
   </xsl:for-each>
</xsl:template>
```

The output from this template is as follows:

```
1 - Is NOT two <br/>
2 - Is two <br/>
3 - Is NOT two <br/>
```

Of course, the `<br/>` tags aren't actually visible in Internet Explorer, but this is what you'd see in Notepad.

Sometimes it's convenient to assign the output of named templates (or less often, a match template) to a variable. You can do this by simply calling the template within the `xsl:variable` directive. Here the value of the variable `var1` is set to the output created within the `xsl:call-template`:

```
<xsl:template match="/">
   <xsl:for-each select="root/*">
      <xsl:variable name="pos" select="position()"/>
      <xsl:variable name="var1">
         <xsl:call-template name="test"/>
      </xsl:variable>
      <xsl:if test="$var1='Fred'">
         <xsl:value-of select="$pos"/>
      </xsl:if>
   </xsl:for-each>
</xsl:template>

<xsl:template name="test">
   <xsl:if test="position()=3">Fred</xsl:if>
</xsl:template>
```

This entry-level template just outputs "3." Get to know XSLT variables. Although they're quite different from variables you're used to, once you understand what they can and can't do, you're likely to find scenarios where they're quite useful.

CAUTION The result of the template is assigned to the variable exactly. If the text *Fred* was on a separate line, the variable would have also contained whitespace, and thus the match would've failed. This is a common error you'll want to watch for.

Using Extension Objects

Extension objects provide a way for the XSLT template to call back into .NET to run methods and access properties. The XSLT specification provides a mechanism for processors to implement additional features, and extension objects are .NET's way of extending XSLT. It's amazingly how easy they are to use. There are just three steps to using extension objects: You need to determine the .NET class you'll call, prepare the template, and pass a reference to the object to the transformation.

The XSLT template needs a namespace to identify the extension objects. I like using an `ext` or `net` prefix to clarify what's going on within the template. To call the methods of the extension object, you just prefix them with this namespace prefix:

```
<xsl:stylesheet version="1.0"
          xmlns:ext="http://www.kadgen/ExtensionObject"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform" >
<snipped>
       <xsl:value-of select="ext:FixName(@name)"/>
```

The `xsl:value-of` directive outputs the result of the `FixName` method of a .NET object. Wow, that was easy!

The final step is fixing .NET to specify a reference to the object. If the methods you want to access are either instance or static methods in the `Utility.Tools` class, you can specify the extension objects as part of the argument list in your .NET code as this:

```
Dim objTool As Object = New Utility.Tools
Dim args As New Xml.Xsl.XsltArgumentList
args.AddExtensionObject("http://www.kadgen/ExtensionObject", objTool)
```

This associates the extension object with a namespace. This namespace much match the namespace in the XSLT document. You can use multiple extension objects in a single XSLT stylesheet. You'll need multiple namespaces, each associated with a different prefix and .NET object.

CAUTION XSLT callbacks allow you to access private methods of the class.

If you use any custom extension to XSLT, you're tying yourself to processing your stylesheets with that tool. In the case of extension objects in .NET, you're tying yourself both to processing with .NET and to having that specific object and method available. That's fine with the generation harness and in many other scenarios, but think about how broadly you'd want to use your stylesheets before employing custom extensions.

CAUTION Don't call methods that use `ParamArray` (`params` in C#) on extension objects. It might work, but the .NET Help says it'll provide unpredictable results.

Outputting Whitespace

Whitespace is the biggest pain in XSLT processing. You're likely to pour more time into adjusting whitespace than you'd like. The rule is simple: Whitespace that stands alone between elements is insignificant whitespace and will not be output by default. All other whitespace is significant and will be output. You read about this earlier in this chapter in the "Introducing Whitespace" section. To experiment with whitespace management, output to text, rather than using Internet Explorer, because HTML further ignores insignificant whitespace in the output. The following are some examples:

```
<xsl:template match="/">
   <xsl:value-of select="'First Line'"/>
   <xsl:value-of select="' - more on the first line'"/> -
   <xsl:value-of select="'Second Line'" />
</xsl:template>
```

The result of the first two `xsl:value-of` directives appear on the same line with the line break between them ignored. That's because the whitespace lies between two elements and therefore is insignificant whitespace. The dash after the second `xsl:value-of` statement makes the line break following it significant whitespace because it's adjacent to text and not between two elements. This whitespace is output. The output of this template is this:

```
First Line - more on the first line -
Second Line
```

Sometimes you need to force a line break between two adjacent elements. You can do this by escaping the carriage return/linefeed characters. The output from the next example is the same as the previous output, but now that I've moved the dash into the attribute value, there's no literal at the end of the second `xsl:value-of` directive, so the escaped carriage-return/line-feed is necessary to place the output of the final `xsl:value-of` on its own line:

```
<xsl:template match="/">
   <xsl:value-of select="'First Line'"/>
   <xsl:value-of select="' - more on the first line -'"/>
   <xsl:text>&#x0d;&#x0a;</xsl:text>
   <xsl:value-of select="'Second Line'" />
</xsl:template>
```

Sometimes you'd like to have a line break in your XSLT file in a location where you don't want whitespace to appear in your output. You can place an otherwise meaningless `xsl:text` element to force the whitespace to be insignificant. This example again has the same output as the two earlier examples. Without the `xsl:text` element, the literal dash on the line with the first `xsl:value-of` element would have placed the results of the second `xsl:value-of` on a separate line:

```
<xsl:template match="/">
   <xsl:value-of select="'First Line'"/> -<xsl:text/>
   <xsl:value-of select="' more on the first line -'"/>
   <xsl:value-of select="'Second Line'" />
</xsl:template>
```

Other whitespace characters, such as spaces and tabs, work similarly.

XPath Marks the Spot

Your XML file is the treasure map, and XPath shows you how to zoom in on exactly the data that's the goal of your current quest. XPath is important in navigating the XML Document Object Model (DOM) within .NET and in XSLT. It's also the basis for XQuery that Microsoft announced for the next version of SQL Server (Yukon). In the same way that it's hard to avoid SQL in today's world, it'll be hard to ignore XPath and XQuery in tomorrow's world.

XPath expressions appear as quoted strings in XSLT. When working with the XML DOM in .NET, `SelectNodes`, `SelectSingleNode`, and `XPathExpression` also use the XPath expressions. Two differences are that XSLT allows variables, and .NET allows you to concatenate the XPath expression at runtime. The XML DOM in

.NET and XSLT also determine the current context a little differently. Within an XSLT stylesheet, there's a current context, determined by previous `xsl:for-each` and `xsl:template` match directives that were processes. `SelectSingleNode` and `SelectNodes` are methods of the `XMLNode` class and use the context of the XML node. In this section, I'll use XSLT for the examples, but you can also test them in .NET.

This section covers XPath in moderate detail. It doesn't try to cover everything, but it covers the core features. Before starting this section, read the sidebar "One-Minute XPath" to give you a general idea of how XPath works.

One-Minute XPath

Let's start with the one-minute version of XPath. It's really not going to be the whole story, but it'll help you get in the right frame of mind. Close your eyes and remember how navigating to different directories at an MS-DOS command line works.[9] A dot represents the current directory, double dots represent the parent directory, and a star represents all files and subdirectory relative to the current directory. Now imagine what a directory tree looks like when printed. Imagine that you put your finger on one location in the directory tree—that's your current directory; and from that point, a double dot is that parent directory. If you're elsewhere in the tree, the parent directory is different.

Where your finger currently is in your imaginary printout is your current directory or current position. XPath calls this the *context,* and context is the stumbling block for many people getting up to speed in XPath. If you get confused about what context is, close your eyes, bring back the image of a simple DOS directory structure, and imagine your XML document as a printed directory structure with your finger at the current context.

XPath supports the familiar dot and double dot syntax for the current and parent context, but these are shortcuts for a formalized syntax.[10] XPath uses the forward slash instead of the backslash, and you need to get that right (the forward slash is in the bottom row of your keyboard and is /).

XPath is a very rich and capable selection technology, so it quickly moves beyond the DOS analogy. The details can get a little jumbled, but the core idea is context, shifting context, and selecting nodes in relation to the current context.

9. If you are unfamiliar with navigating directories from the MS-DOS command line, skip this exercise.

10. Similarly, the asterisk is supported as a shortcut for the child elements.

Exploring XPath Navigation

An XML representation of the British royal family offers a convenient way to demonstrate navigation using XPath. This makes phrases such as *grandparent* immensely logical. However, the same rules would work for any XML document. This XML input for examples in this section is as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/xsl" href="BritishRoyalFamilyTests.xslt" ?>
<Family Name="British royals">
    <Person Name="George VI" DOB="12/14/1895" Sex="Male">
        <Person Name="Elizabeth" DOB="4/21/1921" Sex="Female">
            <Person Name="Charles" DOB="11/14/1948" Sex="Male">
                <Person Name="William" DOB="6/21/1982" Sex="Male" />
                <Person Name="Harry" DOB="9/15/1984" Sex="Male" />
            </Person>
            <Person Name="Anne" DOB="8/15/1950" Sex="Female">
                <Person Name="Peter" DOB="11/15/1997" Sex="Male"/>
                <Person Name="Zara" DOB="5/15/1981" Sex="Female" />
            </Person>
            <Person Name="Andrew" DOB="2/19/1960" Sex="Male">
                <Person Name="Beatrice" DOB="8/8/1988" Sex="Female" />
                <Person Name="Eugenie" DOB="3/23/1990" Sex="Female" />
            </Person>
            <Person Name="Edward" DOB="3/10/1964" Sex="Male" />
        </Person>
        <Person Name="Margaret" DOB="8/21/1930" Sex="Female">
            <Person Name="David" DOB="11/3/1961" Sex="Male" />
                <Person Name="Margarita" DOB="5/14/2002" Sex="Female" />
                <Person Name="Charles" DOB="7/2/1999" Sex="Male" />
            <Person Name="Sarah" DOB="5/1/1964" Sex="Female" />
                <Person Name="Samuel" DOB="7/28/1996" Sex="Male" />
                <Person Name="Arthur" DOB="2/5/1999" Sex="Male" />
        </Person>
    </Person>
</Family>
```

Refer to this input XML as you explore the following examples.

Understanding Context

I hope you've already gotten an idea of context from the DOS exercise in the sidebar "One-Minute XPath." Context is really, really important in XPath, especially when you're working with it in XSLT. The special importance in XSLT stems from using XPath to manage the structure and flow in an XSLT stylesheet, not just to select data for output. The context of a match template is the node currently being processed. You can also change the context with an `xsl:for-each`, with the current context again being the node currently being processed each time through the loop. That's where your finger is in the XML if you're thinking in terms of the DOS exercise.

Using Location Paths

One of the most important types of XPath expressions is a *location path* used for navigation within an XPath document. A location path is made of three parts: the *axis*, *node test,* and *predicate*. The axes and node test determine the resulting context, or the element returned. The predicate allows additional filtering of the possible return elements. Dual semicolons separate the axis from the node path, and square brackets surround the predicate. The axis is more or less the direction to look, the node test is what to return, and the predicate is an extra filter. For example:

```
<xsl:template match="/">
   <xsl:for-each select="descendant::Person[@Name='Anne']">
      <xsl:value-of select="@Name"/>
   </xsl:for-each>
</xsl:template>
```

In this sample, the context is initially the root element because that's the context of the entry-level template. The `descendant` is the direction to look or the axis. The `Person` is the node test so that the resulting node set consists of zero or more `Person` elements. The predicate is `@Name='Anne'`. In English this XPath expression can be stated as, "Process all `Person` nodes that are descendants of the root and have a `Name` attribute of Anne." The result of the `xsl:value-of` is simply "Anne." The axis and the predicate are optional, but the node test is required. If you don't specify the axis, it defaults to `child` (children of the current node). If you don't specify the predicate, all nodes matching the node test are processed.

Using Axes

The axis is the direction to look. If the first character is a slash, this is relative to the root. If the first character is anything else, the axis is relative to the current node, which may or may not be the root. Again, this is consistent with the DOS file path analogy. There are 13 axes. You can find explanations for each in the Visual Studio Help. Table A-4 contains the results of various XPath expressions executed in the context of Anne's node. Rather than repeat explanations, I'll illustrate each axis with an example within Table A-4.

Table A-4. Axes Samples Executed Against the Royal Family Sample XML in the Context of Anne's Element

AXIS	SHORTCUT	EXAMPLE	EXAMPLE RESULTS WITH CONTEXT OF ANNE
ancestor		`<xsl:for-each select= "ancestor::*"> <xsl:value-of select="@Name"/>, </xsl:for-each>`	British royals, George VI, Elizabeth,
ancestor-or-self		`<xsl:for-each select= "ancestor-or-self::*"> <xsl:value-of select="@Name"/>, </xsl:for-each>`	British royals, George VI, Elizabeth, Anne,
attribute	@*	`<xsl:for-each select= "attribute::*"> <xsl:value-of select=”.”/>, </xsl:for-each>`	Anne, 8/15/1950, Female,
child	•	`<xsl:for-each select= "child::*"> <xsl:value-of select="@Name"/>, </xsl:for-each>`	Peter, Zara,
descendant		`<xsl:for-each select= "descendant::*"> <xsl:value-of select="@Name"/>, </xsl:for-each>`	Peter, Zara,

Table A-4. Axes Samples Executed Against the Royal Family Sample XML in the Context of Anne's Element, continued

AXIS	SHORTCUT	EXAMPLE	EXAMPLE RESULTS WITH CONTEXT OF ANNE
descendant-or-self		`<xsl:for-each select= "descendant-or-self::*"> <xsl:value-of select="@Name"/>, </xsl:for-each>`	Anne, Peter, Zara,
following		`<xsl:for-each select= "following::*"> <xsl:value-of select="@Name"/>, </xsl:for-each>`	Andrew, Beatrice, Eugenie, Edward, Margaret, David, Sarah,
following-sibling		`<xsl:for-each select= "following-sibling::*"> <xsl:value-of select="@Name"/>, </xsl:for-each>`	Andrew, Edward,
namespace		`<xsl:for-each select= "namespace::*"> <xsl:value-of select="."/>, </xsl:for-each>`	http://www.w3.org/XML/1998/namespace
parent		`<xsl:for-each select= "parent::*"> <xsl:value-of select="@Name"/>, </xsl:for-each>`	Elizabeth,
preceding		`<xsl:for-each select= "preceding::*"> <xsl:value-of select="@Name"/>, </xsl:for-each>`	Charles, William, Harry,
preceding-sibling		`<xsl:for-each select= "preceding-sibling::*"> <xsl:value-of select="@Name"/>, </xsl:for-each>`	Charles,

Table A-4. Axes Samples Executed Against the Royal Family Sample XML in the Context of Anne's Element, continued

AXIS	SHORTCUT	EXAMPLE	EXAMPLE RESULTS WITH CONTEXT OF ANNE
self		`<xsl:for-each select= "self::*"> <xsl:value-of select="@Name"/>, </xsl:for-each>`	Anne,
	//	`<xsl:for-each select= "//*"> <xsl:value-of select="@Name"/>, </xsl:for-each>`	British royals, George VI, Elizabeth, Charles, William, Harry, Anne, Peter, Zara, Andrew, Beatrice, Eugene, Edward, Margaret, David, Margarita, Charles, Sarah, Samuel, Arthur

Some of the axes have abbreviations or shortcuts. If a shortcut is available, it's commonly used. The last shortcut in Table A-4 doesn't correspond exactly to an axis. Two adjacent slashes mean `descendants-or-self` axis. In addition, it supports a special meaning when it appears at the start of an XPath expression. In that location, it means any descendent of the root element, not the current context. This means it returns all matching nodes anywhere in the document. When it appears other than at the start of the XPath, it indicates the descendants-or-self axis of the previous portion of the XPath statement. The following example returns all persons named Anne appearing anywhere in the document, regardless of the current context when it's processed:

```
<xsl:for-each select="//Person[@Name='Anne']">
   <xsl:value-of select="@Name"/>
</xsl:for-each>
```

The next example outputs "Charles" twice because there are two persons named Charles and nodes are processed regardless of where in the descendants of George VI the name occurred:

```
<xsl:for-each select=" //Person[@Name='George VI']//Person[@Name='Charles']">
   <xsl:value-of select="@Name"/>
</xsl:for-each>
```

Returning Elements Based on Node Tests

The node test specifies what nodes to return. This will be the name of an element or * for all. Regardless of the context, the following example returns all `Person` nodes in the document, which in the sample is all nodes except the `Family` node:

```
<xsl:for-each select="//Person">
   <xsl:value-of select="@Name"/>,
</xsl:for-each>
```

The node test can contain many parts. This is a good time to think back to the DOS directory analogy. You're probably comfortable with long directory paths, and this is no different. For example, the following returns all of the grandchildren of persons directly under the `Family` element. Only George VI is directly under this element, so this returns his ten grandchildren:

```
<xsl:for-each select="/Family/Person/Person/Person">
   <xsl:value-of select="@Name"/>,
</xsl:for-each>
```

The node test is required and determines the resulting context for `xsl:for-each` or `xsl:apply-template` processing.

Selecting with Predicates

The optional predicate applies additional conditions on the matches. Predicates can examine relative elements, use XPath functions, and so on. The following example returns all of the women in the British royal family:

```
<xsl:for-each select="//*[@Sex='Female']">
   <xsl:value-of select="@Name"/>,
</xsl:for-each>
```

You can combine multiple predicates. The results are the intersection of the resulting set. Stated differently, that's all the nodes that satisfy both criteria. The following examples are synonymous and return the women in the family that have names longer than five characters:

```
<xsl:for-each select="//*[@Sex='Female'][string-length(@Name)>5]">
   <xsl:value-of select="@Name"/>,
</xsl:for-each>
```

```
<xsl:for-each select="//*[@Sex='Female' and string-length(@Name)>5]">
   <xsl:value-of select="@Name"/>,
</xsl:for-each>
```

Performance is generally dependent on the processor. Using `and` is documented as being faster than sequential predicates with the Microsoft XML Core Services (MSXML 4.0).

Using Complex Location Paths

You can use predicates to filter nodes at any level within the node test. For example, to retrieve all of the female line grandchildren of George VI (where the child and grandchild were both women), you could use this:

```
<xsl:for-each select="/Family/Person[starts-with(@Name,'George')]/
                  Person[@Sex='Female']/Person[@Sex='Female']">
   <xsl:value-of select="@Name"/>,
</xsl:for-each>
```

The limit to the complexity you can use within XPath is what you and follow-on programmers will understand (although I'm sure there's a length limit, I haven't run into it yet). Your goal is clarity. Depending on your goal, it may be clearer to create a variable or two and not let it get much more complex than this.

Introducing Unions

You can create a union between two sets of nodes. A union includes all nodes of the first clause combined with all nodes of the second. You define a union with the pipe character (`|`):

```
<xsl:for-each select="//Family|Person">
   <xsl:value-of select="@Name"/>,
</xsl:for-each>
```

This example requests any `Family` node in the XML document and any `Person` that's a child of the current node.

NOTE When you don't specify the axis for the second clause of a union, the second clause uses the default of `child`, not the axis of the first clause. If the current node is Anne, the result is the "British royals, Peter, Zara," output.

Summarizing XSLT and XPath

It'd be unfair not to clarify that I just breezed through XSLT and XPath. I haven't touched on everything you'll see in the book or in XSLT or XPath you'll see elsewhere. That wasn't the goal. The goal was to give you enough information that you could build an adequate understanding to figure things out. Combining this, the reference sections that follow, and the Visual Studio Help will give you enough information to figure out the templates and XPath expressions you run across.

XSLT and XPath Reference

Unfortunately, finding information on XSLT and XPath in the Visual Studio Help is difficult unless you know the name of the feature for which you're looking. Once you find the right Help item, it's often quite useful, including the parent element where the directive is valid, appropriate attributes, examples, and so on. To give you insight into this, the tables in this section summarize available XSLT and XPath features. Table A-5 contains available XSLT elements, and Table A-6 contains the XPath functions.

When I'm learning new technologies, I like to know what to initially focus on, so I included my entirely subjective opinion on which directives and functions are most important in relation to the steps of code generation. I'll state this in terms of whether I think they'll be common in the stylesheets you work with (write or use). Because XML and code outputting are different in how they use directives, there's a column for each. All of the elements and functions are useful in some scenarios—these columns indicate how likely it is that you'll encounter them soon.

Table A-5. XSLT Elements

DIRECTIVE	COMMON IN CREATING CODE	COMMON IN CREATING XML	DESCRIPTION
!--	Yes	Yes	Comment within the XSLT. Use this to document your stylesheet. This is an exclamation mark and two dashes, which is sometimes hard to see in the book's code font.
`xsl:apply-imports`			Helps manage template rules when imported rules conflict.
`xsl:apply-templates`	Yes	Yes	Processes match templates for a specified set of nodes.
`xsl:attribute`		Yes	Outputs an XML attribute.

Table A-5. XSLT Elements, continued

DIRECTIVE	COMMON IN CREATING CODE	COMMON IN CREATING XML	DESCRIPTION
`xsl:attribute-set`			Defines a set of attributes. This is rarely used.
`xsl:call-template`	Yes	Yes	Processes a named template.
`xsl:choose`	Yes	Yes	Branching construct allowing multiple branches—a `case` or `switch` statement.
`xsl:comment`		Yes	Outputs a comment node when outputting XML.
`xsl:copy`		Yes	Shallow copy of a node.
`xsl:copy-of`			Deep copy of a node. You don't use this often because you're generally modifying the XML.
`xsl:decimal-format`			Defines how the format-number function behaves.
`xsl:document`			Allows additional XML output documents, not currently available in .NET. Note that there's also a `document` XPath function.
`xsl:element`		Yes	Outputs an XML element.
`xsl:fallback`			Planned for future versioning issues.
`xsl:for-each`	Yes	Yes	Loop through a specified set of nodes.
`xsl:if`	Yes	Yes	Branching construct that allows only one branch option.
`xsl:import`	Yes	Yes	Specified a stylesheet to be processed as part of the current stylesheet. Differs from `xsl:include` in details of how conflicts are resolved—stick to one approach.
`xsl:include`	Yes	Yes	Specified a stylesheet to be processed as part of the current stylesheet. Differs from `xsl: import` in details of how conflicts are resolved—stick to one approach.

Table A-5. XSLT Elements, continued

DIRECTIVE	COMMON IN CREATING CODE	COMMON IN CREATING XML	DESCRIPTION
`xsl:key`			Allows name-value pairs to pre-index nodes in the document for performance. Use to manage complex selection criteria.
`xsl:message`			Allows you to specify text to include in the system-level error message if processing is aborted.
`xsl:namespace-alias`			Allows you to adjust namespace prefixes when you must resolve conflicting prefixes.
`xsl:number`			Automates numbering of items in the output. Note that there's also a `number` XPath function.
`xsl:otherwise`	Yes	Yes	Part of the `xsl:choose` branching construct.
`xsl:output`	Yes	Yes	Specifies the type of output—`xml`, `html`, or `text`.
`xsl:param`	Yes	Yes	Retrieves an argument within a template. Also used as a child of the stylesheet to retrieve stylesheet parameters.
`xsl:preserve-space`	Yes		Indicates that whitespace for specified nodes should be preserved.
`xsl:processing-instruction`			Outputs an XML processing instruction.
`xsl:script`			Defines an inline script block.
`xsl:sort`	Yes	Yes	Specifies sorting for `xsl:for-each` and `xsl:apply-templates`.
`xsl:strip-space`			Indicates that whitespace for specified nodes should be removed.
`xsl:stylesheet`	Yes	Yes	XSLT document root element. Required for XSLT stylesheets.
`xsl:template`	Yes	Yes	Defines both named and match templates.

Table A-5. XSLT Elements, continued

DIRECTIVE	COMMON IN CREATING CODE	COMMON IN CREATING XML	DESCRIPTION
`xsl:text`	Yes		Outputs specified text and is generally used to manage whitespace.
`xsl:transform`			Synonym for `xsl:stylesheet`.
`xsl:value-of`	Yes	Yes	Outputs value specified in XPath expression.
`xsl:variable`	Yes	Yes	Defines a variable.
`xsl:when`	Yes	Yes	Part of the `xsl:choose` branching construct.
`xsl:with-param`	Yes	Yes	Defines parameters that will be passed to match or named templates.

Table A-6. XPath Functions

FUNCTION	COMMON IN CREATING CODE	COMMON IN CREATING XML	DESCRIPTION
`boolean`			Converts the item to a `Boolean`. False, zero, empty strings, and empty node-sets return `false`.
`ceiling`			Returns the next larger integer.
`concat`	Yes	Yes	Combines multiple strings.
`contains`	Yes	Yes	Returns `true` if the first string contains the second.
`count`	Yes	Yes	Number of nodes in the set.
`current`			Returns a node-set containing only the current node.
`document`			Allows additional input XML data to be included.
`element-available`			Returns `true` if the node is an XSLT directive supported in the current release. Intended for future versioning.
`false`			Returns `false`.

Table A-6. XPath Functions, continued

FUNCTION	COMMON IN CREATING CODE	COMMON IN CREATING XML	DESCRIPTION
`floor`			Returns the next smaller integer.
`format-number`			Formats a number based on a format string.
`function-available`			Returns `true` if the passed string is an available function. Intended for future versioning.
`generate-id`			Returns a string suitable for use as an ID. There's no guarantee that it'll be unique from IDs previously in the document.
`id`			Retrieves a node by its ID.
`key`			Retrieves a node by its key.
`lang`			Returns `true` or `false` based on the `xml:lang` element.
`last`	Yes		The number of the last node in the set. This differs from count when the XPath flattens the hierarchy.
`local-name`			Returns the local portion of the current or passed node.
`name`	Yes	Yes	Returns the full name of the passed node. This is particularly valuable during debugging.
`namespace-uri`			Returns the URI of the expanded node.
`node-set`			Converts a result tree to a node-set. You'll use this when creating interim XML for use as a variable—an advanced topic.
`normalize-space`			Removes whitespace from the start and end of the string.
`not`	Yes	Yes	Negates the Boolean value: `true` to `false` or `false` to `true`.

Table A-6. XPath Functions, continued

FUNCTION	COMMON IN CREATING CODE	COMMON IN CREATING XML	DESCRIPTION
`number`			Converts the item to a number. Strings that can't be converted are NaN (not a number). The first node of a node-set is converted to a string, then a number.
`position`	Yes		Position of the node being processed in the node-set.
`round`			Returns the next value further from zero. This differs from floor for negative values.
`starts-with`	Yes	Yes	Returns `true` if the first string starts with the second.
`string`			Converts the item to a string. Returns the first node if node set, `true`/`false` if Boolean. Consider `format-number` for converting numbers to string.
`string-length`	Yes	Yes	Returns the length of the string passed. This is particularly useful to find zero-length strings.
`substring`	Yes	Yes	Selects a subset of characters from the string.
`substring-after`	Yes	Yes	Returns portion of the first string that occurs after the second.
`substring-before`	Yes	Yes	Returns portion of the first string that occurs before the second.
`sum`			Sum of the values of all nodes in a node-set.
`translate`	Yes	Yes	Replaces characters in the first string with character mapped by the second and third strings. Use for changing capitalization.
`true`			Returns `true`.
`system-property`			Returns information about the XSLT processor being used.
`unparsed-entity-uri`			Used to retrieve information about XML entities.

Additional Reading

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

- The book I reach for on XSLT is *XSLT Programmer's Reference,* second edition, by Michael Kay (Wrox Press, 2001)
- *Definitive XML Schema* by Priscilla Walmsley (Prentice Hall, 2001)
- *Definitive XSLT and XPath* by G. Ken Holman (Prentice Hall, 2001)
- *Special Edition Using XML Schema* by David Gulbransen (Que, 2001)
- *The XSL Companion*, second edition, by Neil Bradley (Addison-Wesley, 2002)
- *Visual Basic .NET and XML: Harness the Power of XML in VB .NET Applications* by Rod Stephens and Brian Hochgurtel (Wiley & Sons, 2002)
- *XML Programming* by R. Allen Wyke, Sultan Rehman, and Brad Leupen (Microsoft Press, 2002)
- *XSLT and XPath: A Guide to XML Transformations* by John Robert Gardner and Zarella L. Rendon (Prentice Hall, 2001)
- *XSLT and XPath on the Edge* by Jeni Tennison (Wiley & Sons, 2001)
- You can visit `http://www.W3C.org`, which is the official site of the standards body in charge of X technologies. The site contains an overwhelming amount of information.
- Visit `http://www.zvon.org` for tutorials on X technologies
- Visit `http://www.xmlpitstop.com` for more tutorials on X technologies

APPENDIX B

Dissecting the Metadata Extraction Tools

THIS APPENDIX COVERS DETAILS OF TOOLS presented in Chapter 2. It's like peeling an onion—it shows you more about underlying code. But like looking at the insides of any internal tool, don't let it bring you to tears. Unless you need to make changes, you probably don't have to understand these internal details.

This appendix covers two tools. The first is the template that converts XML Schema Definition (XSD) to something friendly for code generation. Walking through the complex template will help you better understand how XSL Transformations (XSLT) works.

The second tool covered is the SQL-92 metadata extraction tool. This tool takes information available based on the SQL-92 standard and adds information specific to SQL Server. If you're working with Oracle or another SQL-92 tool, you'll need to mimic the SQL Server–specific class. If you're working with Access or another non-SQL-92 tool, you'll need to replace both classes and may want to understand how I structured these before you start. I suggest you use the same pattern, and you'll need to use the same public interface to work with the existing harness tool (discussed in Appendix C).

NOTE The third tool covered in Chapter 2, the metadata merge tool, is discussed only in Chapter 2. Appendix C covers the code generation harness.

Converting XSD into Friendly XML

XSDToFriendlyXML.xslt transforms an XSD describing an Extensible Markup Language (XML) file to the format expected by later code generation. Chapter 2 discusses more about why you want to perform this transform and shows samples of the target output of this stylesheet. Chapter 3 shows how you actually call it from the harness. If you get lost in this walk-through, you might want to read

Appendix A for an understanding of how XSLT and XPath work, as well as a sample XSD that shows the structure of the input document. I'll walk through the entire code of the template, only skipping repetitive portions where indicated.

