

Join the discussion @ p2p.wrox.com



Wrox Programmer to Programmer™



Professional

SharePoint® 2007

Records Management Development

*Managing Official Records with
Microsoft® Office SharePoint® Server 2007*

John Holliday



Programmer to Programmer™

Get more out of **WROX.com**

Interact

Take an active role online by participating in our P2P forums

Wrox Online Library

Hundreds of our books are available online through Books24x7.com

Wrox Blox

Download short informational pieces and code to keep you up to date and out of trouble!

Chapters on Demand

Purchase individual book chapters in pdf format

Join the Community

Sign up for our free monthly newsletter at newsletter.wrox.com

Browse

Ready for more Wrox? We have books and e-books available on .NET, SQL Server, Java, XML, Visual Basic, C#/ C++, and much more!

Contact Us.

We always like to get feedback from our readers. Have a book idea?

Need community support? Let us know by e-mailing [**wrox-partnerwithus@wrox.com**](mailto:wrox-partnerwithus@wrox.com)

Professional SharePoint® 2007 Records Management Development

Introduction	xv
Chapter 1: Official Records	1
Chapter 2: Preparing for Records Management Development.	19
Chapter 3: SharePoint Tools for Managing Records.	39
Chapter 4: The MOSS 2007 Records Center.	99
Chapter 5: Building and Configuring a Records Repository	117
Chapter 6: Populating the Records Repository	171
Chapter 7: Information Management Policy	195
Chapter 8: Information Policy and Record Retention	235
Chapter 9: Information Policy and Record Auditing	255
Chapter 10: Managing Physical Records	281
Chapter 11: Suspending Record Processing Using Holds.	295
Chapter 12: Building and Deploying Custom Routers.	307
Chapter 13: Maintaining Record Integrity.	325
Chapter 14: Managing Electronic Mail Records	355
Chapter 15: Using Workflow to Manage Records	371
Chapter 16: The DoD 5015.2 Add-On Pack.	403
Index	421

Professional

**SharePoint® 2007 Records
Management Development**

**Managing Official Records with
Microsoft® Office SharePoint® Server 2007**

John Holliday



WILEY

Wiley Publishing, Inc.

Professional SharePoint® 2007 Records Management Development: Managing Official Records with Microsoft® Office SharePoint® Server 2007

Published by
Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2009 by Wiley Publishing, Inc., Indianapolis, Indiana

ISBN: 978-0-470-28762-0

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permissions>.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Library of Congress Control Number: 2009930062.

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Wrox Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Microsoft and SharePoint are registered trademarks of Microsoft Corporation in the United States and/or other countries. All other trademarks are the property of their respective owners. Wiley Publishing, Inc., is not associated with any product or vendor mentioned in this book.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

*This book is dedicated to the ever present and ever emerging singularity;
the eternal point-value at which all things are known
and through which all knowledge is shared.*

Credits

Acquisitions Editors

Katie Mohr
Paul Reese

Project Editor

Kelly Talbot

Technical Editors

Dan Attis
Aaron Cutlip
Todd Meister
Stacy Draper

Senior Production Editor

Debra Banninger

Copy Editor

Cate Caffrey

Editorial Director

Robyn B. Siesky

Editorial Manager

Mary Beth Wakefield

Production Manager

Tim Tate

Vice President and Executive Group Publisher

Richard Swadley

Vice President and Executive Publisher

Barry Pruett

Associate Publisher

Jim Minatel

Project Coordinator, Cover

Lynsey Stanford

Composer

Jeffrey Lytle, Happenstance Type-O-Rama

Proofreader

Nancy Carrasco

Indexer

J & J Indexing

Cover Image

© Tetra images/Punchtock

About the Author



John Holliday has more than 25 years of professional software development experience and has been involved in a wide range of commercial software projects from desktop personal information managers to enterprise information systems for companies including IBM, Kodak, Autodesk, Mentor Graphics, Tektronix, and Alcatel. After receiving a bachelor's degree in applied mathematics from Harvard and a J.D. from the University of Michigan, John began researching alternative methods of representing legal relationships and developing a specialized computing language for constructing legal expert systems. Over the years, his interest in knowledge representation has expanded to include all aspects of distributed systems development, with a special focus on intelligent documents, collaboration, and enterprise content management. In addition to his professional career, John is actively involved in humanitarian activities through Works of Wonder International, a non-profit organization he co-founded with his wife, Alice, and the Art of Living Foundation, an international service organization devoted to uplifting human values throughout the world.

Acknowledgments

First of all, I thank my lovely and gracious wife, Alice, without whose love and support I could not survive and without whose patience this book would not have been possible. I also thank my noble and courageous mother, Frances, whose enduring spirit lights my way and warms my heart.

Thanks to Microsoft for creating a breakthrough product and for bringing together the best minds in the industry, some of whom I've had the great good fortune to interact with directly. Mike Ammerlaan, Ryan Duguid, Paul Andrew, Eilene Hao, and Andrew May, to name a few, provided tremendous insight and clarity about the inner workings of the product, and I owe them an immense debt of gratitude.

Reviewing a technical book like this one is no easy task. It takes careful attention to detail and a willingness to challenge the author's fundamental assumptions, as well as the ability to do all of this in an impossibly short amount of time. I cannot thank Dan Attis, Aaron Cutlip, Todd Meister, and Stacy Draper enough for their help and support. I also thank my friends Spencer Harbar, Eric Shupps, Robert Bogue, John Ross, Todd Baginski, Natalya Voskresenkaya, Paul Galvin, Zlatan Dzinic, Jeremy Sublett, Sahil Malik, Brendon Schwartz, and Bradley Smith for many wonderful discussions about ECM and SharePoint development in general.

They say a good teacher always learns more than his or her students, and I must say that I am continually amazed at how much I've learned while delivering SharePoint training for Ted Pattison's Critical Path Training (www.criticalpathtraining.com). Ted has assembled a remarkably talented group of instructors whose high professional standards and willingness to share their knowledge have helped me greatly. In particular, I will be forever indebted to Andrew Connell for introducing me to Ted and for his continued guidance and friendship.

A very special thanks goes out to my friend Eli Robillard (<http://weblogs.asp.net/ERobillard>), whose unique insights into enterprise content management and what it takes to build effective ECM solutions on the SharePoint platform are truly enlightening. Eli spent a great deal of time thinking about the book and helping me to strike the right balance between theory and practice.

I also express my sincere appreciation to Jim Minatel and the folks at Wiley Publishing, whose experience and professionalism are unparalleled in the industry. I especially thank Katie Mohr, whose support for me and for this project has been unwavering from the beginning, even through the many personal and professional challenges that presented themselves along the way. I also give a special "thank you" to my development editor, Kelly Talbot, who was a joy to work with and whose thoughtful advice and insights about the writing process were indispensable.

I know that other writers have echoed the popular phrase, "It takes a village," when describing the experience of bringing a book like this to market. I can only say that I truly understand now what that means. The *village* I'm referring to here is the worldwide network of SharePoint MVPs that I feel privileged to be a part of and whose amazing spirit, dedication, and enthusiasm continue to inspire me. Microsoft has done an incredible job of creating a unique and vibrant community of SharePoint professionals that I hope will continue to thrive and expand.

Contents

Introduction	xv
Chapter 1: Official Records	1
What Are Official Records?	1
Core Records Management Principles	2
Content Modeling	3
Understanding the Content Life Cycle	4
Content Modeling Goals	5
The Role/Activity Modeling Technique	6
Developing a File Plan	14
Identifying Roles and Responsibilities	15
Identifying Applicable Policies and Procedures	15
Identifying Custom Routing and Workflow Requirements	16
Identifying Document Categories and Groups	16
Identifying Document Sources	16
Analyzing Storage Requirements	17
Analyzing Security Requirements	17
Summary	17
Chapter 2: Preparing for Records Management Development	19
Understanding Current U.S. Regulations	19
Setting Up Your Development Environment for Maximum Productivity	20
CAML.NET IntelliSense	22
ECM2007: A Foundation Class Library for Records Management	23
Creating a SharePoint Feature Project Template	25
Leveraging XML Schemas for Maximum Flexibility	32
Generating Schema Classes from within Visual Studio	36
Summary	37
Chapter 3: SharePoint Tools for Managing Records	39
Document Libraries	40
Creating Document Libraries	40
Document Libraries and Property Promotion	41

Contents

Content Types	42
Uses for Content Types	44
Content Type Definitions	47
Creating Content Types	47
Versioning	64
Versioning Rules	66
Check-Out, Check-In, and Versioning	67
Versioning Pitfalls	69
Programmatic Versioning	71
Content Security	83
SharePoint Permissions	83
Strategies for Controlling Access to Content	88
Information Rights Management	90
The Records Management Object Model and API	92
The Official File Web Service	93
Summary	96
Chapter 4: The MOSS 2007 Records Center	99
The Records Management Feature	99
The Records Center Site Definition	102
Records Center Components	103
The Records Repository Users Group	104
The Record Routing Table	104
The Holding Zone	104
The Holds List	105
Workflow Provisioning	105
Default Record Series	105
Default Expiration Actions	106
Records Center File Processing	106
The Core Record Routing Mechanism	106
Custom Record Routing	108
Property Storage	108
Audit History	110
Property Promotion and Demotion	112
Summary	115
Chapter 5: Building and Configuring a Records Repository	117
Creating the Records Center Site	117
Creating a Records Center Manually	118
Creating a Records Center Programmatically	122

Working with Traditional File Plans	125
Creating Document Libraries and Content Types	125
Setting Up the Record Routing Table	129
Building and Using Dynamic File Plans	130
Designing the File Plan Schema	132
A Quick Introduction to InfoPath 2007	134
Generating Serialization Classes	142
Building the File Plan Gallery	157
Executing the File Plan	164
Summary	169
Chapter 6: Populating the Records Repository	171
<hr/>	
Submitting Individual Records	171
Configuring the Farm for Manual Submission	171
Granting Users Permission to Submit Records	172
Testing the Farm Configuration	173
Submitting Records Programmatically	177
Submitting Multiple Records	179
Summary	193
Chapter 7: Information Management Policy	195
<hr/>	
Information Policy Architecture	196
Limitations of Information Policy in SharePoint Server 2007	198
Policies, Policy Features, and Policy Resources	198
The Information Policy Life Cycle	200
Building Custom Policy Features	203
Designing for Extensibility	203
Creating Reusable Policy Components	203
Creating a Printer Control Policy Feature	216
Creating the Policy Feature Class	217
Adding Custom Policy Feature Settings	224
Creating a Print Monitor Add-In for Word 2007	229
Summary	233
Chapter 8: Information Policy and Record Retention	235
<hr/>	
The Expiration Policy Feature	235
Creating Custom Expiration Formulas and Actions	236
Expiration Formulas	238
Expiration Actions	240

Contents

The Expiration Timer Job	243
Planning for Retention	245
Extending the File Plan to Include Expiration Policies	245
Adding Expiration Policy Support to the Dynamic File Plan	245
Summary	253
<hr/> Chapter 9: Information Policy and Record Auditing	<hr/> 255
Understanding Auditing in SharePoint	255
Enabling and Disabling Auditing	257
Managing Audit Entries	258
Audit Reporting	259
Using the Auditing Policy Feature	262
Auditing in a Records Center Site	262
Creating an Audit Viewer Web Part	262
Viewing Historical Audit Records	271
Additional Considerations for Auditing	276
Extending the File Plan Schema to Support Auditing Policy	277
Summary	279
<hr/> Chapter 10: Managing Physical Records	<hr/> 281
Physical Records and List Items	281
Create a Physical Record Content Type	282
Create a Physical Records List	283
Configure Information Policy Features	283
Physical Records and Folders	284
Automating the Process	286
Physical Records and Workflow	292
Summary	294
<hr/> Chapter 11: Suspending Record Processing Using Holds	<hr/> 295
The Holds Architecture	296
Creating and Removing Holds	303
Placing a Hold	303
Removing a Hold	304
The Search & Process API	304
Summary	306