The file opens with the standard XSLT goo. It then adds the `http://www.kadgen/ExtensionObject` namespace for extension objects used by the template. The `http://www.w3.org/1999/XSL/Transform` namespace is required for processing XSLT templates. The input file for this conversion is an XSD, so it contains schema elements resolved by `http://www.w3.org/2001/XMLSchema` and Microsoft-specific schema tags resolved by `urn:schemas-microsoft-com:xml-msdata`. If you infer the XSD using Visual Studio tools, the tools add the Microsoft-specific tags. Other schema inference tools may add different tags, and any namespace appearing in the input XSD should be resolved in the XSLT stylesheet. The `http://www.kadgen/DatabaseStructure` namespace resolves prefixes in the output document:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!– Summary:  Creates friendly XML output. Input is single DataSet XSD !–>
<xsl:stylesheet version="1.0"
          xmlns:ext="http://www.kadgen/ExtensionObject"
          xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
          xmlns:xs="http://www.w3.org/2001/XMLSchema"
          xmlns:dbs="http://www.kadgen/DatabaseStructure"
          xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
```

This template uses utility templates in Support.xslt defined by the `xsl:import` statement. Note that although other templates producing code output to `text`, this template specifies the output as `xml`. This tag provides the standard header and supports indenting to make your output more readable:

```
<xsl:import href="Support.xslt" />
<xsl:output method="xml" encoding="UTF-8" indent="yes" />
<xsl:preserve-space elements="*" />
```

This stylesheet has one entry-level template that matches the root element. This template calls a match template for the root `xs:schema` element and specifies the mode for that template. Because `xs:schema` is the root node, the *entry-level template* will be called only once:

```
<xsl:template match="/">
   <xsl:apply-templates select="xs:schema" mode="DataSet"/>
</xsl:template>
```

The DataSet mode template matches the `xs:schema` elements and outputs the outer elements of the resulting XML. The `xsl:element` directive creates an

XML element in the output document. The name of the root element created in the output document is `DataStructures`, which contains a set of `DataStructure` nodes. This template also calls the `Table` mode template for each child element that contains a nested complex type that in turn contains a choice that contains an element. Whew! In English, this *high-level processing template* calls the `Table` mode template for each contained table:

```
<xsl:template match="xs:schema" mode="DataSet">
   <xsl:element name="dbs:DataStructures">
      <xsl:attribute name="xmlns">http://kadgen/DatabaseStructure</xsl:attribute>
      <xsl:element name="dbs:DataStructure">
         <xsl:attribute name="Name">
            <xsl:value-of select="@id" />
         </xsl:attribute>
         <xsl:element name="dbs:Tables">
            <xsl:apply-templates
                     select="xs:element/xs:complexType/xs:choice/xs:element"
                     mode="Table" >
               <xsl:sort select="name" case-order="upper-first" />
            </xsl:apply-templates>
         </xsl:element>
      </xsl:element>
   </xsl:element>
</xsl:template>
```

The `Table` mode template creates variables for the table name and primary key. It uses a named template to retrieve the primary key. Calling a named template is similar to calling a subroutine—it executes only once each time you call it and the context remains unchanged. The variable `pKeyName` contains anything output by the named template `GetPKeyName` because it's called within this variable's declaration:

```
<xsl:template match="xs:element" mode="Table">
   <xsl:variable name="tablename" select="@name" />
   <xsl:variable name="pKeyName">
      <xsl:call-template name="GetPKeyName">
         <xsl:with-param name="TableName" select="$tablename" />
      </xsl:call-template>
   </xsl:variable>
```

One of the cool things you can do when you execute XSLT using the `XSLTransform` class of the .NET Framework is to call back into the framework using extension objects. The XML specification provides for extensibility, and extension

objects are Microsoft's way of implementing extensibility. The .NET implementation allows you to specify extension objects within the `Xml.Xsl.XsltArgumentList` along with a namespace. You specify the same namespace within your stylesheet. In this case, `http://www.kadgen/ExtensionObject` is assigned to the `ext` prefix in the stylesheet and the object within the `XsltArgumentList`. Once this is set up, it's easy to call methods in the .NET object:

```
<xsl:variable name="fixedname" select="ext:FixName(@name)"/>
```

NOTE Appendix A covers extension objects in more detail.

When you're working with the code generation harness, you specify the extension object as the `AssemblyFileName`, `TypeName`, and `NamespaceURI` in the `XSLTFiles` box of the `XSLTProcess` or `XSLTGeneration` directives. You must process the stylesheet with the correct object available or you'll get an exception when you run the harness script. In this case, the required object is the `Tools` class of the `Utility` assembly. The benefit of calling back to .NET in this case is substantial. It allows consistent processing of names, types, and plurals, which would otherwise be a real pain to synchronize.

NOTE Use extension objects in moderation where you can demonstrate a benefit—such as consistency with tasks performed by .NET classes in your generation process. For example, the `FixName` method allows XSD processing and the `ExtractMetadata` class to handle names consistently.

The remainder of this template outputs the `Table` element. Processing includes three `xsl:apply-templates` calls to create additional elements. The nesting of the output matches the target XML structure, which is important so you can later merge XSD-based metadata and SQL-92 based metadata:

```
<xsl:element name="dbs:Table">
   <xsl:attribute name="Name">
      <xsl:value-of select="$fixedname" />
   </xsl:attribute>
   <xsl:attribute name="OriginalName">
      <xsl:value-of select="$tablename" />
   </xsl:attribute>
   <xsl:attribute name="SingularName">
      <xsl:value-of select="ext:GetSingular($fixedname)" />
```

```
      </xsl:attribute>
      <xsl:attribute name="PluralName">
         <xsl:value-of select="ext:GetPlural($fixedname)" />
      </xsl:attribute>
      <xsl:element name="dbs:TableColumns">
         <xsl:apply-templates select="xs:complexType/xs:sequence/xs:element"
                              mode="Column" />
      </xsl:element>
      <xsl:element name="dbs:TableConstraints">
         <xsl:call-template name="AddPrimaryKey">
            <xsl:with-param name="pKeyName" select="$pKeyName" />
         </xsl:call-template>
         <xsl:element name="dbs:TableRelations">
            <xsl:apply-templates select="//xs:keyref[@refer=$pKeyName]"
                                 mode="GetChildRelations" />
            <xsl:apply-templates   select=

"//xs:keyref[xs:selector/@xpath=concat('.//mstns:',$tablename)]"
                                 mode="GetParentRelations" />
         </xsl:element>
      </xsl:element>
   </xsl:element>
</xsl:template>
```

NOTE Chapter 2 discusses defining multiple names (`Name`, `OriginalName`, `SingularName`, and `PluralName`) for the database tables.

The Column template continues to use the same techniques but is rather long because it outputs a number of things. Again, the template calls the `FixName` method of an XSLT extension object to remove spaces and munge keywords. It outputs the element along with the `Name` and `OriginalName` attributes. The same attributes were output for the `Table` element, and the template uses the same code to create the attribute values. However, the values are different because this template executes in the context of the column, where the previous template executed in the context of the table:

```
<xsl:template match="xs:element" mode="Column">
   <xsl:variable name="fixedname" select="ext:FixName(@name)"/>
   <xsl:element name="dbs:TableColumn">
      <xsl:attribute name="Name">
```

```
            <xsl:value-of select="$fixedname " />
        </xsl:attribute>
        <xsl:attribute name="OriginalName">
            <xsl:value-of select="@name" />
        </xsl:attribute>
```

The ordinal is just the position of the column in the list as contained in the XSD, and if a default value exists, the template outputs it:

```
<xsl:attribute name="Ordinal">
   <xsl:value-of select="position()" />
</xsl:attribute>
<xsl:attribute name="Default">
   <xsl:value-of select="@Default"/>
</xsl:attribute>
```

Because null values are just left out of XML, the `minOccurs=0` attribute indicates that nulls are allowed in the field. The `AllowNulls` attribute is a friendlier way to express this and is consistent with the target structure:

```
<xsl:choose>
   <xsl:when test="@minOccurs='0'">
      <xsl:attribute name="AllowNulls">true</xsl:attribute>
   </xsl:when>
   <xsl:otherwise>
      <xsl:attribute name="AllowNulls">false</xsl:attribute>
   </xsl:otherwise>
</xsl:choose>
```

The template outputs three data types for each column. It preserves the original XSD type to aid later debugging. If the schema is retrieved programmatically using `MissingSchemaAction=AddWithKey`, extra information including the maximum length of string columns will be returned. To store this information, the XSD contains an extra simple type. In this case, the type within the element representing the column is the name of that simple type, not the true type. To resolve this, you need to grab the base type. The `basetype` variable holds this original type:

```
<xsl:variable name="basetype">
   <xsl:choose>
      <xsl:when test="@type">
         <xsl:value-of select="@type" />
      </xsl:when>
      <xsl:otherwise>
```

```
            <xsl:value-of select=".//@base" />
         </xsl:otherwise>
      </xsl:choose>
   </xsl:variable>
```

The sqltype variable holds the SQL type returned from the utility functions available through the extension object. Stuffing the SQL type in a variable allows it to be used multiple times. The template outputs three type values as attributes, along with the maximum length if it's available:

```
      <xsl:variable name="sqltype">
         <xsl:value-of select="ext:GetSQLTypeFromXSDType($basetype)"/>
      </xsl:variable>
      <xsl:attribute name="OriginalType">
         <xsl:value-of select="$basetype" />
      </xsl:attribute>
      <xsl:attribute name="SQLType">
         <xsl:value-of select="$sqltype" />
      </xsl:attribute>
      <xsl:attribute name="NETType">
         <xsl:value-of select="ext:GetNETTypeFromSQLType($sqltype)"/>
      </xsl:attribute>
      <xsl:if test=".//xs:maxLength">
         <xsl:attribute name="MaxLength">
            <xsl:value-of select=".//xs:maxLength/@value" />
         </xsl:attribute>
      </xsl:if>
   </xsl:element>
</xsl:template>
```

The GetPKeyName named template retrieves the name of the primary key constraint within the current DataSet. It also demonstrates a good use for <xsl:for-each>. As discussed in Appendix A, this construct initially feels more comfortable to procedural programmers like us, but exclusive use of it over <xsl:apply-templates> can result in unwieldy templates that are difficult to read. One exception is short, one-or-two line fragments:

```
<xsl:template name="GetPKeyName">
   <xsl:param name="TableName" />
   <xsl:for-each select="//xs:unique[@msdata:PrimaryKey='true']/xs:selector[
                         @xpath=concat('.//mstns:',$TableName)]">
      <!— We are currently in the child node of the primary key, and have
            confirmed there is a primary key —>
      <xsl:value-of select="../@name" />
```

```
   </xsl:for-each>
</xsl:template>
```

CAUTION If you use multicolumn primary keys within this `XSDToFriendlyXML` transformation, you'll need to change this template, perhaps to return a comma-delimited list.

The XPath expression in the `<xsl:for-each>` loop is worth a second look. This returns the `<xs:selector>` node in the document that's a child of an `<xs:unique>` node that's a primary key. In addition, the `<xs:selector>` node must have an attribute matching the table name and a specific prefix. This is an ugly, but effective, way for Microsoft to extend XSDs to include primary key information. Once you've got the `<xs:selector>` node, you output the parent node name—the `<xs:unique>` name.

Once you get the primary key name, you can pass it to the `AddPrimaryKey` template. Because you know the primary key name, it's easy to grab the primary key node and output an element. Looping through the child field nodes lets you also output an element for each child field:

```
<xsl:template name="AddPrimaryKey">
   <xsl:param name="pKeyName" />
   <xsl:for-each select="//xs:unique[@name=$pKeyName]">
      <xsl:element name="dbs:PrimaryKey">
         <xsl:for-each select="xs:field">
            <xsl:element name="dbs:PKField">
               <xsl:attribute name="Name">
                  <xsl:value-of select="substring-after(@xpath,'mstns:')" />
               </xsl:attribute>
               <xsl:attribute name="Ordinal">
                  <xsl:value-of select="position()" />
               </xsl:attribute>
            </xsl:element>
         </xsl:for-each>
      </xsl:element>
   </xsl:for-each>
</xsl:template>
```

Moving on to outputting related tables, the `GetChildRelations` mode template creates a `ChildTable` element. This template runs in the context of the child table in the XSD. The template creates `ChildKeyFields` elements for each child key:

```
<xsl:template match="xs:keyref" mode="GetChildRelations">
   <!-  I am including parent and child fields because otherwise people might
         be lazy, not track down the correct primary ->
   <xsl:variable name="refer" select="@refer" />
   <xsl:element name="dbs:ChildTable">
      <xsl:attribute name="Name">
         <xsl:value-of select="substring-after(xs:selector/@xpath,'mstns:')" />
      </xsl:attribute>
      <xsl:for-each select="xs:field">
         <xsl:element name="dbs:ChildKeyFields">
            <xsl:attribute name="Name">
               <xsl:value-of select="substring-after(@xpath,'mstns:')" />
            </xsl:attribute>
            <xsl:attribute name="Ordinal">
               <xsl:value-of select="position()" />
            </xsl:attribute>
         </xsl:element>
      </xsl:for-each>
   </xsl:element>
</xsl:template>
```

The last template is the `GetParentRelations` template. This template executes in the context of the parent relation. The output XML records each relation as both a child relation and a parent relation. This redundancy makes the later outputting of code much easier. Although long, this template doesn't present any new approaches:

```
   <xsl:template match="xs:keyref" mode="GetParentRelations">
      <xsl:variable name="relationnode" select="." />
      <xsl:variable name="pKey" select="@refer" />
      <xsl:element name="dbs:ParentRelations">
         <xsl:for-each select="//xs:unique[@name=$pKey]/xs:selector">
            <xsl:element name="dbs:ParentRelation">
               <xsl:attribute name="ParentTable">
                  <xsl:value-of select="substring-after(@xpath,'mstns:')" />
               </xsl:attribute>
               <xsl:for-each select="../xs:field">
                  <xsl:element name="dbs:ParentKeyField">
                     <xsl:attribute name="Name">
                       <xsl:value-of select="substring-after(@xpath,'mstns:')" />
                     </xsl:attribute>
                     <xsl:attribute name="Ordinal">
```

```
                        <xsl:value-of select="position()" />
                     </xsl:attribute>
                  </xsl:element>
               </xsl:for-each>
               <xsl:for-each select="$relationnode/xs:field">
                  <xsl:element name="dbs:ChildField">
                     <xsl:attribute name="Name">
                       <xsl:value-of select="substring-after(@xpath,'mstns:')" />
                     </xsl:attribute>
                     <xsl:attribute name="Ordinal">
                        <xsl:value-of select="position()" />
                     </xsl:attribute>
                  </xsl:element>
               </xsl:for-each>
            </xsl:element>
         </xsl:for-each>
      </xsl:element>
   </xsl:template>
</xsl:stylesheet>
```

That's the end of the XSDToFriendlyXML.xslt template. This template is handy for transforming XSD files into the metadata structure expected for later processing.

Extracting Metadata from Databases

Chapter 2 discusses how the base `ExtractMetadata` class and the backend-specific derived class work together to extract metadata using the generic information schema views. This section walks through portions of the `ExtractMetadata` class and most of the SQL Server–specific `SQLExtractMetadata` class. If you just want to use the metadata extraction tool, you probably won't need this walk-through. But, you'll need to understand the code more deeply if you need to change it. You might need to change it to include something I omitted or because you want to take advantage of an alternate database backend. If that alternate backend is a SQL-92 compliant database, you should only need to replace the `SQLExtractMetadata` class. If it isn't a SQL-92 database (such as Access), you'll be starting from scratch, but you should be able to learn from the techniques I use here. If you work on a non-SQL-92 backend and you want it to work with the code generation harness, you'll need to implement the `IExtractMetadata` interface. You also might want to go through these walk-throughs if you'd like to understand the tools you're working with at a deeper level.

Looking at the ExtractMetadata Class

In this section, I'll walk through key sections of the `ExtractMetadata` class. Following this, I'll discuss the `SQLExtractMetaData` class that derives from it and offers backend-specific functionality. I won't go through every line of code, and I won't walk through the utility methods, which are short, straightforward, and pretty much described by their names. Generally, the utility functions take a few lines of code needed to do something such as add an element or attribute in XML and wrap them up in a tidy little function. This leaves the code in `ExtractMetadata` more readable.

As discussed in the Chapter 2, `ExtractMetadata` is an abstract (`MustInherit`) base class for metadata extraction that implements the `IExtractMetaData` interface and provides database metadata through the SQL-92 information schemas. The derived class `SQLExtractMetadata` (shown after the `ExtractMetaData` class) can be instantiated for SQL Server data extraction.

NOTE One of the nice things about Visual Basic .NET (VB .NET) is that I can prefix method calls with just enough of the namespace to let you know where the method came from. Generally, that will be a single namespace. If no namespace is used, in most cases the method resides in the project namespace. Partial namespaces are available in C# only through namespace aliases.

Exploring the Class Opening

The opening of the file is pretty standard. The `KADGen` namespace is the top-level namespace for all of the projects in this book. This makes it easier to access other projects that are part of this solution. The class implements the `IExtractMetadata` interface:

```
Option Strict On
Option Explicit On

Imports System
Imports System.Data
Imports System.Diagnostics
Imports KADGen
```

```
' Class Summary: Metadata extracton based on the SQL-92 INFORMATION_SCHEMA.
'                 Tested only against SQL Server

Public MustInherit Class ExtractMetaData
   Implements IExtractMetaData
```

Walking through the class fields, there are declarations for variables that store the name of the server (used in the connection string of derived classes) and two private fields that back Boolean properties. One of these properties indicates whether the metadata uses the full catalog/schema/name pattern or the simple name. The other property indicates whether the full source code of functions and stored procedures should be included in the metadata. Generally, these should both be false so names are readable and stored procedures aren't included in the metadata. The `mXMLDoc` and `mDataSet` variables store the output document and the DataSet you're building that document from. To determine the resultsets produced by stored procedures, you have to run the stored procedures. The `mSelectPatterns` and `mSetSelectPatterns` variables allow you to run only the stored procedures that produce values, not ones that perform other actions, such as inserting records. The `ErrorResponseEnum` simplifies indicating whether to continue or abort processing if an exception occurs:

```
#Region "Class level declarations"
   Protected mServerName As String
   Private mUseVerboseNames As Boolean
   Private mUseProcContents As Boolean = True
   Protected mXMLDoc As Xml.XmlDocument
   Protected mDataSet As Data.DataSet
   Protected mSelectPatterns() As String
   Protected mSetSelectPatterns() As String
   Protected mRemovePrefix As String

   Private Enum ErrorResponse
      Continue
      Abort
   End Enum

#End Region
```

Declaring MustOverride Methods

When you override this class, you have to override two overloaded versions of the `DataTableFromSQL` method and the `RunStoredProc` method. One overload is a

function that returns a DataTable and the other is a sub that takes a DataTable and fills it. The base class calls these methods to fill tables with information from the SQL-92 information schema views:

```
#Region "MustOverrides"
   Protected MustOverride Function DataTableFromSQL( _
                ByVal SQLText As String, _
                ByVal databaseName As String, _
                ByVal tablename As String) _
                As Data.DataTable
   Protected MustOverride Sub DataTableFromSQL( _
                ByVal SQLText As String, _
                ByVal tablename As String, _
                ByVal dt As Data.DataTable)
   Protected MustOverride Function RunStoredProc( _
                   ByVal spname As String, _
                   ByVal nodeParams As Xml.XmlNode, _
                   ByVal databaseName As String) _
                   As Data.DataSet
#End Region
```

I'll skip the `UserVerboseNames`, `UserProcContents`, and `ServerName` public properties that just wrap backing fields and implement members of `IExtractMetaData`.

Implementing CreateMetadata

The public method `CreateMetaData` is the access point for creating metadata. This method takes six parameters. If the `skipStoredProcs` parameter is true, no stored procedure information is created. This improves performance in the initial metadata creation step because only the required stored procedures are executed. The `selectPatterns` and `setSelectPatterns` provide the ability to selectively execute only stored procedures that return recordsets. If `removePrefix` is true, the class removes any lowercase letters that appear at the start of names. This is helpful if you prefix table or column names with the part of the application, such as `appUser` or `accInvoice`. I designed the class interface to support multiple databases, but this hasn't been tested, and the harness interface doesn't yet support it. This method returns an XML document that the harness outputs:

```
Public Function CreateMetaData( _
            ByVal skipStoredProcs As Boolean, _
            ByVal selectPatterns As String, _
            ByVal setSelectPatterns As String, _
            ByVal removePrefix As String, _
```

```
            ByVal lookupPrefix As String, _
            ByVal ParamArray databaseNames() As String) _
            As Xml.XmlDocument _
            Implements IExtractMetaData.CreateMetaData
```

Each XML document can contain only one root element, and the `CreateMetaData` method uses a utility method to create a root element named `MetaDataRoot`:

```
mXMLDoc = New Xml.XmlDocument
Dim nodeRoot As Xml.XmlElement = mXMLDoc.CreateElement("dbs", _
          "MetaDataRoot", "http://kadgen/DatabaseStructure")
nodeRoot.Attributes.Append(Utility.xmlHelpers.NewBoolAttribute(mXMLDoc, _
             "FreeForm", True))
```

CreateElement

The `CreateElement` method adds a new element with the name specified as the second argument (such as `DataStructure`) to an XML document including a `Name` attribute with a value matching the third argument. `CreateElement` returns the new element, which is used to add additional XML elements as child nodes.

The method retrieves a list of databases and processes each one. You'll generally want to pass a single or small number of database names to the `CreateMetaData` method so you focus your metadata to what you're interested in. If no database name is passed, metadata is collected for all databases in the SQL Server instance. This code then sets the class-level variables for `selectPatterns`, `setSelectPatterns`, and `RemovePrefix`:

```
Dim nodeDatabase As Xml.XmlElement
' Retrieve database names if needed
If (databaseNames.GetLength(0) = 0) Then
   Dim dtDatabases As Data.DataTable = GetDatabases()
   ReDim databaseNames(dtDatabases.Rows.Count - 1)
   For i As Int32 = 0 To dtDatabases.Rows.Count - 1
      databaseNames(i) = dtDatabases.Rows(i)("CATALOG_NAME").ToString
   Next
End If
Me.SetSelectPatterns(selectPatterns, setSelectPatterns)
Me.mRemovePrefix = removePrefix
```

The `CreateXmlDeclaration` method of the .NET Framework's `XMLDocument` class creates the standard XML header in the output document. After appending the previously created root element, the method creates a new element named `DataStructures`:

```
mXMLDoc.AppendChild(mXMLDoc.CreateXmlDeclaration("1.0", "UTF-8", ""))
mXMLDoc.AppendChild(nodeRoot)
Dim node As Xml.XmlElement = Me.CreateElement("DataStructures")
nodeRoot.AppendChild(node)
```

For each database used in the extraction, the existing values are cleared from the class-level DataSet, and it's refilled with information retrieved from the current database in the `FillMetaDataSet`. `TableInsertPriority` and `TableDeletePriority` establish the correct order for adding and deleting records from child tables. This code then calls a series of methods to add child nodes that each returns a specific set of metadata. This walk-through covers only the `GetAllTablesXML` and its routines. The others are quite similar. The walk-through of `SQLExtractMetadata` covers the `GetProviderSpecificXMLNodes`:

```
        For Each name As String In databaseNames
            mDataSet = Nothing
            nodeDatabase = Me.CreateElement("DataStructure", name)
            node.AppendChild(nodeDatabase)
            FillMetaDataSet(name)
            TableInsertDeletePriority.appendTableInsertPriority(mDataSet)
            TableInsertDeletePriority.appendTableDeletePriority(mDataSet)
            nodeDatabase.AppendChild(Me.GetAllTablesXML(lookupPrefix))
            nodeDatabase.AppendChild(Me.GetAllUserDefinedTypesXML())
            If Not skipStoredProcs Then
                nodeDatabase.AppendChild(Me.GetAllStoredProcXML(name))
                nodeDatabase.AppendChild(Me.GetAllFunctionsXML())
            End If
            nodeDatabase.AppendChild(Me.GetAllViewsXML())
            GetProviderSpecificXMLNodes(nodeDatabase, _
                  mDataSet.Tables("Database").Rows(0)("CATALOG_NAME").ToString, _
                  mDataSet.Tables("Database").Rows(0)("SCHEMA_NAME").ToString)
            nodeDatabase.AppendChild(Me.GetHierarchyXML())
        Next
        Return mXMLDoc
    End Function

#End Region
```

Looping Through All Tables

The GetAllTablesXML loops through the Tables table of the DataSet calling GetTableXML to return the information on each table. The GetTableXML method appends element information for each table to the Tables grouping node:

```
 Protected Overridable Function GetAllTablesXML(ByVal lookupPrefix As String) _
                As Xml.XmlNode
   Dim nodeTables As Xml.XmlElement = Me.CreateElement("Tables")
   nodeTables.Prefix = "dbs"
   For Each rowTable As Data.DataRow In mDataSet.Tables("Tables").Rows
      ' Each row represents a database table
      nodeTables.AppendChild(GetTableXML(rowTable, "Table", lookupPrefix))
   Next
   Return nodeTables
End Function
```

Outputting XML for Each Table

The GetTableXML method returns an XML element containing information on the table. The FixName method removes spaces and munges keywords. The NewElementWithName method creates this element. NewElementWithName checks the UserVerboseName property to determine whether the table name (and the name for other elements in other parts of this class) should be a simple name or a combination of the database, schema, and table name. Use the verbose name when you place multiple databases in a single XML metadata file and there might be naming conflicts. The singular and plural versions of the name allow simpler naming of later classes and variables in code:

```
Protected Overridable Function GetTableXML( _
                ByVal rowTable As Data.DataRow, _
                ByVal modeName As String, _
                ByVal lookupPrefix As String) _
                As Xml.XmlNode
    Dim tableName As String = rowTable("TABLE_NAME").ToString
    Dim rowColumns() As Data.DataRow = mDataSet.Tables("Columns").Select( _
                        TableMatch(rowTable))
    Dim nodeColumns As Xml.XmlNode
    Dim originalName As String = rowTable("TABLE_NAME").ToString
```

```
    Dim isLookup As Boolean
    Dim name As String = Utility.Tools.FixName( _
                        rowTable("TABLE_NAME").ToString, mRemovePrefix)
    Dim singularName As String = Utility.Tools.GetSingular(name)
    Dim nodeTable As Xml.XmlNode = NewElementWithName(modeName, name, rowTable, _
                        "TABLE")
    nodeTable.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                        nodeTable.OwnerDocument, "OriginalName", _
                        originalName))
    nodeTable.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                        nodeTable.OwnerDocument, "SingularName", _
                        singularName))
    nodeTable.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                        nodeTable.OwnerDocument, "PluralName", _
                        Utility.Tools.GetPlural(name)))
    nodeTable.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                        nodeTable.OwnerDocument, "DisplayName", _
                        GetDisplayName(rowColumns, singularName)))
    nodeTable.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                        nodeTable.OwnerDocument, "Caption", _
                        MakeCaption(singularName)))
```

CAUTION The routines that create singulars and plurals are simple and rely on English language rules. If you work in a different language than English, these rules won't work.

The `lookupPrefix` parameter allows you to specify a prefix that marks the lookup tables in your application. For example, if you use `lku` as the prefix, any tables in your database that start with this prefix will automatically be marked as `IsLookup`. This means less manual entry in the Object Relational Mapping (ORM) file:

```
If (lookupPrefix.Length > 0) AndAlso originalName.StartsWith(lookupPrefix) Then
   isLookup = True
   nodeTable.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                     nodeTable.OwnerDocument, "IsLookup", _
                     "true"))
End If
```

The GetTableXML method is used for both tables and views. When used for tables, the insert/delete priority is included in the metadata output. This allows you to peform these actions on tables in the right order:

```
If modeName = "Table" Then
   nodeTable.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                       nodeTable.OwnerDocument, "DeletePriority", _
                       rowTable("DELETE_PRIORITY").ToString))
   nodeTable.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                       nodeTable.OwnerDocument, "InsertPriority", _
                       rowTable("LOAD_PRIORITY").ToString))
End If
nodeColumns = nodeTable.AppendChild(Me.CreateElement(modeName & "Columns"))
```

GetTableXML calls several child methods. These get information on column, table privileges, table constraints, as well as provider-specific information:

```
  For Each rowColumn As Data.DataRow In rowColumns
     nodeColumns.AppendChild(GetColumnXML(rowTable, rowColumn, modeName, isLookup))
   Next
   nodeTable.AppendChild(nodeColumns)
   nodeTable.AppendChild(GetPrivilegesXML(modeName, mDataSet.Tables( _
                         "TablePrivileges").Select(TableMatch(rowTable))))
   nodeTable.AppendChild(GetTableConstraintsXML(modeName, rowTable))
   GetProviderSpecificXMLNodes(nodeTable, rowTable("TABLE_CATALOG").ToString, _
                         rowTable("TABLE_SCHEMA").ToString, _
                         rowTable("TABLE_NAME").ToString, _
                         modeName)
   Return nodeTable
End Function
```

Outputting XML for Each Column

Looking at the GetColumnXML method, the NewElementWithName again appends either a simple name or a verbose name. This method copies information about the column into attributes of the TableColumn or ViewColumn element of the metadata output. Empty default values aren't copied. Because there's one row in the view for each column in the underlying table, the variable naming gets weird.

The parameter `rowColumn` underscores how confusing this issue can be. It might help if you mentally read `rowColumn` as "row for the column's information." This code copies the ordinal position, default value, data type, and max length into the XML. It also translates the nullable value into a standard Boolean:

```
Protected Overridable Function GetColumnXML( _
                ByVal rowtable As Data.DataRow, _
                ByVal rowColumn As Data.DataRow, _
                ByVal modeName As String, _
                ByVal isLookup As Boolean) _
                As Xml.XmlNode
    Dim isPrimaryKey As Boolean
    Dim nodeColumn As Xml.XmlNode = NewElementWithName(modeName & "Column", _
                    Utility.Tools.FixName(rowColumn("COLUMN_NAME").ToString, _
                                    Me.mRemovePrefix), _
                    rowColumn, "TABLE")
    nodeColumn.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                    nodeColumn.OwnerDocument, "OriginalName", _
                    rowColumn("COLUMN_NAME").ToString))
    nodeColumn.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                    nodeColumn.OwnerDocument, "Caption", _
                    MakeCaption(rowColumn("COLUMN_NAME").ToString)))
    Dim rows() As Data.DataRow
    nodeColumn.Attributes.Append(Utility.xmlHelpers.NewAttribute(mXMLDoc, _
                    "Ordinal", rowColumn("ORDINAL_POSITION").ToString))
    Dim def As String = rowColumn("COLUMN_DEFAULT").ToString
    If def.Length > 0 Then
       nodeColumn.Attributes.Append(Utility.xmlHelpers.NewAttribute(mXMLDoc, _
                    "Default", def))
    End If
    If rowColumn("IS_NULLABLE").ToString.ToLower = "yes" Then
       nodeColumn.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                    mXMLDoc, "AllowNulls", "true"))
    Else
       nodeColumn.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                    mXMLDoc, "AllowNulls", "false"))
    End If
```

Both the SQL data type and the .NET type are output to the XML. The SQL data type helps in debugging and in creating stored procedures. You can use the

.NET type in output code in place of the aliases that individual languages provide for them. Including the .NET type rather than a language-specific type keeps your metadata language neutral:

```
nodeColumn.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                    mXMLDoc, "SQLType", Utility.Tools.GetSQLTypeFromSQLType( _
                                    rowColumn("DATA_TYPE").ToString)))
 nodeColumn.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                    mXMLDoc, "NETType", Utility.Tools.GetNETTypeFromSQLType( _
                                    rowColumn("DATA_TYPE").ToString)))
 nodeColumn.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                    mXMLDoc, "MaxLength", _
                    rowColumn("CHARACTER_MAXIMUM_LENGTH").ToString))
 isPrimaryKey = Me.IsPrimaryKey(rowtable, rowColumn)
 nodeColumn.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                    mXMLDoc, "IsPrimaryKey", _
                    isPrimaryKey.ToString.ToLower))
 If isLookup And Not isPrimaryKey Then
    nodeColumn.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                       mXMLDoc, "UseForDesc", _
                       "true"))
 End If
```

The attributes for user-defined types (domains in the information schema views) are included only if they exist:

```
If rowColumn("DOMAIN_NAME").ToString.Length > 0 Then
    nodeColumn.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                    mXMLDoc, "UDTCatalog", _
                    rowColumn("DOMAIN_CATALOG").ToString))
    nodeColumn.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                    mXMLDoc, "UDTOwner", rowColumn("DOMAIN_SCHEMA").ToString))
    nodeColumn.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                    mXMLDoc, "UDTName", rowColumn("DOMAIN_NAME").ToString))
 End If
```

Diving into the column creation, you may think you've drilled down to the root of the process, but there are two deeper nodes. Both column privileges and column constraints appear as child nodes of the column node. This method also includes any provider-specific data, such as extended properties in SQL Server:

```
nodeColumn.AppendChild(GetPrivilegesXML(modeName & "Column", _
                       mDataSet.Tables("ColumnPrivileges").Select( _
                                    ColumnMatch(rowtable, rowColumn))))
```

```
    rows = mDataSet.Tables("ColumnConstraints").Select( _
                                    ColumnMatch(rowtable, rowColumn))
    If rows.GetLength(0) > 0 Then
       nodeColumn.AppendChild(GetCheckConstraintsXML("Check", rows))
    End If
    GetProviderSpecificXMLNodes(nodeColumn, rowtable("TABLE_CATALOG").ToString, _
                        rowtable("TABLE_SCHEMA").ToString, _
                        rowtable("TABLE_NAME").ToString, modeName, _
                        rowColumn("COLUMN_NAME").ToString)
    Return nodeColumn
End Function
```

Outputting XML for Column Privileges

The `GetPrivilegesXML` method returns nodes that contain privileges defined on a column or table. Each member of the array of DataRow objects contains a single privilege definition of grantor, grantee, and privilege. The `GetPrivilegesXML` method places this information as attributes on the returned node:

```
Protected Overridable Function GetPrivilegesXML( _
              ByVal modeName As String, _
              ByVal rowPrivileges() As Data.DataRow) _
              As Xml.XmlNode
    Dim nodePrivileges As Xml.XmlNode = Me.CreateElement(modeName &
"Privileges")
    Dim nodePrivilege As Xml.XmlNode
    For Each rowPrivilege As Data.DataRow In rowPrivileges
       nodePrivilege = Me.CreateElement(modeName & "Privilege")
       nodePrivilege.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                      mXMLDoc, "Grantor", rowPrivilege("GRANTOR").ToString))
       nodePrivilege.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                      mXMLDoc, "Grantee", rowPrivilege("GRANTEE").ToString))
       nodePrivilege.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                      mXMLDoc, "Type", rowPrivilege("PRIVILEGE_TYPE").ToString))
       nodePrivileges.AppendChild(nodePrivilege)
    Next
    Return nodePrivileges
 End Function
```

Outputting Check Constraints

As discussed in Chapter 2, the structure of constraints in the information schemas is a little more convoluted. There are four types of constraints—check constraints, primary key constraints, unique constraints, and referential constraints.

The `modeName` parameter indicates whether the constraints are table or column constraints. The `GetCheckConstraintsXML` method is called for both the table and column nodes and relies on the calling routine to create the array of `DataRows` representing each constraint. This array contains the rows from the `CONSTRAINT_COLUMN_USAGE` view (held in the DataTable named `ColumnConstraints`), which is just the name of the constraint. So the `GetCheckConstraintsXML` uses the `Select` method of the table to retrieve an array of the actual check constraint rows. This comes ultimately from the `CHECK_CONSTRAINTS` view (represented by the `CheckConstraints` DataTable). The value of each check clause is output to the XML. You might want to parse the check constraint before storing it in the metadata, but how you want to parse this will depend on your data. You can modify or override this method to provide appropriate parsing:

```
Public Overridable Function GetCheckConstraintsXML( _
                 ByVal modeName As String, _
                 ByVal rowConstraints() As Data.DataRow) _
                 As Xml.XmlNode
    Dim nodeConstraints As Xml.XmlNode
    Dim nodeConstraint As Xml.XmlNode
    Dim rowChecks() As Data.DataRow
    nodeConstraints = Me.CreateElement(modeName & "Constraints")
    For Each rowConstraint As Data.DataRow In rowConstraints
       rowChecks = mDataSet.Tables("CheckConstraints").Select( _
                      ConstraintMatch(rowConstraint))
       If rowChecks.GetLength(0) > 0 Then
          nodeConstraint = Me.CreateElement(modeName & "Constraint")
          For Each rowCheck As Data.DataRow In rowChecks
             nodeConstraint.Attributes.Append( _
                   Utility.xmlHelpers.NewAttribute(mXMLDoc, "Clause", _
                   rowCheck("CHECK_CLAUSE").ToString))
          Next
          nodeConstraints.AppendChild(nodeConstraint)
       End If
    Next
    Return nodeConstraints
 End Function
```

Outputting Table Constraints

Table constraints are a little more complicated because there are several types of table constraints and each requires one or more levels of indirection. I'll continue with the divide-and-conquer approach to complexity, so the GetTableConstraintsXML simply calls a method for each type of constraint it deals with. Naming these methods in the form Add*xxx* (instead of Get*xxx*) indicates that the parent node will be passed, anticipating that the routine will add child nodes only if appropriate. This works better in this case than returning a node because the child node may or may not logically exist and you don't want a bunch of empty elements cluttering up the XML metadata file:

```
Protected Overridable Function GetTableConstraintsXML( _
                ByVal modeName As String, _
                ByVal rowTable As Data.DataRow) _
                As Xml.XmlNode
   Dim nodeConstraints As Xml.XmlNode
   nodeConstraints = Me.CreateElement(modeName & "Constraints")
   AddPrimaryKeyXML(nodeConstraints, rowTable)
   AddRelationsXML(modeName, nodeConstraints, rowTable)
   AddUniqueConstraintXML(modeName, nodeConstraints, rowTable)
   AddTableCheckConstraintXML(nodeConstraints, rowTable)
   Return nodeConstraints
End Function
```

Outputting Primary Key Information

Looking at the AddPrimaryKeyXML, it calls GetPrimaryKeyConstraint to retrieve the primary key constraint name from the CONSTRAINT_TABLE_USAGE, held in the TableConstraints DataTable. If GetPrimaryKeyConstraint returns a row, AddPrimaryKeyXML creates a child node for the primary key. AddPrimaryKeyXML uses the primary key constraint name for a second lookup in the KeyColumns table and passes the resulting node list to the AddKeyColumns method. This lookup is into the KeyColumns table that contains columns for all the keys, including primary keys, unique keys, and foreign keys. To ensure consistency, all the key types use the AddKeyColumns method to build the XML key column nodes. I'm not including the AddKeyColumns method in this walk-through because it just loops through the columns of the key, adding the name of each column and its ordinal position:

```
 Protected Overridable Sub AddPrimaryKeyXML( _
                 ByVal nodeParent As Xml.XmlNode, _
                 ByVal rowTable As Data.DataRow)
```

```
    Dim rowPrimaryKey As Data.DataRow = GetPrimaryKeyConstraint(rowTable)
    If Not rowPrimaryKey Is Nothing Then
       Dim nodeConstraint As Xml.XmlNode = Me.CreateElement("PrimaryKey")
       AddKeyColumns(nodeParent.OwnerDocument, nodeConstraint, _
                        mDataSet.Tables("KeyColumns").Select( _
                                 ConstraintMatch(rowPrimaryKey)), "PKField")
       nodeParent.AppendChild(nodeConstraint)
    End If
 End Sub
 Protected Overridable Function GetPrimaryKeyConstraint( _
                 ByVal rowtable As Data.DataRow) _
                 As Data.DataRow
    Dim rowCOnstraints() As Data.DataRow
    rowCOnstraints = mDataSet.Tables("TableConstraints").Select( _
                    TableMatch(rowtable) & " AND CONSTRAINT_TYPE='PRIMARY KEY'")
    Select Case rowCOnstraints.GetLength(0)
       Case 0
          ' No problem, just return nothing
       Case 1
          Return rowCOnstraints(0)
       Case Else
          Throw New System.Exception( _
                    "More than one primary key found on table " & _
                    rowtable("TABLE_NAME").ToString)
    End Select
 End Function
```

The `AddUniqueConstraint` method is similar to the `AddPrimaryKey`. The difference is that there might be multiple unique keys on a single table, so `AddUniqueConstraint` retrieves an array of constraints and loops through them. Because it's similar to the `AddPrimaryKey` method, I'm not including the `AddUniqueConstraint` method.

Outputting Relations

Foreign key relations are a bit trickier because they add another level of indirection. It's useful to include both parent and child relations along with each table in the target metadata. This simplifies later templates. Although it's a little extra code now, keeping later templates simple is an important goal of metadata extraction. The child relations for the current table include all the tables that use this table in a foreign key. Finding all of the child relations for the current table requires finding the table's primary key by looking in the `REFERENTIAL_CONSTRAINTS`

view (contained in the `ReferentialConstraints` table) for all rows containing this primary key as the unique side of the constraint. Multiple tables will often contain this primary key as the unique constraint, so you're likely to get multiple rows for some tables. Looping through each of these rows, the method looks up the constraint on the nonunique (child) side of the relation. A final `Select` returns an array of key columns for the child relation's key. You can retrieve the child table name from any of the columns in this key:

```
Protected Overridable Sub AddRelationsXML( _
                 ByVal modeName As String, _
                 ByVal nodeparent As Xml.XmlNode, _
                 ByVal rowtable As Data.DataRow)
    Dim xmlDoc As Xml.XmlDocument = nodeparent.OwnerDocument
    Dim nodeRelations As Xml.XmlNode = Me.CreateElement(modeName & "Relations")
    Dim nodeRelation As Xml.XmlNode
    Dim nodeRelated As Xml.XmlNode
    Dim rowKeys() As Data.DataRow
    Dim rowConstraints() As Data.DataRow
    Dim rowRefConstraints() As Data.DataRow
    Dim selectClause As String

    ' To create child relations,
    ' -find the primary key,
    ' -then find the referential constraints with this as unique side of the key
    ' -finally, get those relations keys
    Dim rowPrimaryKey As Data.DataRow = GetPrimaryKeyConstraint(rowtable)
    If Not rowPrimaryKey Is Nothing Then
       ' Child relations are possible
       rowRefConstraints = mDataSet.Tables("ReferentialConstraints").Select( _
                       ConstraintMatch(rowPrimaryKey, "", "UNIQUE_"))
       If rowRefConstraints.GetLength(0) > 0 Then
          nodeRelation = Me.CreateElement("ChildTables")
          For Each rowRefConstraint As Data.DataRow In rowRefConstraints
             rowKeys = mDataSet.Tables("KeyColumns").Select(ConstraintMatch( _
                          rowRefConstraint))
             nodeRelated = Me.CreateElement("ChildTable", Utility.Tools.FixName( _
                          rowKeys(0)("TABLE_NAME").ToString, _
                          Me.mRemovePrefix))
             AddKeyColumns(xmlDoc, nodeRelated, rowKeys, "ChildKeyField")
             nodeRelation.AppendChild(nodeRelated)
          Next
          nodeRelations.AppendChild(nodeRelation)
       End If
    End If
```

To find all the parent or foreign keys, you need to select the FOREIGN_KEY constraints from the TableConstraints table (from CONSTRAINT_TABLE_USAGE). Looking each of these constraints up in the ReferentialConstraints table (from the REFERENTIAL_CONSTRAINTS view) gives you the corresponding parent constraint as the unique side of the foreign key relation. The unique side of this constraint points to the primary key of the foreign key table. From this constraint, you can retrieve the key columns of the constraint:

```
  ' To find all of the parent relations
   ' - Find all the foriegn key relations for this table
   ' - look up these referential constraints
   ' - get those relations keys
   rowConstraints = mDataSet.Tables("TableConstraints").Select( _
                    TableMatch(rowtable) & " AND CONSTRAINT_TYPE='FOREIGN KEY'")
   If rowConstraints.GetLength(0) > 0 Then
      nodeRelation = Me.CreateElement("ParentTables")
      For Each rowConstraint As Data.DataRow In rowConstraints
         rowRefConstraints = mDataSet.Tables("ReferentialConstraints").Select( _
                    ConstraintMatch(rowConstraint))
         For Each rowRefConstraint As Data.DataRow In rowRefConstraints
            rowKeys = mDataSet.Tables("KeyColumns").Select( _
                    ConstraintMatch(rowRefConstraint, "UNIQUE_", ""))
            nodeRelated = Me.CreateElement("ParentTable")
            nodeRelated.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                    xmlDoc, "Name", _
                    Utility.Tools.FixName(rowKeys(0)("TABLE_NAME").ToString, _
                              Me.mRemovePrefix)))
            AddKeyColumns(xmlDoc, nodeRelated, rowKeys, "ParentKeyField")
            nodeRelation.AppendChild(nodeRelated)

            rowKeys = mDataSet.Tables("KeyColumns").Select( _
                    ConstraintMatch(rowRefConstraint))
            AddKeyColumns(xmlDoc, nodeRelated, rowKeys, "ChildField")
         Next
      Next
      nodeRelations.AppendChild(nodeRelation)
   End If
   nodeparent.AppendChild(nodeRelations)
End Sub
```

Adding the table check constraints is similar to adding the column check constraints earlier, so they aren't included. You can find the similar code for column check constraints in the section "Outputting Check Constraints."

Looking Back at the ExtractMetadata Class

That's the end of the walk-through of the `ExtractMetadata` class. The `ExtractMetadata` class has 1,213 lines of code (wrapped at about 80 characters, including white-space), and this walk-through covered fewer than 400 lines. Although that's only about a third of the code, it covered a set of methods that represent the basic structure of the metadata extraction process. Much of the remaining code performs similar actions on other parts of the metadata (such as stored procedures) or provides supporting utility methods. The beauty of the `INFORMATION_SCHEMA` views is that they're consistent across all database backends. Working with them, you can continue to dissect the database definition by focusing on it at a gradually more detailed level in a stepwise approach. This divide-and-conquer approach methodically steps through a set of rows and calls another method to step through child rows. This continues down the hierarchy represented in the information schema views.

In theory, everything in this `ExtractMetadata` class will work with any backend that supports the SQL-92 information schemas as long as you override a few methods. In the next section, you'll see an example of overriding these methods for SQL Server. Remember that you can override anything in `ExtractMetadata` if it makes things work better for you. Be sure you keep backend-specific stuff in the derived class and only change the base class when you're making improvements valid for all SQL-92 backends.

Looking at the Derived SQL Server–Specific Class

The `SQLExtractMetadata` class derives from the `ExtractMetadata` class. It depends on `ExtractMetadata` to do the vast majority of the work and adds backend specific methods that fall into two categories. Some of these methods provide necessary functionality for housekeeping duties such as interacting with the connection. Other methods extend the functionality to provide information that isn't part of the SQL-92 standard. This SQL Server–specific functionality provided in this class includes support for auto-incrementing columns and extended properties. If you're using a version of SQL Server prior to SQL 2000, extended properties aren't supported and you'll have to create a version of this class that doesn't call the unsupported functions.

You'll need to derive a similar class for any other backend you need to support. To work with these classes, the backend will have to support the SQL-92 standard `INFORMATION_SCHEMA` views, or you'll have to create matching views. Because you may need to create your own version, this walk-through includes the entire `SQLExtractMetadata` class.

The option and imports statements are included for completeness. As expected, this class is derived from `ExtractMetadata`, and it supports no additional interfaces:

```
Option Strict On
Option Explicit On

Imports System
Imports System.Data
Imports System.Diagnostics

' Class Summary: Metadata extraction for SQL Server.
'                 Assumes Windows authentication

Public Class SQLExtractMetaData
   Inherits ExtractMetaData
```

There are three field declarations. These allow caching of data for both extended properties and column data to improve performance:

```
#Region "Class level declarations"
   Dim mdtExtendedColumns As New Collections.Hashtable      ' Also hold
parameters
   Dim mdtColumnInfo As New Collections.Hashtable      ' Also hold parameters
   Dim mdtStoredProcPrivileges As New Data.DataTable
#End Region
```

Exploring the Constructor

By including the `serverName` as a parameter to the constructor, I make it more difficult to instantiate an invalid object (it could still point to a nonexistent or invalid database). I prefer to require all necessary information in the constructor so instantiated objects are less likely to exist in an invalid state:

```
#Region "Constructors"
   Public Sub New(ByVal serverName As String)
      MyBase.New()
      mServerName = serverName
   End Sub
#End Region
```

Getting a Connection

The SQLExtractMetadata class provides one function for all access to the connection. Because .NET pools database connections and no transactions are involved, there's no real disadvantage to re-creating the connection whenever it's used. Using a single GetConnection method ensures that the connection string is identical for each call to the database, which is advantageous because connections pooling is based on identical connection strings:

```
Protected Function GetConnection( _
                ByVal databaseName As String) _
                As SqlClient.SqlConnection
   Return New SqlClient.SqlConnection( _
             "workstation id=" & mServerName & _
             ";packet size=4096;integrated security=SSPI" & _
             ";data source=" & mServerName & _
             ";persist security info=False;initial catalog=" & databaseName)
End Function
```

Implementing MustOverride Methods

You must override three methods in classes derived from the ExtractMetadata class. These are two overloads of the DataTableFromSQL method and the RunStoredProc method. The first is a function that returns a DataTable. It simply creates a SQLDataAdapter using the SQL string passed to the function and fills a new DataTable. The Fill method of the DataAdapter opens and closes the connection:

```
Protected Overloads Overrides Function DataTableFromSQL( _
                ByVal SQLText As String, _
                ByVal databaseName As String, _
                ByVal tablename As String) _
                As Data.DataTable
   Dim da As New SqlClient.SqlDataAdapter(SQLText, GetConnection(databaseName))
   Dim dt As New Data.DataTable
   da.Fill(dt)
   dt.TableName = tablename
   Return dt
End Function
```

The second overload takes a DataTable and fills it. The code for this is similar to the previous overload of the method:

```
Protected Overloads Overrides Sub DataTableFromSQL( _
                ByVal SQLText As String, _
                ByVal databaseName As String, _
                ByVal dt As Data.DataTable)
   Dim da As New SqlClient.SqlDataAdapter(SQLText, GetConnection(databaseName))
   da.Fill(dt)
End Sub
```

The `RunStoredProc` method runs stored procedures to determine the returned recordsets. Because you don't want any side effects appearing in your database, and I can't guarantee that your select stored procedures don't also perform other actions or that you've supplied the correct patterns for these stored procedure types, I wrap the `Fill` method that executes the stored procedure in a transaction and always rolls back the result. This will slow down the extraction process a bit but provides an important margin of safety:

```
Protected Overrides Function RunStoredProc( _
                   ByVal spname As String, _
                   ByVal nodeParams As System.Xml.XmlNode, _
                   ByVal databaseName As String) _
                   As System.Data.DataSet
   Dim ds As New Data.DataSet
   Dim da As New SqlClient.SqlDataAdapter
   Dim cmd As New SqlClient.SqlCommand
   Dim def As Object
   Dim name As String
   Dim tr As SqlClient.SqlTransaction

   cmd.Connection = GetConnection(databaseName)
   cmd.CommandType = CommandType.StoredProcedure
   cmd.CommandText = spname
   For Each node As Xml.XmlNode In nodeParams
      Select Case CStr(Utility.Tools.GetAttributeOrEmpty(node, "Type")).ToLower
         Case "char", "varchar", "nchar", "nvarchar"
            def = ""
         Case "uniqueidentifier"
            def = DBNull.Value
         Case "datetime"
            def = "1/1/1873"
         Case Else
            def = 0
```

```
        End Select
        name = Utility.Tools.GetAttributeOrEmpty(node, "Name")
        cmd.Parameters.Add(New SqlClient.SqlParameter(name, def))
    Next
    da.SelectCommand = cmd
    cmd.Connection.Open()
    cmd.CommandTimeout = 5
    tr = cmd.Connection.BeginTransaction(IsolationLevel.Serializable)
    cmd.Transaction = tr
    da.Fill(ds)
    tr.Rollback()
    cmd.Connection.Close()
    Return ds
End Function
```

Supporting Extended Properties

Overriding the remaining methods of the `ExtractMetadata` class is optional because the base class provides these methods, even though in some cases they're empty. The SQL Server implementation overrides these methods to provide access to extended properties and information about auto-incrementing columns. This class overrides the `FillMetaDataSet` method to add DataTables with extended properties for the database and its objects (tables, stored procedures, and so on). Building these methods at the start of the process allows the data to be cached and reused. Before the method adds these tables, it calls the base class `FillMetaDataSet` method to fill the DataSet with standard information from the information schema views. This class later overrides the three `GetProviderSpecificXMLNode` methods to output the extended property information.

The `fn_listextendedproperty` function of SQL Server 2000 retrieves the extended properties values from any database structural element. The parameters passed determine which object's extended properties are retrieved. For example, passing all nulls returns information on the database. Chapter 2 discusses the `fn_listextendedproperty` function further. You can see what each call to `fn_listextendedproperty` returns if you note the new table name (such as `ExtDatabase`) in the `FillMetaDataSet` method:

```
Protected Overrides Sub FillMetaDataSet(ByVal name As String)
    mdtExtendedColumns = New Collections.Hashtable
    MyBase.FillMetaDataSet(name)
    mdataset.Tables.Add(DataTableFromSQL( _
                "SELECT * FROM ::fn_listextendedproperty(null, null, " & _
                "null, null, null, null, null)", name, "ExtDatabase"))
```

```
    mdataset.Tables.Add(DataTableFromSQL( _
                "SELECT * FROM ::fn_listextendedproperty(null, 'User', " & _
                "'dbo', 'Table', null, null, null)", name, "ExtTable"))
    mdataset.Tables.Add(DataTableFromSQL( _
                "SELECT * FROM ::fn_listextendedproperty(null, 'User', " & _
                "'dbo', 'Procedure', null, null, null)", name, "ExtStoredProc"))
    mdataset.Tables.Add(DataTableFromSQL( _
                "SELECT * FROM ::fn_listextendedproperty(null, 'User', " & _
                "'dbo', 'View', null, null, null)", name, "ExtView"))
    mdataset.Tables.Add(DataTableFromSQL( _
                "SELECT * FROM ::fn_listextendedproperty(null, 'User', " & _
                "'dbo', 'Function', null, null, null)", name, "ExtFunction"))
    mdtStoredProcPrivileges = DataTableFromSQL( _
               "sp_helprotect", name, "StoredProcPrivileges")
End Sub
```

NOTE This method doesn't retrieve extended properties for columns and parameters because they're retrieved when they're first needed.

The base class provides three different overloads to the `GetProviderSpecificXMLNodes` method, all of which `SQLExtractMetadata` overloads. These provide access at the database, object (table, stored procedure, and so on), or element (column or parameter) level. At all levels, the `SQLExtractMetadata` classes retrieve extended properties from SQL Server. At the column level, it also retrieves the auto-increment status of the columns.

The database-level `GetProviderSpecificXMLNodes` demonstrates how to use extended properties. The derived `FillMetaDataSet` (shown previously) already filled the `ExtDatabase` DataTable. All rows in the `ExtDatabase` table refer to the current database, so you don't need to filter the records. For each extended property, the method creates an `ExtendedProperty` XML element along with a `Name` and `Value` attribute:

```
Protected Overloads Overrides Function GetProviderSpecificXMLNodes( _
                ByVal node As System.Xml.XmlNode, _
                ByVal databaseName As String, _
                ByVal schemaName As String) _
                As System.Xml.XmlNode
```

```
    Dim dt As DataTable = mdataset.Tables("ExtDatabase")
    Dim nodeProps As Xml.XmlNode
    Dim nodeProp As Xml.XmlNode
    Dim xmlDoc As Xml.XmlDocument = node.OwnerDocument
    If dt.Rows.Count > 0 Then
        nodeProps = node.AppendChild(Me.CreateElement("ExtendedProperties"))
        For Each row As Data.DataRow In dt.Rows
            nodeProp = nodeProps.AppendChild(Me.CreateElement( _
                   "ExtendedProperty", row("name").ToString))
            nodeProp.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                   mxmlDoc, "Value", row("value").ToString))
        Next
    End If

End Function
```

Retrieving extended properties at the table level is similar. I'm using the word *table* for simplicity, but this actually retrieves provider specific data for tables, stored procedures, and views. The `modeName` contains the type of the object—table, stored procedure, and so on. There are a few differences in retrieving the data for a stored procedure, so the function calls a subroutine for the right type:

```
Protected Overloads Overrides Function GetProviderSpecificXMLNodes( _
                ByVal node As System.Xml.XmlNode, _
                ByVal databaseName As String, _
                ByVal schemaName As String, _
                ByVal tableName As String, _
                ByVal modeName As String) _
                As System.Xml.XmlNode
    If modeName = "StoredProc" Then
        GetExtraStoredProcInfo(node, databaseName, schemaName, tableName, modeName)
    Else
        GetExtraTableInfo(node, databaseName, schemaName, tableName, modeName)
    End If
End Function
```

The `tableName` parameter refers to a database object, which could be any of the expected types. Each one of these object types is loaded into a different DataTable. The method finds the correct DataTable based on its name, determined by concatenating `ext` to the `modeName`, which will be `extTable`, `extFunction`, or `extView`. `Select` filters the data rows because the DataTable contains the

extended properties for all objects, such as all database tables. The method loops through the array of rows creating an `ExtendedProperty` element with the name and value of each extended property:

```
Protected Overridable Function GetExtraTableInfo( _
            ByVal node As System.Xml.XmlNode, _
            ByVal databaseName As String, _
            ByVal schemaName As String, _
            ByVal tableName As String, _
            ByVal modeName As String) _
            As System.Xml.XmlNode

   Dim extTableName As String = "Ext" & node.LocalName
   Dim dt As Data.DataTable = mdataset.Tables(extTableName)
   Dim nodeProps As Xml.XmlNode
   Dim nodeProp As Xml.XmlNode
   Dim xmlDoc As Xml.XmlDocument = node.OwnerDocument
   Dim rows() As Data.DataRow = dt.Select("objname='" & tableName & "'")
   If dt.Rows.Count > 0 Then
      nodeProps = node.AppendChild(Me.CreateElement("ExtendedProperties"))
      For Each row As Data.DataRow In rows
         nodeProp = nodeProps.AppendChild(Me.CreateElement( _
               "ExtendedProperty", row("name").ToString))
         nodeProp.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
               mxmlDoc, "Value", row("value").ToString))
      Next
   End If
End Function
```

Currently the `GetExtraStoredProcInfo` method retrieves privileges and doesn't also retrieve extended properties. You could change the `GetExtraStoredProcInfo` to also call `GetExtraTableInfo` if you used extended properties on stored procedures. For now, it just retrieves the privileges information from the database:

```
Protected Overridable Function GetExtraStoredProcInfo( _
            ByVal node As System.Xml.XmlNode, _
            ByVal databaseName As String, _
            ByVal schemaName As String, _
            ByVal tableName As String, _
            ByVal modeName As String) _
            As System.Xml.XmlNode
```

```
    ' Get Stored Proc Privileges
    Dim rowPrivileges As Data.DataRow()
    rowPrivileges = mdtStoredProcPrivileges.Select("Object = '" & tableName & "'")
    Dim nodePrivileges As Xml.XmlNode = Me.CreateElement(modeName & "Privileges")
    Dim nodePrivilege As Xml.XmlNode
    For Each rowPrivilege As Data.DataRow In rowPrivileges
        nodePrivilege = Me.CreateElement(modeName & "Privilege")
        nodePrivilege.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                        mXMLDoc, "Grantor", rowPrivilege("Grantor").ToString))
        nodePrivilege.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                        mXMLDoc, "Grantee", rowPrivilege("Grantee").ToString))
        nodePrivilege.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                        mXMLDoc, "Type", rowPrivilege("Action").ToString))
        nodePrivilege.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                       mXMLDoc, "ProtectType", rowPrivilege("ProtectType").ToString))
        nodePrivileges.AppendChild(nodePrivilege)
    Next
    node.AppendChild(nodePrivileges)

End Function
```

Although similar, retrieving extended properties for columns and parameters is a little more complex. Again, the word *table*, such as `tableName`, refers to the object under consideration, which may be a function, stored procedure, table, or view. Similarly, the word *column* refers to the level-two element, which in English means it could be either a column or a parameter.

This is where the cached data in the class's protected fields come into play. When this method is called for the first column in the table (or parameter in the stored procedure), it doesn't find information in the cache. It retrieves the information from the database and stores it in the cache. This happens twice, once for the auto-incrementing information and once for the extended properties. Later columns or parameters use the cached data. The `FillMetaDataSet` method clears the cache when it reloads data. Once the DataTable is available, you can find the rows relevant to this column or parameter. A conditional statement within a loop selects the relevant records for the auto-incrementing flag. A `Select` statement filters rows for extended properties:

```
Protected Overloads Overrides Function GetProviderSpecificXMLNodes( _
                ByVal node As System.Xml.XmlNode, _
                ByVal databaseName As String, _
                ByVal schemaName As String, _
```

```
                ByVal tableName As String, _
                ByVal modeName As String, _
                ByVal columnName As String) _
                As System.Xml.XmlNode
   Dim dt As Data.DataTable
   Dim nodeProps As Xml.XmlNode
   Dim nodeProp As Xml.XmlNode
   Dim xmlDoc As Xml.XmlDocument = node.OwnerDocument
   Dim rows() As Data.DataRow
   Dim extSQL As String
```

For auto-incrementing information, check the cache and fill the DataTable if needed:

```
If mdtColumnInfo.Contains(tableName) Then
     dt = CType(mdtColumnInfo(tableName), Data.DataTable)
  Else
     dt = DataTableFromSQL("sp_columns [" & tableName & "]", databaseName, _
              "columns")
     mdtColumnInfo.Add(tableName, dt)
  End If
```

Loop through the rows, and output auto-incrementing column information:

```
For Each row As Data.DataRow In dt.Rows
     If CStr(row("COLUMN_NAME")).tolower = columnName.ToLower Then
        If CStr(row("TYPE_NAME")).IndexOf("identity") >= 0 Then
           node.Attributes.Append( _
                   Utility.xmlHelpers.NewAttribute( _
                      node.OwnerDocument, "IsAutoIncrement", "true"))
        End If
     End If
  Next
```

For extended properties, check the cache and fill the DataSet if needed. The SQL select statement depends on the type of object being retrieved, which is held in the `modeName`:

```
If mdtExtendedColumns.Contains(tableName) Then
    dt = CType(mdtExtendedColumns(tableName), Data.DataTable)
 Else
```

```
    Select Case modeName
       Case "Table"
          extSQL = "SELECT * FROM ::fn_listextendedproperty(null, " & _
                   "'User', 'dbo', 'Table', '" & tableName & _
                   "', 'Column', null)"
       Case "View"
          extSQL = "SELECT * FROM ::fn_listextendedproperty(null, " & _
                   "'User', 'dbo', 'View', '" & tableName & _
                   "', 'Column', null)"
       Case "StoredProc"
          extSQL = "SELECT * FROM ::fn_listextendedproperty(null, " & _
                   "'User','dbo', 'Procedure', '" & tableName & _
                   "', 'Parameter', null)"
       Case "Function"
          extSQL = "SELECT * FROM ::fn_listextendedproperty(null, " & _
                   "'User', 'dbo', 'Function', '" & tableName & _
                   "', 'Parameter', null)"
    End Select
    dt = DataTableFromSQL(extSQL, databaseName, "ExtProp")
    mdtExtendedColumns.Add(tableName, dt)
 End If
```

Finally, loop through the rows outputting extended property information for the column.

```
  rows = dt.Select("objname='" & columnName & "'")
      If rows.GetLength(0) > 0 Then
         For Each row As Data.DataRow In rows
            node.Attributes.Append( _
                      Utility.xmlHelpers.NewAttribute( _
                            node.OwnerDocument, _
                            row("name").ToString, row("value").ToString))
         Next
      End If
   End Function
```

Ignoring Specific Tables

Overriding one last method allows you to exclude specific tables. You could also allow them in the metadata and clean them up during the ORM transformation

process. Building SQL Server data diagrams creates a dtproperties table that isn't marked as a system table. That isn't part of the application's metadata and is likely to confuse template programmers, so I skip it in the metadata extraction step. Overriding the GetTables method allows you to remove the dtproperties table before metadata processing starts:

```
Protected Overrides Function GetTables( _
                      ByVal databaseName As String) _
                      As System.Data.DataTable
   Dim dta As Data.DataTable
   dta = MyBase.GetTables(databaseName)
   ' This is an ugly hack because the data diagram table dtProperties
   ' is marked by SQL as a user table
   Dim rows() As Data.DataRow
   rows = dta.Select("TABLE_NAME = 'dtproperties'")
   If rows.GetLength(0) > 0 Then
      dta.Rows.Remove(rows(0))
   End If
   Return dta
End Function
```

Looking Back at the SQLExtractMetadata Class

Although there's a lot to look at in the base class, it's designed to be easily extensible, as shown by the SQLMetadataExtraction class. You can create a derived class to extract metadata from another backend or modify this class as needed. This class must override the DataTableFromSQL and RunStoredProc.

APPENDIX C

Details of the Code Generation Harness

Chapter 3 showed how you use a code generation harness. This appendix shows you how that harness application works. The harness has two purposes—it allows you to edit the Extensible Markup Language (XML) generation script and executes this script displaying the results. Together these offer the one-click code generation that's one of the key principles of effective code generation. Figure C-1 shows how two inputs—the XML file and the corresponding XML Schema Definition (XSD)—are combined for both to edit and to execute the script through the script engine.

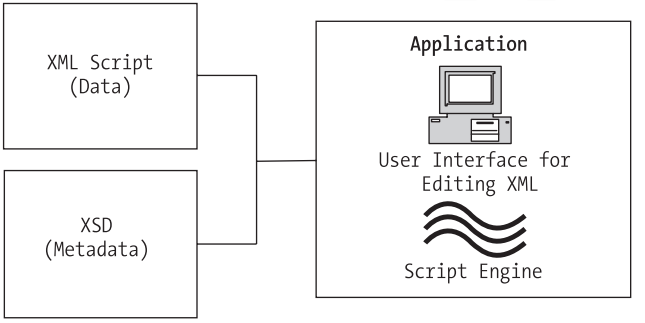


Figure C-1. An overview of the code generation harness

The code generation harness is an interesting application because it uses dynamic data-driven techniques. Although there are certainly some cases where dynamic data-driven techniques are useful, appropriate scenarios are fairly limited because of some serious disadvantages as follows:

- Mediocre performance.
- No strong typing possible.

- Requires an understanding of uncommon techniques (reflection).
- Awkward incorporation of handcrafted code.
- Quite difficult to test and debug.

These are also three significant benefits of dynamic data-driven techniques as follows:

- No recompile/redeploy on metadata changes.
- Application rarely or never needs modification once built.
- A single application runs many different scenarios.

The code generation harness is an unusual application that's immune to the disadvantages of dynamic data-driven applications and can take full advantage of the benefits. The harness couldn't care less what you're doing except that you give it an XSD that defines the XML you'll edit. You've got to understand reflection anyway to work on the harness because of its use in processing specified methods, particularly for brute-force and CodeDOM generation. Testing is relatively limited because you're doing a small number of things—just collecting data from the user and running the generation processes.

I'll warn you that this isn't the final word in code generation harnesses. My goal was to offer enough complexity to be "real world" but limit the complexity wherever I could to keep the scope of the book focused. You might want to put more work into how errors are logged and into extensibility, for example. Although you may evolve the code generation harness, consider how those changes will affect existing generated applications and possibly maintain a static copy within each project, at least as it moves into deployment. Within a single development project, it's important that the one-click generation next year be exactly like one-click generation the last time the project was fully tested. Subtle changes to the harness or the tools it uses can result in bugs *in your application* that you aren't going to catch unless you do full-scale testing when you later make even the smallest change and regenerate. Alternatively, you could use your source control tool to mark the version used in generating a particular application.

This chapter is split into two parts. The first section covers the data-driven User Interface (UI) that edits XML documents. The second part discusses details of the generation process.

Building a Dynamic Data-Driven Application

As a dynamic data-driven application, the code generation harness uses an XSD to determine what to display. From the perspective of the editor, you're editing elements and complex types described by the XSD. Figure C-1 shows that the XSD and XML script combine to drive editing and running the script. Looking back at Figure 3-1, you'll see that there's a TreeView on the left and a series of data-entry controls, many grouped in frames. Each frame represents a complex type. Appendix A shows the XSD for the harness as an example and includes a walk-through of its elements. Figure C-2 shows the UML diagram of the code generation harness. Most of these classes are performing WinForms duties.

NOTE There are several thousand lines of code that run the UI of this application. About 350 of these lines are in the derived classes in the `GenerationHarness` project, mostly performing housekeeping tasks. There are about 2,200 lines of code in the base classes of the `WinSupport` project. The remaining code performs miscellaneous functions.

Looking at SimpleMDIParentBase and HarnessForm

When you start the application, the Multiple Document Interface (MDI) parent form named `HarnessForm` opens. This form handles the menu options, particularly opening and closing files.

If you open an existing XML script file or create a new file, `HarnessForm` opens `GenerationForm` as its MDI child and loads the file. I'll talk in terms of XML script files, but the harness is based on a generic view of XSD files and can edit many XML files, as long as you provide the right XSD.

`HarnessForm` is derived from `SimpleMDIParentBase`, which does almost all of the work. To determine the child form to load, `SimpleMDIParentBase` calls `FormForFileType` in `SimpleMDIParentBase`. `HarnessForm` overrides this method to return a new `GenerationForm`.[1] `HarnessForm` also overrides the `NewFile` method to provide specifics for opening script files.

1. This design specifically allows a single derived form to open many different types of files within the MDI parent. The type of form is based on the file type.

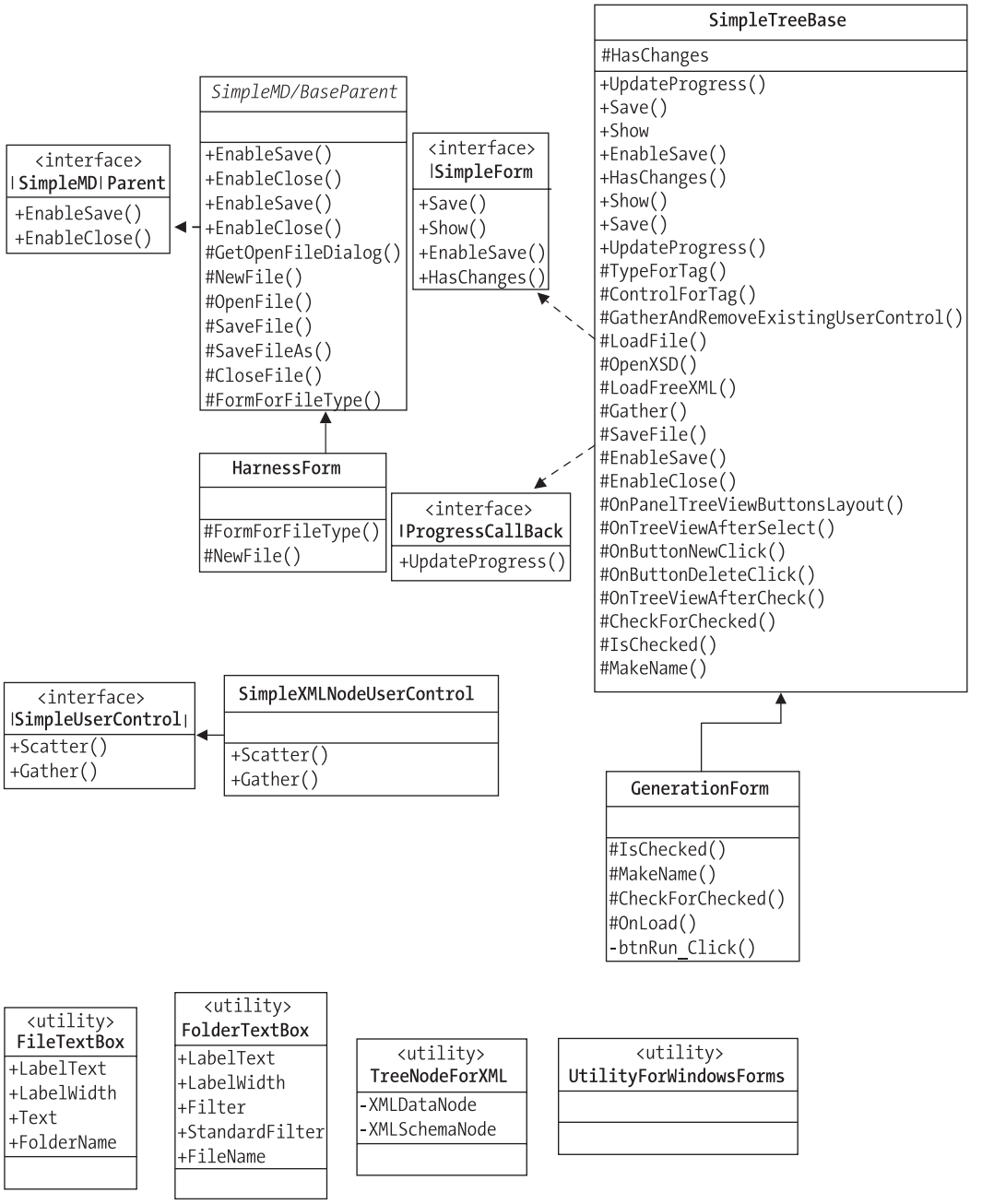


Figure C-2. A UML static model of the UI of the code generation harness

Exploring SimpleTreeBase

The SimpleTreeBase class does almost all the work of displaying the child form. GenerationForm derives from it and overrides methods that access the Checked attribute because this attribute is contained on the Standard element, not the main element as expected by the base class. It also overrides the MakeName method to provide specific behavior. For each node, the SimpleTreeBase instantiates a SimpleXMLNodeUserControl to display in the right portion of the screen (refer to Figure 3-1).

Loading the XML

The most interesting part of SimpleTreeBase is how it loads and manages the XML data. The data is loaded as an XML document, which means the data is contained in a series of XMLNode objects. Each TreeView node stores a reference to the XMLNode it represents as well as the node for the corresponding XMLNode in the XSD schema. Because there are two pieces of information to store, the Tag property doesn't work well. So, I derived the TreeNodeForXML class from the Windows.Forms.TreeNode class. The derived class adds properties to hold the extra information. This also allows an easy avenue for later extensibility.

NOTE .NET includes an XML.Schema namespace with classes to manage schemas. I'm not using this and instead am treating the XSD as an XML document. The extra XML.Schema namespace doesn't offer much benefit in the specific context of using the XSD to drive the dynamic data-driven UI. Because the XML.Schema namespace has about 70 classes and treats the XSD in a unique way (consistent with the W3C XSD specification rather than treating the XSD as an XML document), using these classes would have brought in a ton of new concepts that have no relevance to code generation. If you're building schemas from scratch in code, by all means explore that namespace.

Looking at the Show Method

To initiate loading the XML, the MDI parent code calls one of the two overloaded Show methods of the SimpleTreeBase form. These methods call the protected LoadFile method:

```
Public Overloads Sub Show( _
                ByVal xmlFileName As String) _
                Implements WinFormSupport.ISimpleForm.Show
```

```
    LoadFile(xmlFileName)
    Me.Text = "Generation Script - " & IO.Path.GetFileName(xmlFileName)
    HasChanges = False
    mCurrentTreeNode = TreeView.Nodes(0)
End Sub

Public Overloads Sub Show( _
                ByVal xmlDoc As Xml.XmlDocument, _
                ByVal path As String) _
                Implements WinFormSupport.ISimpleForm.Show
    LoadFile(xmlDoc, path)
End Sub
```

Looking at LoadFile

The `LoadFile` method has two overloads. The first `Show` method calls the first `LoadFile` overload. This opens the XML document with the validating reader, assigns the filename to a class field, and calls the second `LoadFile` overload. Using the validating reader ensures that the XML matches the XSD schema used to display it. The validating reader determines the XSD to validate from the `xsi:schemaLocation` attribute in the XML file. This location should match the XSD opened for editing—explained next. The `mXMLStyle` field allows later extensibility to other styles of editing:

```
Protected Overridable Sub LoadFile( _
                ByVal xmlFileName As String)
    Dim xmlDoc As New Xml.XmlDocument
    Dim textReader As Xml.XmlTextReader
    Dim validReader As Xml.XmlValidatingReader
    mXSDDoc = Nothing
    Try
        mXMLFileName = xmlFileName
        textReader = New Xml.XmlTextReader(xmlFileName)
        validReader = New Xml.XmlValidatingReader(textReader)
        validReader.ValidationType = Xml.ValidationType.Schema
        Select Case mXMLStyle
            Case XMLStyle.Free
                xmlDoc.Load(validReader)
                LoadFile(xmlDoc, IO.Path.GetDirectoryName(xmlFileName))
            Case Else
                Return
        End Select
```

```
    Catch ex As System.Exception
        Throw
    Finally
        textReader.Close()
    End Try
End Sub
```

NOTE An invalid `schemaLocation` path is the most common error when opening a harness script.

After validating the XML, this method calls the second `LoadFile` overload. The second overload takes an XML document and a string pathname. This overload assumes you've already validated the XML. If you call this second overload and your XML doesn't match the XSD, you'll either run into errors or potentially corrupt your XML file. This method finds the XSD file and calls a method to load the XSD as an XML document and also manages some namespaces:

```
Protected Overridable Sub LoadFile( _
                ByVal xmlDoc As Xml.XmlDocument, _
                ByVal xFilepath As String, _
                ByVal basePath As String, _
                ByVal path As String)
    Dim node As Xml.XmlNode
    Dim fileName As String
    Dim fullFileName As String
    Dim xfileFullPath As String
    Dim nsmgrXSD As New Xml.XmlNamespaceManager(New Xml.NameTable)
    nsmgrXSD.AddNamespace("xs", "http://www.w3.org/2001/XMLSchema")
    nsmgrXSD.AddNamespace("kd", "http://kadgen.com/KADGenDrivingx.xsd")
    Dim nsmgrXML As Xml.XmlNamespaceManager = _
                Utility.xmlHelpers.BuildNamespaceManagerForDoc(xmlDoc)
    mXMLDoc = xmlDoc
    node = mXMLDoc.SelectSingleNode("kg:GenerationScript", nsmgrXML)
    If mXMLDoc.ChildNodes.Count > 1 Then
```

NOTE The `http://kadgen.com/KADGenDrivingx.xsd` namespace is one I use to add extra information to schemas. All use of this namespace is optional, and if you ignore it in your schema, this application should still run fine.

NOTE The `BuildNamespaceManager` method finds all namespace/prefix combinations in the document and builds a corresponding namespace manager. This isn't a particularly fast operation, but it's necessary to establish the correct namespace generically.

I'm loading the XSD using the `schemaLocation` property to ensure the schema used when validating the XML document is the one used to display it. As discussed in Appendix A, the `schemaLocation` is a space-delimited list. In this context, only the first file location is used.[2] The harness editing mechanism doesn't support merging of multiple schemas:

```
If Not node.Attributes.GetNamedItem("schemaLocation", _
                "http://www.w3.org/2001/XMLSchema-instance") _
                Is Nothing Then
   Dim temp() As String
   temp = node.Attributes.GetNamedItem("schemaLocation", _
                "http://www.w3.org/2001/XMLSchema-instance"). _
                      Value.Split(" "c)
   fullFileName = temp(1)
End If
```

TIP The `LookupNamespace` method of the `XMLNamespaceManager` class uses an atomized string to enhance performance, so you can't just pass a literal string to it. This is poorly documented in the Help.

Once you retrieve the filename, the name of the XSD is passed to the `OpenXSD` method. This method opens the XSD as an XML document stored in a class field. This allows all methods and properties of the class to access both the XML and the XSD document and prepares the class to load the data in the `LoadFreeXML` method:

```
      If Not fullFileName Is Nothing Then
         If IO.File.Exists(fullFileName) Then
            OpenXSD(fullFileName)
```

2. You could enhance this to pick the correct schema at any location in the `schemaLocation` string; however, the current version of the tool will have problems with XML documents that are described by multiple schemas anyway.

```
            Else
               xfileFullPath = Utility.Tools.FixPath(xFilepath, _
                     basePath, Nothing, Nothing)
               fullFileName = IO.Path.Combine(xfileFullPath, _
                     IO.Path.GetFileName(fullFileName))
               If IO.File.Exists(fullFileName) Then
                  OpenXSD(fullFileName)
               End If
            End If
         End If
         LoadFreeXML(nsmgrXSD)
      End If
End Sub
```

I'm not including the `OpenXSD` method because it's simple and just opens the file as an XML document.

Loading Data

The `LoadFreeXML`[3] method does three things; it loads the XML data into the tree, loads XSD information into the TreeView's menu so you can add new nodes, and shows the form:

```
Protected Overridable Sub LoadFreeXML():
   LoadFreeXMLData()
   LoadTreeMenuXSD ()
   Me.Show()
End Sub
```

The `LoadFreeXMLData` method finds the root element of the document. Searching the child nodes allows an unlimited number of XML comments at the head of the file. `LoadFreeXMLData` passes the root element to the recursive function `LoadFreeXMLDataNode`:

```
Protected Sub LoadFreeXMLData(ByVal nsmgr As Xml.XmlNamespaceManager)
   Dim node As Xml.XmlNode
   mTreeNodeLoadDepth = 0
   For Each node In mXMLDoc.ChildNodes
      If node.NodeType = Xml.XmlNodeType.Element Then
```

3. This somewhat unusual method name is because I had anticipated needing a more specific approach that could not be accomplished with the generic approach presented here. That need did not arise, but the artifact of "Free" in this method name remains.

```
            Exit For
        End If
    Next
    If Not node Is Nothing Then
        LoadFreeXMLDataNode(node, TreeView.Nodes, nsmgr)
    End If
End Sub
```

The LoadFreeXMLDataNode method uses the MakeTreeNode method to create a new TreenodeForXML for the base node. Because TreeNodeForXML is derived from the Windows.Forms.TreeNode class, you can insert it into the TreeView as though it was a TreeNode. Unless the kd:NoChildrenInTree attribute is set to true, the method recursively calls itself to build a nested tree:

```
Protected Sub LoadFreeXMLDataNode( _
                    ByVal node As Xml.XmlNode, _
                    ByVal treenodelist As Windows.Forms.TreeNodeCollection, _
                    ByVal nsmgr As Xml.XmlNamespaceManager)
    Dim treenode As TreeNodeForXML
    Me.mTreeNodeLoadDepth += 1
    treenode = MakeTreeNode(treenodelist, node, mTreeNodeLoadDepth <= 2, nsmgr)
    Dim xmlSchemaNode As Xml.XmlNode = _
                Utility.xmlHelpers.GetSchemaForNode( _
                      node.LocalName, mXSDDoc)
    If Utility.Tools.GetAttributeOrEmpty(xmlSchemaNode, "", "kd:NoChildrenInTree", _
                nsmgr) <> "true" Then
        For Each childNode As Xml.XmlNode In node.ChildNodes
            If childNode.NodeType = Xml.XmlNodeType.Element Then
                LoadFreeXMLDataNode(childNode, treenode.Nodes, nsmgr)
            End If
        Next
    End If
    Me.mTreeNodeLoadDepth -= 1
End Sub
```

Creating the TreeNode

The MakeTreeNode method creates the TreeNodeForXML and adds this to the collection. The MakeName function inserts spaces before capital letters, which almost always makes a nice human-readable name. The TreeNodeForXML calls MarkChecked to display checks. MarkChecked is called at several points, including when you're

loading the data. The important method call here is the `GetSchemaForNode` method, discussed in the section "Loading the Schema." The `CanAdd` property indicates whether child nodes can later be added to this TreeNode:

```
Private Function MakeTreeNode( _
               ByVal nodeCollection As Windows.Forms.TreeNodeCollection, _
               ByVal node As Xml.XmlNode, _
               ByVal canAdd As Boolean, _
               ByVal nsmgr As Xml.XmlNamespaceManager) _
               As TreeNodeForXML
   Dim xmlSchemaNode As Xml.XmlNode = _
                   Utility.xmlHelpers.GetSchemaForNode(node.LocalName, _
                   mXSDDoc)
   Dim treenode As New TreeNodeForXML(MakeName(node), node, xmlSchemaNode)
   nodeCollection.Add(treenode)
   MarkChecked(treenode, xmlSchemaNode, node, nsmgr)
   treenode.CanAdd = canAdd
   Return treenode
End Function
```

The `MarkChecked` method displays a checkbox. If you use the checked features of the TreeView, you'll have a checkbox displayed beside every entry. In the generation harness, it'd be confusing to display a checkbox beside `FilePaths` or `Standard` blocks. You can emulate checkboxes with icons to display them only on appropriate nodes. The `CanChecked` and `IsChecked` methods indicate what icons to display, and you can override these methods in derived forms. `GenerationForm` overrides `IsChecked` to retrieve the `Checked` attribute from the `Standard` attribute subset:

```
   Private Sub MarkChecked( _
              ByVal treenode As Windows.Forms.TreeNode, _
              ByVal xmlSchemaNode As Xml.XmlNode, _
              ByVal xmlNode As Xml.XmlNode, _
              ByVal nsmgr As Xml.XmlNamespaceManager)
      Const noCheckImage As Int32 = 0
      Const checkImage As Int32 = 1
      Const uncheckImage As Int32 = 2
      If Not Me.CanCheck(xmlSchemaNode, nsmgr) Then
         treenode.ImageIndex = noCheckImage
         treenode.SelectedImageIndex = noCheckImage
      ElseIf Me.IsChecked(xmlNode) Then
         treenode.ImageIndex = checkImage
         treenode.SelectedImageIndex = checkImage
```

```
        Else
            treenode.ImageIndex = uncheckImage
            treenode.SelectedImageIndex = uncheckImage
        End If
    End Sub

#End Region
```

Loading the Schema

Because the XSD document is loaded as an XML document, you use XML Document Object Model (DOM) mechanisms to retrieve data. The `GetSchemaForNode` method uses the `SelectSingleNode` method to retrieve the schema element for the specific element in the XML data document. This method retrieves an element in the XSD corresponding to the element in the data document:

```
Public Shared Function GetSchemaForNode( _
                ByVal nodeName As String, _
                ByVal xsdDoc As Xml.XmlDocument) _
                As Xml.XmlNode
    If Not xsdDoc Is Nothing Then
        Dim namespaceManager As New Xml.XmlNamespaceManager( _
                        New System.Xml.NameTable)
        namespaceManager.AddNamespace("xs", "http://www.w3.org/2001/XMLSchema")
        Return xsdDoc.SelectSingleNode("//xs:element[@name='" & nodeName & "']", _
                        namespaceManager)
    End If
End Function
```

NOTE Two independent XML documents are used to create the display. One is the data document, and the other XML document happens to be an XSD.

The `LoadTreeMenuXSD` method loads XSD information needed to later create context-dependent menus for adding child nodes. These menus need to list only children valid for the user to add to the current node. To provide this feature, the `LoadTreeMenuXSD` fills a class field with the root menu options and a hash table with the child nodes menu options. Each of these entries contains an array with

the menu options for a single unique node type. The current XSD supports different menus for the `GenerationScript`, `Section`, and `FilePaths` nodes. The array returned by the base node is available within `LoadTreeMenuXSD` and is assigned to the `mBaseMenus` field.

Popup child menus don't behave correctly if they don't contain at least one entry, so I add the dummy menu item Fred:

```
Protected Sub LoadTreeMenuXSD(ByVal nsmgr As Xml.XmlNamespaceManager)
    Dim baseNode As Xml.XmlNode
    If Not mXSDDoc Is Nothing Then
       mChildAutos = New Collections.Hashtable
       baseNode = mXSDDoc.ChildNodes(1).ChildNodes(0)
       mBaseMenus = GetMenuInfo(baseNode, nsmgr)
       'GetAutoInfo(baseNode, nsmgr)
       'add a dummy entry so the menu will open
       mnuTreeViewAdd.MenuItems.Add("Fred")
    End If
End Sub
```

`LoadTreeMenuXSD` uses the recursive `GetMenuInfo` method. This method looks for elements within the complex type that have child elements. This allows you to ignore the terminal leaf elements in the XSD. For each node, the method adds a structure with the required information to the array of menu options for this node type and then adds these child menu arrays to the `mChildMenus` field of the class:

```
Private Function GetMenuInfo( _
             ByVal node As Xml.XmlNode, _
             ByVal nsmgr As Xml.XmlNamespaceManager) _
             As menuStuff()
   Dim childNodeList As Xml.XmlNodeList
   Dim childNode As Xml.XmlNode
   Dim menuStuffItems() As menuStuff
   Dim menuStuff As menuStuff
   Dim autoStuff() As Xml.XmlNode
   ' The following selects elements with at least one child
   childNodeList = node.SelectNodes( _
            "xs:complexType/*/xs:element[*]", nsmgr)
   If childNodeList.Count > 0 Then
      ReDim menuStuffItems(childNodeList.Count - 1)
      For i As Int32 = 0 To menuStuffItems.GetUpperBound(0)
         childNode = childNodeList(i)
         If Utility.Tools.GetAttributeOrEmpty(childNode, "", "kd:auto", nsmgr) _
                      <> "true" Then
```

```
            menuStuff = New menuStuff(childNode, Utility.Tools.SpaceAtCaps( _
                      Utility.Tools.GetAttributeOrEmpty(childNode, "name")))
            menuStuffItems(i) = menuStuff
            mChildMenus.Add(Utility.Tools.GetAttributeOrEmpty(childNode, "name"), _
                         GetMenuInfo(childNode, nsmgr))
         End If
      Next
```

The `mChildAutos` collection is also built from the schema. This hash table contains the child elements that are automatically created when a new parent element is later added by the user:

```
      childNodeList = node.SelectNodes( _
               "xs:complexType/*/xs:element[@kd:auto='true']", nsmgr)
      If childNodeList.Count > 0 Then
         ReDim autoStuff(childNodeList.Count - 1)
         For i As Int32 = 0 To childNodeList.Count - 1
            autoStuff(i) = childNodeList(i)
         Next
         mChildAutos.Add(Utility.Tools.GetAttributeOrEmpty(node, "name"), _
                            autoStuff)
      End If
   End If

   Return menuStuffItems
End Function
```

At this point, the XML is loaded in the TreeView and arrays are in place to support the user adding new elements of the appropriate types.

Popping Up the Add Menu

Before looking at how to display an element, I'll show the tricks behind the dynamic menu that lets the user add nodes. Both adding and deleting elements requires that you know the context the user has in mind. Getting the active node is different when the user right-clicks vs. using the buttons at the bottom of the panel. It's fair to assume that if the user right-clicks, they want to work with the element they right-clicked, and if they use the buttons at the bottom, they want to work with the selected element.

To track what the user right-clicks, all mouse down events on the TreeNode set the value of the `mCurrentTreeNode` class field. That's a little ugly, but it's

effective in making the TreeNode under the mouse available within the right-click pop-up event:

```
Private Sub TreeView_MouseDown( _
                    ByVal sender As System.Object, _
                    ByVal e As System.Windows.Forms.MouseEventArgs) _
                    Handles TreeView.MouseDown
    ' When the user puts mousedown on this node for any reason,
    ' save it for possible use by a context menu
    Dim treeNode As Windows.Forms.TreeNode
    treeNode = TreeView.GetNodeAt(e.X, e.Y)
    If Not treeNode Is Nothing Then
       mCurrentTreeNode = treeNode
    End If
 End Sub
```

The context menu's pop-up event calls the `OnTreeViewAddPopup` method. This method calls the `FillAddMenu` with the correct TreeNode:

```
Protected Overridable Sub OnTreeViewAddPopup( _
                ByVal sender As Object, _
                ByVal e As System.EventArgs)
   FillAddMenu(mCurrentTreeNode, mnuTreeViewAdd)
End Sub
```

The `FillAddMenu` figures out the appropriate `XMLSchemaNode` to use to create the menu. This is nontrivial because clicking a leaf node adds a sibling, clicking a higher node adds a child element, and clicking the base element adds a child element from options retrieved from a different field. The `AddMenuStuff` method actually fills the context menu:

```
Private Sub FillAddMenu( _
                ByVal treeNode As Windows.Forms.TreeNode, _
                ByVal menu As Windows.Forms.Menu)
    Dim name As String
    Dim obj As Object
    Dim treeNodeXML As TreeNodeForXML = CType(treeNode, TreeNodeForXML)
    name = Utility.Tools.GetAttributeOrEmpty(treeNodeXML.XMLSchemaNode, "name")
    obj = mChildMenus(name)
    If Not obj Is Nothing Then
       AddMenuStuff(menu, obj)
```

```
    Else
       ' Try the next one up
       If Not treeNode.Parent Is Nothing Then
          treeNodeXML = CType(treeNode.Parent, TreeNodeForXML)
          name = Utility.Tools.GetAttributeOrEmpty(treeNodeXML.XMLSchemaNode,  _
                   "name")
          obj = mChildMenus(name)
          If Not obj Is Nothing Then
             AddMenuStuff(menu, obj)
          Else
             AddMenuStuff(menu, mBaseMenus)
          End If
       Else
          AddMenuStuff(menu, mBaseMenus)
       End If
    End If
 End Sub
```

Whew! That was a lot of work setting up for this fairly simple `AddMenuStuff` method. Because you've already done most of the work, the `AddMenuStuff` method just needs to cruise through the array adding the menu items and the handler to use when the user clicks each menu option:

```
Private Sub AddMenuStuff( _
                ByVal menu As Windows.Forms.Menu, _
                ByVal obj As Object)
   Dim mnu As MenuItemWithXSD
   Dim menuStuffs() As menuStuff
   menuStuffs = CType(obj, menuStuff())
   menu.MenuItems.Clear()
   For Each menuStuff As menuStuff In menuStuffs
      If (Not menuStuff.Text Is Nothing) AndAlso _
                        menuStuff.Text.Trim.Length > 0 Then
         mnu = New MenuItemWithXSD( _
                        menuStuff.Text, menuStuff.XSDNode)
         AddHandler mnu.Click, AddressOf AddItemClick
         menu.MenuItems.Add(mnu)
      End If
   Next
End Sub
```

Adding Elements

When the user adds a new element, the application needs to add a new `TreeNode` and a new XML node to the document. The new XML element needs to be legal according to the XSD. If the current node doesn't allow additions, the method looks up the tree until it finds a node that does allow new child elements:

```
Private Sub AddItemClick( _
               ByVal sender As Object, _
               ByVal e As System.EventArgs)
   Dim treeNode As TreeNodeForXML
   Dim mnu As MenuItemWithXSD = CType(sender, MenuItemWithXSD)
   Dim xmlDataNode As Xml.XmlNode
   Dim xsdNode As Xml.XmlNode = mnu.XSDNode
   Dim canAdd As Boolean
   If TypeOf sender Is MenuItemWithXSD Then
      If Not mXSDDoc Is Nothing Then
         treeNode = CType(mCurrentTreeNode, TreeNodeForXML)
         Do While Not treeNode.CanAdd
            treeNode = CType(mCurrentTreeNode.Parent, TreeNodeForXML)
         Loop
```

The `AddItemClick` method then creates a new namespace manager and calls the `MakeNewNode` method that does the actual work of adding the node:

```
         If Not treeNode Is Nothing Then
            ' Infer add status
            If treeNode.Nodes.Count > 0 Then
               canAdd = CType(treeNode.Nodes(0), TreeNodeForXML).CanAdd
            Else
               canAdd = False
            End If
            ' Now we know where to stick the new treenode, make new xmldatanode
            Dim nsmgr As New Xml.XmlNamespaceManager(New Xml.NameTable)
            nsmgr.AddNamespace("xs", "http://www.w3.org/2001/XMLSchema")
            nsmgr.AddNamespace("kd", "http://kadgen.com/KADGenDrivingx.xsd")
            MakeNewNode(mnu.XSDNode, treeNode, canAdd, nsmgr)
         End If
      End If
   End If
End Sub
```

In the MakeNewNode method, the Utility.Tools.MakeNewNodeFromXSDNode (discussed in the section "Making a Generic Node") creates the new XML data node. The new node needs a preliminary name, and the method creates one from the modified name of the current XML node's name:

```
Private Sub MakeNewNode( _
                    ByVal xsdNode As Xml.XmlNode, _
                    ByVal parentTreeNode As TreeNodeForXML, _
                    ByVal canAdd As Boolean, _
                    ByVal nsmgr As Xml.XmlNamespaceManager)
   Dim xmlDataNode As Xml.XmlNode = _
                Utility.Tools.MakeNewNodeFromXSDNode( _
                mXMLDoc, thisNamespace, "kg", xsdNode)
   ' Need a dummy name
   Dim attrName As Xml.XmlAttribute = CType( _
                xmlDataNode.Attributes.GetNamedItem("Name"), _
                Xml.XmlAttribute)
   Dim xsdNodes() As Xml.XmlNode
   Dim newTreeNode As TreeNodeForXML
   If Not attrName Is Nothing Then
      attrName.Value = Utility.Tools.SpaceAtCaps( _
                Utility.Tools.StripNamespacePrefix(xmlDataNode.Name))
   End If
```

MakeTreeNode method (introduced earlier) creates the new TreeNode and adds it to the requested parent:

```
parentTreeNode.XMLDataNode.AppendChild(xmlDataNode)
newTreeNode = MakeTreeNode(parentTreeNode.Nodes, xmlDataNode, canAdd, nsmgr)
```

Certain nodes are required as part of the parent node, and the parent node isn't legal without these child elements. I mark these required elements with the kd:autos attribute in the XSD and build a hash table of required elements organized by their parent node. The MakeNewNode method creates these required elements before exiting:

```
   xsdNodes = _

CType(mChildAutos(Utility.Tools.StripNamespacePrefix(xmlDataNode.Name)),
                Xml.XmlNode())
```

```
    If Not xsdNodes Is Nothing Then
        For Each xsdAutoNode As Xml.XmlNode In xsdNodes
            Dim newXmlDataNode As Xml.XmlNode = _
                        Utility.Tools.MakeNewNodeFromXSDNode( _
                        mXMLDoc, thisNamespace, "kg", xsdAutoNode)
            MakeTreeNode(newTreeNode.Nodes, newXmlDataNode, canAdd, nsmgr)
        Next
    End If
End Sub
```

Making a Generic Node

Creating an XML node based on an XSD node is a generic problem, so its solution is in the `Utility.Tools` class. It's another recursive algorithm. You'll find that recursive algorithms are often helpful when working with the nested levels of XML documents.

The `xsdElementNode` that is passed specifies the element to create. Creating the element as the `dataNode` variable is easy enough; remaining sections of the routine add the required attributes and child nodes:

```
Public Shared Function MakeNewNodeFromXSDNode( _
            ByVal xmldoc As Xml.XmlDocument, _
            ByVal namespaceURI As String, _
            ByVal prefix As String, _
            ByVal xsdElementNode As Xml.XmlNode) _
            As Xml.XmlNode
    Dim name As String = Utility.Tools.GetAttributeOrEmpty( _
                        xsdElementNode, "name")
    Dim attrName As String
    Dim dataNode As Xml.XmlElement = xmldoc.CreateElement( _
                        prefix & ":" & name, namespaceURI)
    Dim xsdAttrNodeList As Xml.XmlNodeList
    Dim attrValue As String
    Dim xsdNodeList As Xml.XmlNodeList
    Dim xsdTypeNode As Xml.XmlNode
    Dim xsdAttrTypeNode As Xml.XmlNode
    Dim nsmgr As New Xml.XmlNamespaceManager(xmldoc.NameTable)
    Dim typeName As String = Utility.Tools.GetAttributeOrEmpty( _
                        xsdElementNode, "type")
```

To add attributes and child nodes, you need to know whether the `complexType` is anonymous or named because you use different XPath expressions to retrieve anonymous and named `complexType` elements:

```
nsmgr.AddNamespace("xs", "http://www.w3.org/2001/XMLSchema")

If xsdElementNode.Attributes.GetNamedItem("type") Is Nothing Then
    ' This is an anonymous type
    xsdTypeNode = xsdElementNode.SelectSingleNode("xs:complexType", nsmgr)
Else
   xsdTypeNode = xsdElementNode.SelectSingleNode( _
                        "/xs:schema/xs:complexType[@name='" & typeName & "']", _
                        nsmgr)
End If
```

A big part of the `MakeNewNodeFromXSD` method deals with default values for each attribute. You can't ignore default values because empty strings would fail validation if the value must be in a specific enumeration, the value is Boolean, and so on:

```
If Not xsdTypeNode Is Nothing Then
     xsdAttrNodeList = xsdTypeNode.SelectNodes("xs:attribute", nsmgr)
     For Each xsdAttrNode As Xml.XmlNode In xsdAttrNodeList
        attrName = Utility.Tools.GetAttributeOrEmpty(xsdAttrNode, "name")
        typeName = Utility.Tools.GetAttributeOrEmpty(xsdAttrNode, "type")
        attrValue = ""
        If typeName.StartsWith("xs:") Then
           Select Case typeName
              Case "xs:int"
                 attrValue = "0"
              Case "xs:boolean"
                 attrValue = "false"
           End Select
        Else
           ' A simple type, assume its a simple type of enumeration, grab first
           xsdAttrTypeNode = xsdElementNode.SelectSingleNode( _
                        "/xs:schema/xs:simpleType[@name='" & typeName & _
                        "']/xs:restriction/xs:enumeration", nsmgr)
           If Not xsdAttrTypeNode Is Nothing Then
              attrValue = GetAttributeOrEmpty(xsdAttrTypeNode, "value")
           End If
```

```
        End If
        dataNode.Attributes.Append(Utility.xmlHelpers.NewAttribute( _
                        xmldoc, attrName, attrValue))
    Next
```

Once the attributes are set for the new node, the method does a recursive call to itself for each child node:

```
    xsdNodeList = xsdTypeNode.SelectNodes( _
                            "xs:sequence/xs:element|xs:all/xs:element", nsmgr)
        ' I'm not currently supporting choice because I am not sure what you'd do
        For Each xsdChildNode As Xml.XmlNode In xsdNodeList
           dataNode.AppendChild(MakeNewNodeFromXSDNode( _
                            xmldoc, namespaceURI, prefix, xsdChildNode))
        Next
      End If
      Return dataNode
   End Function
```

And the fully functioning new element is returned.

Deleting elements is simpler and uses standard `Windows.Forms` techniques. The additional issue is that you have to delete both the TreeNode and the XML element. I won't walk through that code because I don't think it presents new ideas.

Displaying an Element

The right side of the form displays details corresponding to each element in the TreeView. This is a dynamic data-driven display, again using the XSD to determine how to structure the display. To review how the script is organized, each script contains one or more *sections*. Each section contains a series of *directives*. There are a finite number of directives defined in the XSD and if you create new directives, you must extend the XSD as well. Each directive can contain attributes on the directive and child elements containing defined sets of attributes. This organization greatly simplifies the maintenance of the XSD. The UI for each directive contains group boxes for each defined child element and controls within these group boxes for each attribute.

When the user selects an attribute in the tree, the TreeView's `AfterSelect` event handler calls the `OnTreeViewAfterSelect` method. This method calls the `ControlForTag` method to instantiate the control to display. This is designed for

extensibility and the ability to display other types of controls, but right now, all directives are displayed using the `SimpleXMLNodeUserControl` control:

```
Protected Overridable Sub OnTreeViewAfterSelect( _
                ByVal sender As Object, _
                ByVal e As System.Windows.Forms.TreeViewEventArgs)
   Dim obj As Object
   Dim nodeData As Xml.XmlNode
   Dim nodeSchema As Xml.XmlNode
   If TypeOf e.Node Is TreeNodeForXML Then
      obj = ControlForTag(CType(e.Node, TreeNodeForXML).XMLDataNode)
      nodeData = CType(e.Node, TreeNodeForXML).XMLDataNode
      nodeSchema = CType(e.Node, TreeNodeForXML).XMLSchemaNode
   Else
      obj = ControlForTag(e.Node.Tag)
      If TypeOf e.Node.Tag Is Xml.XmlNode Then
         nodeData = CType(e.Node.Tag, Xml.XmlNode)
      End If
```

Once the right[4] control has been instantiated, the XML data needs to be placed in the control. The process of moving data from a store to UI controls is often called a *scatter* operation. The signature of the scatter called depends on the control type; again, there's extensibility built in here. The converse operation of moving from the UI controls to the data is called *gather*. `GatherAndRemoveExisitngUserControl` cleans up any previously displayed user controls after saving the users changes. After this is cleaned up, the method adds the new control to the controls collection of the tab on the right side of the form (`tbpInstructions`):

```
   End If
   If TypeOf obj Is Windows.Forms.Control Then
      If TypeOf obj Is SimpleXMLNodeUserControl Then
         CType(obj, SimpleXMLNodeUserControl).Scatter(nodeData, nodeSchema)
      ElseIf TypeOf obj Is WinFormSupport.ISimpleUserControl Then
         CType(obj, WinFormSupport.ISimpleUserControl).Scatter(nodeData)
      End If
      GatherAndRemoveExistingUserControl()
      tbpInstructions.Controls.Add(CType(obj, Windows.Forms.UserControl))
   End If
End Sub
```

4. *Right* refers both to "correct" and "on the right side of the user interface display."

Instantiating the Right Control

The short ControlForTag method determines the correct type and instantiates the control used to display the data. The TypeForTag method supports extensibility to other control types but right now returns the SimpleXMLNodeUserControl. The method uses reflection to instantiate the control:

```
Protected Overridable Function ControlForTag( _
                ByVal obj As Object) _
                As Object
   Dim type As System.Type
   Dim constructorInfo As Reflection.ConstructorInfo
   Dim retObj As Object
   type = TypeForTag(obj)
   If Not type Is Nothing Then
      ' We need to instantiate it here
      constructorInfo = type.GetConstructor(System.Type.EmptyTypes)
      retObj = constructorInfo.Invoke(New Object() {})
   End If
   Return retObj
End Function
```

The TypeForTag is overridden in the derived GenerationForm. The overriding method in GenerationForm just returns the type for the control to instantiate:

```
Protected Overrides Function TypeForTag(ByVal obj As Object) As System.Type
   Return GetType(GenerationDIrectiveUserControl)
End Function
```

Removing the Old Control

Before displaying the new control, you need to save any data currently displayed and remove the current control. The section “Gathering Data” explains the Gather method:

```
Protected Sub GatherAndRemoveExistingUserControl()
   If tbpInstructions.Controls.Count > 0 Then
      Me.Gather()
      tbpInstructions.Controls.RemoveAt(0)
   End If
End Sub
```

Scattering Data into Controls

The `Scatter` method of `SimpleXMLNodeUserControl` moves the data into the UI controls. There are a couple of overloads to this method. `OnTreeViewAfterSelect` calls the first one, which supplies a zero default for the top parameter and calls the overloaded version that does the real work:

```
Public Function Scatter( _
                ByVal xmlDataRow As Xml.XmlNode, _
                ByVal xmlSchemaNode As Xml.XmlNode) _
                As Int32
    Me.Tag = xmlDataRow
    Return Scatter(Me, 0, xmlDataRow, xmlSchemaNode)
 End Function
```

NOTE Earlier code was from the `SimpleTreeBase` form class. This code is from the `SimpleXMLNodeUserControl` class.

The `Scatter` method is a bit long but worth walking through. Arguments to the method include the parent control that'll contain the newly added controls. `Scatter` is another recursive method, so the parent control might be either the current `SimpleXMLNodeUserControl` (`Me` or `this`) or a contained group box. This method, and the methods it calls, scatters the data into the user control. As part of doing this it also creates the controls that'll contain the data.

The value of the `top` variable is modified as controls are positioned on the form and represents the top position for the next control to display. This value of `top` at the end of the `Scatter` method is returned:

```
Public Function Scatter( _
                ByVal parentControl As Windows.Forms.Control, _
                ByVal top As Int32, _
                ByVal xmlDataRow As Xml.XmlNode, _
                ByVal xmlSchemaNode As Xml.XmlNode) _
                As Int32
    Dim xsdNodeList As Xml.XmlNodeList
    Dim xTypeName As String
    Dim name As String
    Dim special As String
    Dim extra As Object
    Dim xpathString As String
    Dim ctl As Windows.Forms.Control
```

```
    Dim attrNode As Xml.XmlNode

    top += margin
    parentControl.Controls.Clear()
    If Not xmlDataRow Is Nothing Then
```

The code generation harness is designed for use with a passed `xmlSchemaNode`, but if it isn't passed, the current attributes are displayed. The actual controls are instantiated and loaded in the `NewLabeledControl` method:

```
If xmlSchemaNode Is Nothing Then
   For Each attr As Xml.XmlAttribute In xmlDataRow.Attributes
      If Not attr.Name.ToLower = "checked" Then
         ctl = NewLabeledControl(parentControl, attr.Name, _
                "String", "", "", False, top, attr.Value, Nothing)
         ctl.Tag = attr
         top = ctl.Bottom + margin
      End If
   Next
Else
```

If the XSD schema element is present, it's a vastly preferable source for the control layout specifics because it's richer, containing things such as type information, and because it's complete. If you add new items to the schema as you extend it,[5] they might not yet be present in the data node, but the corresponding controls will be displayed, and when the `Gather` method is called, the new items are included in the XML document:

```
Dim xsdNsmgr As Xml.XmlNamespaceManager = _
              Utility.Tools.BuildNameSpaceManager(xmlSchemaNode, True)
 xsdNsmgr.AddNamespace("kd", "http://kadgen.com/KADGenDrivingx.xsd")
 ' At this point, we may be looking at either a schema or a type
 If xmlSchemaNode.LocalName = "element" Then
    Dim sType As String = Utility.Tools.GetAttributeOrEmpty( _
              xmlSchemaNode, "type")
    If sType.Trim.Length = 0 Then
       xpathString = "xs:complexType/xs:attribute"
    Else
       xpathString = "//xs:complexType[@name='" & _
                        sType & "']/xs:attribute"
```

5. If the new items are required, you won't be able to open the XML file in the harness. You'll need to open it in a more forgiving editor and add the required elements before you can open it in the harness.