Chapter 12: Building and Deploying Custom Routers	307
Building Custom Routers	307
Creating a Simple Filtering Router	310
Installing the Router	313
Activating the Router	313
Creating a Tracking Router	314
Creating a Redirecting Router	317
Extending the File Plan Schema to Support Custom Routing	322
Summary	323
Chapter 13: Maintaining Record Integrity	325
Building a Content Validation Framework	325
Defining a Validation Schema	327
Building Validation Components	330
Using the Validation Framework with a Self-Validating Proposal Content Type	342
Using the Validation Framework to Build a Validating Router	347
Summary	353
Chapter 14: Managing Electronic Mail Records	355
Configuring Exchange 2007	357
Handling the Folders in Outlook 2007	368
Handling Missing Properties	368
Summary	369
Chapter 15: Using Workflow to Manage Records	371
A SharePoint Workflow Primer	373
Official Records, Workflow, and Complexity	376
Primitive Activities	377
Useful Activities for Records Management	380
Domain-Specific Records Management Activities	381
Workflow Modeling	381
Building a Workflow Activity Library	383
Testing Your Activities	384
Building Workflow Activities for Records Management	387
Executing a File Plan	388
Validating List Item Metadata	391

Contents

Extending SharePoint Designer to Use Records Management Workflows	396
Deploying and Registering the Activity Library	396
Creating the .ACTIONS File	396
Summary	401
Chapter 16: The DoD 5015.2 Add-On Pack	403
Requirements Addressed by the Add-On Pack	403
New Concepts	404
Enhanced Search	408
Installing the Add-On Pack	409
The Components Installed by the Add-On Pack	411
The RecordCenterRouter Feature	411
The RecordCenterAddonPack_Web Feature	411
The RecordCenterAddonPack_SiteWorkflows Feature	412
Custom Field Types and Field Controls	412
Content Types	413
Timer Jobs	413
Workflows	414
STSADM Commands	417
The Records Center E-Mail Router	418
Summary	420
Index	421

Introduction

Despite its power and flexibility, SharePoint presents many challenges for building enterprise content management (ECM) solutions. This is partly because of the inherently complex nature of application development on the SharePoint platform in general, but mainly because the out-of-the-box tools that SharePoint provides (such as content types, site columns, lists, etc.) are defined at such a low level that it is often difficult for developers to find the right balance between building reusable components that capture the semantics they need when crafting their solutions and writing applications directly using CAML and .NET code. These challenges are even greater for records management (RM) development because the platform provides no built-in support for file planning or content life-cycle management.

Records management on the SharePoint platform relies on several different components that together provide a comprehensive set of tools for building RM solutions, but each of these components can also be understood as a separate and distinct technology. As an example, Information Policy plays an important role in records management but also stands alone as a powerful component of any ECM solution.

This book, therefore, has two goals. The first is to explain the Microsoft Office SharePoint Server (MOSS) records management architecture and to share the most effective design methodologies and development strategies I've found for building ECM/RM solutions on the SharePoint platform. The second is to describe the underlying technologies and core components of the platform in sufficient detail to enable readers to apply the design methodologies and development techniques effectively to any SharePoint solution.

Who This Book Is For

This book is for SharePoint developers who are tasked with planning and implementing solutions involving documents that have been designated as *official records* at some point during their life cycle. These kinds of solutions typically include requirements such as controlling how long certain kinds of documents remain in the system, determining where those documents are stored, keeping track of what happens to documents after they have been placed into document libraries, and so on. In order to focus on such high-level requirements without getting lost in the details of SharePoint solution development, the book assumes that the reader is already familiar with the fundamentals of developing SharePoint solutions using Visual Studio.

While it is not necessary to be a *seasoned* SharePoint developer with years of experience, the reader should understand the basic steps involved in creating SharePoint features, writing custom code against the SharePoint object model, and deploying that code into the SharePoint run-time environment. The reader should also be comfortable using C# or Visual Basic .NET within the Visual Studio integrated development environment to create ASP.NET web applications.

It is not necessary for the reader to be familiar with general records management principles or with specific regulatory requirements. The book will bring the reader up to speed on the relevant regulations and will also introduce a conceptual framework for understanding them in the context of SharePoint development.

What This Book Covers

No records management solution is complete without a file plan, but there is little guidance available to help developers understand the file planning process or to incorporate a consistent file planning methodology into the development cycle. This book fills that gap by introducing a file planning methodology based on a set of concrete file plan components that enable developers to incorporate file plans into the solutions themselves, as parts of information policies, workflows, or custom features.

Unlike other SharePoint development books that focus only on the mechanics of building solutions, this book introduces *content modeling* as an integral part of the development process. This approach helps the reader to better understand enterprise content in the context of the roles it is intended to support, and to identify the “high value targets” to focus on first. The book then guides the reader through the development of a consistent methodology for identifying official records and creating effective file plans, understanding the out-of-the-box tools provided by SharePoint for managing official records, and understanding the limitations of the current architecture, including ways to work around them wherever possible.

The book also presents an overall development strategy that seeks to leverage the power of the SharePoint object model to create reusable components. Because of the breadth of the SharePoint platform and the inherent complexity of ECM solutions, developers often fall into the habit of writing “throw-away” code that can only be applied to a single solution or that must be copied and pasted from one solution to another. This tendency is even greater when a developer is forced to use a declarative XML-based language like CAML that is not compiled and that therefore does not offer the same level of reuse that most developers are used to. This book seeks to counteract that tendency and increase developer productivity while simplifying the design and implementation of complex solutions by developing and then continually extending a core set of ECM components in an ECM *foundation class library*. The components that are developed throughout the book can be extended easily and applied repeatedly, not only to records management solutions, but also to a wide range of SharePoint development projects.

How This Book Is Structured

This book is about building MOSS 2007 records management solutions, but it also includes some additional material related to SharePoint development in general. The early chapters build a conceptual framework that includes a content modeling methodology and also introduces several tools and techniques that are referred to throughout the book. It is therefore recommended that you read the first three chapters first, and then read the other chapters, focusing on the subsystems that interest you. Chapter 2 introduces the ECM2007 foundation class library, which is gradually extended throughout the book. It also introduces the dynamic file plan component, which is also extended throughout the book as a way of tying together the different phases of the content life cycle. The remaining chapters focus on specific parts of the MOSS records management architecture and need not be read in any particular order.

- ❑ **Chapter 1: Official Records** — Chapter 1 provides a conceptual framework for understanding MOSS records management and introduces the concept of *content modeling*, which helps to develop more effective records management solutions. It also describes the file planning process and lays the foundation for the remaining chapters.
- ❑ **Chapter 2: Preparing for Records Management Development** — Chapter 2 starts by examining some of the key U.S. legislation that drives many records management solutions. It then walks through the process of setting up a Visual Studio development environment for an optimal

development experience, shows you how to create your own SharePoint Feature project template, and introduces the ECM2007 foundation class library that is referred to and extended throughout the book.

- ❑ **Chapter 3: SharePoint Tools for Managing Records** — Chapter 3 provides an overview of the built-in components on the SharePoint platform that are most important for building records management solutions. It covers core concepts such as document libraries, content types, versioning, and security, showing how to add support for these components to the ECM2007 foundation class library. Finally, the chapter introduces the SharePoint records management namespaces and object model and describes the Official File Web Service.
- ❑ **Chapter 4: The MOSS 2007 Records Center** — Chapter 4 provides a deep dive into the architecture and implementation of the MOSS Records Center site definition, which provides most of the built-in support for records management in the SharePoint environment. It introduces each component of the site definition and describes how incoming records are processed.
- ❑ **Chapter 5: Building and Configuring a Records Repository** — Chapter 5 builds on Chapter 4 and describes how to set up and configure a Records Repository, including the steps required for creating the necessary document libraries and content types and setting up the Record Routing Table. The chapter also introduces the concept of using a dynamic file plan for configuring the repository programmatically. It then further extends the ECM2007 foundation class library to include custom file planning components.
- ❑ **Chapter 6: Populating the Records Repository** — Chapter 6 looks at the Records Repository from the perspective of users, administrators, and client applications that need to submit documents to the Repository. It walks through the steps needed to configure the SharePoint Farm for manual submission, and it shows how to submit records programmatically to the Repository. It also shows how to extend the SharePoint UI to support the submission of more than one record at a time.
- ❑ **Chapter 7: Information Management Policy** — Chapter 7 provides a broad introduction to MOSS Information Management Policy, which underlies much of the advanced records management capabilities of the SharePoint platform. It describes the Information Policy architecture and gives a detailed explanation of each component, showing how to create custom policy features and how to build libraries of reusable Information Policy components.
- ❑ **Chapter 8: Information Policy and Record Retention** — Chapter 8 builds on Chapter 7 and shows how Information Policy is used to control record retention. The chapter provides a detailed explanation of how the built-in Expiration Policy Feature works and shows how to extend it with custom expiration formulas and actions. It then shows how to extend the dynamic file plan introduced in Chapter 5 to include record retention policies.
- ❑ **Chapter 9: Information Policy and Record Auditing** — Chapter 9 takes a similar approach to Chapter 8, showing how Information Policy is used to control record auditing. It starts by describing the built-in auditing features of the SharePoint platform and shows how to work with the SharePoint auditing components and audit entries. It then shows how to extend the dynamic file plan components from Chapter 5 to include record auditing support.
- ❑ **Chapter 10: Managing Physical Records** — Chapter 10 shows how to use the MOSS Records Management features to handle physical records by managing them as list items from within the Records Center site. It then shows how to leverage the Information Policy framework in conjunction with SharePoint workflows to reduce the complexity inherent in coordinating the physical and electronic components.

- ❑ **Chapter 11: Suspending Record Processing Using Holds** — Chapter 11 describes the Holds architecture and shows how to create and remove holds, associate them with records, and leverage the built-in holds reporting and processing functionality. It also introduces the Search & Process API, which can be used to quickly locate records to apply or remove holds.
- ❑ **Chapter 12: Building and Deploying Custom Routers** — Chapter 12 introduces the MOSS Record Routing framework and shows how to build and deploy custom routers that enhance the built-in support for processing incoming records. It adds custom routing support to the ECM2007 foundation class library and then shows how to use custom routers for filtering, tracking, and redirecting records. Finally, the chapter further extends the dynamic file plan components from Chapter 5 to support custom routing.
- ❑ **Chapter 13: Maintaining Record Integrity** — Chapter 13 approaches records management from a higher level and focuses on the problem of ensuring that incoming records have valid content and metadata. It starts by extending the ECM2007 foundation class library to include an extensible content validation framework, which is then used to associate custom validation rules with standard SharePoint components such as content types and Information Policy features. The chapter then shows how to use the Validation framework from within a custom router.
- ❑ **Chapter 14: Managing Electronic Mail Records** — Chapter 14 explains the built-in support provided by MOSS and Exchange 2007 for managing electronic mail records. It introduces the Messaging Records Management framework and walks through the required steps for setting up Exchange 2007 to route official e-mail messages to a MOSS Records Repository using Exchange Managed Folders. The chapter also includes code that shows how to set up managed folders programmatically and further extends the dynamic file plan components from Chapter 5 to include support for setting up Exchange.
- ❑ **Chapter 15: Using Workflow to Manage Records** — Chapter 15 introduces the concept of using SharePoint workflows to manage the complex interactions that are often involved in ECM/RM solutions. After providing a brief overview of the SharePoint workflow architecture, the chapter describes the workflow activities that are included in the Microsoft SharePoint ECM Starter Kit for working with official records. The chapter then identifies additional workflow activities that are useful for records management and shows how to build a custom workflow activity library for building records management solutions that uses many of the components, such as the dynamic file plan and the custom validation framework, that were created in earlier chapters. Finally, the chapter shows how to extend SharePoint Designer to add records management support to declarative, no-code workflows.
- ❑ **Chapter 16: The DoD 5015.2 Add-On Pack** — Chapter 16 provides a deep dive into the architecture and implementation of the DoD 5015.2 Add-On Pack. It starts by describing the additional requirements imposed by the DoD 5015 standard and shows how each of these requirements is fulfilled by the components installed by the Add-On Pack. The chapter then walks through the installation and configuration of the Add-On Pack and describes each of its components and features in detail, including custom field types and controls, content types, timer jobs, and workflows.