```
    End If
 Else
    xpathString = "xs:attribute"
 End If
 xsdNodeList = xmlSchemaNode.SelectNodes(xpathString, xsdNsmgr)
```

The `xsdNodeList` contains the definition for the attributes within the `complexType`, and several variables are set to different aspects of the XSD information:

```
For Each node As Xml.XmlNode In xsdNodeList
   name = Utility.Tools.GetAttributeOrEmpty(node, "name")
   xTypeName = Utility.Tools.StripPrefix( _
                Utility.Tools.GetAttributeOrEmpty(node, "type"))
   special = Utility.Tools.GetAnnotOrEmpty(node, "Special", xsdNsmgr)
   attrNode = xmlDataRow.Attributes.GetNamedItem(name)
```

The next `Select Case` determines whether the type should have special handling in the UI. Again, it's entirely based on the XSD definition for the directive:

```
Select Case xTypeName.ToLower
   Case "string", "int", "bool"
   Case "filename"
      special = "file"
   Case "directory"
      special = "directory"
   Case Else
      Dim comboList() As String
      comboList = Utility.Tools.StringArrayFromNodeList( _
                xmlSchemaNode.SelectNodes( _
                      "//xs:simpleType[@name='" & xTypeName & _
                      "']/xs:restriction/xs:enumeration", xsdNsmgr))
      If comboList.GetLength(0) > 0 Then
         special = "combobox"
         extra = comboList
      End If
End Select
```

The `NewLabeledControl` instantiates the new control. It uses the `special` and `extra` values set previously to determine the correct control to instantiate. In the case of combo boxes, the `NewLabeledControl` also fills the combo box. Continuing

through all of the attributes defined in the XSD loads all of the controls into the current container, which is either a user control or a `GroupBox`:

```
   ctl = NewLabeledControl( _
         parentControl, _
         name, _
         xTypeName, _
         special, _
         Utility.Tools.GetAttributeOrEmpty(node, "", "kd:Desc", xsdNsmgr), _
         (Utility.Tools.GetAnnotOrEmpty(node, "Required", xsdNsmgr) _
                  = "1"), _
         top, _
         Utility.Tools.GetAttributeOrEmpty(xmlDataRow, name), _
         extra)
   top = ctl.Bottom + margin
   ctl.Tag = attrNode
Next
```

Another call to `SelectNodes` fills the `xsdNodeList` with the child elements (previously it contained the attributes). This code loops through these elements finding those children that are themselves `complexTypes`:

```
xsdNodeList = xmlSchemaNode.SelectNodes( _
                "xs:complexType/xs:sequence/xs:element", xsdNsmgr)
Dim dataNsmgr As New Xml.XmlNamespaceManager( _
                xmlDataRow.OwnerDocument.NameTable)
dataNsmgr.AddNamespace("kg", "http://kadgen.com/KADGenDriving.xsd")
Dim grp As Windows.Forms.GroupBox
Dim childXSDNode As Xml.XmlNode
Dim childDataNode As Xml.XmlNode
For Each xsdNode As Xml.XmlNode In xsdNodeList
   xTypeName = Utility.Tools.GetAttributeOrEmpty(xsdNode, "type")
   If xTypeName.Trim.Length > 0 Then
      childXSDNode = xsdNode.SelectSingleNode( _
                "//xs:complexType[@name='" & xTypeName & "']", xsdNsmgr)
      childDataNode = xmlDataRow.SelectSingleNode("kg:" & xTypeName, _
                dataNsmgr)
      If Not childXSDNode Is Nothing Then
```

Each child that's a complexType is displayed in its own GroupBox. After positioning the GroupBox, a recursive call to Scatter fills it with the attributes of the complexType:

```
                    grp = New Windows.Forms.GroupBox
                    grp.Text = xTypeName
                    grp.Top = top + margin
                    grp.Width = parentControl.Width - (3 * margin)
                    grp.Tag = childDataNode
                    grp.Anchor = Windows.Forms.AnchorStyles.Top Or _
                          Windows.Forms.AnchorStyles.Left Or _
                          Windows.Forms.AnchorStyles.Right
                    parentControl.Controls.Add(grp)
                    top += Me.Scatter(grp, margin * 7, childDataNode, _
                          childXSDNode) + (3 * margin)
                 End If
              End If
           Next
        End If
    End If
    If top > 0 Then
        parentControl.Height = top + 2 * margin
    End If
    Return top
End Function
```

I'm going to skip walking through the NewLabeledControl method. It's about 90 lines of code instantiating different user controls based on the special and extra values passed and the type of the attribute being displayed, along with setting properties specific to each one. You can walk through it in the download if you're interested in how it works.

Gathering Data from Controls

When the user leaves the node, the user control is removed, and the values entered by the user need to be stored in the XML document. Gather checks the type of the only control in the tab page (tbpInstructions), casts it, and calls Gather of the SimpleXMLNodeUserControl class:

```
Protected Sub Gather()
    If tbpInstructions.Controls.Count > 0 Then
       If TypeOf tbpInstructions.Controls(0) Is ISimpleUserControl Then
          CType(tbpInstructions.Controls(0), _
                           WinFormSupport.ISimpleUserControl).Gather()
```

```
        End If
    End If
End Sub
```

The `Gather` method of the user control calls another overload that recursively moves data from the UI control into the XML document:

```
Public Sub Gather() Implements ISimpleUserControl.Gather
   Gather(Me)
End Sub
```

Finally, you get to work! `Gather` relies on the `XmlNode` contained in the tag of the parent element, such as the `SimpleXMLNodeUserControl` and its contained group boxes:

```
Public Sub Gather(ByVal parentControl As Windows.Forms.Control)
    Dim attrName As String
    Dim node As Xml.XmlNode
    Dim attr As Xml.XmlAttribute

    If TypeOf parentControl.Tag Is Xml.XmlNode Then
       node = CType(parentControl.Tag, Xml.XmlNode)
       For Each ctl As Windows.Forms.Control In parentControl.Controls
```

If the control is a group box, recursively call this method to retrieve the contained attributes:

```
If TypeOf ctl Is Windows.Forms.GroupBox Then
   Gather(ctl)
Else
```

If the control is anything other than a group box, the name of the attribute is contained within the name of the control. Retrieving this allows creation of a new attribute if needed. Because if the control doesn't contain a reference to an attribute in the XML document, it's created. If the control is a checkbox, the checked status is retrieved from the checked property. Otherwise, the method stores the control's `Text` property value in the attribute:

```
If ctl.Name.Length > 3 Then
   attrName = ctl.Name.Substring(3) ' remove prefix
   If ctl.Tag Is Nothing Then
      attr = node.OwnerDocument.CreateAttribute(attrName)
      node.Attributes.Append(attr)
```

```
                Else
                   attr = CType(ctl.Tag, Xml.XmlAttribute)
                End If
                Select Case ctl.Name.Substring(0, 3)
                   Case "chk"
                      attr.Value = CType(ctl, _
                                Windows.Forms.CheckBox).Checked.ToString.ToLower
                   Case Else
                      attr.Value = ctl.Text
                End Select
             End If
          End If
       Next
    End If
End Sub
```

NOTE `SimpleXMLNodeUserControl` contains a reference to the node in the XML document. So, when you modify the node contained in the `SimpleXMLNodeUserControl` variable, the node in the `XMLDoc` variable automatically reflects this change because they contain two pointers to the same object.

That's it for the `Gather` method. Keeping the XML document up-to-date makes file operations such as Save and Save As extremely easy. Keeping all of the data neatly tucked in the form allows true MDI behavior with many script files open or even the same script file open in two windows. But be careful of that last scenario. There's no collision detection, and the last file will win if both versions of the file are edited.

Executing the Script

Once you've got the XML in place, either using the editing tool described previously or manually editing the XML file, you're ready to run the code generation. A button on the `GenerationForm` initiates the public `RunGeneration` method of the `Generation` class of the `GenerationSupport` project. Figure C-3 shows an overview of the processing. Okay, that's a pretty intimidating diagram, but I'll provide a high-level overview and then walk though key segments of the code in the sections that follow.

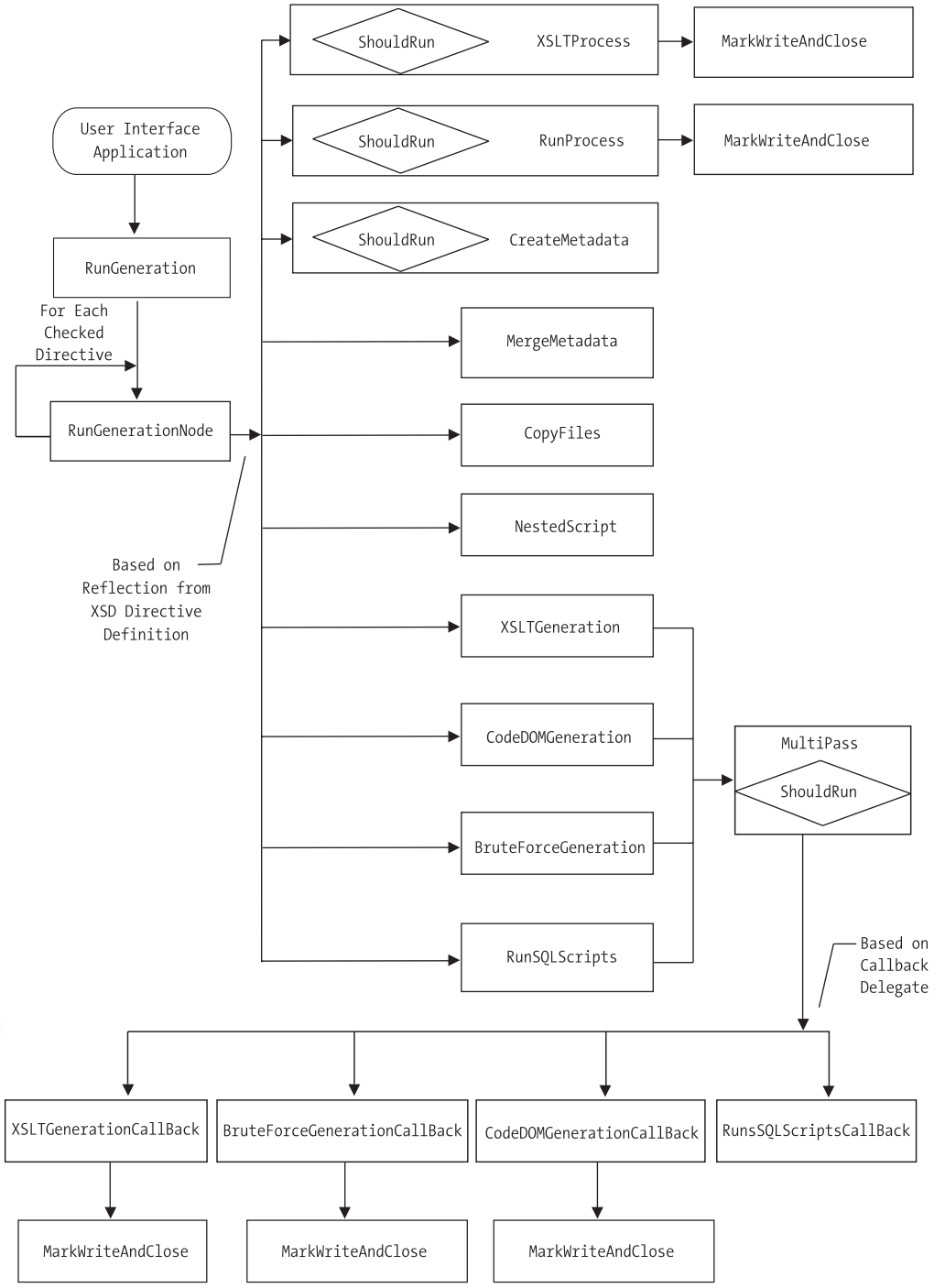


Figure C-3. A schematic layout of code generation execution

The RunGeneration method loops through the directives in the XML file, calling RunGenerationNode for each. The RunGenerationNode method processes all checked directives. RunGenerationNode uses the name of the element to determine the method to run for the directive and uses reflection to invoke the method. There are currently ten directives defined, as shown in Figure C-3. The XSLTProcess, RunProcess, and CreateMetadata methods each call the ShouldRun method to check whether overwriting the output file is appropriate. ShouldRun checks whether anyone has edited the file. MergeMetadata, CopyFiles, and NestedScripts run without this check.

XSLTGeneration, CodeDOMGeneration, BruteForceGeneration, and RunSQLScripts each produce multiple files based on an XPath expression. Each of these methods sets some class-level fields, determines a delegate for a callback, and calls the MultiPass method passing the delegate to the method that performs the specific type of processing. The MultiPass method manages the list of nodes to process, determines the output filenames, calls ShouldRun, manages source control, and performs other duties consistent for the four styles of MultiPass processing. The callback methods of XSLTGeneration, CodeDOMGeneration, and BruteForceGeneration all create streams, and the MarkWriteAndClose method handles writing these streams to disk.

Like all the files in samples associated with this book, Option Strict and Option Explicit are on. This reflection stuff can be confusing enough without missing compiler errors if you mess up a type:

```
Option Strict On
Option Explicit On

Imports System
```

Outputting Log Information

A class field of the OutputInfo structure is available throughout the Generation class for log information. The following fragment shows the information currently available, with these private fields exposed through properties that I'm not including:

```
Public Structure OutputInfo
   Public StartTime As DateTime  'Fields aren't displayed by default in the grid
   Public EndTime As DateTime
   Private mSection As String
   Private mDirectiveName As String
   Private mDirectiveType As String
   Private mSelectFile As String
   Private mInputFile As String
```

```
   Private mTemplate As String
   Private mStatus As String
   Private mOutputRule As String
   Private mCountSelected As Int32
   Private mCountCreated As Int32
   Private mCountOutput As Int32
```

Looking at RunGeneration

The RunGeneration class opens by declaring a delegate. The MultiPass processing later uses this delegate GenerateDelegate.

```
Public Class Generation

#Region "Class level declarations"
   Private Delegate Function GenerateDelegate( _
                ByVal outputFileName As String, _
                ByVal nsmgr As Xml.XmlNamespaceManager, _
                ByVal node As Xml.XmlNode, _
                ByVal nodeSelect As Xml.XmlNode) _
                As IO.Stream
```

I'm going to skip the rest of the class level declarations—there are about 20, so they'd be fairly tedious. Where you see a variable prefixed with an m in the code I walk through, it's a class-level (module-level) variable.

The only public method in the Generation class is RunGeneration, and it's probably also the messiest. XPath statements are used because of the sort features they provide. The calling code has to implement the IProgressCallback interface and pass itself to RunGeneration, which allows the calling form to later display progress updates via the callback:

```
#Region "Public Methods and Properties"
   Public Function RunGeneration( _
                   ByVal xmlDoc As Xml.XmlDocument, _
                   ByVal xsdDoc As Xml.XmlDocument, _
                   ByVal callBack As Utility.IProgressCallback, _
                   ByRef cntDone As Int32, _
                   ByRef output As Collections.ArrayList, _
                   ByVal xmlDocFileName As String) _
                   As Collections.ArrayList
      mLogEntries = New Collections.ArrayList
```

I'll skip the declarations and assigning passed parameters to class fields. The `RunGeneration` method saves the current directory and sets the default directory to the directory of the harness script. This ensures that all filenames are relative to this directory. It also sets up a source control object. This should gracefully degrade if SourceSafe isn't available, and you can write an extension for other source control options. These parts of the method opening are as follows:

```
currentDirName = Environment.CurrentDirectory
Environment.CurrentDirectory = _
                    IO.Path.GetFullPath(IO.Path.GetDirectoryName(xmlDocFileName))
SetupSourceSafe()
```

The `nodeItSection` variable used for the outer loop represents each section that's processed. An inner loop processes each child of the section, representing each directive:

```
nodeItSection = Utility.NodeItHelpers.GetNodeIt(xpathNav, _
                    "//kg:Section[kg:Standard/@Checked='true']", nsmgr)
Do While nodeItSection.MoveNext
   sectionName = Utility.NodeItHelpers.GetChildAttribute(nodeItSection.Current, _
                    "kg:Standard", "Name", nsmgr)
   nodeItDirective = Utility.NodeItHelpers.GetSortedNodeIt( _
                    nodeItSection.Current, _
                    "./*", nsmgr, "Ordinal", True, False, True)
   Do While nodeItDirective.MoveNext
```

The `mCurrentOutput` variable's methods hold values later output to the user. The `Standard` element is needed to extract the `Checked` attribute that determines if the directive should be processed by `RunGenerationNode`. After the node is processed, a callback to the class that called the method updates the progress bar for the user:

```
      mCurrentOutput = New OutputInfo
      mCurrentOutput.Section = sectionName
      mCurrentOutput.DirectiveName = Utility.NodeItHelpers.GetChildAttribute( _
                    nodeItDirective.Current, "kg:Standard", "Name", nsmgr)
      mCurrentOutput.DirectiveType = nodeItDirective.Current.LocalName
      mCurrentOutput.CountSelected = nodeItSection.Count
      If Utility.NodeItHelpers.GetChildAttribute(nodeItDirective.Current, _
                  "kg:Standard", "Checked", nsmgr) <> "false" Then
         mCntDone += 1
         If cntNodes = 0 Then
            callBack.UpdateProgress(100)
```

```
            Else
               percentDone = CInt(100 * mCntDone / cntNodes)
               If percentDone > 100 Then
                  percentDone = 100
               End If
               callBack.UpdateProgress(percentDone)
            End If
            callBack.UpdateCurrentNode(CType(nodeItDirective.Current, _
                     Xml.IHasXmlNode).GetNode)
            ' This is also updated in Multipass
            callBack.UpdateCurrentFile("")
            RunGenerationNode(nodeItDirective.Current, nsmgr, xsdDoc)
         Else
            mCurrentOutput.Status = "Not Checked"
         End If
         mOutput.Add(Me.mCurrentOutput)
      Loop
   Loop
   cntDone = mCntDone
```

At the end of the routine, the current directory is reset before returning control to the UI:

```
      ' Pop current directory
      Environment.CurrentDirectory = currentDirName
      Return mLogEntries
   End Function
#End Region
```

Looking at RunGenerationNode

The `RunGenerationNode` method starts the processing. It's an amazingly simple routine—just determining the `methodInfo` corresponding to the directive and then invoking it. If the method doesn't exist, no exception is raised:

```
Private Sub RunGenerationNode( _
                ByVal nav As Xml.xpath.XPathNavigator, _
                ByVal nsmgr As Xml.XmlNamespaceManager, _
                ByVal xsdDoc As Xml.XmlDocument)
   Try
      Dim node As Xml.XmlNode = CType(nav, Xml.IHasXmlNode).GetNode
```

```
        Dim nodeXSD As Xml.XmlNode = Utility.xmlHelpers.GetSchemaForNode( _
                                node.LocalName, xsdDoc)
        Dim methodInfo As Reflection.MethodInfo
        If node.LocalName <> "Standard" Then
           methodInfo = Utility.Tools.GetMethodInfo(nodeXSD, GetType(Generation), _
                                  nsmgr, mLocalSettings.GetBasePath(), mDocPath, _
                                  mNodeFilePaths)

        End If
        If Not methodInfo Is Nothing Then
           methodInfo.Invoke(Me, New Object() {node, nsmgr})
        End If
    Catch ex As System.Exception
        Diagnostics.Debug.WriteLine(ex.ToString)
        Throw
    End Try
End Sub
```

Looking at Tools.GetMethodInfo

The `GetMethodInfo` method and the `GetSpecifiedType` method it calls are important because they're used throughout the generation process when method or type information is needed via reflection. Reflection allows you to specify assemblies, types, and methods as strings and then find the corresponding methods using the assembly metadata. Using this information, you can create objects and invoke methods.

The code generation harness expects the assembly, type, and method to be specified with attributes named `AssemblyFileName` or `AssemblyName`, `TypeName`, and `MethodName`, respectively. There's a little bit of complexity in retrieving the names of the assemblies, types, and methods. Depending on the context, the names might be retrieved from the main element or one of the child elements. Because the actual code to resolve the type is messy and isn't terribly important to how the overall process works, I'll skip those details and just show the reflection call that retrieves the methods after the strings are set to the right values:

```
methodInfo = type.GetMethod(name, Reflection.BindingFlags.Instance Or _
                   Reflection.BindingFlags.Static Or _
                   Reflection.BindingFlags.IgnoreCase Or _
                   Reflection.BindingFlags.Public Or _
                   Reflection.BindingFlags.NonPublic)
Return methodInfo
```

Calling SinglePass Directives

There are five single pass or simple directives: `XSLTProcess`, `RunProcess`, `CreateMetadata`, `MergeMetadata`, and `CopyFiles`. I'll just walk through the most complex—`XSLTProcess`. Even `XSLTProcess` is pretty simple. The method retrieves filenames for the output file, input XML, and the XML Transformations (XSLT) template:

```
Private Sub XSLTProcess( _
                ByVal node As Xml.XmlNode, _
                ByVal nsmgr As Xml.XmlNamespaceManager)

    Dim basePath As String = mLocalSettings.GetBasePath()
    Dim outputFile As String = Utility.Tools.FixPath( _
                        Utility.Tools.GetAttributeOrEmpty( _
                               node, "kg:SinglePass", "OutputFile", nsmgr), _
                        basePath, mDocPath, mNodeFilePaths)
    Dim inputXML As String = Utility.Tools.FixPath( _
                        Utility.Tools.GetAttributeOrEmpty( _
                               node, "kg:XSLTFiles", "InputFileName", nsmgr), _
                        basePath, mDocPath, mNodeFilePaths)
    Dim XSLTFileName As String = Utility.Tools.FixPath( _
                        Utility.Tools.GetAttributeOrEmpty( _
                               node, "kg:XSLTFiles", "XSLTFileName", nsmgr), _
                        basePath, mDocPath, mNodeFilePaths)
```

Extension objects allow XSLT templates to call back to .NET, as discussed further in Appendix A. This lets you minimize redundant functionality in XSLT and .NET for things such as building plurals and changing types. Limitations in the UI restrict you to only one extension object per XSLT process, but this isn't a limitation in extension objects or the harness processing. If an extension object is specified for the directive, a namespace must also be specified. This namespace must exactly match the namespace used to call the object in the XSLT stylesheet for the callback function to run successfully.

XSLT processing, as well as all of the code generation mechanisms that create a stream, use `ShouldRun` to determine if the file should be created. If it should be, the stream is created by the `GenerateViaXSLT` method. This stream is passed to the `MarkWriteAndClose` method that calculates the hash, writes the file, and closes the stream. `SSCheckInFile` handles source control if a source control object is registered. The `madeFile` variable tracks whether a file was actually created:

```
    Dim extObjects() As ExtObject = GetExtObjects(node, nsmgr)
    Dim stream As IO.Stream
```

```
    Dim madeFile As Boolean
    Dim countOutput As Int32
    Dim outputType As String = Utility.Tools.GetAttributeOrEmpty(node, _
                "kg:OutputRules", "OutputFileType", nsmgr)
    CurrentOutputStart("None", inputXML, XSLTFileName, 1, _
                Me.GetOutType(node).ToString)
    Try
       If ShouldRun(outputFile, node) Then
          If SSCheckOutFile(node, basePath, outputFile) Then
             Try
                stream = CodeGenerationSupport.XSLTSupport.GenerateViaXSLT( _
                                XSLTFileName, inputXML, outputType, extObjects)
                madeFile = MarkWriteAndClose(stream, node, outputFile)
             Finally
                SSCheckInFile(node, basePath, outputFile, madeFile)
             End Try
          End If
       End If
       If madeFile Then
          countOutput = 1
       End If
       CurrentOutputEnd("Done", 1, countOutput)

    Catch ex As System.Exception
       mCurrentOutput.Status = "Error"
       Throw
    End Try
End Sub
```

Looking at ShouldRun

The `ShouldRun` method determines whether output conditions warrant building the file. Chapter 3 explains the rules for how the `OutputGenType` attribute and output conditions interact. The start of the method determines the generation type (which category of overwriting behavior to use) and the output type (VB, C#, or XML). It also calls a method that opens the file and checks the internal hash to see if anyone has modified the file. I won't walk through the `IsFileChanged` method because it does pretty much what you'd expect—opens the file, strips the header, rebuilds a hash, and compares it with the hash that was previously stored in the file. Note that this style of hashing is great for a situation such as this where malicious tampering with the hash isn't expected:

```
Private Function ShouldRun( _
                 ByVal outputfile As String, _
                 ByVal node As Xml.XmlNode) _
                 As Boolean
    Dim outType As Utility.OutputType
    Dim fileChanged As Utility.FileChanged
    If node.Name = "kg:RunSQLScripts" Then
       ' Should Run is handled later
       Return True
    Else
       Dim nsmgr As Xml.XmlNamespaceManager = GetNameSpaceManager( _
                             node.OwnerDocument, "kg")
       Dim nodeStandard As Xml.XmlNode = node.SelectSingleNode( _
                             "kg:OutputRules", nsmgr)
       outType = Me.GetOutType(node)
       fileChanged = Utility.HashTools.IsFileChanged(outputfile, _
                             mProjectSettings.GetCommentStart(outType), _
                             mProjectSettings.GetCommentEnd(outType), _
                             mProjectSettings.GetHeaderMarker(), _
                             mProjectSettings.GetHashMarker())
       Return ShouldRun(Me.GetGenType(node), fileChanged, outputfile, False)
    End If
 End Function
```

A second overload of `ShouldRun` determines whether the file can be overwritten and what message to record for the user. This overload of `ShouldRun` consists of a rather ugly set of nested select statements that test for the various possible combinations to do what's specified in Chapter 3. Ultimately, this method returns `True` or `False`, indicating whether the calling code should perform processing to replace the file. `ShouldRun` also outputs messages as appropriate. You can look at this method in the download if you want to see details of how it works.

Calling MultiPass Directive

Because the four multipass directives are similar, I'll just walk through the XSLT generation. This will give you enough understanding so that the details of the brute-force and CodeDOM generation, and `RunSQLScript` will make sense.

`RunGeneraationNode` calls `XSLTGeneration`. It sets some class-level fields that'll speed further processing and calls the `MultiPass` method, passing a delegate to the `XSLTGenerationCallback` method that does the actual processing. The methods initiating the process and the callback methods are unique for each type of

multipass processing, but all four types use the same `MultiPass` method. Delegates offer a simple way to organize the functionality and maximize reuse.

For XSLT processing the input metadata can be loaded once and stored in a class field as an `XPathNavigator`. Loading the potentially large input file many times would significantly degrade processing performance. The XSLT stylesheet can also be loaded once and stored in a class field:

```
Private Sub XSLTGeneration( _
                ByVal node As Xml.XmlNode, _
                ByVal nsmgr As Xml.XmlNamespaceManager)
   Dim basePath As String = mLocalSettings.GetBasePath()
   Dim metadataFileName As String = Utility.Tools.FixPath( _
                Utility.Tools.GetAttributeOrEmpty( _
                          node, "kg:XSLTFiles", "InputFileName", nsmgr), _
                basePath, mDocPath, mNodeFilePaths)
   Dim xmlDoc As New Xml.XmlDocument
   Dim xsltFile As String = Utility.Tools.FixPath( _
                Utility.Tools.GetAttributeOrEmpty( _
                          node, "kg:XSLTFiles", "XSLTFileName", nsmgr), _
                basePath, mDocPath, mNodeFilePaths)
   CurrentOutputStart("None", metadataFileName, xsltFile, 1, _
                Me.GetOutType(node).ToString)
   Try
      xmlDoc.Load(metadataFileName)
      mNavMetadata = xmlDoc.CreateNavigator
```

To prepare for parameters to be passed to the XSLT stylesheet, you have to know what the parameters are. The `GetXSLTParams` method opens the XSLT stylesheet as an XML document and retrieves the `xsl:param` elements. Information about the parameters is stored in a class field. `XSLTGenerationCallback` later adds values for the node it's processing. `GetExtObjects` performs the same task it performed in `XSLTProcess`, except that the array of extension objects is stored in a class field for reuse:

```
mParams = CodeGenerationSupport.XSLTSupport.GetXSLTParams(xsltFile, _
             basePath)
mExtObjects = GetExtObjects(node, nsmgr)
mXSLTTransform = New Xml.Xsl.XslTransform
mXSLTTransform.Load(xsltFile)
```

The address of the method that performs the actual XSLT generation—the delegate pointing to `XSLTGenerationCallback` is passed to the `MultiPass` method. When the `MultiPass` method has processed all the nodes, the `XSLTGeneration` method clears the class fields before exiting:

```
        MultiPass(node, nsmgr, AddressOf XSLTGenerationCallBack)
        mNavMetadata = Nothing
        mParams = Nothing
        mXSLTTransform = Nothing
        mCurrentOutput.Status = "Done"
    Catch ex As System.Exception
        Throw
    End Try
End Sub
```

Catching and rethrowing the exception allows you to set a breakpoint to catch any exceptions locally. Reflection makes it a little harder to debug, so this technique is sometimes useful (although it's ugly).

The node passed to the `MultiPass` method is the node of the directive in the harness script file. The `MultiPass` method does most of the work for managing files during code generation:

```
Private Sub MultiPass( _
                ByVal node As Xml.XmlNode, _
                ByVal nsmgr As Xml.XmlNamespaceManager, _
                ByVal delegGenerate As GenerateDelegate)
    Dim basePath As String = mLocalSettings.GetBasePath()
    Dim xmlDoc As New Xml.XmlDocument
    Dim nodeList As Xml.XmlNodeList
    Dim outputFileName As String
    Dim attr As Xml.XmlNode
    Dim stream As IO.Stream
```

Four values are retrieved from the directive node to specify fragments of processing and the output filename:

```
Dim outputDir As String = Utility.Tools.FixPath( _
                      Utility.Tools.GetAttributeOrEmpty( _
                            node, "kg:MultiPass", "OutputDir", nsmgr), _
                      basePath, mDocPath, mNodeFilePaths)
Dim outputFilePattern As String = Utility.Tools.GetAttributeOrEmpty( _
                      node, "kg:MultiPass", "OutputFilePattern", nsmgr)
Dim selectXPath As String = Utility.Tools.FixFilter( _
                      Utility.Tools.GetAttributeOrEmpty( _
                      node, "kg:MultiPass", "SelectXPath", nsmgr), _
                      mNodeFilters)
Dim selectFile As String = Utility.Tools.FixPath( _
                      Utility.Tools.GetAttributeOrEmpty( _
                            node, "kg:MultiPass", "SelectFile", nsmgr), _
                      basePath, mDocPath, mNodeFilePaths)
```

The value of the `selectFile` variable specifies the XML file that contains the nodes to process, and the `selectXPath` variable specifies the XPath expression used to select nodes to process. The `selectFile` is generally the metadata file used in processing. An `XmlNodeList` named `nodeList` contains the nodes selected for processing. For example, the `selectXPath` expression might select all the tables in the metadata document for further processing:

```
Dim madeFile As Boolean
If selectFile.Trim.Length > 0 Then
   If (Not Me.mCurrentOutput.InputFile Is Nothing) AndAlso _
                Me.mCurrentOutput.InputFile.Trim.Length = 0 Then
      Me.mCurrentOutput.InputFile = selectFile
   End If
   xmlDoc.Load(selectFile)
   Dim nsmgrMeta As Xml.XmlNamespaceManager = _
                   Utility.Tools.BuildNameSpaceManager( _
                         xmlDoc, node, "kg:MultiPass", nsmgr, "Select")
   nsmgrMeta.AddNamespace("ffu", "http://kadgen/FreeFormForUI.xsd")
   nsmgrMeta.AddNamespace("ui", "http://kadgen.com/UserInterface.xsd")
   nodeList = xmlDoc.SelectNodes(selectXPath, nsmgrMeta)
   Me.mCurrentOutput.CountSelected = nodeList.Count
```

The `outputFilePattern` contains the pattern for the output filename. Each output filename must be unique or output files will overwrite each other:

```
For Each nodeSelect As Xml.XmlNode In nodeList
   madeFile = False
   outputFileName = IO.Path.Combine(outputDir, _
                ParseFilenamePattern(outputFilePattern, nodeSelect, _
                nsmgrMeta))
   mCallBack.UpdateCurrentFile(outputFileName)
   Utility.Tools.MakePathIfNeeded(outputFileName)
```

Armed with the filename, the `ShouldRun` method returns whether the file should be created, as discussed earlier. If the processing should be performed, the specified delegate is invoked. For `XSLTGeneration`, the delegate calls `XSLTGenerationCallback`. Finally, `MarkWriteAndClose` handles hashing the title and writing it to disk:

```
         If ShouldRun(outputFileName, node) Then
            If SSCheckOutFile(node, basePath, outputFileName) Then
               Try
```

```
                    stream = delegGenerate.Invoke(outputFileName, nsmgr, node, _
                                          nodeSelect)
                    If Not stream Is Nothing Then
                       madeFile = MarkWriteAndClose(stream, node, outputFileName)
                    End If
                    Me.mCurrentOutput.CountCreated += 1
                    If madeFile Then
                       Me.mCurrentOutput.CountOutput += 1
                    End If
                 Finally
                    SSCheckInFile(node, basePath, outputFileName, madeFile)
                 End Try
              End If
           End If
        Next
    End If
End Sub
```

The actual processing for the node is very simple. The `SetParameters` method stuffs the current values of the specified attributes into the parameters held in a class field and the previously saved field values are passed to the `GenerateViaXSLT` method for processing:

```
Private Function XSLTGenerationCallBack( _
                ByVal outputFileName As String, _
                ByVal nsmgr As Xml.XmlNamespaceManager, _
                ByVal node As Xml.XmlNode, _
                ByVal nodeSelect As Xml.XmlNode) _
                As IO.Stream
   SetParameters(nodeSelect, outputFileName)
   Dim outputType As String = Utility.Tools.GetAttributeOrEmpty(node, _
                "kg:OutputRules", "OutputFileType", nsmgr)
   Return CodeGenerationSupport.XSLTSupport.GenerateViaXSLT( _
                         mXSLTTransform, mNavMetadata, outputType, _
                         mExtObjects, mParams)
End Function
```

The other two code generation mechanisms closely parallels the approach shown here for XSLT generation. `RunSQLScripts` is similar but runs existing scripts rather than creating new files, so it skips some steps of the process.

Looking Back at the Generation Harness

The code generation harness consists of two pieces—the UI and script execution. Both rely on reflection techniques.

The UI dissects an XSD schema and displays results based on the schema. The UI code consists of a base and derived class, with the base class doing the majority of the work. If you have another need for an XML editor, you could derive separate classes based on your needs, paralleling the classes in the `CodeGenerationHarness` project.

The execution engine relies on reflection to determine the methods to run based on element names and attribute values within the generation harness script. This technique is useful when various actions need to be performed based on a script and extensibility of the script is valuable.

EXCERPT

APPENDIX D

CodeDOM Issues, Tips, and Tricks

Chapter 3 introduces CodeDOM techniques. It shows you how to build a CodeDOM graph and how to output code using this graph in any supported language.

That gives you an idea of how to use the CodeDOM. Unfortunately, a number of ancillary issues plague CodeDOM development. Many of these issues are related to the CodeDOM supporting only the least common denominator in code. Others are specific shortcomings in how the CodeDOM works.

Extending the CodeDOM

As you look at the problems with the CodeDOM, you may wonder if you could improve on the Microsoft-provided CodeDOM. If you're considering this, carefully explore the distinction between what Microsoft seems to have just left out and what's a significant difference between languages. Microsoft's existing CodeDOM solutions are designed to work with any language, even ones that aren't yet created. If you modify the CodeDOM, you won't have this support for the full family of .NET languages. But, if you just care about Visual Basic .NET (VB .NET), C#, and perhaps a few other languages, you could create a more complete solution.

This task isn't easy. In fact, I've helped my son work with the CodeDOM to express a symbolic, language-neutral representation of existing code. He ran into significant problems working with the `CodeGenerator` classes that Microsoft provided. He explains the issues involved in the sidebar "Extending Providers."

Extending Providers

Extending the CodeDOM through inheritance lets you create a more accurate and useful symbolic representation of your code. You extend the CodeDOM by creating new classes derived from the CodeDOM's `CodeObject` class. The classes that create the code graph are designed for logical extension with the exception of attributes.[1]

Unfortunately, outputting this extended code graph to real code is problematic because the `CodeGenerator` classes are written with a black-box approach, rather than as an extensible model. The language-specific `CodeGenerator` classes provided by Microsoft won't know what to do with any extended `CodeObject` classes. There are three ways to work around this as follows:

- Create a wrapper around the Microsoft `CodeGenerator` that supports the `ICodeGenerator` interface.
- Preprocess the CodeDOM graph to distill it to standard `CodeObject` classes and snippets.
- Create your own `CodeGenerator` from the ground up.

The first option is by far the easiest to implement but unfortunately the least useful. You just need to implement the methods of `ICodeGenerator` and forward all calls you don't want to handle to the appropriate method in a `CodeGenerator` object held in a private field in your wrapper class. This contained private `CodeGenerator` object is the standard CodeDOM provider for your target language. You place any extra logic for outputting extended `CodeObject` classes in your wrapper methods. The usefulness of this option is extremely limited, however, because once the call is forwarded to the wrapped generator, you lose all control—it doesn't come back until it's done with the block. For example, let's say you have an extended `CodeExpression` that you want to output and you call your wrapper at the `CodeCompileUnit` level. The wrapper would forward this call to the standard generator, and the standard generator wouldn't know what to do with your extended expression. Any methods in your wrapper class that output custom code would never be called. I've seen wrappers used to hack a solution for outputting regions, but it's really ugly. If you find a useful way to use wrapped CodeDOM generators, let me know.

Although it isn't an elegant solution, preprocessing the CodeDOM graph provides a more practical way to output extended `CodeObject` classes without having to rewrite the entire provider. To use this technique, you first create a complete copy of the CodeDOM graph for every output language. Preprocess this graph before giving it to the generator, and replace all `CodeObject`

1. Classes in the CodeDOM graph, except attributes, derive from CodeObject. It isn't clear why, but attributes derive from `System.Object`.

extensions with snippets containing their output code. Keep in mind that the output code may need to include the output code of child nodes. This option can work fairly well in certain situations, such as creating extended expressions, because the output is small and contained on one line. However, if you're trying to create an extension to output a `For Each` loop, this method becomes significantly more difficult because you must include all the statements inside the loop in the snippet string. This requires that you worry about outputting a broad range of content and details such as the indentation on each line.

If you're doing a lot of extension of the CodeDOM graph, the best choice for output is probably to rewrite the `CodeGenerator`. It does involve a lot of parallel work to what Microsoft has already done, and you have to write a `CodeGenerator` for every language you support.[2] But, if you write your own providers, you can create a much simpler and extensible model. To implement this solution, you create a class that implements `ICodeGenerator`. Add methods (probably protected scope) to this class that output each type derived from `CodeObject`—both the original CodeDOM classes and your custom derived classes. This includes classes such as `CodeExpression` that don't themselves output anything but instead examine the type of the expression and make a call to more specific output method for something such as `CodeCastExpression`. Use `IndentedTextWriter` in your new provider to help output cleanly formatted text. Another benefit of re-creating the generator is that it allows you to easily fix any disagreements you have with Microsoft's implementation—such as removing the annoying extra line breaks in the output of C# code.

—Ben Joeris
Programmer

Looking at Broad Issues with the CodeDOM

Because the purpose of the CodeDOM is to output code in all languages, it can't work with compiler-specific features of a given language.[3] If you can't accept this, don't work with the CodeDOM. You may not know where your compiler does extra work on your behalf,[4] so the following sections cover the compiler features you need to avoid when you're using the CodeDOM.

2. If you aren't outputting to more than one language, why are you using the CodeDOM? You actually may have other reasons to use the CodeDOM if you can extend it to be a fully symbolic model of your code, but your effort will be far more useful if you output to at least the core languages.

3. In some cases, you can use snippets to output language-specific features, but that trashes the idea of language-independent code outputting.

4. If you want more information, you can find details about your language, including what it does on your behalf, in its language specification. Both VB .NET and C# have formal specifications included in the .NET Help.

Overall documentation for the CodeDOM is decent but minimal. All of my efforts in it have felt a bit like getting pounded in big mean white water while I was in a small boat. If you find a way to accomplish any of things in this section (short of snippets) or think I've left things out, please let me know, and I'll put corrections on the Apress Web site.

Shortcutting Namespaces (Imports/using)

The VB .NET `Imports` and C# `using` aren't equivalent. The VB .NET `Imports` statement must appear at the top of the file, and it applies to the whole file. C# allows you to place the parallel `using` statement within a namespace. Also, VB .NET allows partial namespaces, but C# doesn't. For example, the following VB code is valid:

```
Imports System
Dim dt As Data.DataTable
```

But the parallel C# code raises a compiler error:

```
using System;
Data.DataTable dt;
```

To accommodate this difference with CodeDOM templates, you have to use the C# approach. If namespaces are used, you have to replace the entire namespace. Thus, you need the template to output the following:

```
using System.Data;
DataTable dt;
```

The VB provider is smart enough to always place the `Imports` at the top of the file, regardless of whether you put it in a namespace. However, achieving matching results (putting the `using` statements outside the namespace) for the `using` statement of C# requires a bit of a trick. As you saw earlier, `CodeCompileUnit` contains a collection of `CodeNamespace` objects. If you create a namespace with an empty string as the namespace name, it acts as a container for other code but doesn't actually output the namespace statement in either VB .NET or C#.[5] The trick is to create this unnamed namespace as the first namespace in the namespace collection and place the `CodeNamespaceImport` object in this unnamed namespace. Create one or more additional named `CodeNamespace` objects to hold the actual code for your file.

5. In other words, it acts as a default namespace.

To support namespace aliasing in the `Imports/using` statements, use this syntax:

```
nSpace.Imports.Add(New CodeNamespaceImport("AliasName = System"))
```

This syntax is cheating the CodeDOM. C# and VB .NET just happen to alias imported namespaces similarly. VB .NET uses this:

```
Imports AliasName = System
```

And C# uses this:

```
using AliasName = System;
```

The CodeDOM has no clue that you're aliasing a namespace. It just blindly outputs whatever string you give as the namespace name.

Declaring Base Types and Interfaces

The `CodeTypeDeclaration` object contains a property called `BaseTypes` that contains a `BaseTypes` collection. That may imply multiple inheritance, but of course the .NET Framework doesn't support multiple inheritance. Instead, the first member is the actual base type from which you're deriving. Any additional types are interfaces. The following CodeDOM code:

```
Dim cls As New CodeTypeDeclaration("TestClass")
cls.BaseTypes.Add("System.Object")
cls.BaseTypes.Add("ITest")
cls.BaseTypes.Add("ITest2")
nspace.Types.Add(cls)
```

produces this VB .NET code:

```
Public Class TestClass
     Inherits Object
     Implements ITest, ITest2
 End Class
```

and this C# code:

```
public class TestClass : object, ITest, ITest2 {
}
```

To specify a class with no base class, list `System.Object` as the first of the `BaseTypes`.

Implementing Interfaces

There's a little more ugliness here. In VB .NET, you specify the interface methods implemented by each class method using the `Implements` keyword. C# uses name matching. In the CodeDOM, you specify implementation via the `ImplementationTypes` of each member, which you need to do for VB .NET, even where it doesn't affect the C# output. The CodeDOM doesn't allow you to have different names for the implementation and the interface method, which is supported in VB .NET, but not C#.

VB .NET also has more flexibility in scoping, and private or friend implementations for public interfaces are common. C# requires interface implementations to be public but provides a different mechanism for private implementations. This is represented in the CodeDOM by `PrivateImplementationType`, of which there can only be one for each method. Because the VB .NET provider doesn't understand this single special private interface for the method, it creates a method with the name *`<interface>_<methodname>`*. Unfortunately, both of the language-specific compilers output the results of a method with `PrivateImplementationType` without an explicit scope. The default scope of VB .NET is public, and the default scope of C# is private. Thus, the ugly named method is accessible in addition to the intended name accessible through the interface in VB .NET. That looks like a CodeDOM bug.

Working with the Scope and Type of Members

The scope and other information for class members is determined using the `Attributes` property. The `Attributes` property allows you to combine multiple flags indicating whether the member is shared (static), private, public, and so on. Some of the scope possibilities, such as `FamilyAndAssembly`, supported by the CodeDOM aren't supported by VB .NET or C#. `FamilyAndAssembly` is output as `assembly (Friend)` in both languages, and the additional restriction to derived members is ignored.

VB .NET and C# treat nonoverridden members with the same name as a member in the base class differently. VB .NET requires you to explicitly state your intent and raises a compiler warning unless you specify either the `Overloads`—for the Microsoft Intermediate Language (MSIL) `hidebysig`—or `Overrides` modifier. C# outputs the MSIL `hidebysig` modifier unless you use the `overrides` modifier (it does this even when there isn't a corresponding base class method). Thus, you'll need to include the `Overloaded` attribute at times you might not expect if

you're more familiar with C#. As long as you're testing in VB .NET, the compiler will tell you when you need to add it.

In C#, you can use the `new` modifier (not constructor) to specify the MSIL `hidebysig` that makes the default explicit. The language provider for VB .NET appears to handle the `new` attribute incorrectly. It outputs `Shadows` instead of `Overloads`. `Shadows` is referred to as *hide by name* and corresponds to the lack of a `newslot` modifier in MSIL.[6] This isn't the same as the C# `new` modifier or MSIL's `hidebysig` and appears to be a second bug[7] in the VB language-specific provider.

Assigning Variables Before Use

VB .NET implicitly assigns default values to all value type variables. For example, the following declaration:

```
Dim j as System.Int32
```

results in a variable `j` that's initialized to zero. C# doesn't do this, and you must explicitly assign values to all variables prior to use to avoid compiler errors. When working with the CodeDOM, you have to follow the C# approach to produce code valid in both languages. The most convenient way to do this is providing initialized values as part of all declarations.

Comparing Assignment and Equality

VB .NET uses the same character for assignment and equality. Thus, the following code:

```
Dim bexp As New CodeDom.CodeBinaryOperatorExpression( _
                New CodeDom.CodeVariableReferenceExpression("iValue"), _
                CodeDom.CodeBinaryOperatorType.Assign, _
                New CodeDom.CodeBinaryOperatorExpression( _
                      New CodeDom.CodePrimitiveExpression(42), _
                      CodeDom.CodeBinaryOperatorType.Add, _
                      New CodeDom.CodePrimitiveExpression(23)))
Dim ifBlock As New CodeDom.CodeConditionStatement(bexp)
ifBlock.TrueStatements.Add(WriteLineExpression("True Executed"))
mbr.Statements.Add(ifBlock)
```

6. `Shadows` or "hide by name" isn't supported in C#.

7. There are actually surprisingly few bugs in the CodeDOM.

outputs the perfectly valid VB .NET code:

```
If (iValue  _
             = (42 + 23)) Then
    System.Console.WriteLine("True Executed")
End If
```

However, the corresponding C# code output is illegal:

```
if ((iValue
             = (42 + 23))) {
    System.Console.WriteLine("True Executed");
}
```

This is invalid C# code because `==` must be used for equality. The problem is the inadvertent use of an assign expression when a binary expression with the equality operator was needed.

Outputting Whitespace

You can add whitespace via CodeDOM snippets. You can't remove any whitespace inserted by the language provider. From a practical perspective, you'll have to just live with the whitespace produced by default. You can't even control the awkward line break between `iValue` and assigning `42+23` in the previous example.

The two exceptions are setting the size of the indent and controlling blank lines between members. You set both these values with the `CodeGenerationOptions` value passed to the `GenerateCodeFromCompileUnit` during code generation. Set the indent size with this:

```
CodeGenerationOptions.IndentString = "    "
```

Blank lines are controlled with the `BlankLineBetweenMembers` property. The default value is `True`. For C#, you can also control whether open curly brackets appear on the same line. Having them appear on the same line is the default and specified with the following:

```
CodeGenerationOptions.BracingStyle = "Block"
```

To specify curly brackets on the next line, use this:

```
CodeGenerationOptions.BracingStyle = "C"
```

The code generation harness included with the book doesn't currently support these features. Extending the XML Schema Definition (XSD) and the harness's mechanism for outputting code from CodeDOM templates would be a great exercise in learning how the harness handles CodeDOM templates.

Using Regions and Other Compiler Directives

You have to use snippets to create regions and other compiler directives via CodeDOM. If you do this, however, you no longer have a language-neutral CodeCOM graph. You'll also have to ensure that you close any compiler directives such as regions that need matching open and close constructs. Although regions are an important part of creating code that simplifies debugging using the other code generation techniques, you'll have to balance this benefit against the language-specific nature of snippets. To limit the language dependency, you could extend the provider as briefly described in the section "Extending the CodeDOM." If you're just trying to manage regions, the preprocessing approach described in the sidebar is probably the easiest.

Avoiding MyClass

The VB .NET `MyClass` keyword allows you to specify a method in the current class, regardless of whether it was overridden in a child class. This allows linking of specific methods that really need to work together. The CodeDOM doesn't support `MyClass`.

Avoiding Local Static Variables

VB .NET (and C++) supports local static variables. Visibility of these variables is scoped to the local method, but their lifetime is similar to shared class fields—the lifetime of the application. The CodeDOM doesn't support local static variables.[8]

---

8. The .NET Framework doesn't support local static variables. Instead, they're implemented in VB .NET with compiler sleight of hand and class fields.

Outputting Option Strict and Option Explicit

VB .NET code, whether generated manually or from a code generator, should almost always include `Option Strict On` and `Option Explicit On`. `Option Explicit` indicates that variable declarations are required, and `Option Strict` indicates whether the compiler prohibits nonwidening implicit type conversions. Both C# and VB .NET allow widening conversions (such as `System.Int16` to `System.Int32`).

CAUTION With `Option Strict On`, VB .NET is actually "stricter" than C# because of C#'s operator overloading. This forces explicit cases in all nonwidening conversions. The best way to find these cases is by outputting to both languages and checking that the code output to each language compiles.

By default the VB .NET CodeDOM provider outputs `Option Strict Off` and `Option Explicit On`. This happens to be the same as the default when building a normal VB .NET project in Visual Studio. However, many developers don't include these statements and rely on the project-level defaults that are often changed to `Option Strict On` and `Option Explicit On` in the Project Properties dialog box. This isn't the approach taken by the VB .NET CodeDOM provider. You can't keep the provider from outputting these lines, and the explicit settings the provider outputs override any project settings you've set. You have to explicitly output the settings you want.

Use the `UserData` of the compile unit to specify values for VB .NET option settings. The code is as follows:

```
compileUnit.UserData.Add("AllowLateBound", False)
compileUnit.UserData.Add("RequireVariableDeclaration", True)
```

Astound your Visual Basic friends at cocktail parties. This use of `UserData` is basically impossible to find on the Web at the time I'm writing this book (I found it by reading the MSIL on a rainy Saturday afternoon).

Exploring Language-Specific Details

The tables in this section summarize features that may not be intuitive or easy to find. Table D-1 describes how a number of VB .NET features work with the CodeDOM. Table D-2 describes how several C# features work with the CodeDOM. These tables are subjective, and you'll probably extend these lists if you work

extensively with the CodeDOM. The longer list for VB .NET indicates that I've worked more with VB .NET in the CodeDOM.

Table D-1. Implementation of Selected VB .NET Features in the CodeDOM

VB LANGUAGE FEATURE	SUPPORTED	CODEDOM APPROACH
`Sub`	Yes	`CodeMemberMethod` with no `ReturnType` assigned.
`For`	No	Use `CodeIterator` to create a `Do While` loop.
`For Each`	No	Rewrite to use `CodeIterator`, probably along with variable assignments.
`Module`	No	Use a normal class with all shared methods.
`Me`	Yes	`CodeThisReferenceExpression`.
`MyBase`	Yes	`CodeBaseReferenceExpression`.
`MyClass`	No	Not supported in the CodeDOM.
`Select Case`	No	Restructure target code output to use conditionals (`If` statements).
`ElseIf`	No	Restructure target code to use nested conditionals.
& (string concatenation)	No	Use the addition operator.
`And`	Yes	`CodeBinaryOperatorType.BitwiseAnd`.
`Or`	Yes	`CodeBinaryOperatorType.BitwiseOr`.
`AndAlso`	Yes	`CodeBinaryOperatorType.BooleanAnd`.
`OrElse`	Yes	`CodeBinaryOperatorType.BooleanOr`.
`With`	No	Use explicit object references.
`WithEvents`	No	Events must be declared and attached.
`Handles`	No	Events must be declared and attached.
`Static` (local variable declaration modifier)[a]	No	None. The closest replacement is a class-level field.

[a] This feature preserves a variable's value across multiple calls to the function throughout the life of the application and is not related to the meaning of static in C#.

Table D-1. Implementation of Selected VB .NET Features in the CodeDOM, continued

VB LANGUAGE FEATURE	SUPPORTED	CODEDOM APPROACH
Project namespaces	No	You can't modify any project properties via the CodeDOM. Use explicit namespaces within the CodeDOM output. These will appear within the file. Use caution because VB .NET concatenates project and local namespaces.
Project imports	No	Existing project imports are respected, but you can't add new ones. Use file-level imports.
Inline declarations in loops[b]	No	Predeclare variables.
`Overloads`	Yes	C# implicitly assumes overloading in locations where VB .NET requires it. Add the `Overloads` attribute when needed to support VB .NET output.
Type suffixes on literals	No	Use explicit type casting.
`MustInherit`	Yes	`TypeAttributes = TypeAttributes.Abstract`.
`NotInheritable`	Yes	`TypeAttributes = TypeAttributes.Sealed`.
Nested class `Friend`	No	Although some of the type attributes would imply that this is supported, it isn't.
Nested class `Protected`	No	Although some of the type attributes would imply that this is supported, it isn't.
Nested class `Protected Friend`	No	Although some of the type attributes would imply that this is supported, it isn't.
Method `Friend`	Yes	`Attributes = MemberAttributes.Assembly`.
`Protected`	Yes	`Attributes = MemberAttributes.Family`.

[b] This is a new VB .NET 1.1 feature and allows code such as this:

```
For j As Int32 = 0 To 10
```

which is probably a moot point because the CodeDOM doesn't support `For` loops.

Table D-1. Implementation of Selected VB .NET Features in the CodeDOM, continued

VB LANGUAGE FEATURE	SUPPORTED	CODEDOM APPROACH
`Protected Friend`	Yes	`Attributes = MemberAttributes.AssemblyOrFamily`.
`Shadows` (overload by name)	No	`Attributes = MemberAttributes.New` creates `Shadows` in VB .NET and `new` in C#. These aren't equivalent, and this appears to be a bug.
`MustOverride`	Yes	`Attributes = MemberAttributes.Abstract`.
`NotOverridable`	Yes	`Attributes = MemberAttributes.Final`.
`Option Strict On`	Yes	Assign `False` to `AllowLateBound` in the namespace `UserData`.
`Option Explicit On`	Yes	Assign `True` to `RequireVariableDeclaration` in the namespace `UserData`.
`SyncLock`	No	I didn't discover a way to provide thread locking via the CodeDOM.
Assigning a return value via the method name	No	This isn't supported in the CodeDOM. Restructure your code to use the `Return` statement, recognizing that this represents different logic.
Nested namespaces	No	Not supported in the CodeDOM.
- (unary negation)	No	Use `-1*operand`.
<< (bitwise shift left)	No	Use `CType(LeftOperand * System.Math.Pow(2, RightOperand), System.Int32)`.
>> (bitwise shift right)	No	Use `CType (LeftOperand * System.Math.Pow(2, -1 *RightOperand), System.Int32)`.
`Not`	No	Use `operand = False`.
<> (value inequality)	No	Use `( LeftOperand = RightOperand ) = False`.

Table D-1. Implementation of Selected VB .NET Features in the CodeDOM, continued

VB LANGUAGE FEATURE	SUPPORTED	CODEDOM APPROACH
`Xor`	No	For Boolean operands use `(LeftOperand = RightOperand) = False`. I don't see a solution for bitwise `Xor`.
`[]`	No	The VB .NET `CodeDOMProvider` will escape keywords, but you can't specify that variable names be escaped.

Table D-2. Implementation of Selected C# Features in the CodeDOM

C# LANGUAGE FEATURE	SUPPORTED	CODEDOM APPROACH
`using` (namespace shortcutting)	Yes	`CodeNamespaceImport`.
`using` (automatic disposal of local variables)	No	Explicitly handle disposal.
Nested `internal`	No	Although some of the type attributes would imply that this is supported, it isn't.
Nested `protected`	No	Although some of the type attributes would imply that this is supported, it isn't.
Nested `internal protected`	No	Although some of the type attributes would imply that this is supported, it isn't.
`switch`	No	Restructure target code output to use conditionals.
`sizeof`	No	Not supported in the CodeDOM.
`stackalloc`	No	Not supported in the CodeDOM.
`lock`	No	I didn't find a way to do thread syncing via the CodeDOM.
Unsafe code	No	Not supported.
Inline declarations	No	Predeclare variables.
Open brackets on next line	Yes	`CodeGenerationOptions.BracingStyle="C"`.

Table D-2. Implementation of Selected C# Features in the CodeDOM, continued

C# LANGUAGE FEATURE	SUPPORTED	CODEDOM APPROACH
`@`-quoted strings	No	Not supported by the CodeDOM, except when escaped by the provider to handle keywords.
`[]`-non-array indexer	No	Although you could force the square brackets by treating the string as an array, in the resulting code this will work only in C#, not VB .NET.
Operator overloading	No	Restructuring code to use methods because operator overloading is likely to fail or give different results in VB .NET.
Nested namespaces	No	Not supported in the CodeDOM.
Unsigned integers and other non-CLS[a]-compliant types	No	You can create these types via the CodeDOM. However, the output will be illegal in other languages if you use any operators in relation to non-CLS-compliant types.
`-` (unary negation)	No	Use `-1*operand`.
`<<` (bitwise shift left)	No	`(System.Int32)(LeftOperand * System.Math.Pow(2, RightOperand))`.
`>>` (bitwise shift right)	No	`(System.Int32)(LeftOperand * System.Math.Pow(2, -1 *RightOperand))`.
`not`	No	`operand == false`.
`!=` (value inequality)	No	`(LeftOperand == RightOperand ) == false`.
`xor`	No	For Boolean operands use `(LeftOperand = RightOperand) = false`. I don't see a solution for bitwise `xor`.

[a] Common Language Specification (CLS) identifies a minimum set of types that languages must support via keywords. C# extends these to include additional types, including the unsigned integers.

Testing in All Target Languages

It's absolutely essential you test in all your target languages and understand nuances between the languages. Without this understanding, you may wind up with unexpected runtime errors or incorrect logic arising from language subtleties, such as the way C# overloads operators when working with intrinsic values. The following VB .NET code doesn't execute the `DoSomething` method:

```
Dim i As Int32 = 2
Dim j As Int32 = 3
If i/j = 0 Then
   DoSomething()
End If
```

But the corresponding C# code executes the `DoSomething` method:

```
int i = 2;
  int j = 3;
  if (i/j == 0 )
      Console.WriteLine("true");
```

In these code fragments, VB .NET returns `0.666` for `i/j` (to the precision of a `Double`), and C# returns zero. That's because VB .NET uses explicit floating point (`/`) and integer (`\`) division operators and C# overloads the single operator depending on the type of the operands. You'll also see a different reaction if `j` is zero. Because this is a floating-point division in VB .NET, you can successfully perform the comparison, but C#'s integer division throws an exception. Nuances of how operators work with intrinsic types is the most hazardous areas of crossing code over between the languages, but the many differences between the languages can result in different behavior for complex operations.

NOTE If you're using the CodeDOM to produce J# code, plan to do some extra testing and research because I haven't worked with the J# output and don't know what additional issues there are.

In addition to the subtle differences between code that appears identical, you can create dozens of different ways to output code that'll compile in one but not both languages—and that doesn't even touch on warnings because C# catches unused variables and inaccessible code, and the VB .NET compiler doesn't. It's essential that you test the compile in all target languages early and often to catch problems and adjust your habits. You also need to test the resulting code. Microsoft used the technology in a relatively narrow usage, and there aren't many other people out there shaking it down.

Index

Symbols

($) dollar sign, 582
$primarykey variable, 136
$primarykeyname variable, 136
$remaining variable, 145
& (string concatenation) operator, 701
&nbps (nonbreaking space character), 499
* (asterisk), 279, 355
(@) sign, 49, 174, 581, 588
(@*) attribute, 334
@Name attribute, 463
@-quoted strings feature, 705
[] feature, 704
[]-non-array indexers, 705
| (pipe character), 229, 360, 601
+= operator, 425
< character, 548
< operator, 416
<% delimiter, 496
<% Page...> tag, 472
<%# delimiter, 497
<< (bitwise shift left) operator, 703, 705
<> operator, 416
<> (value inequality) operator, 703
= (equals) operator, 415, 416
!= (value inequality) operator, 705
>> (bitwise shift right) operator, 703, 705
~ (tilde), 487
! directive, 602
/ (slash), 465, 594
- (unary negation) operator, 703, 705

A

abstract filenames, 218
abstract paths, 119, 238–39
AccessVia attribute, 313
AcquireRequestState event, 489
ActionItem class, 446, 447, 448
ActionItem object, 446
ActionMenuItem class, 446
AddExtensionObject, 225
AddFile method, 260, 263
AddHandler statement, 425
AddItemClick method, 663
AddKeyColumns method, 631
AddMenuStuff method, 661–62
AddObject template, 337
AddParameters method, 394
AddPrimaryKey method, 632
AddPrimaryKey template, 616
AddPrimaryKeyXML, 631
Address element, 588
AddUniqueConstraint method, 632
AddWithKey setting, 96
ADO.NET structures, 96
AllChildren element, 327
Allitemtype element, 304
AllObjects element, 304, 323, 335
AllowNulls attribute, 325, 583, 614
AllProperties element, 304, 325
AlternateProcess element, 224, 231
ancestor axis, 597
ancestor-or-self axis, 597
AnchorStyle property, 464
And operator, 701
AndAlso operator, 409, 701
anonymous type, 565
ANSI_NULLS option, 347, 350
application debugging, 266–70
application frameworks, 183–85
ApplyEdit method, 449
ApplyHash routine, 149
Apress Web site, 36
architecture, designing, 26
arrays
  assigning values, 171–72
  creating, 170
  declaring variables, 169–70
AS clause, 360
asp namespace, 494
ASP.NET, 472–507
  additional reading about, 507
  code generation strategies for, 482–505
    allowing string-free navigation, 486–91
    establishing derivation strategy, 482–85
    putting user controls to work, 492–93
    specifying UI details in UIInfo.XML, 492

developing for, vs. developing for Windows, 473–78
laying out pages, 478–81
overview, 472
server controls, 482
templates, 493–505
overview, 493–94
WebEdit.xslt, 500–505
WebSelect.xslt, 494–99
user controls, 481
Assembly element, 300, 318, 322–24
AssemblyAttribute template, 338
AssemblyFileName attribute, 125, 222, 223, 224, 245, 612
AssemblyInfo element, 300, 341
AssemblyInfo object, 301
AssemblyName attribute, 125, 224, 225, 245
AssignFromReader method, 396
Assignment class, 387, 388, 430
Assignment object, 310, 311
assignments, outputting, 167–68
Assignments table, 306
Assignment object, 310
asterisk (*), 279, 355
attribute axis, 597
attributeFormDefault attribute, 557
attributeName attribute, 548
attributes, 305, 546, 546–47
Attributes property, 45, 159, 696
AttributeUsage attribute, 440
automating
flow of database structure changes, 298
source control, 255–63
accessing SourceSafe wrapper object, 260–63
initiating source control tasks, 256–60
overview, 255–56

B

backing store, 148
base class, 190–92, 203
BaseEdit class, 457
BaseMain MDI form, 443, 444
BaseSelect class, 457
BaseSelectForm class, 447
basetype variable, 614
BaseTypes collection, 695
BaseUserControl class, 449
BaseXMLFileName attribute, 221, 232
benefits of code generation, 34–35, 215. *See also* code generation harness
binary expression, 167
binding of data, 428–29
BirthDate property, 91
bitwise shift left (<<) operator, 703, 705
BlankLineBetweenMembers property, 698
boolean directive, 605

 element, 472

 tag, 589, 590
BrokenRules system, 392, 414
BrokenRules.Assert method, 413
brute-force generation, 5, 147–55
creating classes, 150–55
creating entry-level method, 151–53
creating support template, 154–55
overview, 150
region methods, 153–54
and file streams, 148–49
generating “Hello World” via, 8–10
generating simple class via, 41–44
using IndentTextWriter, 150
brute-force templates, debugging, 280
BruteForceGeneration directive, 220, 221, 233, 234, 235, 678
btnAdd<childName>_Click method, 468
btnSave_Click event, 449
btnSave_Click method, 449
Build element, 242, 315, 318, 319, 348
Build File List directive, 244
BuildAction, 244, 253
BuildClasses, 132
BuildColumn element, 359, 360, 373
BuildGraph template, 11
BuildJoin template, 360, 361, 362–63
BuildMenus event, 447
BuildMenus method, 441, 444, 445
BuildNamespaceManager method, *654*
BuildProject, 263
BuildRecordset element, 316, 327, 355, 361
BuildSingleJoin template, 360, 362, 363
BuildSSPath method, 261
BuildTable element, 359, 360, 361
built-in templates, 580
BusinessBase class, 383, 386, 405, 406, 410
BusinessCollectionBase class, 383, 386
BusinessObject attribute, 456, 495
BusinessObject parameter, 458, 495
businessobject variable, 455
BusinessObjectAssemblyList method, 444
BusinessObjectCollection class, 191
BusinessPropertiesAndMethods template, 406, 408
ButtonColumn control, 498

C

C# programming language, 531–32
C++ programming language, 532
Can<Operation> method, 416
CanAdd property, 391–92, 657
CanCreate method, 393, 416
CanCreate property, 446
CanDelete method, 393
CanEdit property, 391–92
CanPrivileges template, 410, 416
CanRemove property, 391–92
CanRetrieve method, 393
CanSelect method, 416
CanUpdate method, 393
Caption attribute, 325, 437
Caption parameter, 98
cardinality, 566
careful naming, 66, 95, 99, 207
Catch clause, 262
ceiling directive, 605
CHECK_CONSTRAINTS view, 88, 630
Checked attribute, 116, 222, 223, 651
CheckIn method, 259, 260, 262
CheckOut attribute, 256
CheckOut method, 258, 260
CheckOutProject attribute, 256, 257
Child attribute, 324, 404
child axis, 597
child elements, 222–29
  MultiPass element, 226–27
  OutputRules element, 227–29
  overview, 222–23
  Parameter element, 229
  Process element, 224
  SinglePass element, 226
  Standard element, 223
  XSLTFiles element, 225
child relations, retrieving, 93
ChildCollection element, 324, 327, 338, 464, 468, 469
ChildCollection object, 327
childcollection variable, 469
ChildCollectionInfo object, 327
ChildField element, 83
ChildKeyFields element, 616
childnamespace, 468
childobject variable, 468
ChildOf attribute, 324
ChildTable element, 616
ChildTableName attribute, 327
chkIsDirty control, 462, 467
class creation, 38–39
  brute-force generation, 150–55
    creating entry-level method, 151–53
    creating simple class, 41–44
    creating support template, 154–55
    overview, 150
    region methods, 153–54
  via codeDom, 44–47
  XSLT code generation, 130–46
    creating entry-level template, 132
    creating high-level processing template, 132–35
    creating match templates, 141–42
    creating named templates, 135–37
    managing whitespace, 137–39
    specifying XSLT parameters, 131
    supporting stylesheets, 142–43
    using conditional variables, 140–41
    using looping, 140–41
    using recursive templates, 144–46
    XSLT templates, 47–50
ClassDeclarations template, 406, 460
ClassLevelDeclaration template, 140
class-specific code, 29
ClassViaBruteForce class, 41
ClassViaBruteForce.GenerateOutput method, 41
ClassViaCodeDOM method, 44
Class.xslt template, 47
clean nesting, 548
CloseSP template, 355, 357–58
CLR (Common Language Runtime) exceptions, 188
code attributes, 440
code generation, 3–51. *See also* benefits of code generation
  additional reading about, 51
  debugging generation process, 270–80
    generated output, 272–73
    generation harness, 275–76
    harness directives, 274
    metadata, 274
    templates, 275–80
  five principles of, 31–37
    benefits of code generation, 34–35
    benefits of handcrafted code, 34. *See also* metadata extraction
    implement one-click generation, 33–34. *See also* outputting code
    metadata is distinct step, 33
    overview, 31–32
    you have control over template, 32
    five steps of, 24–31
    collecting metadata, 27. *See also* metadata extraction
    designing architecture, 26–27
    implementation and testing, 30–31
    outputting code, 27–28. *See also as main entry*
    overview, 24–25
    writing handcrafted code, 28–30

future of, 527–37
developing new vision of code, 531–33
extracting metadata from requirements, 531
standardizing code generation interactions, 528–30
Visual Studio, 534–37
generating simple program, 7–20
overview, 7–8
via brute force, 8–10
via CodeDOM, 10–15
via XSLT templates, 15–20
mechanisms for, 5–6, 20–23
overview, 3–5
performing real-world code generation, 38–50
collecting XML metadata, 39–41
creating simple class, 38–39
generating simple class via brute force, 41–44
generating simple class via codeDom, 44–47
generating simple class via XSLT templates, 47–50
overview, 38
selling to management, 510–20
estimating code generation projects, 515
evaluating code generation solutions, 516–20
overview, 510–11
sending letter, 511–14
upcoming features, 520–27
key .NET 2.0 (Whidbey) features, 522–27
key SQL server features in Yukon, 520–22
code generation harness, 114–25, 216–36, 647–90
abstract file paths, 119
avoiding too many steps, 115–16
building dynamic data-driven application, 649–76. *See also* SimpleTreeBase class
overview, 649
SimpleMDIParentBase and HarnessForm, 649–50
child elements, 222–29
MultiPass element, 226–27
OutputRules element, 227–29
overview, 222–23
Parameter element, 229
Process element, 224
SinglePass element, 226
Standard element, 223
XSLTFiles element, 225
code generation tool, 117–19
directives, 220–22, 230–36
basic types of, 219–20
BruteForceGeneration directive, 233
CodeDOMGeneration directive, 234
CopyFiles directive, 234–35
CreateMetadata directive, 231
MergeMetadata directive, 232
NestedScripts directive, 236
RunProcess directive, 230–31
RunSQLScripts directive, 235
XSLTGeneration directive, 232–33
XSLTProcess directive, 230
executing the script, 676–90
calling MultiPass directive, 685–89
calling SinglePass directives, 683–84
GetMethodInfo method, 682–83
outputting log information, 678–79
RunGeneration class, 679–81
RunGenerationNode method, 681–82
ShouldRun method, 684–85
extending, 122–23
harness building blocks, 218
harness script, 123–25
overview, 16, 114–15, 215–16, 647–48
overwriting files, 121
providing parameters, 120–21
reusing, 264
script directives, 116–17
code generation mechanics, 125–75. *See also* XSLT code generation
overview, 125–26
target and metadata files, 126–30
code generation tool, 117–19
code serialization, 424
code snippets, 523
CodeArgumentReferenceExpression, 173, 174
CodeArrayCreateExpression, 170
CodeArrayIndexerExpression, 171
CodeAssignStatement, 174
Codebehind attribute, 485
Codebehind file, 479
codebehndProjectEdit class, 483, 485
CodeBinaryOperatorExpression, 174
CodeCompileUnit, 11, 44, 158, 159, 161, 164
CodeConstructor, 159, 160
CodeDOM, 691–706
assigning variables before use, 697
avoiding local static variables, 699
avoiding MyClass keyword, 699
comparing assignment and equality, 697–98

declaring base types and interfaces, 695–96
extending, 691–93
implementing interfaces, 696
language-specific details, 700–705
outputting Option Strict and Option Explicit, 700
outputting whitespace, 698–99
shortcutting namespaces, 694–95
testing in all target languages, 706
using regions and other compiler directives, 699
working with scope and type of members, 696–97
CodeDOM generation, 155–75
building CodeDOM graph, 164–72
accessing enums, 168–69
assigning array values, 171–72
building code structure, 165–66
creating arrays, 170
creating objects, 169
declaring array variables, 169–70
declaring variables, 165
outputting statements, 166
overview, 164
using CodePrimitiveExpressions, 164
enhancing, 535–36
“Hello World” via, 10–15
keyword use, 174
nature of output, 173–74
overview, 155–58
parentheses use, 174
simple class via, 44–47
snippets, 172–73
testing in all target languages, 174–75
UserData, 172
using object hierarchy, 158–63
expressions, 161–63
members, 160–61
overview, 158–59
statements, 161
types, 159–60
CodeDOM graph, 158, 159
CodeDOM templates, 280
CodeDOM.Compiler namespace, 14, 150
CodeDOMGeneration directive, 220, 221, 234, 235
CodeDOMGeneration method, 678
CodeDomProvider parameters, 11, 166
CodeDOMSerializer, 156, 205, 379, 424
CodeEntryPointMethod, 13, 160
CodeExpressionStatement class, 163
CodeFieldDeclaration, 169
CodeGenerationHarness.exe file, 118
CodeGenerationOptions method, 698
CodeGenerator, 12
CodeGenerator classes, 691–93
CodeGeneratorOptions object, 12
CodeMemberField object, 45
CodeMemberMethod, 159, 160
CodeMemberProperty, 46, 159
CodeMethodInvokeExpression, 163
CodeNamespace objects, 694
CodeNamespaceImport object, 694
CodeObject class, 536
CodeObject classes, 692–93
CodePrimitiveExpression, 164
CodePropertySetValueReferenceExpression, 46
CodeSnippetExpression, 173
CodeSnippetStatement, 173
CodeSnippetTypeMember, 173
CodeTypeConstructor, 159, 160
CodeTypeDeclaration, 45, 159, 695
CodeTypeMember, 159, 160, 161
CodeTypeRef object, 167
CodeTypeReference, 45, 46, 167, 169
CodeTypeReferenceExpression, 167
CodeVariableDeclarationStatement, 165
CodeVariableReferenceExpression, 165, 173, 174
collaboration-based programming, 513, 537–39
Collection class, 525
CollectionBase class, 133
CollectionChild attribute, 324
CollectionName attribute, 324
CollectionRoot attribute, 324
Column attribute, 360
Column element, 41, 42, 49, 74
COLUMN_DOMAIN_USAGE view, 88
COLUMN_PRIVILEGES view, 88
ColumnChanging event, 37, 204
ColumnInfo property, 201
columnName property, 141
columnNameColumnInfo method, 141
columns, 643
columns variable, 495
COLUMNS view, 88
ColumnSPList template, 359–60, 374
ComboBox control type, 437
CommaIfNotFirst template, 143
comment markers, 211–12
comments, 67
Common Language Runtime (CLR) exceptions, 188, 189, 190
CompanyName column, 98
Comparison template, 415
compiler errors, 266
Complex types, 76
complexType elements, 666
complexTypes method, 673–74

Component Object Model (COM) type library, 66
Component-based, Scalable, Logical Architecture. *See* CSLA (Component-based, Scalable, Logical Architecture)
compositors, 566
concat directive, 605
concat function, 497
conditional variables, 140–41
CONSTRAINT_COLUMN_USAGE view, 88, 630
CONSTRAINT_TABLE_USAGE view, 88
constraints, 89–93
Constructors method, 153
contains directive, 605
context, 594
ControlForTag method, 669
ControlPropertyInfo template, 436
Controls collection, 467
ControlTypeInfo template, 438
CopyFile directive, 234, 235, 244
CopyFiles directive, 220, 221, 234–35
CopyRootAttributes template, 455
count directive, 605
Create Solution tool, 285–87
create stored procedures, 64, 364–67
CreateBusinessObject method, 397
CreateMetadata directive, 117, 220, 221, 224, 231
CreateMetadata element, 226
CreateMetaData method, 621–23
CreatePattern attribute, 321
CreateXmlDeclaration method, 623
Criteria class, 394, 396, 397, 398, 411
Criteria object, 395, 397, 398, 411
CriteriaBase, 397, 398
CSLA (Component-based, Scalable, Logical Architecture), 26, 308, 378–79, 380–86
  four base models, 384–86
  isolating data structure details in, 381–82
  overview, 380
  process, 382–84
CSLAEditCollection.xslt template, 386
CSLAEditSwitchable.xslt template, 386
CSLAList.xslt template, 386
CSLANameValueList.xslt template, 386
CSLAServer.DataPortal class, 397
current directive, 605
Custom_AfterSelect stored procedure, 209
Custom_BeforeSelect stored procedure, 209
CustomerCollection class, 199
Customers class, 29, 41

D

data binding vs. data scattering, 428–29
Database attribute, 221, 222, 231
Database Extraction tool, 72
databases. *See also* ORM (Object Relation Mapping); SQL-92 databases (SQL Server)
  designing for easier code generation, 346
  as metadata source, 60–61
DataBind method, 466
Databind<ChildName> method, 469
DataGrid, 433, 462, 464, 466, 468, 498
DataGridTableStyle object, 469
dataNode variable, 665
DataPortal class, 383, 398
DataPortal object, 383
DataPortal_Fetch method, 383, 395, 398
DataPortal.Fetch method, 383
DataReader, 314, 395, 396
DataRows array, 630
DataSets, strongly typed, 36–37, 115, 535
DataSource method, 466
DataStructure element, 72, 102, 342, 622, 623
DataStructure node, 611
DataTable, 36, 71, 76–77
DataTableFromSQL method, 620, 637, 646
DateTime property, 437
DateTimePicker control type, 437
dbs namespace, 352
dbs:Name variable, 586
dbs:NETType variable, 586
dbs:Table node, 586
dbs:TableColumns element, 586
debugging, 265–80
  applications, 266–70
  generation process, 270–80
    generated output, 272–73
    generation harness, 275–76
    harness directives, 274
    metadata, 274
    templates, 275–80
  getting valid compile, 265–66
  overview, 265
  providing support for, 194
decoupling, 200
Default attribute, 326
default namespace, 557
DefaultPrivilege element, 321, 322
DefaultPrivileges attribute, 309
DefaultPrivileges element, 318, 319, 321
DELETE statement, 368, 369
delete stored procedures, 368–70
DeletePattern attribute, 321
derived class, 192–93, 199

derived methods, 203
descendant axis, 597
descendant-or-self axis, 598
descendants-or-self axis, 599
deserialization, 379
design tools, as metadata source, 64–65
designing architecture, 26, 26–27
designing metadata, 69–70
directives, 216, 667
directory structure, organizing files via, 236–40
  finding effective directory organization, 236–38
  managing template projects, 239–40
  using abstract and relative paths, 238–39
DirectoryInfo object, 248
div division, 465
DLLs (Dynamic Link Libraries), 65, 194, 382
document directive, 605
document function, 576
Document Object Model (DOM), 21
Document Type Definition (DTD), 65
DOM (Document Object Model), 21
DOMAIN_CONSTRAINTS view, 89
DOMAINS view, 89
DoSomething method, 706
DTD (Document Type Definition), 65
dtproperties table, 646
dynamic data-driven applications, building, 649–76. *See also* SimpleTreeBase class
  overview, 649
  SimpleMDIParentBase and HarnessForm, 649–50
Dynamic Link Libraries (DLLs), 65, 194, 382
dynamic loading, 391
dynamic stored procedure, 93
dynamic techniques, 123
dynamic user interface creation, 442–52
  dynamic menus, 443–49
  generic EditForm, 449–51
  handling primary keys, 452
  overview, 442
  using flow layout in WinForms, 451–52

E

/e:<elementName> switch, 75
Edit and Continue (E&C), 524
edit forms, generating
  building edit layout template, 458–69
    building InitializeComponent, 462–68
    creating event handlers, 468–69
    declaring parameters and variables, 458–59
    generating spit code region, 460–62
    outputting class declarations, 460
    using entry templates, 459–60
  building generated UserControl template, 469–70
  overview, 458
Edit method, 448
Editable subdirectory, 253
EditForm, 442, 447, 449–51
element-available directive, 605
elementFormDefault attribute, 557
ElementInfo element, 332
elementName attribute, 548
elements, 305, 340
elements, XML, 545–46
ElseIf statement, 701
Empty attribute, 326
Empty directories parameter, 287
EmptyForColumn template, 408
End Function construct, 532
End While construct, 532
EndCurrentEdit method, 449
Ended property, 313
Enterprise Templates, 6, 23
<entity>Edit class, 500
<entity>Select class, 494
entry-level method, 151–53
entry-level template, 132, 610
enums, 168–69
EnvDTE namespace, 68, 157, 167
equals (=) operator, 415, 416
Equals method, 393, 415
ErrorResponseEnum, 620
escaping, 549
estimating code generation projects, 515
evaluating code generation solutions, 516–20
EventHandler template, 468
EventHandlers template, 460
events, using to provide behavior, 203–5
exception reporting, 187–89
Exclude attribute, 249, 250, 251
Exclude directories parameter, 287
Explicit attribute, 314
expressions, 161–63
ExtDatabase DataTable variable, 640
extended properties, 98
ExtendedProperty element, 640, 642
eXtensible Markup Language. *See* XML (eXtensible Markup Language)
extension objects, 225
extension objects, XSLT, 591–92
ExtensionObject namespace, 610
external metadata sources, 63–64

extracting metadata. *See* metadata extraction
ExtractMetadata class, 84, 619–35
declaring MustOverride methods, 620–21
and FixName method, 612
implementing CreateMetadata method, 621–23
looping through all tables, 624
opening, 619–20
outputting check constraints, 630
outputting primary key information, 631–32
outputting relations, 632–34
outputting table constraints, 631
outputting XML for column privileges, 629
outputting XML for each column, 626–29
outputting XML for each table, 624–26
overview, 619

F

false directive, 605
Family element, 600
Family node, 600, 601
FamilyAndAssembly method, 696
Fetch method, 383, 396
field variable, 45
fields, 161, 406
fil:Dir element, 490
File element, 247
file header, XSLT, 577–78
file paths, abstract, 119, 238–39
file streams, 148–49
FileCodeModel, 157
FileInfoXML method, 249
FileList element, 246–47, 248
FileListXML, 247–49
fileName, 120, 154
fileName stylesheet, 143
fileNode parameter, 252
FileOpen method, 151, 154
FileOpen template, 133, 134, 143
FilePath element, 119, 218, 238, 244
FilePaths element, 218
Files element, 243
files, organizing via directory structure, 236–40
finding effective directory organization, 236–38
managing template projects, 239–40
using abstract and relative paths, 238–39
FileStream, 148
FileTools class, 245, 248
Fill method, 136, 137, 637, 638
FillAddMenu method, 661
FillMetaDataSet method, 623, 639, 640, 643
FirstName field, 182
FirstName property, 183
five principles of code generation, 31–37
benefits of code generation, 34–35
benefits of handcrafted code, 34. *See also* metadata extraction
implement one-click generation, 33–34. *See also* outputting code
metadata is distinct step, 33
overview, 31–32
you have control over template, 32
five steps of code generation, 24–31
collecting metadata, 27. *See also* metadata extraction
designing architecture, 26–27
implementation and testing, 30–31
outputting code, 27–28. *See also as main entry*
overview, 24–25
writing handcrafted code, 28–30
FixName method, 591, 612, 613, 624
floor directive, 606
flow constructs, XSLT, 583–87
flow layout for WinForms, 527
fn_listextendedproperty function, 639
following axis, 598
following-sibling axis, 598
Folsom, Nancy, 268
Foo property, 205
For Each statement, 701
For statement, 701
FORCEOVERWRITE phrase, 206
foreign key tables, 92
FOREIGN_KEY constraint, 634
Form attribute, 492
Format event, 426
format-number directive, 606
FormInfo template, 455, 456
Forms element, 438
frameworks, 183–84, 193, 528–29
FreeForm attribute, 108, 434
freeform metadata, creating, 102–7
creating column details, 102–3
defining stored procedure intent, 103
managing freeform metadata files, 104–7
overview, 102
providing UI information, 103–4
FreeForm="true" attribute, 107
Friend template, 135
FullAlphaName property, 29
FullName attribute, 247
function-available directive, 606

G

gather, 668
Gather method, 669, 671, 674–76
GatherAndRemoveExisitngUserControl method, 668
gathering, 428
gen<objectName>Edit class, 468
gen<objectName>Select class, 458
genCSLAEditCollection.xslt template, 386
genCSLAEditSwitchable stylesheet, 403–17
  checking privileges, 416–17
  comparing values, 415–16
  declaring the class, 406–8
  high-level processing template, 405–6
  opening declarations and entry-level template, 404–5
  outputting for business properties and methods, 408–10
  outputting property procedures, 410–15
  overview, 403
genCSLAEditSwitchable.xslt template, 386, 401–3
genCSLAList.xslt template, 386
genCSLANameValueList.xslt template, 386
genCustomerCollection class, 199
genDateTime parameter, 120, 143, 154
genEdit stylesheet, 466
Generate attribute, 309, 318, 321
GenerateClassViaCodeDOM method, 44
GenerateCodeFromCompileUnit method, 12
generate-id directive, 606
GenerateOutput method, 8, 41
GenerateViaCodeDOM method, 11, 12
GenerateViaXSLT method, 17, 47, 257, 683–84
Generation class, 224
GenerationForm method, 649, 657
GenerationSupport assembly, 250
generics, 524–25
GenInputMetadata.xml file, 437
GenInput.xml, 299, 398
genNavigate class, 487, 490, 491
genNavigateURL class, 487, 490
genProject, 401–3, 482
genProjectAssignment class, 388
genResourceAssignment class, 388
genServerTransfer generated class, 487, 490, 491
genShippers node, 250
GenType subdirectories, 237, 238, 267
Get attribute, 313
Get method, 46
Get property, 412
Get<BusinessObject> method, 447
Get<BusinessObjectName> method, 383
GetAllTablesXML, 623, 624
GetAttributeOrEmpty method, 151
GetCheckConstraintsXML method, 630
GetChildRelations mode template, 616
GetColumnXML method, 626
GetConnection method, 637
getdate( ) method, 91
GetExtraStoredProcInfo method, 642
GetExtraTableInfo method, 642
GetFileListXML method, 245, 248
GetHashCode method, 393
GetItem method, 262
GetItemStatus method, 262
GetMembers method, 434
GetMenuInfo method, 659–60
GetMethodInfo method, 682–83
GetObjectType, 398
GetOLEDBSchemaTable method, 61
GetPageName template, 490, 491
GetParentRelations template, 617
GetPKeyName template, 611, 615
GetPlural method, 404
GetPrimaryKeyConstraint method, 631
GetPrivilegesXML method, 629
GetProject method, 383, 447
GetProviderSpecificXMLNode method, 623, 639, 640
GetSchemaForNode method, 657, 658
GetSingular method, 404
GetSourceCodeHash method, 535
GetSpecifie Type method, 682
GetStatements collection, 46
GetStream method, 151
GetTableConstraintsXML method, 631
GetTables method, 646
GetTableXML method, 624, 626
GetType method, 161, 165, 167
GetXMLSchema method, 75
GetXSLTParams method, 686
“ghost” file (in Solution Explorer), 253
Global Unique Identifiers (GUIDs), 241–42, 282, 285, 305, 306, 346, 415
GotDotNet Web site, 67
grandparent, 595
Grant statement, 321, 357
Grantee attribute, 322
GroupBox control, 464, 465, 466, 467, 673
GUIDs (Global Unique Identifiers), 241–42, 282, 285, 305, 306, 346, 415

H

handcrafted code, 20, 28–30
  application frameworks, 183–85
  benefits of, 34
  creating task-based categories of, 185–94
    incorporating exception reporting, 187–89
    overview, 185
    providing class-specific code functionality with derived class, 192–93
    providing debug support, 194
    providing reusable functionality with base class, 190–92
    running startup code, 185–87
    using code with insufficient or interspersed patterns, 190
    using components, 189
    using utility methods, 189–90
  mechanisms for isolating, 196–97. *See also* inheritance
  overview, 179–80
  and stored procedures, 207–10
  using comment markers, 211–12
  using JScript, 210–11
  vs. using metadata, 180–83
  using regions as alternative for, 205–7
  working with, 194–96
Handles clause, 157, 425
Handles clause, 701
harness. *See* code generation harness
harness directives, 274
Harness9.xml harness script, 457
HarnessForm, 649–50
hash mechanisms, 534–35
HashOutput attribute, 223, 347
Header control, 479
"Hello World" program
  generating via brute force, 8–10
  generating via via CodeDOM, 10–15
  generating via XSLT templates, 15–20
HelloWorldViaCodeDOM.BuildGraph method, 13
hidebysig modifier, 696–97
HideInUI attribute, 323
hiding, 424
high-level processing template, 132, 611
Home method name, 491
html directive, 578
HTML (Hypertext Markup Language), 68, 210
HTTPApplication object, 489, 490
HTTPRequest, 473, 476
Hypertext Markup Language (HTML), 68, 210

I

i variable, 171
IBusinessObject interface, 441
id directive, 606
IDataReader interface, 314
identity (auto-increment) primary keys, 346
IDictionary object, 172
IDisposable class, 256
IDoNotUseInUI_<Whatever> interface, 381
IExtractMetadata interface, 618, 619, 621
IFullNameProvider interface, 183
IList interface, 428
image type, 141
implementation of code generation, 30–31
ImplementationTypes method, 696
import parameter, 154
Imports element, 13, 15, 242, 243
Imports list, 405
imports parameter, 143, 145
Imports statement, 144, 145, 154, 243, 694
Imports System statement, 7, 143
Imports/using statements, 695
Include element, 247
Include node, 243, 251, 252
IncorporateInfo.xslt, 341
IncorporateTableInfo stylesheet, 373
incriteria parameter, 411
Indent property, 150
IndentedTextWriter, 12, 22, 41, 42, 43, 693
IndentTextWriter, 150
inference, 74
information schema views, 86–93
  modifying mappings, 93
  overview, 86
  specific schemas, 87–89
  and SQL-92 terminology, 86–87
  using constraints, 89–93
INFORMATION_SCHEMA view, 635
inheritance, 198–205
  instantiating most derived class, 199
  and Object-Oriented Programming fundamentals, 198
  overview, 198
  using derived methods to extend behavior, 203
  using events to provide behavior, 203–5
  using validation as example of overriding behavior, 200–203
Inherits attribute, 405, 479
Inherits class, 479

InitializeComponent method, 423, 424, 462–68
Inline declarations, 704
Inline declarations in loops, 702
inline SQL, 93
InnerException, 280
InputFileName attribute, 223
INSERT statement, 364, 366
insignificant whitespace, 138, 550
IntelliSense, 76
intimate merges, 102
invocation list, 424
IO.Path class, 17
IsAutoIncrement, 359, 366
IsCheckedOut method, 262
IsDirty flag, 409
IsEnum property, 159, 169
IsInRole method, 410
IsInterface property, 159
IsLoaded property, 391
IsLookup attribute, 323, 437
IsPostBack property, 476
IsPrimaryKey attribute, 326
IsRequired attribute, 325
IsRoot code attribute, 445
IsStruct property, 159
iSum variable, 167
IsValid flag, 409
IsValid property, 392
iValue variable, 171

J

Join attribute, 363
JOIN clause, 362, 363
Join element, 361, 363
JoinOn element, 317
JoinTable attribute, 363
JScript, 210–11

K

KADDrivingMetada.xsd, 118
KADGen namespace, 619
key directive, 606
KEY_COLUMN_USAGE view, 88, 92, 93
KeyColumns table, 631
keywords, 174

L

/l: <languageName> switch, 75
lang directive, 606
Large attribute, 106
last directive, 606
LastName field, 182
LastName property, 183
lasttabindex variable, 464
layout class, 432
lazy loading, 391
lbl literal, 463
Lhotka, Rockford, 377–78
listeners, collection of, 424
listXML variable, 251
Load event, 526
LoadFile method, 651–52, 652–55
LoadFreeXML method, 654, 655
LoadFreeXMLData method, 655–56
LoadFreeXMLDataNode method, 655–56
LoadTreeMenuXSD method, 658–59
LoadVBFiles method, 251, 252
local static variables, 699
local-name directive, 606
Location property, 463, 466
lock feature, 704
logical tables, 96
Login method, 444
lookup values, 385
LookupNamespace method, 654
looping, 140–41
looupPrefix parameter, 625
lstRules control, 462, 467

M

madeFile variable, 257, 683–84
Main form, 444
Main MDI form, 443
MainBase MDI form, 444
MakeNewNode method, 663–64
MakeNewNodeFromXSD method, 666–67
MakeNewVBIncludeNode, 252
MakeTreeNode method, 656–58, 664–65
MakeUpperCase template, 579
ManagerEmployeeID, 346
MapDataStructure attribute, 307, 308, 320, 336
mapping. *See* ORM (Object Relation Mapping)
MappingFileName attribute, 221
MappingRoot element, 308, 309, 310, 318, 319–21, 322, 328, 334
MarkChecked method, 656
MarkWriteAndClose, 257
match attribute, 580
match clause, 581
match templates, 132, 141–42
MaxLength attribute, 326, 359
maxOccurs="unbounded", 567
mBaseMenus field, 659
mBusinessObjectType type, 447
mChildAutos collection, 660

mChildMenus field, 659
mCurrentOutput methods, 680–81
mCurrentTreeNode class field, 660–61
mDataSet variable, 620
MDI form, 457
MDIParent form, 187
Me keyword, 701
mechanisms for code generation, 5–6, 20–23
mEditUCType, 447
members, 160–61
mEnded class, 314
Merge File Lists directive, 245
MergeFreeForm class, 108
MergeMetadata directive, 220, 221, 232, 249
MergeNode method, 109
merging metadata, 107–10
MergingFileName attribute, 221
MergingXMLFileName file, 232
metadata. *See also* metadata extraction
  controlling visibility of, 299–300
  creating, 333
  creating code generation–friendly metadata, 298–99
  creating freeform metadata, 102–7
    creating column details, 102–3
    defining stored procedure intent, 103
    managing freeform metadata files, 104–7
    overview, 102
    providing UI information, 103–4
  debugging, 274
  designing, 69–70
  as distinct step, 33
  entering manually, 62
  extracting from requirements, 531
  vs. handcrafted code, 180–83
  merging, 107–10, 333
  SQL server –specific, 97–98
  standardizing, 529
  and WinForms, 432–39
    adding manual metadata for, 437–39
    merging additional metadata, 433–34
    overview, 432–33
    reflection-based metadata for UIs, 434–35
    transforming metadata for easy UI generation, 435–37
metadata extraction, 27, 39–41, 53–111
  additional reading about, 111
  creating freeform metadata, 102–7
    creating column details, 102–3
    defining stored procedure intent, 103
    managing freeform metadata files, 104–7
    overview, 102
    providing UI information, 103–4
  establishing own XML design guidelines, 70–72
  merging metadata, 107–10
  and metadata design, 69–70
  metadata sources, 56–68. *See also* XSD
    databases, 60–61. *See also* SQL-92 databases (SQL Server)
    design tools (such as UML), 64–65
    entering metadata manually, 62
    existing applications and source code, 66–68
    external, 63–64
    overview, 56–58
    schema definitions, 59–60
    Web Services, 62
  overview, 53–55
  reasons for extracting, 68
  tools for, 72–73. *See also* SQL-92 metadata extraction tool; XSDToFriendlyXML.xslt
MetaDataRoot element, 622
Method Friend scope, 702
MethodName attribute, 222, 224
Microsoft Intermediate Language (MSIL), 6