What You Need to Use This Book

This book assumes that the reader has moderate proficiency in using C# and the Visual Studio development environment to build ASP.NET web applications. It also assumes that the reader is familiar with the basic steps involved in building and deploying SharePoint features and solution packages using Visual Studio.

The code modules and examples in the book were developed on a Windows Server 2003 system (Enterprise Edition with Service Pack 2), Office SharePoint Server 2007 with Service Pack 1, Office 2007 Ultimate, Visual Studio 2008 with Service Pack 1, and the Visual Studio 2008 Software Development Kit. The downloadable Visual Studio solution files that accompany the book require the .NET Framework 3.5, and InfoPath 2007 is required for working with the dynamic file planning components that are introduced in Chapter 5.

The examples in Chapter 16 were developed in a separate development environment in which the DoD 5015.2 Add-On Pack was installed. As mentioned in that chapter, installing the Add-On Pack irreversibly alters the MOSS environment. I therefore highly recommend that you create and configure a separate system for working through those examples to avoid having to redeploy your primary development environment.

There are several additional tools that are popular among SharePoint developers, and I typically use many of them when building SharePoint solutions. These include the SharePoint Debugger Feature (www.codeplex.com/features), DebugView from SysInternals (www.sysinternals.com), SPTraceView (sptraceview.codeplex.com), Spence Harbar's Application Pool Manager (www.harbar.net), and SharePoint Manager (www.codeplex.com/spm). Although not a requirement for working through the code examples, I also highly recommend Carsten Keutmann's WSP Builder (wspbuilder.codeplex.com), Lutz Roeder's .NET Reflector (www.red-gate.com/products/reflector), and my own CAML.NET IntelliSense (<http://code.msdn.microsoft.com/camlintellisense>).

Conventions

To help you get the most from the text and keep track of what's happening, we've used several conventions throughout the book.

Boxes like this one hold important, not-to-be forgotten information that is directly relevant to the surrounding text.

Notes, tips, hints, tricks, and asides to the current discussion are offset and placed in italics like this.

As for styles in the text:

- ❑ We show keyboard strokes like this: [Ctrl]+A.
- ❑ We show URLs and code within the text like this: `persistence.properties`.
- ❑ We present code in two different ways:

We use a monospace type with no highlighting for most code examples.

We use gray highlighting to emphasize code that's particularly important in the present context.

Source Code

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. All of the source code used in this book is available for download at www.wrox.com. Once at the site, simply locate the book's title (either by using

Introduction

the Search box or by using one of the title lists), and click on the “Download Code” link on the book’s detail page to obtain all the source code for the book.

Because many books have similar titles, you may find it easiest to search by ISBN; this book’s ISBN is 978-0-470-28762-0.

Once you download the code, just decompress it with your favorite compression tool. Alternately, you can go to the main Wrox code download page at www.wrox.com/dynamic/books/download.aspx to see the code available for this book and all other Wrox books.

Errata

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, such as a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata, you may save another reader hours of frustration, and at the same time, you will be helping us provide even higher-quality information.

To find the Errata page for this book, go to www.wrox.com and locate the title using the Search box or one of the title lists. Then, on the book details page, click on the Book Errata link. On this page, you can view all errata that have been submitted for this book and posted by Wrox editors. A complete book list, including links to each book’s errata, is also available at www.wrox.com/misc-pages/booklist.shtml.

If you don’t spot “your” error on the book’s Errata page, go to www.wrox.com/contact/techsupport.shtml and complete the form there to send us the error you have found. We’ll check the information and, if appropriate, post a message to the book’s Errata page and fix the problem in subsequent editions of the book.

p2p.wrox.com

For author and peer discussion, join the P2P forums at p2p.wrox.com. The forums are a web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At <http://p2p.wrox.com> you will find a number of different forums that will help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to p2p.wrox.com and click on the Register link.
2. Read the terms of use and click Agree.

3. Complete the required information to join as well as any optional information you wish to provide and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

You can read messages in the forums without joining P2P, but in order to post your own messages, you must join.

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the Web. If you would like to have new messages from a particular forum e-mailed to you, click on the “Subscribe to this Forum” icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

1

Official Records

Records management is driven primarily by regulatory compliance and the need to reduce the risk of exposure to legal liability for improperly managing information. The prospect of paying significant fines for not adhering to an increasing array of regulatory requirements provides a strong incentive for companies to implement comprehensive records management solutions. But there can be equally significant costs associated with implementing those solutions that must also be considered. There is the potential for lost productivity as knowledge workers spend more and more time focusing on records management issues. There is also the potential for increased IT costs as additional time and energy must be devoted to building and maintaining the records management infrastructure.

That infrastructure has requirements as well. Not only must it support the identification and handling of *official records*, but it must also provide the same tools for collaboration, approval, and workflow that other knowledge workers employ in their day-to-day work. Compliance officers are knowledge workers too, and the job of managing official records can be even more daunting than creating them.

Microsoft has published a comprehensive records management planning guide that is essential reading for anyone involved in the development of records management policies and procedures, regardless of whether you are using MOSS as an implementation platform or not. You can download the guide from <http://go.microsoft.com/fwlink/?LinkID=92720>.

What Are Official Records?

Don't you hate it when you ask someone a question and they respond with the generic answer, "that depends"? I often find myself thinking, "This person obviously doesn't know the answer to my question and is just stalling for time so they can come up with something intelligent to say!" You see where I'm going with this, right? When thinking about records management, the first question that often comes to mind is, "What is an official record, anyway?"

Chapter 1: Official Records

When does an ordinary document become something more? When does it “officially” become something you have to treat specially because of some legislation or corporate policy or other business process that may govern what you might otherwise do with it? Well, I hate to do this to you right off the bat, but the only answer I can give you is, “that depends.” Let’s dig a little deeper, and you’ll see what I mean.

To answer this simple question, there is a lot to consider. Is there something about the document itself that determines whether it is considered an official record? In the simplest case, that determination might flow from a special set of values in one or more document properties. In other cases, it might come from the presence of a particular kind of content within the document. For example, the document might contain sensitive information that only certain people should be allowed to see. Ultimately, particularly in the context of highly collaborative environments like SharePoint, the final determination of what constitutes an official record may have as much to do with how a document is used as it does with the document content or with the metadata associated with it.

Traditional records management systems are focused primarily on archival mechanisms. We could characterize them as *location-based*, where the physical location of the document is the dominant consideration.

If physical storage were our only concern, then building records management solutions would be fairly simple. Assuming we had a consistent platform for associating metadata with a file, we would basically only need an established process for storing the file in a secure repository so we could find it again easily. Obviously, we would also need to apply some discipline in categorizing the metadata according to a given set of rules, but this would be a relatively straightforward process.

The more challenging scenarios involve concurrent processes that may have nothing at all to do with the policies or rules we are trying to enforce. This is where SharePoint distinguishes itself significantly from other systems. Dealing with official records in the context of dynamically changing metadata without interfering with critical business processes represents a quantum shift in the way we think about official records and the software components needed to manage them effectively.

Core Records Management Principles

What are the core principles of records management? Here, I’m talking about the essential characteristics of any records management system. At a minimum, there are four: confidentiality, information integrity, adherence to policy, and auditability. If any one of these capabilities is missing, then that system does not provide the necessary control points for ensuring that a given organization is meeting the regulatory requirements driving the decision to implement the system.

In the SharePoint environment, we can add one additional core requirement: high availability. This gets back to the notion that whatever controls you put in place for managing official records should have minimal impact on normal business operations. For collaborative documents that support multiple business processes, it means that you must implement solutions that do not hinder those business processes or increase their cost.

Now look at these principles of records management more closely:

- ❑ **Confidentiality** — Confidentiality requires that the records management system must ensure that strict access controls are maintained for any official record so that only those persons and groups with appropriate permissions are able to view the record. Any deviation from such access controls must also be properly monitored and recorded.

- ❑ **Information Integrity** — The records management system should provide a way to check the integrity of a record’s metadata as well as its content. This could require the enforcement of rules that govern the range of property values that are considered *valid* for a given set of metadata fields. The system must also ensure that neither the content nor the metadata associated with records is altered after they have been placed into the repository.
- ❑ **High Availability** — Official records must be available at all times to support processes that may or may not be related to core business functions, such as litigation support or compliance research. This requirement of high availability often underscores the need to decouple the Records Repository from other enterprise information stores so that any request for official records is not tied to critical business processes.
- ❑ **Adherence to Policy** — *Adherence to policy* means that there are rules that govern how particular types of records must be handled by the system. For policies to be implemented effectively, they must be defined clearly and must also support administrative proof that a given policy was followed for a given record instance.
- ❑ **Auditability** — The auditability requirement means that there must be an efficient and secure mechanism for keeping track of everything that happens to a record. This typically includes changes to the way in which the auditing itself is implemented. Thus, audit records are also treated as official records. Consequently, the records management system must provide tools for ensuring that the auditing features are also tamper-proof and that the audit records are securely maintained.

Content Modeling

What are the essential characteristics of content that enable effective records management? Is there a methodology we can use for deciding what metadata to associate with a given content element? Ultimately, we want tools that not only enable us to build detailed models of our metadata design, but also to bind the resulting metadata to our code consistently and in a reusable fashion so we can avoid having to build the same model repeatedly.

What we have today is a pretty haphazard definition of what metadata consists of and how it should be defined. There are different kinds of metadata coming in from legacy document management systems and from unstructured documents. The properties attached to those documents haven’t necessarily been defined in any consistent or rigorous manner. Consequently, when we’re building records management solutions in SharePoint, we run into the problem of matching columns to properties and making sure that each document being managed by our solution has appropriate data in the right places. This can be a big problem, and it can get very complicated. There are third-party tools cropping up that purport to provide mechanisms for mapping properties, and thus the concern is not so much about the actual mechanics, but about the methodology used for figuring out which properties are important for a given scenario and deciding what we’re going to map those properties to.

Another way that metadata is being determined now is that there are often many different copies of the same document floating around in the enterprise. In a perverted kind of way, the more duplication there is of a particular document, the more we could say that document may increase in value. Because it is being used so much, it must have more value than other documents. It becomes a sort of *high-value target* as a starting point for further analysis. By examining frequently copied documents, we may develop a rudimentary understanding for how the document is being used at different times. As an example, let’s say that we have an annual report that is being created consistently every year. But then

at some point, new management takes control of the company and they need new information, and the annual report morphs into a different kind of document that includes additional reports. Another example might be an expense report that has new expense items that must be included. Whenever a document that is attached to one business process is applied to different business processes, the result is that the metadata gets messy very quickly. Certain properties support either the original process or the new one, while other properties support both.

Generally, there is no systematic method for determining what metadata is needed for a given business purpose. Yet the paradigm keeps evolving, and we're moving forward with this content explosion. We have no choice but to move more and more content into SharePoint based on a pretty informal methodology. To deal with this, it helps to think about how the metadata is going to be used. If it is simply for classification and categorization, it may not be as important to have formal methodology for figuring out what the metadata should be. But if it is going to be used to drive business process automation, it becomes vitally important to have a methodology so that we can consistently determine or derive the correct metadata because we need to coordinate multiple processes and the activities being performed by people connected to those processes. Where is that coordination going to come from? It's going to come from something attached to the document or to some other object that describes the document. In a traditional document management system, we might have a database record that includes this metadata. In SharePoint, we can attach the metadata directly to the document, associate it with a list or content type, or embed it into a workflow activity.

Understanding the Content Life Cycle

The content life cycle provides a useful model of the way humans interact with information, and it can be represented as a simple sequence consisting of four phases: creation, review-and-edit, publication, and final disposition. Depending on the type of content being modeled and the way it is used, the review-and-edit phase may be repeated as often as needed.

It is worth noting also that this is a recursive model, meaning that the entire sequence can be embedded within any individual phase to drill deeper into the analysis of the content-driven workflow. For example, the process of creating an annual report could be modeled as a sequence of steps starting with the gathering of initial metadata to create the document instance, review-and-edit of the content and underlying assumptions, approval and publication of a series of drafts, and then final disposition to a shredder or long-term storage repository.

But the process of gathering the metadata could itself be modeled in the same way. This is because what we call *content* depends on many factors, not the least of which is the context of the analysis. Figure 1-1 illustrates the basic sequence.

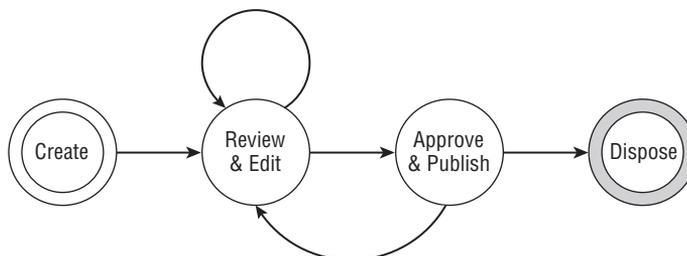


Figure 1-1: Typical content life cycle.

The content life cycle can help a lot when building ECM solutions because it provides a context within which to understand and tease out the requirements needed to deal with chaos. The content life cycle informs further analysis of those requirements, even though our current tools for creating content provide no consistent data model that can be leveraged to answer basic questions about the information we use every day to perform common business functions. The key lies in the fact that content drives all business processes. If we can clarify just what kinds of interrelationships tend to help or hinder those processes, we can define and identify critical metadata that can then be used to manage the transformation of content from one stage to another.

At each stage of this life-cycle model (creation, review-and-edit, publication, and final disposition), different tools may be used to manage the production or consumption of information at that stage. Having the right tools at the right stage can have a significant impact on whether a given business process is successful. It can certainly influence how quickly an organization can respond to changes in the underlying information. Therefore, it is useful to explore ways to leverage whatever can be known about the life cycle of a particular kind of content to understand how that content relates both to the individual producers and consumers of that content and the environment surrounding how that content is used.

Consider what happens during the content creation phase. Whether using a server or client application, the way that content is created is driven at least in part by the role being played by each person who interacts with the content or with the initial metadata used to generate that content. As an example, if a team is producing a periodic report such as an annual report for a company or a quarterly report for a department, then the appropriate tools must be available for each team member to create the required metadata before the report can be created (whether it is generated or being written by hand). Once it is created, it then moves on to the next phase in its life cycle and may require different tools at that point.

What's interesting is that the role being played by each team member as well as other essential characteristics of the content itself help to define what tools are needed at each stage. This intersection between the content life cycle, the roles of each content producer and consumer, and the business process to which the content is being applied defines a "nexus of opportunity" that can be exploited to achieve new levels of business efficiency. The trick will be to find an effective technique that pulls all of the required elements together in a way that engages all stakeholders. First, we'll lay out some high-level goals for our content modeling technique, and then we'll look at role/activity modeling, which addresses these goals.

Content Modeling Goals

Goals that you should keep in mind as you develop a content modeling methodology generally include the following:

1. Understand the essential characteristics of content that can enable effective management and control.
2. Develop a methodology for deciding what metadata to associate with a given content element.
3. Develop tools for binding content metadata to code in consistent and repeatable patterns.
4. Build an effective strategy for using content to support business process automation.
5. Be precise and not require heavy IT skills or sophisticated modeling tools.

Ideally, the modeling methodology should not require heavy technical skills. The target audience will include both developers and knowledge workers who typically understand business processes better than IT does. We don't want to introduce a modeling technique that is overly technical. At the same time, it has to be precise enough to support the kinds of quantitative analysis we need for software component building in order to respond to the business requirements.

The final goal is that we don't want to have to use sophisticated modeling tools. We should be able to create these models easily using something no more complicated than Microsoft Excel. We should also be able to create visual models easily that relate directly to the actual modeling activity using something like Microsoft Visio so that we can easily share the model with knowledge workers, developers, and architects.

The Role/Activity Modeling Technique

We use role/activity modeling to identify the content that drives specific business processes, and then we classify the content so that we can map it to different parts of the solution. The remaining content is grouped for highest efficiency, such as by predominant role, security requirements, source location, and so on. Finally, we identify and build domain-specific components (such as timer jobs, event receivers, and workflow activities) around those content elements. This results in a set of components that is more tightly coupled to the business process being automated and greatly simplifies the construction of solutions around that business process.

This technique is particularly useful for workflow development because the components you identify can be implemented as custom workflow activities that can then be added to SharePoint Designer to enable business analysts to easily add domain-specific workflow support to any SharePoint portal.

The role/activity modeling exercise begins with a guided discussion to identify the major roles that will participate in the solution and the primary responsibilities that will be assigned to those roles. Then we walk through a very informal, but still structured analysis to determine what the main activities are that are required in order to fulfill those responsibilities. Within each of these activities, we then determine what information is essential to performing the tasks we have enumerated. As it turns out, this is a very useful analysis when dealing with any kind of portal because in the portal environment, you are presenting users with tools for performing tasks that fulfill responsibilities to which various roles have been assigned. And so that maps pretty nicely into, for example, an Active Directory permission-granting environment wherein you are providing certain tools to certain individuals and information is constantly flowing into and out of the system. Figure 1-2 shows the steps of the technique.

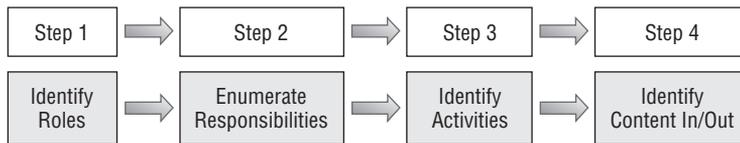


Figure 1-2: Role/activity modeling.

Here, we are making a few basic assumptions about the portal environment, such as the ability to capture metadata in the form of lists, the ability to control access to documents, the ability to build comprehensive workflows, and so on. Typically, what is needed prior to doing this kind of analysis is to build the necessary semantic tools to do the envisioning at a higher level. From the perspective of a business

analyst who may be new to the SharePoint environment, an essential ingredient is helping them to develop a realistic understanding of what features SharePoint provides out-of-the-box and what it does not, and then move on to a broader strategic vision that includes guidance for how those features can be applied to solving the particular business problem. In order to use the analysis as a foundation for building solutions, it is also necessary to identify reusable patterns that can be combined to reduce the overall complexity of a given problem by making it possible to decompose the problem into smaller, more manageable pieces.

I like to think of SharePoint in two dimensions: space and time. If you view it from the spatial dimensions, then the *share point* is the location in space at which you place information to be shared with others. It's like a glorified file share. It's a physical location for sharing information. Alternatively, if you look at it from the temporal dimension, then the *share point* is the point in time when collaboration becomes essential to your business process. In other words, there is a point in time when the complexity of working with shared content becomes so great that you have no viable alternative but to share the information. The solution requires more consciousness, so you have no choice but to collaborate. *SharePoint* is therefore a great name for this platform because it truly captures the spatial and temporal complexities that are inherent in the work we do and the activities we have been tasked with.

Consider the platform itself. We have a set of sites that support large numbers of users. We have a fixed topology that imposes a certain structure on the information stored within it. Perhaps there is a top-level site with departmental sites underneath it and end-users have the ability to create their own sites beneath those. With so many sites being created and so much information being stored in so many ways, the spatial constraint is to reduce the unnecessary duplication of information. But it's a temporal constraint as well. We want to enable users to find and reuse information as much as possible, just in the interest of time. Otherwise, they will fall into the same behavioral patterns they used when all they had were file shares. The reason there were so many duplicate copies of the same document was because someone couldn't find that first copy they created last year that was exactly on point for what they are doing now. Even if it was only 80 percent on point, they just don't have time to look for it.

Enumerating Roles

We need a way to determine what the workflow patterns look like because there are many ways to structure information to support those patterns. But there can be a real disconnect between the collaboration platform and what people in the organization are being tasked with. In many ways, collaboration presents a paradigm shift for the average information worker, who is used to working on a separate *island* of information. Now, there are other *nearby islands* they need to collaborate with, but they are still thinking in terms of *my island* and *your island*. They are not yet comfortable with the *shared repository* idea, especially when it potentially affects their ability to deliver work product.

The role/activity modeling exercise helps to clearly state and enumerate the roles that are involved. There are clearly defined responsibilities. That's an embedded relationship. You can't have the same responsibility in more than one role. By enumerating the list of responsibilities associated with each role, you have a way to identify the primary forces working against the collaboration initiative. When it comes down to it, people are motivated by their need to fulfill their perceived responsibilities. Adopting a new paradigm requires effort. It requires time to learn the new tools. It requires a commitment to learning. It is easier to persuade people to invest the additional time and energy if they think the new paradigm will better enable them to fulfill those responsibilities.

If you roll out a portal that does not facilitate the individual's ability to perform his or her assigned tasks, you are increasing the noise they have to deal with, and you are increasing the effort they must

expend. There is then little incentive for them to adopt the new paradigm. One of the strategies that Microsoft started with was to integrate the collaboration platform tightly with Office. This gives you the ability to build on knowledge they already have. You have the ability to *extend* rather than *shift* the paradigm. This works well for both the collaboration and aggregation activities.

Enumerating Responsibilities, Activities, and Tasks

The role/activity modeling exercise can really help to clarify and focus the structure of an information portal. Beyond that, it can also provide a foundation for building analytical models that can guide the construction of component libraries so they provide the essential tools required by users to meet specific business objectives. Figure 1-3 illustrates the relationships between the different parts of the model.