middle tier business objects
  genCSLAEditSwitchable stylesheet, 403–17
    checking privileges, 416–17
    comparing values, 415–16
    declaring the class, 406–8
    high-level processing template, 405–6
    opening declarations and entry-level template, 404–5
    outputting property procedures, 410–15
    outputting template for business properties and methods, 408–10
    overview, 403
  GenInput.xml structure, 398
  genProject business object target, 401–3
middle tier templates, 377–419
  changing base architecture, 387–98
    changing private scope to protected scope, 389–90
    creating predictable names, 394
    deriving business objects, 387–88
    handling invalid values, 391–92

handling special types, 396–98
including parent primary key in child objects, 390
overriding ToString, Equals, and GetHashCode, 393
providing CanAdd, CanRemove, and CanEdit properties, 391–92
providing parameterless constructors, 388–89
providing single location for state changes, 395–96
supporting multiple criteria, 394–95
tracking whether the object is loaded, 391
using internal methods for assignments, 395
Component-based, Scalable, Logical Architecture (CSLA), 380–86
CSLA process, 382–84
four base models, 384–86
isolating data structure details in, 381–82
overview, 380
overview, 377–79
minOccurs attribute, 583
minOccurs=0 attribute, 77, 583, 614
Miscellaneous node, 240
MissingSchemaAction property, 96
MissingSchemaAction=AddWithKey, 614
mnu variable, 446
mObject data, 451
mode attribute, 578, 580, 581, 585
modeName parameter, 630, 641, 644
"Modified Outside the Source Editor" dialog boxes, 217–18
Module feature, 701
msdata prefix, 555, 556, 562
MSDN libraries, 65
mSelectFormType type, 447
mSelectPatterns variable, 620
mSetSelectPatterns variable, 620
msgstart variable, 411
MSIL (Microsoft Intermediate Language), 6
mstns prefix, 555
msxsl:nodeset function, 342
MultiPass directive, 272, 685–89
MultiPass element, 124
attributes of, 116, 220, 223
and BruteForceGeneration directive, 233
description of, 223
explanation of, 226–27
and SinglePass element, 222
and XSLTGeneration directive, 232
MultiPass method, 678, 685–87
MustInherit class, 84, 199, 311, 405, 423, 619
MustInherit keyword, 702
MustOverride keyword, 703
MustOverride method, 256, 620–21, 637–39
mXML tyle field, 652–53
mXMLDoc variable, 620
MyBase keyword, 701
MyBase.OnLoad method, 426
MyClass keyword, 699, 701

N

Name attribute, 81, 581, 596, 613, 622, 640
name directive, 606
Name parameter, 132, 458
Name property, 463, 466
named templates, 132, 135–37
named type, 565
Namespace attribute, 308, 320
namespace axis, 598
namespaces, 551–59
default namespace, 557
namespace prefixes, 555–56
namespace scope, 552–53
in .NET, 558–59
overview, 551–52
shortcutting, 694–95
tempuri.org, 553–55
NamespaceURI, 612
NamespaceURI attribute, 223, 225
namespace-uri directive, 606
name-value pairs, 418
NameValueList class, 386
naming delimiters, 99–100
naming rules, 100–101
Navigate class, 487, 488, 489
NavigateURL class, 487, 489
NavigateURL property, 478
NavigateURL template, 491
Navigation class, 490
NestedScript directive, 222
NestedScripts directive, 235, 236
.NET 2.0 (Whidbey) features, 522–27
NETType attribute, 82, 141, 326, 437
NETType property, 437
NewElementWithName method, 624, 626
NewLabeledControl method, 671, 672–73
NewLine template, 142, 355
NewSelectForm method, 447
NoChildCollections element, 305, 310
node variable, 45
nodeItSection variable, 680

nodeList variable, 45
nodes, 565
nodeSelect, 120
node-set directive, 580, 606
nodeTemp variable, 153
NoGen attribute, 310, 455, 456
Noitemtype element, 305
normalize-space directive, 606
normalize-space function, 145
not function, 606
Not operator, 703
not operator, 705
NotInheritable keyword, 702
NotOverridable keyword, 703
NoUI .NET code attribute, 435
NoWinUI .NET code attribute, 435
number directive, 607

O

/o:<directoryName> switch, 75
obj variable, 163
Object element, 336, 466, 468, 469
  and AllObjects element, 323
  and Assembly elements, 322
  attributes supported by, 324
  and ORM, 337, 338, 340, 341
  and overriding inferred details for customization, 300
  and stored procedures, 318
Object Management Group (OMG) Web site, 65
Object mode template, 459, 460
Object parameter type, 489
Object Relation Mapping. *See* ORM (Object Relation Mapping)
ObjectAttribute template, 338
ObjectAttributes template, 335, 336, 337
ObjectAttributesUpdate template, 338
ObjectContents template, 335, 337
ObjectInfo element, 300, 304, 324, 332, 341
objectname variable, 495
<objectName>Select class, 458
Object-Oriented Programming (OOP), 198
objects, creating, 169
ObjectType property, 397
Obsolete attribute, 106
OIM (Open Information Model) standard, 79
OLEDBDataReader class, 396
OMG (Object Management Group) Web site, 65
ON clause, 355
On<EventName> method, 425
OnBtnCancelClick method, 426
OnBtnSaveClick event, 449
OnBtnSaveClick method, 449
OnCancel method, 426
one-click generation, 33–34. *See also* outputting code
OnFooChanged method, 205
OnLoad event, 484
OnLoad method, 444, 526
OnTreeViewA Popup method, 661
OOP (Object-Oriented Programming), 198
op_equality method, 415
op_inequality method, 415
Open Information Model (OIM) standard, 79
OpenSP template, 354, 356–57, 369, 371
OpenSP utility template, 356
Operator overloading feature, 705
Option Explicit, 678, 700, 703
Option Explicit statement, 143
Option Strict, 9, 172, 174, 526, 678, 700, 703
Or operator 701
Orders class, 41
Orders element, 567, 568
Ordinal attribute, 222, 223, 325, 327
OrElse operator 701
organizing files via directory structure, 236–40
  finding effective directory organization, 236–38
  managing template projects, 239–40
  using abstract and relative paths, 238–39
orgtablename variable, 371
OriginalName attribute, 80, 81, 134, 366, 613
origtablename variable, 366, 369
orm namespace, 352
ORM (Object Relation Mapping), 295–318
  automating flow of database structure changes, 298
  benefits of, 296–301
  controlling metadata visibility, 299–300
  creating code generation–friendly metadata, 298–99
  describing complex stored procedures, 299
  downloading scripts, 343
  morphing ORM into generation-friendly metadata, 328–43
    creating metadata, 333
    merging metadata, 333
    overview, 328–29

processing IncorporateInfo.xslt, 341
processing ORMBuildOne.xslt, 338–40
processing ORMBuildTwo.xslt, 340
processing ORMFilterForGen.xslt, 341–42
processing ORMObjectOne.xslt, 338
processing ORMObjectTwo.xslt, 338
ORM file examples, 301–18
applying different stored procedure privileges, 315–16
defining special types, 312–15
example database, 305–6
generating joined tables, 316–17
limiting children, 310–11
managing responsibility, 304–5
ORM holding object details, 303
simplest version of ORM, 306–8
specifying handcrafted stored procedures, 317–18
specifying individual properties, 312
specifying minimal stored procedure information, 308–9
unique or shared child objects, 311
overriding inferred details for customization, 300–301
overview, 295–96
setting characteristics of generated code via ORM flags, 301
simplifying, 297–98
structure and attributes, 318–27
overview, 318–19
setting default privileges, 321–22
specifying assembly details, 322–27
specifying MappingRoot, 319–20
orm: Privileges collection, 356
orm:Build element, 354
orm:BuildColumn element, 360
ORMBuildOne.xslt stylesheet, 338–40
orm:BuildRecordset element, 360
ORMBuildTwo.xslt stylesheet, 340, 342
ORMColumnSPList template, 359
ORMFilterForGen.xslt, 341–42
ORMJoinList template, 360–62
orm:Object element, 219, 310, 455, 463, 495, 503
ORMObjectOne.xslt, 338
ORMObjectTwo.xslt, 338
orm:Property element, 356, 434, 463, 502
orm:SafeDataProperty element, 314
orm:SPSet element, 354
OutputFile attribute, 223
OutputFilePattern attribute, 124, 223, 227, 235
outputFilePattern method, 688
OutputFileType attribute, 223
OutputGenType attribute, 121, 223, 227, 347, 524
OutputProjectFile, 244
OutputRules element
attributes of, 223, 256, 257, 347
description of, 223
explanation of, 227–29
outputting code, 27–28, 113–77. *See also* brute-force generation; code generation harness; CodeDOM generation
code generation mechanics, 125–75. *See also* XSLT code generation
overview, 125–26
target and metadata files, 126–30
overview, 113–14
OutputXMLFileName attribute, 221
Overloads keyword, 702
Overloads modifier, 696–97
Overridable class, 84, 85
Overridable keyword, 193
overridableOnBtnAdd<childName> method, 468
overrides, 249–50
Overwrite directive, 228
Overwrite rule, 228
overwriting files, 121

P

Page attribute, 492
Page directive, 479, 485
Page object, 489, 490
ParamArray, 592
Parameter attribute, 223, 287
Parameter child element, 229
Parameter element, 223, 229
PARAMETERS view, 88
parent axis, 598
parent relations, 92
parentheses, 174
ParentKeyField element, 83
ParentObject element, 338
Parse event, 426
partial classes, 526–27
Pascal casing, 71
paths, 238–39
pipe character (|), 229, 360, 601
pKeyName variable, 611
Plural attribute, 81
PluralName attribute, 134
position directive, 607
position function, 463, 466, 589

preceding axis, 598
preceding-sibling axis, 598
Price property, 160
primary keys, 92, 346
PrimaryKey element, 136
primaryKey node, 154
primaryKeyName variable, 153
Principal object, 410
principles of code generation. *See* five principles of code generation
Private attribute, 106
PrivateImplementationType method, 696
privileges parameter, 357
prjXML file, 251
Process attribute, 151
Process directive, 124, 125
Process element, 222, 224, 233, 274
programs
  generating simple programs, 7–20
    generating "Hello World" via brute force, 8–10
    generating "Hello World" via CodeDOM, 10–15
    generating "Hello World" via XSLT templates, 15–20
    overview, 7–8
  as metadata source, 66–68
Project Assignment class, 430
project files
  building, 240–54
    directing process, 244–46
    FileList, 246–47
    FileListXML, 247–49
    overview, 240–41
    project file structure, 241–44
    providing overrides, 249–50
    running Template, 250–53
  maintaining by hand, 253–54
Project imports feature, 702
project life cycle, 539–40
project management, 280–87
  carrying experience into next applications, 283–87
  developing strategy for reuse, 283–84
  overview, 280–81
  performing postmortem, 282–83
  template projects, 239–40
  using existing project, 281–82
Project namespaces feature, 702
Project node, 227
Project object, 310, 390, 395, 455
Project Properties dialog box, 243
ProjectAssignment class, 387, 388, 390
ProjectAssignment object, 310, 311, 390
ProjectEdit class, 485, 500
ProjectID, 355, 390
ProjectResource object, 394
ProjectResources collection, 396
ProjectSelect class, 494
ProjectSupport class, 250
property attribute, 437
Property element, 300, 301, 313, 324, 325, 326, 452
Property Object, 325, 326
Property template, 408
PropertyControlInfo template, 454, 455
PropertyInfo element, 300, 313, 332
PropertyInfo object, 301, 325, 326
propget variable, 412
propset variable, 412
Protected Overridable method, 203
prototype, code generation, 35
Public template, 135
PublicAndFriend region, 153
PublicAndFriend template, 136

Q

Query Analyzer, 79

R

Read method, 395
Read statement, 396
ReadOnly attribute, 106, 324, 411, 452, 502
ReadOnly property, 452
readonly variable, 411, 412
ReadOnlyCollectionBase class, 386
Recordset element, 328
recursion, debugging, 279–80
recursive templates, 144–46
RecursiveImports template, 144, 145, 146
refactoring, 523–24
References element, 242
referential constraints, 92
REFERENTIAL_CONSTRAINTS view, 88, 92, 93, 632, 634
ReferentialConstraints table, 633, 634
reflection, 66, 381
reflection-based metadata, 434–35
Reflection.Emit namespace, 6
region methods, 153–54
regions, 205–7
Register directive, 479
relative paths, 238–39
remaining variable, 145, 279, 280
Replace(oldGUID,newGUID) method, 285
ReportsOnly attribute, 106
RequiredFieldValidator, 503
resource assignment, 311
Resource Assignment class, 430
Resource object, 310

ResourceAssignment class, 387, 388
ResourceAssignment object, 311
ResourceAssignments collection, 310
Response property, 490
Response.Redirect method, 477, 478, 486
Result property, 447, 448
ResumeLayout method, 467, 468
retrieve stored procedures, 350–63
  overview, 350
  retrieve stored procedure stylesheet, 352–56
  retrieve target, 350–52
  supporting stored procedure templates, 356–63
    BuildJoin template, 362–63
    BuildSingleJoin template, 363
    closing stored procedure with CloseSP template, 357–58
    opening stored procedure with OpenSP template, 356–57
    ORMJoinList template, 360–62
    outputting columns with ColumnSPList template, 359–60
    outputting parameters with SPParameters template, 358–59
RetrievePattern attribute, 320
RETURN statement, 355
Rich Text Format (RTF), 68
Rights attribute, 322, 356
Role property, 312
RoleName property, 433, 434, 437
Root attribute, 404, 441, 442
Root directory parameter, 287
Root element, 324
RootAttribute class, 441
round directive, 607
ROUTINE_COLUMNS view, 88
ROUTINES view, 88
rowColumn parameter, 627
rows, 347
RTF (Rich Text Format), 68
rules, XML, 547–48
runall variable, 335, 336, 455, 456
RunGeneration class, 679–81
RunGenerationNode method, 678, 680, 681–82
RunProcess directive, 221, 229, 230–31, 245, 248, 286
RunProcess element, 226
RunSP element, 317, 318, 340, 354, 356
RunSPParam element, 354, 359, 371
RunSQL directive, 348
RunSQL process, 347
RunSQLScript method, 678, 685
RunSQLScripts directive, 220, 222, 235, 236
RunStoredProc method, 620, 637, 638, 646
runtime application interfaces, 529–30

S

SafeDataProperty element, 314
SafeDataReader, 314, 396
SafeDataRetrieve element, 314
Save method, 393, 449, 451
Scatter method, 670–71
scatter operation, 668
scattering of data, 428–29
schema definitions, 59–60
schemaLocation attribute, 570, 573
schemaLocation property, 654
Schemas property, 569
SCHEMATA view, 87
script directives, 116–17
ScriptName attribute, 222
Section element, 218, 557, 558
sections, 116, 667
Seek method, 149
select attribute, 278, 580, 581, 587, 589
Select Case statement, 672, 701
select clause, 358, 581, 588
Select filter, 641
SELECT literal, 355
Select method, 630
SELECT statement, 208, 355, 359, 363, 374, 643
select stored procedure, 94
SelectFile attribute, 124, 223, 226
selectFile variable, 688
selection file, 121
SelectNamespace attribute, 124, 223, 226
SelectNode method, 151
SelectNodes method, 42, 593, 594
SelectNSPrefix attribute, 124, 223, 226
selectPatterns, 621, 622
SelectSingleNode method, 109, 151, 153, 593, 594, 658
SelectXPath attribute, 124, 223, 272
selectXPath variable, 688
self axis, 599
self-describing data, 543
Serializable attribute, 442
serialization, 205, 379
Server attribute, 221, 222, 231
Server DataPortal class, 383
Server property, 490
Server.DataPortal class, 396
ServerName property, 621, 636
ServerTransfer class, 487, 488, 489
Server.Transfer method, 478, 486

Set attribute, 313
SET clause, 475
SET FMTONLY ON switch, 95
Set<PropertyName> method, 390, 415
SetInventory method, 160
SetParameters method, 689
SetSelect stored procedures, 94, 309, 372–74, 494
SetSelectColumn element, 359, 360, 374
SetSelectPattern attribute, 320
setSelectPatterns, 621, 622
SetSelectWhereClause element, 373, 374
SetStatement collection, 46
Settings element, 242, 243
Settings node, 242
Shadows, 697, 703
ShouldRun method, 257, 678, 684–85, 688–89
Show method, 651–52
significant whitespace, 138, 550
SimpleDataContainer.xslt stylesheet, 130, 142–43
SimpleMDIParentBase, 649–50
SimpleTreeBase class, 651–76
  adding elements, 663–65
  displaying elements, 667–76
    gathering data from controls, 674–75
    instantiating right control, 669
    removing old control, 669
    scattering data into controls, 670–74
  LoadFile method, 652–55
  LoadFreeXMLData method, 655–56
  loading the schema, 658–60
  MakeTreeNode method, 656–58
  making generic node, 665–67
  popping up add menu, 660–62
  Show method, 651–52
SimpleXMLNodeUserControl class, 674–75, 676
SinglePass directive, 230
SinglePass element, 223, 226, 250, 683–84
singleton categories, 29
Singular attribute, 81
SingularName attribute, 134, 135, 137, 586, 613
singularName variable, 152
Size property, 466
sizeof keyword, 704
Skip attribute, 105
skipStoredProcs parameter, 621
slash (/), 465, 594
SmartDate class, 312, 313
SmartDate object, 313
snippets, 172–73
Solution Explorer. *See also* Visual Studio
  deleting files, 253
  finding files, 238
  opening XSLT templates, 239
  subdirectories, 238
  updating Solution Explorer list, 237
Solution filename parameter, 287
source code
  getting metadata directly from, 67–68
  as metadata source, 66–68
source control, 254–65
  automating, 255–63
    accessing SourceSafe wrapper object, 260–63
    initiating source control tasks, 256–60
    overview, 255–56
  extending, 263–64
  overview, 254–55
SourceBase, 258, 262
SourceFileName attribute, 221
SourceForge Web site, 67
sources of metadata
  databases, 60–61
  design tools, 64–65
  entering metadata manually, 62
  existing applications and source code, 66–68
  external, 63–64
  overview, 56–58
  schema definitions, 59–60
  Web Services, 62
SourceSafe object, 260–63
sp_help procedures, 61, 86, 98
special characters, and XML, 548–49
SpecialType element, 313, 314, 318, 319
SPInputMetadata.xml file, 348, 371–72
SPInput.xml, 299
spit code region, 423
Split method, 154
spname variable, 353, 357
SPParameters template, 358–59, 369, 371
SPParameters utility template, 358
SPSet element, 315, 318, 338, 348, 355, 356, 362
SPSupport.xslt file, 353
SQL server features in Yukon, 520–22
SQL-92 databases (SQL Server), 78–104, 79
  alternatives to, 79
  careful naming, 99–101
  extracting stored procedures schemas, 93–97
    determining stored procedure intent, 94–95
    retrieving stored procedure recordsets, 95–97
  information schema views, 86–93

modifying mappings, 93
overview, 86
specific schemas, 87–89
and SQL-92 terminology, 86–87
using constraints, 89–93
metadata extraction tool, 79–86
overview, 79
security, 85–86
target structure, 80–83
tool architecture, 84–85
overview, 78
working with SQL server –specific metadata, 97–98
SQL-92 metadata extraction tool, 618–46
ExtractMetadata class, 619–35
declaring MustOverride methods, 620–21
implementing CreateMetadata method, 621–23
looping through all tables, 624
opening, 619–20
outputting check constraints, 630
outputting primary key information, 631–32
outputting relations, 632–34
outputting table constraints, 631
outputting XML for column privileges, 629
outputting XML for each column, 626–29
outputting XML for each table, 624–26
overview, 619
overview, 618
SQLExtractMetadata class, 635–46
constructor, 636
getting connection, 637
ignoring specific tables, 645–46
implementing MustOverride methods, 637–39
overview, 635–36
supporting extended properties, 639–45
SQLDataAdapter, 637
SQLDataReader class, 396
SQLDataReader object, 314
SQLExtractMetadata class, 84, 85, 97, 619, 623, 635–46
constructor, 636
getting connection, 637
ignoring specific tables, 645–46
implementing MustOverride methods, 637–39
overview, 635–36
supporting extended properties, 639–45
SQLMetadataExtraction class, 646
SQLType attribute, 359, 436
sqltype variable, 615
SQLUtility.getdate( ) function, 91
srcSafeINI file, 261
SSAddFile method, 259
SSCheckIn method, 259
SSCheckInFile method, 259, 683–84
SSCheckOutFile method, 257, 258, 259
stackalloc keyword, 704
Standard element, 116, 117, 124, 222, 223
starts-with directive, 607
Startup class, 13
startup code, 185–87
StatefulPrivate attribute, 106
StatelessPrivate attribute, 106
statements, 159, 161, 166
Statements collection, 166, 170
Static keyword, 701
static method, 233
steps of code generation. *See* five steps of code generation
stored procedures, 207–10, 345–75. *See also* retrieve stored procedures
additional reading about, 375
applying different stored procedure privileges, 315–16
building stored procedure metadata, 348–50
create stored procedures, 364–67
delete stored procedures, 368–70
describing complex, 299
generation process, 347–48
SetSelect stored procedures, 372–74
specifying handcrafted stored procedures, 317–18
specifying minimal information, 308–9
update stored procedures, 370–72
stored procedures schemas, extracting, 93–97
determining stored procedure intent, 94–95
retrieving stored procedure recordsets, 95–97
StoredProcedure element, 342
storedProcName_Custom_BeforeSelect stored procedure, 208
Stream class, 148
StreamWriter, 8, 9, 17, 18
String class, 145
string directive, 607
String parameter, 579
StringBuilder, 150
string-free navigation, 487
string-length directive, 607
string-length function, 413
StripPath template, 146, 353

StripPathAndExtension template, 353
strongly typed DataSets, 36–37, 115, 535
stylesheets, 142–43, 576–77
Sub statement, 701
substring directive, 607
sum directive, 607
support templates, creating, 154–55
Support.xslt, 354
switch statement, 704
SyncLock feature, 703
System.CodeDOM namespace, 167
System.ComponentModel.Component interface elements, 189
System.Object class, 191, 192, 489
system-property directive, 607
System.String property, 313
System.Xml.Xsl namespace, 17

T

/t:<typeName> switch, 75
TabIndex property, 464, 465, 466
Table element, 41, 42, 74, 126, 612, 613
Table mode template, 611
TABLE_CONSTRAINT view, 92
TABLE_CONSTRAINTS view, 87
TABLE_PRIVILEGES view, 87
TableColumn element, 80, 82, 126, 372, 626
TableColumn node, 136, 153
TableConstraints element, 136, 153
TableConstraints table, 631, 634
TableDeletePriority, 623
TableInsertPriority, 623
TableName attribute, 366, 369, 374
tableName parameter, 48, 576, 641, 643
tablenameDataRow classes, 37
TABLES view, 87
target files, 26, 126–30
TargetFileName attribute, 221
TargetLanguage attribute, 221, 234
targetNamespace attribute, 560, 565, 570, 571, 573, 574
Task attribute, 318, 366
Task List, 240, 241
TempCustomersOrders DataSet, 568
templates, 5, 578–83
  C++, 524
  control over, 32
  debugging, 275–80
  entry-level, 132
  generating “Hello World” via, 15–20
  generating simple class via, 47–50
  high-level processing, 132–35
  managing template projects, 239–40
  match templates, 141–42, 580–82
  named, 135–37
  named templates, 578
  overview, 578
  passing parameters, 582–83
  recursive, 144–46
  refactoring code and/or snippets to, 536–37
  sharing, 528–29
  support templates, 154–55
tempuri.org, 553–55
test attribute, 583
testing of code generation, 30–31
text directive, 578
Text method, 314
Text property, 466
TextBox control, 437, 453
tilde (~), 487
Timesheet table, 306
tns namespace, 559
TODO item, 258
Tools class, 151, 612
ToolTips, 79, 273
top variable, 463, 670–71
ToString method, 393
Transact SQL (T-SQL), 79, 210, 298, 345, 521
TransactionForRetrieve attribute, 320
TransactionsForRetrieve attribute, 308
TransactionType attribute, 308, 320
Transform method, 18, 577
translate directive, 607
translate function, 579
TreeNodeForXML class, 651, 656
TreeView, 188
Trim method, 145
true directive, 607
Try/Catch block, 187, 269, 275
Try/Catch/Finally block, 262
T-SQL (Transact SQL), 79, 210, 298, 345, 521
Type attribute, 45, 49, 565
Type property, 45
Type suffixes on literals feature, 702
type variable, 411
Type variable, 584
TypeAttributes property, 159
TypeForTag method, 669
TypeLibs, 67
TypeName attribute, 222, 223, 224, 225, 612
TypeOf operator, 192
TypeOf..Is operator, 423
types, 159–60
types array, 441
typesafe function pointer, 425

U

uc1 tag prefix, 479, 494, 496
uc1:header user control, 501
UDT (User-Defined Type), 313
ui:AllForms element, 438
ui:Form element, 438, 455, 456, 458, 495
ui:FormInfo element, 455
ui:Forms element, 438
UIInfo template, 438, 454
UIInfo.xml file, 437, 455, 492
ui:Root element, 438
ui:UIRoot element, 453, 455, 495
UML (Unified Modeling Language), 6, 58, 64–65, 198
unary negation (-) operator, 703, 705
unique key constraints, 92
UNIQUE_CONSTRAINT fields, 92, 93
unnamed type, 565
unparsed-entity-uri directive, 607
Unsafe code feature, 704
UntilEdited type, 228, 272
UPDATE command, 475
UPDATE statement, 368, 371
update stored procedures, 370–72
UpdateFlag, 475
UpdatePattern attribute, 321
UpdateProjectFile method, 250
urn:schemas-microsoft-com:xml-msdata namespace, 556, 610
UseForDesc attribute, 326, 372, 373, 393, 457
UseForNew attribute, 326
user interfaces. *See* WinForms
user interface templates, 453–70. *See also* edit forms, generating
  generating MDI form, 457
  generating select forms, 457–58
  overview, 453
  transforming metadata with WinInputTransform.xslt, 453–57
UserControl template, 469–70
UserData method, 172, 536, 700
User-Defined Type (UDT), 313
UserEdit page, 492
UserID parameter, 136, 137
UserProcContents property, 621
UserVerboseName property, 621, 624
UseSPSet element, 315, 338
usetype variable, 411
Utility assembly, 245, 248, 404
utility methods, 189–90
Utility/Tools class, 404
Utility.Tools class, 591, 665

V

Validate method, 203
Validate<PropertyName> method, 392, 411, 414
ValidateName property, 392, 411
validation, 200–203
Validation method, 526
Value attribute, 640
value inequality (<>) operator, 703
var1 variable, 590
varbinary type, 141
variables
  array variables, 169–70
  conditional, 140–41
  declaring, 165
  XSLT, 588–90
VB .NET Imports statement, 694
vbproj file, 246, 251, 252
VIEW_COLUMN_USAGE view, 89
VIEW_TABLE_USAGE view, 89
ViewColumn element, 626
VIEWS view, 89
virtual keyword, 193
Visio code generation, 6
Visual Basic 6, 190
Visual Studio. *See also* Solution Explorer
  absolute positioning, 451
  automation model of, 68, 157
  callstack display, 273
  Class View, 238
  code generation problems, 482–83, 485
  debugging tools of, 270
  debugging with, 270, 278
  deleting files, 253–54
  and deserialization, 424
  Document Object Model (DOM) of, 21
  editing XSLT files in, 240
  future of, 534–37
  and GUIDs, 241–42, 285
  including and excluding files, 253
  managing information in Settings, Imports, and References sections, 243
  managing template projects, 239–40
  “Modified Outside the Source Editor” dialog boxes, 217
  and MustInherit class, 423
  next version of, 67, 534–37
  options for generating code, 6
  and refactoring, 523–24
  and serialization, 205, 379

spit code region, 460, 468
and strongly typed DataSets, 36, 37
subdirectories in, 238
use of CodeDOM generation, 8, 156
viewing XML files, 247
Whidbey version of, 67
XML validation, 274, 275
and XSDs, 59, 74, 75, 76, 106, 107, 122, 308

W

Web Forms validator, 392
Web Service Description Languages (WSDLs), 58, 62–63, 307
Web user interface. *See* ASP.NET
WebEdit.xslt, 500–505
WebForms. *See* ASP.NET
WebSelect.xslt, 494–99
Web.UI.Page, 472, 493
Web.UI.UserControl class, 472, 493
well-formedness constraints, 548
WHERE clause, 355, 373, 374, 522
WHERE literal, 355
WhereClause element, 355, 369, 372
Whidbey (.NET 2.0) features, 67, 522–27
whitespace, 137–39, 207, 550, 592–93
WinDiff tool, 26
Windows applications, 186
Windows Form Designer generated region, 156, 423
Windows.Forms, 199, 202, 204, 205
WINDOWS.FORMS namespace, 421
Windows.Forms validator, 392
Windows.Forms.Control, 423
WindowsFormsDesigner template, 460
Windows.Forms.Form, 423
Windows.Forms.UserControl, 427
WinForms, 421–70
architecture of, 422–29
data binding vs. data scattering, 428–29
design-time support, 424
events and methods in derived forms, 424–26
implementing user controls, 427
inheritance hierarchy, 423
interacting with Visual tools, 429
overview, 422–23
developing code generation strategies for, 429–52. *See also* dynamic user interface creation; metadata, and WinForms
building new derivation strategy, 430–32
overview, 429
user controls, 439–42
overview, 421–22
user interface templates, 453–70. *See also* edit forms, generating
generating MDI form, 457
generating select forms, 457–58
overview, 453
transforming metadata with WinInputTransform.xslt, 453–57
WinInputMetadata.xml, 457
WinInputTransform.xslt stylesheet, 436, 437–38, 453–57
With keyword, 701
WithEvents keyword, 701
Word application, 26
Write method, 154
WriteLine method, 8, 9, 14, 152
WriteLine statement, 42
WriteLineAndIndent method, 152
WriteLineAndOutdent method, 152
WriteLineNoTabs method, 150
writers fittings, 148
WSDL.EXE, 63
WSDLs (Web Service Description Languages), 58, 62–63, 307

X

XCopy deployment, 118
XMI standard, 65
XML. schema namespace, *651*
XML comments, 67
xml directive, 578
XML (eXtensible Markup Language), 543–59. *See also* XSDs (XML Schema Definitions); XSLT
additional reading about, 608
attributes, 546–47
elements, 545–46
handling special characters, 548–49
namespaces, 551–59
default namespace, 557
namespace prefixes, 555–56
namespace scope, 552–53
in .NET, 558–59
overview, 551–52
tempuri.org, 553–55
overview, 543–44
rules, 547–48
whitespace, 550
working with minimal feedback, 550–51
XML Merge tool, 73
XML metadata. *See* metadata
xml output type, 353
XML Schema Definitions. *See* XSDs (XML Schema Definitions)

XmlAttribute class, 547
XMLAttribute object, 547
xmlDoc.Load method, 569
XmlDocument class, 42, 47, 546, 547, 558, 568, 569, 623
XmlDocument object, 561
XmlDocument.Load method, 549
XMLElement object, 545, 546
XmlLinkedNode, 546
xml-msdata namespace, 578
XmlNamespaceManager, 558
XMLNamespaceManager class, 654
XmlNameSpaceManager.AddNamespace method, 559
XMLNode class, 546, 547, 594
XmlNode method, 675
XMLNode objects, 651
XMLNodeList, 42
xmlns attribute, 552, 557, 570, 571, 578
XMLSchema namespace, 578
XmlSchemaCollection, 569
XmlTextReader, 569
XMLTextWriter, 578
XmlValidatingReader, 569, 573
Xml.XmlDocument object, 545
Xml.XmlNode, 546
Xml.XmlValidatingReader, 568
Xml.Xsl.XsltArgumentList, 612
Xml.Xsl.XslTransform object, 577
Xor operator, 704
xor operator, 705
XPath, 16, 17, 593–607
  additional reading about, 608
  functions, 605–7
  overview, 593–94
  XPath navigation, 595–601
    context, 596
    location paths, 596–601
XPathExpression, 593
XPathNavigator objects, 17
XQuery statements, 522
xs:all directive, 567
xs:all element, 566, 567
xs:choice element, 566, 567, 574
xs:choice elements, 77
xsdElementNode method, 665
XSD.EXE tool, 36, 59, 74, 75, 107
xsdNodeList method, 672, 673
XSDs (XML Schema Definitions), 59–60, 61, 184, 210, 559–74
  cardinality, 566
  changes to, 122
  and child elements, 222
  and Code Generation tool, 117–18
  compositors, 566–67
  converting into XML, 609–18
  editing in Visual Studio, 571–74
  and legal syntax for XML files, 308
  managing freeform metadata files using, 106–7
  overview, 559–65
  role in code generation, 74–78
  structure of, 76–78
  targetNamespace, 570–71
  and XML validation, 568–70
  XSD nodes, 565
xsdschema.xsd file, 573
XSDToFriendlyXML.xslt template, 60, 72, 74, 80
xs:element, 565, 566, 567
xsi:schemaLocation, 573
xsi:schemaLocation attribute, 274
xsl: import description, 603
xsl prefix, 556, 578
xsl:apply-imports directive, 602
xsl:apply-template directive, 600
xsl:apply-templates directive
  as cause of problems, 278
  description of, 602
  and entry-level template creation, 132
  looping through code with, 49
  looping with, 584, 585, 586
  and match templates, 580, 581
  and retrieve stored procedure stylesheet, 354
  and stored procedure templates, 356, 358
  working with variables, 589
  and XML copying details, 334
  and XSD conversion into XML, 612
  vs. xsl:for-each directive, 140
  and XSLT template creation, 134
xsl:attribute directive, 339, 602
xsl:attribute-set directive, 603
xsl:call-template directive, 134, 354, 356, 579, 589, 590, 603
xsl:call-template element, 535
xsl:call-templates, 585
xsl:call-templates directive, 132
xsl:choose directive, 134, 141, 145, 147, 583, 584, 603
xsl:comment directive, 577, 603
xsl:copy directive, 603
xsl:copy-of directive, 603
xsl:decimal-format directive, 603
xsl:document directive, 603
xsl:element directive, 339, 496, 603, 610
xsl:fallback directive, 603
xsl:for-each description, 604, 615
xsl:for-each directive
  as cause of problems, 278
  changing context with, 596
  description of, 603
  and event handler creation, 468

looping with, 49, 461, 584–86
and match template creation, 141
and ObjectAttributes template, 335
and retrieve stored procedure stylesheet, 354
vs. xsl:apply-templates directive, 140
xsl:if directive, 134, 137, 138, 141, 147, 373, 374, 583, 603
xsl:import directive, 131, 132, 576, 588, 603
xsl:include directive, 131, 576, 588, 603
xsl:key directive, 604
xsl:message directive, 604
xsl:namespace-alias directive, 604
xsl:number directive, 604
xsl:otherwise directive, 145, 604
xsl:output directive, 604
xsl:param directive, 278, 411, 604
<xsl:param…> element, 120
xsl:preserve-space directive, 604
xsl:preserve-space element, 353, 578
xsl:processing-instruction directive, 604
xsl:script directive, 604
xsl:sort directive, 586, 604
xsl:sort element, 582
xsl:strip-space directive, 604
xsl:stylesheet directive, 604
xsl:stylesheet element, 352, 576, 578
<xsl:stylesheet…> element, 120
XSLT, 574–93
additional reading about, 608
extension objects, 591–92
file header, 577–78
flow constructs, 583–87
outputting information, 587–88
outputting whitespace, 592–93
processing, 575–76
stylesheets, 576–77
templates, 578–83
generating “Hello World” via, 15–20
generating simple class via, 47–50
match templates, 580–82
named templates, 578
overview, 5, 578
passing parameters, 582–83
variables, 588–90
XSLT code generation, 130–47
creating classes, 130–46
creating entry-level template, 132
creating high-level processing template, 132–35
creating match templates, 141–42
creating named templates, 135–37
managing whitespace, 137–39
specifying XSLT parameters, 131
supporting stylesheets, 142–43
using conditional variables, 140–41
using looping, 140–41
using recursive templates, 144–46
overview, 130
XsltArgumentList, 225, 612
xsl:template directive, 134, 135, 278, 594, 604
xsl:text directive, 138, 139, 605
xsl:text element, 358, 407, 464, 593
XSLTFileName attribute, 223
XSLTFiles element, 124, 222, 223, 224, 225, 232
xslt:for-each directive, 585
XSLTGeneration directive, 220, 221, 225, 232–33, 235, 612
XSLTGeneration method, 678, 686–87
XSLTGenerationCallback method, 685
XSLTProcess directive, 220, 221, 225, 230, 257, 612, 683
XSLTProcess element, 226
XSLTransform class, 611
xsl:transform directive, 605
xsl:transform element, 120, 576, 578
XSL/Transform namespace, 578
XslTransform object, 17
xsl:value-of directive
and accessing information in metadata file, 142
description of, 605
and location paths, 596
Name and Type attributes output by, 49
and named template creation, 135
outputting information, 587
outputting whitespace, 592, 593
and XSLT template creation, 134
xsl:variable directive, 589, 590, 605
xsl:when directive, 584, 605
xsl:with-param directive, 605
xs:maxOccurs attribute, 566, 567
xs:minOccurs attribute, 566, 567
xs:minOccurs=“0” attribute, 566, 567, 574
xs:schema element, 610
xs:schema node, 566, 567, 610
xs:selector node, 616
xs:sequence element, 566, 574
xs:sequence node, 566

Y

Yukon, 520–22