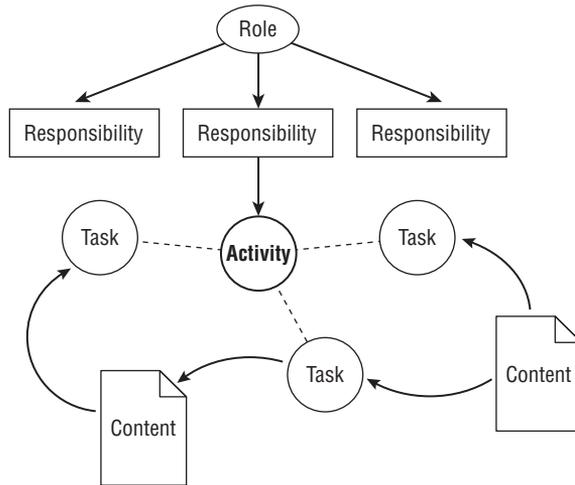


Figure 1-3: Role/activity modeling elements.

People don't really have a choice with regard to their responsibilities. It's part of their job descriptions. They must fulfill their responsibilities in order to satisfy their job requirements. The following table shows some common examples of roles and responsibilities.

Group	Role	Responsibilities
Administration	Administrative Assistant	Managing communications and scheduling activities to alleviate executive information workload
	Accountant	Collecting, auditing, and reviewing financial documents and related information Creating financial reports
	Bookkeeper	Collecting and distributing funds and issuing receipts Maintaining financial records

Group	Role	Responsibilities
Compliance	Operations	Ensuring that all operations are running smoothly Troubleshooting problems as they arise
	Comptroller	Ensuring that legal documents, contracts, and releases are properly signed, received, and filed
	Compliance Officer	Ensuring that the organization acts in accordance with governmental rules and laws as well as internal policies and guidelines Ensuring that governmental reporting requirements are met
Media/PR	Outreach	Contacting and maintaining relationships with the press
	Marketing	Identifying advertising opportunities Creating marketing plans Developing PR campaigns Interfacing with graphic designers Developing branding requirements and specifications
	Press Coordinator	Developing and approving talking points for media spokespeople Coordinating with the media outreach specialist Developing and maintaining lists of media spokespeople Developing and approving press materials Developing and maintaining a database of media contacts Interfacing with the publications group to ensure that materials are created
Human Resources	HR Manager	Identifying the personnel needs of the organization Recruiting, hiring, and managing the staff
	Mediator/ Counselor	Reviewing trends to identify potential problems Pulling together the appropriate resources to help people deal with difficult situations Creating a mechanism to enable people to find the support they need Acting as a mediator/facilitator
	Team Builder	Identifying needs, creating teams, and inspiring people to join them Creating and designing activities to build team identity and cohesiveness and sustainability within the team

Continued

Chapter 1: Official Records

Group	Role	Responsibilities
	Skills Manager	Creating and disseminating training programs and materials to keep staff and teachers up-to-date with new policies, programs, and legal requirements Identifying gaps in skills and developing tools for updating individual skills and ensuring that mandated policies are effectively applied

Getting people to simply articulate their understanding of what their responsibilities are can be helpful in itself. You can organize them easily in a simple spreadsheet with columns for Role, Responsibilities, Tasks/Activities, and Inputs and Outputs. You don't have to organize them into an elaborate model — just simply enumerate them. Later on you can project them onto a site topology that identifies the most appropriate tools that support those activities. They don't have to be connected into a workflow, either, although the ultimate tool would be one that ties the inputs and outputs to other activities within the same business process. That would yield a modeling platform that provides everything needed to map the process all the way down to the individual content elements flowing into and out of the system. Until we have such a tool, just getting users to articulate their understanding of the roles and responsibilities can be valuable.

A Simple Example

As an example, consider the role of compliance officer in an organization. The compliance officer is responsible for ensuring that the system is properly enforcing the core requirements of any records management system, namely, confidentiality, auditability, high availability, and so on. Figure 1-4 illustrates the technique applied to this role.

These are unique responsibilities, because the same responsibility cannot be associated with more than one role. Also, high availability in the context of records management does not mean up time, which might be the responsibility of an operations staff or system administrators. It means ensuring that the information needed to manage a given set of records is always available and that the ability to comply with regulatory requirements is not compromised by day-to-day interactions with the affected documents.

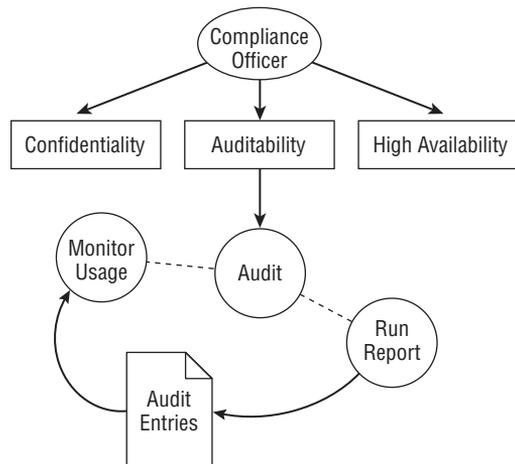


Figure 1-4: Role/activity modeling example.

Within the *auditability* responsibility, we want to derive a list of tasks that are essential to fulfilling that responsibility. When we are done, we want to be able to say, “If I perform these tasks, then the responsibility is fulfilled.” So the tasks must be tied directly to the fulfillment. Within each task, there may be several activities that are part of the task. This just gives us a way to capture iterative processes that taken together make up the set of actions that must be performed. Keep in mind that each activity may feed other tasks. They may be ordered or unordered — it depends on the task.

Within each activity, there is information that is essential to performing the activity. Without that information, the activity cannot be done. We need to enumerate those pieces of information as well. Similarly, as a result of performing an activity, there is new information that is produced. We can say that every activity must produce new information or it is not an activity we are interested in modeling. However, metadata can also fulfill this requirement, so the very fact that an activity was initiated can be information that is produced as a by-product of performing the activity.

Typically, list items, documents, or e-mails are produced or consumed by the activities when modeling human interactions. The same technique can be applied to system level workflows; however, different kinds of information may be produced or consumed — perhaps more fine-grained. For example, when dealing with business-to-business workflows, we may be talking about packets of information in the form of WCF messages instead of actual documents.

After going through this process, you can now build a site topology that presents all of the tools and information needed for a records manager or project lead to fulfill the responsibilities to which they’ve been assigned. You could provide a *Records Manager’s Workspace* that includes links for performing a specific set of tasks. You could go further and limit the available tasks to only those that have been enumerated during the role/activity modeling exercise. You could also think about the web parts or custom Web Services needed to support those activities, whether they are presented sequentially or in random order.

For any given model, the level of granularity needed when defining the roles can vary. They can be specific to an industry, especially in the context of regulatory compliance that often involves roles whose responsibilities are driven directly by constraints imposed by a particular piece of legislation. One approach is to actually define the roles as off-shoots of each regulation. So, for example, we could define roles such as *HIPAA Compliance Officer* and *SOX Compliance Officer* and model the solution based on the specific regulatory issues faced by those users. Such an analysis would naturally lead to HIPAA-specific web parts that simplify searching for records that are affected by HIPAA rules, or a SOX-specific file plan that captures SOX-related metadata that, in turn, is used to set up the Records Center site to work with those records, and so on. Figure 1-5 depicts what this might look like.

The end-users involved in a records management application are not users of the Records Center site, but users of the collaboration portal — collaborative end-users. These are the people who are interacting with documents daily. These documents may or may not be promoted to official record status. In the simplest case, there are scenarios wherein the end-user selects a document and sends it manually to the Records Repository. Other scenarios involving end-users might require specialized tools beyond the simple send to repository functionality provided by the SharePoint UI. As an example, you might want to enable users to send many documents to the Records Center at once, and consequently may need to provide tools that allow them to specify all of the metadata for those records in a single batch.

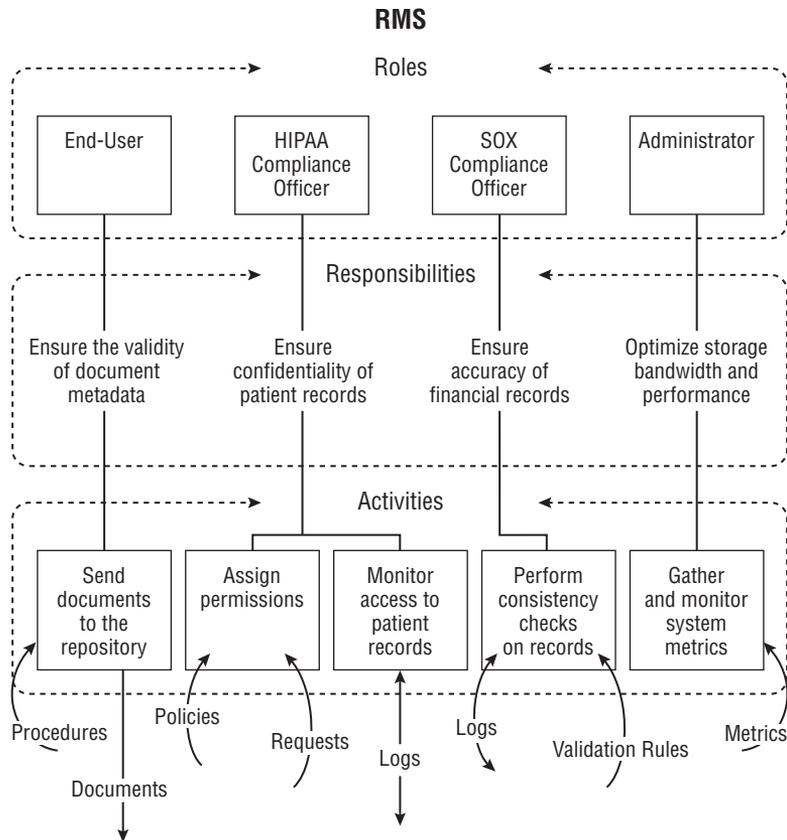


Figure 1-5: RMS role activity model.

Factors to Consider When Applying the Technique

There are several useful factors to consider when using role/activity modeling:

- ❑ The discovery process empowers knowledge workers.
- ❑ Models can be created quickly using familiar tools, making the modeling exercise approachable by business analysts, developers, and end-users.
- ❑ Modeling artifacts (such as spreadsheets and Visio diagrams) directly support software component building and can help clarify and drive other development activities.

Discovery Empowers Knowledge Workers

Just by going through this discovery process, knowledge workers may suddenly understand what their responsibilities are, whereas before the exercise, their understanding of this might have been vague. It forces everyone to be on the same page in terms of identifying what the requirements are for a particular set of content elements and responsibilities. Another thing to observe is that it's very approachable both by business analysts and by those who are not business analysts, but who know what their

business responsibilities are. By being engaged in the role/activity modeling exercise early in the solution development cycle, knowledge workers can gain deeper insights into their business function and understand their existing responsibilities in new ways that can improve their performance. Common understanding forces everyone onto the same page, enabling them to share responsibilities more effectively and work better as a coordinated team, lessening the perceived need to take ownership of parts of the process that are beyond their immediate control.

Roles and Activities Are Easily Understood

In the case of one large volunteer organization (with more than 10,000 members), we pulled together people from IT, people from management, and people from the board of directors and the oversight group, as well as people who were being assigned these tasks, such as accounting and HR, with a wide range of skills and experience, but everyone in the room was comfortable with Excel. No one pushed back on the process. This enabled us to reach consensus quickly and figure out exactly what they needed in this particular portal environment.

Without a methodology like this, it all falls on IT to figure out what everyone needs. Experience in several different projects has shown that there can be a substantial time lag between when IT comes up with a design and when they present it to those who are going to use the SharePoint portal in their day-to-day activities. This increases the chance for failure if there was never any clear enumeration of roles and responsibilities. Applying this methodology can really help with user adoption of any kind of portal design.

Content modeling and data validation are interrelated. When building a content model, there is a natural progression from the role/activity modeling exercise to the construction of a more detailed XML schema that describes the content and its interactions with various business processes. From that point onward, an API can be developed gradually using generated classes that are extended with domain-specific interfaces as they are discovered.

Modeling Artifacts Support Component Building

After the role/activity modeling exercise is complete, you will end up with some number of spreadsheets and diagrams that describe the model elements and how they are related. We can call these *artifacts* of the modeling exercise. What's really interesting is that by defining these artifacts in a structured way, we now have tools that we can use to go ahead and begin declaring our content types. The modeling exercise has already identified the major content types as well as their interdependencies. What is left is the actual fields that make up each type. The field derivation is fairly straightforward at this point. Just examining the tasks and looking at what they consume and produce is enough to identify existing site columns and field types or to design new ones. Finally, the context provided by the roles and responsibilities enables us to create separate schemas that describe each type in context as it moves from one phase of its life cycle to the next.

This means that you can build XML schemas to support various layers of the solution. As an example, consider a meeting agenda for a board of directors. This information is required by the board secretary, but is also required by the individual board members. So the meeting agenda has a life cycle that interacts with both roles. Therefore, when defining the metadata for the meeting agenda, you can focus on the metadata that is specific to the board secretary and consider that metadata separately from the same meeting agenda when it is being used by a board member. Likewise for any of the other items, you can easily see when a particular content element is being referenced or consumed by multiple roles.

As we apply the role/activity modeling technique to records management, you will see that this ability to capture high-level details of the content life cycle in a schema that maps each phase to a discrete set of roles and activities makes the XML schema an important tool for building any kind of content management solution. In the next section, we'll examine what are the most important elements of such a schema that could be used specifically for managing official records.

Developing a File Plan

At the heart of any records management system is the *file plan*. Records managers and compliance officers are accustomed to creating file planning worksheets that describe the kinds of documents that their organization will treat as *official records*. The *file plan* describes where each type of record should be stored, how long it should be kept, and the manner and conditions under which it will be archived or destroyed. A traditional file plan may also include additional information that is used to categorize documents and to assign tasks to the persons responsible for managing each record type.

The fundamental concept of an *official record* is intended to convey the notion that at some point in its life cycle, a document may serve as evidence of some transaction that has taken place within the organization. For instance, when a contract or legal agreement is signed, then the contract itself serves as evidence of the agreement. It's not the *only* evidence of the agreement, but taken together with other evidence, it can serve to clarify the intentions of both parties. Therefore, it should be possible to somehow *freeze* the document in its current state such that a snapshot of the document is stored in the system so that it can be retrieved and reviewed later.

For certain scenarios, taking an actual snapshot of the document may suffice. However, storing only an image of the document is not as useful as keeping all of its metadata, macros, embedded objects, and so on.

The following table lists the basic elements of a file plan, as defined by Microsoft. They provide a starting point for analyzing documents and describing the types of records needed to manage them. The resulting worksheet can then be used to design an effective MOSS Record Repository.

Element	Description
Record Type	This is the name of the record type and typically matches the name of the content type associated with the document.
Description	This is a brief description of the record type that should be targeted at content managers so they understand the rationale for "promoting" the document to an <i>official record</i> .
Media	This describes the format in which the record is stored, such as MP3, HTML, Word 2007, and so on.
Category	This is a general categorization that can be used to group similar record types together, for example, when deciding which document libraries will house submitted documents.
Retention	This is a statement of the required retention period for records of this type. The retention statement will ultimately be used to determine how the expiration policy for the document is configured. It should therefore include all of the information needed to perform that configuration.

Element	Description
Disposition	This is a description of what will happen to the document when its designated retention period ends. This will be used to select the appropriate expiration action for the document and will therefore also influence the configuration of the expiration policy applied to the document.
Contact	This is the name of the compliance officer or content administrator assigned to documents of this type.

Typically, this information is captured in a spreadsheet and then is referred to by all members of the compliance team, which may include lawyers, business analysts, and IT personnel. This spreadsheet can become a useful artifact because it can be easily uploaded to a document library, revised, approved, and then used to drive the manual construction and maintenance of the Records Repository. From this one document, all of the required content types can be identified and linked to the appropriate routing types. Custom routers can also be built and installed if necessary, and document retention policies as well as other information policies can be configured and tested.

Ultimately, we'd like to move beyond the manual model represented by the *static* file plan described above toward a more automated approach, wherein a *dynamic file plan* is used to drive the process of adding the required components into an existing Records Repository. An automated or semi-automated approach would fit in well with the day-to-day operation of a typical Records Center by enabling compliance officers and content managers to deal more effectively with constantly changing requirements and regulations. Ideally, we'd like to publish the file plan using a set of SharePoint features. But before we can delve into the mechanics of building dynamic file plans using the MOSS Records Center API, we first need to cover the manual steps that are required to set up a Records Repository using the out-of-the-box Records Center site template.

Identifying Roles and Responsibilities

The process of associating roles and responsibilities with the file plan flows very naturally from the role/activity modeling exercise. We simply copy the roles into the file plan, describing each one using information that we've captured in the spreadsheet. We can follow a similar process when enumerating the responsibilities, keeping in mind our rule that there can be only one role associated with each responsibility. By enumerating the roles within the file plan, we can more easily see how different aspects of the plan are affected by and serve the various roles.

In this way, we can think of file planning as an extension of the role/activity modeling exercise, the key difference being that whereas file planning is content-centric, role/activity modeling is more process-driven. This translation from process- to content-centric views while creating the file plan is key to developing a clear understanding of how a particular kind of content drives the business process. The role/activity modeling exercise then becomes an essential preliminary step so that we thoroughly understand the process before attempting to refine our understanding of each content element that feeds it.

Identifying Applicable Policies and Procedures

To develop a comprehensive file plan that can support different operations at various stages of the overall processing life cycle for a record, it is important to identify the policies and procedures that govern what happens to the record at each stage. One approach might be to define the policy very simply as

just a descriptive text statement attached to each record. That text could then be presented to the content manager or to other persons involved in the document processing sequence, and could explain the purpose of the policy and the reasons for its inclusion, perhaps tying it back to a particular piece of legislation or to a particular organizational rule or procedure.

Another approach might be to attempt a more granular description of the policy by breaking it down into its constituent parts. Using the expressive power of XML, for instance, we could begin with a Policy element and then identify different parts of the policy as nested subelements. Those subelements might refer to other elements in the file plan, such as the roles and responsibilities most affected by the policy, and so on.

Identifying Custom Routing and Workflow Requirements

As we develop a deeper understanding of how a particular content element feeds the business process, we're really starting to talk about workflow. We're looking specifically at how a particular content element flows out of the collaborative environment and into the more structured and restrictive environment of the Records Repository. We are basically using the file plan to define a path for the record to follow, and as part of that process we are examining what happens to the record as it moves along that path.

This is where custom routing requirements may begin to emerge. If, for example, we determine that the primary requirement for handling a given set of documents is to keep track of what happens to them, then we may need enhanced tracking that is more detailed or comprehensive than the tracking mechanisms that are provided out-of-the-box. For other record types, such as patient records, the dominant requirement might be confidentiality, in which case, we may need a special router that attaches more restrictive permissions to the record when it is stored.

Identifying Document Categories and Groups

Categorization is a natural part of any content processing mechanism. We do this automatically whenever we decide what name to give a folder that will contain new documents that we create. We do the best we can to identify the proper folder, and we typically name the folders based on our current understanding of the document's intended purpose. Later, the primary purpose of the document may change, and we often end up copying the document to a different folder and renaming it.

When building a file plan, it is important to capture as much information as possible about the groups a given record might belong to as well as the categories it might be associated with. This supports the construction of different views of the record while it is being processed. You can also use the categorization information to define queries for retrieving and manipulating only those documents you are interested in.

Identifying Document Sources

The process of finding documents that match the file plan can be highly subjective. Describing the sources of those documents in an objective way can help to simplify the process and can also provide a foundation for building expertise around the file plan that is not dependent on any one person. One way to do this is to include information that can be used to create a query for finding the documents to which the plan applies.

This is technically more of a document management than a records management activity, because it applies to *active* documents that have not yet moved into the records management process. However,

capturing the relevant metadata, location, keywords, and other properties that determine whether the plan applies to a given document is an important part of the planning process, and the file plan is the most convenient location for storing this information.

Analyzing Storage Requirements

Storage information can be included in the file plan to support capacity planning for the Records Repository and is related to the identification of document sources. Using a query to identify and locate record candidates would allow you also to calculate the probable size of the Repository before actually constructing it. This information might also influence how the Repository is organized.

Analyzing Security Requirements

Security and access constraints should also be included in the file plan so that permissions can be set up automatically for certain documents. For example, if you're developing a file plan for healthcare documents that are governed by the Health Insurance Portability and Accountability Act (HIPAA), then confidentiality rules may apply such that when the records are processed, only certain people are allowed to see them. Having a description of these rules in the file plan enables any processing components to assign the required permissions to the appropriate people or groups.

Another aspect of security is the identification of the persons who will be responsible for the management of the record once it moves into the repository. Thus, there may be multiple levels of security that must be described. One approach is to identify a *records manager* with a certain set of permissions and then to add a separate list of roles and their permissions.

Summary

Records management is driven by regulatory compliance, which imposes core constraints on any records management system. These include confidentiality, information integrity, high availability, adherence to well-established policies, and auditability. The MOSS Records Management infrastructure provides tools that support each of these requirements.

This chapter introduced the concept of *content modeling*, which helps to develop more effective records management solutions by identifying the key content elements that drive a business process. The role/activity modeling technique was introduced as a straightforward methodology that identifies the primary roles and responsibilities involved in a business process and then maps them to the tasks and content elements needed to fulfill them. This approach is easily understandable to business analysts as well as to knowledge workers and IT staff, which makes it an ideal candidate for bridging the gap between the groups involved in the development of business process automation solutions. The role/activity modeling technique can be applied easily using readily available tools like Excel and Visio and offers a way to ensure consistent binding of metadata to content elements, the people who interact with them, and the business processes that produce and consume them.

At the heart of any records management strategy is the *file plan*, which is a document that describes the criteria that determines which documents shall be treated as *official records* and how they should be processed. This chapter enumerated the essential parts of an effective file plan and showed how the file plan can provide a strong foundation for further development activities.

2

Preparing for Records Management Development

There are some basic considerations that come into play when building a solution that will support records management activities. The first things to think about are the roles involved in the overall workflow. The business processes for records management may include collaborative workflows, but they may also include other workflows that are driven by compliance needs or auditing needs and that generally involve closer oversight and have stricter rules associated with them. So in this chapter, you'll see some of the key roles that are involved in a few of the more common regulatory compliance scenarios.

Some tools and techniques for SharePoint development will be identified that can help to maximize your productivity when building records management solutions. More specifically, we'll look at ways to enhance the Visual Studio development environment by installing some popular utilities that are freely available on the Internet, taking advantage of some off-the-shelf tools that are included with the Visual Studio product, and building some additional tools of our own.

Understanding Current U.S. Regulations

Since records management is driven primarily by regulatory compliance, our solution development strategy depends on first understanding the relevant regulations so that we have a better idea of the roles and responsibilities facing organizations having to deal with records management. These regulations specify rules for handling e-mail, privacy, accuracy in financial reporting, identity security, and other records management issues. Following are some of the most important U.S. regulations that you might need to consider in developing your solution development strategy:

- ❑ **DoD 5015.2** — The DoD 5015 standard is often referred to as the *gold standard* for records management software systems in the United States because it is so well-known and widely used. It provides detailed guidance for the proper management of official records and defines many of the criteria that companies use to determine whether a given software system includes the features needed to manage records in a way that satisfies the strict

requirements of government agencies. These criteria include the management of e-mail records, the creation and processing of complex document retention rules, and the ability to automate the periodic review of records that are deemed critical to the operation of an enterprise. For more information, see www.dtic.mil/whs/directives/corres/pdf/501502std.pdf.

- ❑ **Health Insurance Portability and Accountability Act (HIPAA)** — This legislation was enacted in 1996 to protect health insurance coverage for workers and their families when they change or lose their jobs. It includes specific provisions governing the creation and distribution of electronic health records that potentially cover all health care transactions and requires the establishment of national standards pertaining to the security and privacy of those transactions. These standards include the establishment of national identifiers for employers, health care providers, and health insurance plans. For more information, see www.hipaa.org/.
- ❑ **Sarbanes-Oxley (SOX)** — There are 11 provisions, or *Titles*, in this bill, two of which directly affect the management of financial reports. Title III requires that corporate officers take individual responsibility for the accuracy of corporate financial reports and specifically requires that they personally certify and approve those reports every quarter. Title IV requires internal controls for assuring the accuracy of financial reports and requires both audits and reports on those controls. For more information, see www.soxlaw.com.
- ❑ **Gramm-Leach-Bliley Act (GLBA)** — This legislation was enacted by Congress in 1999 to protect the privacy of personal financial information maintained by banks and other financial institutions. In particular, the so-called “Safeguards Rule” requires financial institutions to develop and maintain an information security plan to protect each client’s private information. As part of this plan, the rule requires the institution to develop actual safeguards for the information and establishes procedures for verifying the effectiveness of those safeguards. For more information, see www.ftc.gov/privacy/privacyinitiatives/glbaact.html.

This book does not include any material that references the Model Requirements for the Management of Electronic Records (MoReq) specification that is gaining popularity in the European Union. While the MoReq and MoReq2 specifications provide a thorough treatment of the key issues involved in the design and construction of electronic document and records management systems, a detailed examination of MoReq in the context of MOSS is beyond the scope of this book. For more information about the MoReq specification, visit www.moreq2.eu.

Setting Up Your Development Environment for Maximum Productivity

Although Visual Studio is widely regarded as the de facto standard development environment for building SharePoint solutions, it does not include many of the tools needed to build real-world solutions. The good news is that there are many great tools available on the Internet, but they must be downloaded and installed separately. These tools are concerned primarily with creating various SharePoint components, such as web parts, site and list definitions, and generating SharePoint solution packages. The following list is provided to point you to a minimal set of tools to get you started:

A more complete list of developer tools and resources is available on the SharePoint Developer Network website. This is a public portal I created to support professional SharePoint developers. The site includes training resources, job listings, and other content that you may find useful. For more information, visit <http://sharepointdeveloper.org>.

- ❑ **Visual Studio Extensions for SharePoint** (www.microsoft.com/downloads/details.aspx?familyid=7BF65B28-06E2-4E87-9BAD-086E32185E68) — This is an extensive set of tools designed by Microsoft to provide support for creating SharePoint solutions in Visual Studio. It includes project templates for web parts, site and list definitions, and other components. It also includes a solution generator and tools for converting existing SharePoint components such as lists and columns into the CAML needed to create new ones.
- ❑ **WSP Builder** (www.codeplex.com/wspbuilder) — This is a very popular tool for creating SharePoint solution packages that many consider a “must have” for SharePoint development. It integrates directly into the Visual Studio IDE and works by reading files from a designated folder to build a solution manifest, which is ultimately compiled into a WSP file ready to be deployed into the SharePoint farm.
- ❑ **SharePoint Server 2007 SDK** (www.microsoft.com/downloads/details.aspx?FamilyId=6D94E307-67D9-41AC-B2D6-0074D6286FA9) — This is the official SharePoint Software Development Kit from Microsoft. It includes documentation as well as essential resources such as conceptual overviews, “How Do I?” programming tasks, developer tools, code samples, references, and the Enterprise Content Management (ECM) starter kit.
- ❑ **Windows SharePoint Services 3.0 SDK** (www.microsoft.com/downloads/details.aspx?familyid=05E0DD12-8394-402B-8936-A07FE8AFAFFD) — The Windows SharePoint Services 3.0 software development kit (SDK) contains conceptual overviews, programming tasks, samples, and references to guide you in developing solutions based on Microsoft Windows SharePoint Services 3.0.
- ❑ **STSDEV** (www.codeplex.com/stsdev) — This tool was created by Ted Pattison to help with the initial creation of SharePoint components like features and web parts. It does this through a set of Visual Studio project and item templates as well as custom MSBuild targets that automatically deploy the SharePoint components when the project is built.
- ❑ **SharePoint Features Project** (www.codeplex.com/features) — This site is maintained by Scot Hillier and contains an ever-growing collection of very useful features for both developers and administrators. To date, there are close to 50,000 downloads of the solution packages, which are well organized. Source code for all of the features is included.
- ❑ **Application Pool Recycler** (www.harbar.net/articles/APM.aspx) — This is a free System Tray utility developed by Spencer Harbar for providing quick access to common IIS tasks that are useful on a SharePoint development machine, such as recycling selected application pools. It also includes code for *warming up* the URLs you are working on whenever a recycle or ISS Reset occurs, giving you much better performance (and therefore productivity) during development.
- ❑ **SPTraceView** (<http://hristopavlov.wordpress.com/2008/06/19/sptraceview-lightweight-tool-for-monitoring-the-sharepoint-diagnostic-logging-in-real-time/>) — SPTraceView monitors in real time all SharePoint diagnostic tracing (also called *ULS tracing*) and can notify you using balloon-style messages in the tray bar when any information of particular interest to you is sent (traced) by any of the MOSS services and components. Because SPTraceView processes the tracing in real time, you can identify errors and events as they happen. That is as soon as you interact with the SharePoint GUI when testing/debugging your custom SharePoint solutions including web parts, event receivers, workflows, and all other SharePoint technology components.

CAML.NET IntelliSense

Working with raw XML can be difficult for .NET developers because the lack of a compiler means that you can make trivial mistakes that waste a lot of time. One way to increase productivity is to add the SharePoint schemas to Visual Studio so you get IntelliSense when editing CAML files. This is a good first step, but we can take it even further by copying the core schema files and then extending them so they provide more information. Specifically, we can enhance them by adding the detailed guidance for each CAML element from the SharePoint SDK.

Using Annotated Schemas

The code download for this book includes the CAML.NET IntelliSense installer package. This package installs an enhanced annotated version of the WSS.XSD file into the Visual Studio IDE.

You can download the CAML.NET IntelliSense installer directly from the MSDN Code Gallery at <http://code.msdn.microsoft.com/camlintellisense>.

The main idea behind the CAML.NET IntelliSense project is to extend the core schemas in two ways:

1. Gather as much information as possible about each element and attribute and place it into `xs:annotation` elements so that it pops up in context while editing.
2. Identify and replace as many `xs:string` types as possible with enumerated types so the valid values for each attribute also pop up in context while editing.

For example, using the core WSS schemas directly, we get a dropdown like that shown in Figure 2-1 while editing a `Field` element.



Figure 2-1: Using the core WSS schemas.

That is better than nothing, but it does not say what the attribute is for, so unless you're already familiar with it, you have to refer back to the SDK either online or in the compiled help (CHM) file. On the other hand, using the enhanced annotated schema, we can provide more information in context. We only have to refer to the SDK documentation when it's really needed. Figure 2-2 shows the result of using the enhanced version.

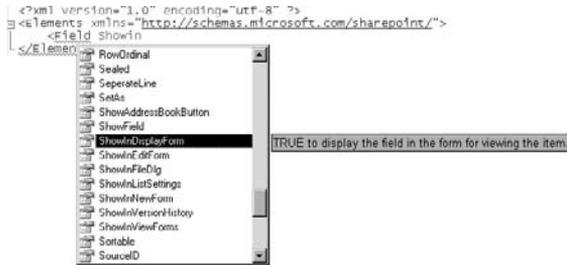


Figure 2-2: Using the enhanced schema.

In addition to annotating the schemas with the corresponding SDK documentation for each element, we can go one step further and change the plain vanilla `xs:string` types to custom enumerated types so we get the list of valid choices for the attribute values. This comes in very handy and also helps to eliminate those nasty little typos that can be really hard to find.

There are lots of places in the SDK documentation where the expected attribute values are listed in the description, but are not enumerated in the schema. This again means that we have to go digging through the documentation to discover what SharePoint expects the attribute values to be. This can be an enormous waste of time — even if you have a pretty good idea of what the values are. Figure 2-3 shows the effect when each enumeration is also annotated so that you get contextual help for the available values.

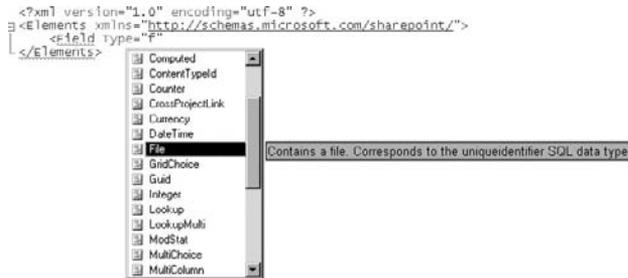


Figure 2-3: Annotated enumerations.

ECM2007: A Foundation Class Library for Records Management

I'm a strong believer in building reusable tools as much as possible. Developers who are new to SharePoint sometimes fall into the habit of thinking that, since they are working in XML as well as their .NET language of choice (C# or VB.NET), reuse is no longer an integral part of the development life cycle. However, by using disciplined component-building, we can overcome many of the perceived shortcomings of the typical SharePoint Records Management development life cycle. We'd like as much as possible to identify components and capture them in a set of reusable tools so that we don't waste time re-building the same functionality over and over again. When it comes to SharePoint Records Management development, sometimes that is more easily said than done. One recurring problem is

Chapter 2: Preparing for Records Management Development

figuring out how to integrate declarative elements written in XML with imperative elements written in the .NET language in a way that facilitates reuse without introducing too much additional complexity.

Throughout this book, I have tried to strike a balance between making the code samples understandable while at the same time showing how to create a set of *foundation classes* that can be extended easily to support your own custom solution requirements now and in the future. Therefore, we will build a component library called *ECM2007* that defines some base classes and interfaces that make it easier to build SharePoint solutions. The library includes the following namespaces, interfaces, and components:

Namespace	Class/Interface	Description
ECM2007	<code>ISharePointObject</code>	Provides an abstraction for any SharePoint object.
	<code>SharePointObject</code>	Base class for other SharePoint components. Inherits from <code>System.Configuration.Install.Installer</code> to enable quick deployment.
	<code>SharePointList</code>	Static wrapper class for <code>SPList</code> that provides utility methods for accessing list data
	<code>ItemEventReceiver</code>	Base class for declaring item event receivers. Provides a default implementation of <code>SPItemEventReceiver</code> that uses reflection to determine which event receiver methods have been implemented by the derived class.
ECM2007. ContentTypes	<code>SharePointContentType</code>	Custom attribute class that allows any .NET class to act as a template for a content type. Supports automatic wiring of event receivers, loading of document templates and XML documents from embedded resources, and data binding for fields.
	<code>FieldRefAttribute</code>	Custom attribute class used to mark up public properties that are mapped to columns on a SharePoint list or content type
ECM2007. InformationPolicy	<code>ISharePointPolicy</code> , <code>ISharePointPolicyFeature</code> , <code>ISharePointPolicyResource</code>	Abstractions for information management policy components
	<code>SharePointPolicy</code>	Base class for declaring information policies

Namespace	Class/Interface	Description
	SharePointPolicyFeature	Base class for declaring information policy features. This class provides a default implementation of the <code>IPolicyFeature</code> interface, allowing derived classes to implement only the methods they require.
	SharePointPolicyResource	Base class for implementing policy resources
ECM2007. RecordsManagement	SharePointRouter	Base class for implementing custom routers. This class provides a default implementation of the <code>IRouter</code> interface, allowing derived classes to override selected methods.
ECM2007. Versioning	SPListEx, SPListItemCollectionEx	Static classes that declare extension methods for list items and list item collections related to versioning

The code for most of these components is shown and explained in the appropriate sections of this book. For those that are not explained in detail, you can refer directly to the comments included in the downloadable solution files that accompany this book.

Creating a SharePoint Feature Project Template

Often when building SharePoint solutions, it is useful to have tools that generate some of the code for you. This is especially true when you are writing raw CAML code or you need to structure your project a certain way. During the process of developing records management solutions, you will typically work in a virtual machine that is running both MOSS and the Visual Studio IDE. Anything you can do to shorten the edit/compile/debug cycle will directly affect your day-to-day productivity.

To create components that will be installed into SharePoint using a feature, you have to copy files and other elements into a special folder that is commonly known as the *12 hive*. This folder is where the SharePoint product is installed, and it is located at `%programfiles%/common files/microsoft shared/web server extensions/12` on the server machine. The easiest way to ensure that the files in your project folder are copied into the correct locations within the 12 hive is to mimic the same folder structure within your Visual Studio project. This means that every time you create a new SharePoint feature project, you will have to create the appropriate folder structure within Visual Studio.

Creating the correct project folder structure for your feature is only part of the story, however. You must also create a special set of files that includes the feature manifest, one or more element manifests, a feature receiver class, an optional image file to use as the feature logo, and so on. If a feature receiver is included in your solution, then the feature manifest must also specify the correct assembly name and public key token, and the assembly must be registered in the Global Assembly Cache (GAC) after it is built.

Using a Visual Studio project template can automate the entire process of setting up the project, generating the required files, and entering the required text into those files — reducing it all to a single step. I

Chapter 2: Preparing for Records Management Development

like to take it one step further and create a wizard to capture the data needed to generate all of the files. That way, I don't have to worry about any of the details in my daily work. Figure 2-4 shows the dialog presented by the Feature Wizard to gather information about the feature project, and the resulting feature project is shown in Figure 2-5.

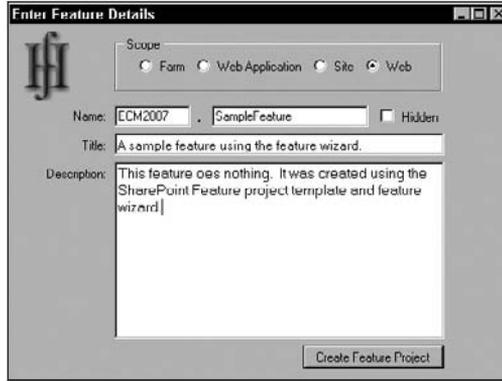


Figure 2-4: SharePoint Feature Wizard input form.

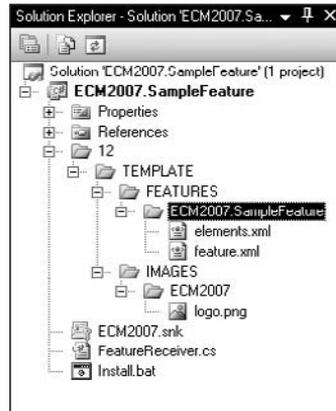


Figure 2-5: Generated SharePoint feature project.

To create a Visual Studio project template, you need two things: a vstemplate file that describes the different elements that will be included in the template, and an example of the project you intend to create. The vstemplate file, along with the files of the sample project, are added to a zip file and copied into the Visual Studio installation folder.

Listing 2-1 shows the code for a SimpleFeature.vstemplate file that sets up a SharePoint feature project. This template is based on a sample project that is included in the code download for the book. The sample project files referenced from this template are located in the VisualStudio/SimpleFeature folder of the ECM2007 project.

Listing 2-1: SimpleFeature.vstemplate

```
<VSTemplate Version="2.0.0" xmlns="http://schemas.microsoft.com/developer/
  vstemplate/2005" Type="Project">
  <TemplateData>
    <Name>SharePoint Feature</Name>
    <Description>A basic SharePoint feature project</Description>
    <ProjectType>CSharp</ProjectType>
    <ProjectSubType>SharePoint</ProjectSubType>
    <SortOrder>1000</SortOrder>
    <CreateNewFolder>true</CreateNewFolder>
    <DefaultName>SharePoint Feature</DefaultName>
    <ProvideDefaultName>true</ProvideDefaultName>
    <LocationField>Enabled</LocationField>
    <EnableLocationBrowseButton>true</EnableLocationBrowseButton>
    <Icon>__TemplateIcon.ico</Icon>
  </TemplateData>
  <TemplateContent>
    <Project TargetFileName="$safeprojectname$.csproj"
      File="SimpleFeature.csproj" ReplaceParameters="true">
      <Folder Name="12" TargetFolderName="12">
        <Folder Name="TEMPLATE" TargetFolderName="TEMPLATE">
          <Folder Name="FEATURES" TargetFolderName="FEATURES">
            <Folder Name="SimpleFeature" TargetFolderName="SimpleFeature">
              <ProjectItem ReplaceParameters="true" TargetFileName=
                "elements.xml">elements.xml</ProjectItem>
              <ProjectItem ReplaceParameters="true" TargetFileName=
                "feature.xml">feature.xml</ProjectItem>
            </Folder>
          </Folder>
          <Folder Name="IMAGES" TargetFolderName="IMAGES">
            <Folder Name="ECM2007" TargetFolderName="ECM2007">
              <ProjectItem ReplaceParameters="false" TargetFileName=
                "logo.png">logo.png</ProjectItem>
            </Folder>
          </Folder>
        </Folder>
      </Folder>
      <ProjectItem ReplaceParameters="false" TargetFileName=
        "ECM2007.snk">ECM2007.snk</ProjectItem>
      <ProjectItem ReplaceParameters="true" TargetFileName=
        "FeatureReceiver.cs">FeatureReceiver.cs</ProjectItem>
      <ProjectItem ReplaceParameters="true" TargetFileName=
        "Install.bat">Install.bat</ProjectItem>
      <Folder Name="Properties" TargetFolderName="Properties">
        <ProjectItem ReplaceParameters="true" TargetFileName=
          "AssemblyInfo.cs">AssemblyInfo.cs</ProjectItem>
      </Folder>
    </Project>
  </TemplateContent>
  <WizardExtension>
    <Assembly>ECM2007, Version=1.0.0.0, Culture=neutral, PublicKeyToken=
      eb8a6a1622425a15</Assembly>
    <FullClassName>ECM2007.FeatureWizard</FullClassName>
  </WizardExtension>
</VSTemplate>
```

Chapter 2: Preparing for Records Management Development

The `WizardExtension` element at the bottom of the template associates it with the custom Feature Wizard described above. The `ECM2007.FeatureWizard` class implementation is shown in Listing 2-2. It implements the `IWizard` interface, which Visual Studio uses to communicate with the Wizard at run time.

Listing 2-2: ECM2007.FeatureWizard implementation

```
using System;
using System.Collections.Generic;
using System.Windows.Forms;
using EnvDTE;
using Microsoft.SharePoint;
using Microsoft.VisualStudio.TemplateWizard;

namespace ECM2007
{
    public class FeatureWizard : IWizard
    {
        const string RootFolderName = "12";
        const string DefaultFeatureFolderName = "SimpleFeature";

        private string feature_name = string.Empty;
        private string feature_prefix = "Custom";
        private string feature_folder = string.Empty;
        private string feature_title = "SharePoint Feature";
        private string feature_description = string.Empty;
        private bool feature_hidden = false;
        private SPFeatureScope feature_scope = SPFeatureScope.Web;
        private FeatureDetailsForm m_inputForm;

        /// <summary>
        /// Recursively searches for an item having the default feature folder name
        /// and changes it to the value in feature_name.
        /// </summary>
        bool AdjustFeatureFolderName(ProjectItem item)
        {
            if (item.Name.Equals(DefaultFeatureFolderName))
            {
                item.Name = feature_folder;
                return true;
            }
            else
            {
                foreach (ProjectItem subItem in item.ProjectItems)
                {
                    if (AdjustFeatureFolderName(subItem))
                        return true;
                }
            }
            return false;
        }

        #region IWizard Members

        /// <summary>
        /// Called before opening any item that has the OpenInEditor attribute.
        /// </summary>
```

Chapter 2: Preparing for Records Management Development

```
/// <param name="projectItem"></param>
void IWizard.BeforeOpeningFile(ProjectItem projectItem)
{
}

/// <summary>
/// Called after a new project is generated.
/// </summary>
/// <param name="project"></param>
void IWizard.ProjectFinishedGenerating(Project project)
{
    // Search for the "12" folder.
    foreach (ProjectItem item in project.ProjectItems)
    {
        switch (item.Kind)
        {
            case EnvDTE.Constants.vsProjectItemKindPhysicalFolder:
                if (item.Name.Equals(RootFolderName))
                    AdjustFeatureFolderName(item);
                break;
        }
    }
    // Adjust the project name
    project.Name = feature_folder;
}

/// <summary>
/// Called for item templates, not for project templates.
/// </summary>
/// <param name="projectItem"></param>
void IWizard.ProjectItemFinishedGenerating(ProjectItem projectItem)
{
}

/// <summary>
/// Called after the project is created.
/// </summary>
void IWizard.RunFinished()
{
}

/// <summary>
/// Called when the wizard is started.
/// </summary>
/// <param name="automationObject">The root DTE object -
    used for customization.</param>
/// <param name="replacementsDictionary">A collection of all
    pre-defined parameters in the template.</param>
/// <param name="runKind">A WizardRunKind parameter that contains
    information about what kind of template is being
    used.</param>
/// <param name="customParams">An object array that contains a set of
    parameters passed to the wizard by Visual Studio.</param>
void IWizard.RunStarted(object automationObject, Dictionary<string,
```

Continued

Listing 2-2: ECM2007.FeatureWizard implementation (continued)

```
        string> replacementsDictionary, WizardRunKind runKind,
        object[] customParams)
    {
        try
        {
            // Retrieve some parameters for use later.
            feature_name = replacementsDictionary["$safeprojectname$"];
            feature_title = replacementsDictionary["$projectname$"];

            // Display a form to the user to collect data needed to
            // configure the project.
            m_inputForm = new FeatureDetailsForm();
            m_inputForm.Hidden = feature_hidden;
            m_inputForm.FeatureName = feature_name;
            m_inputForm.FeatureTitle = feature_title;
            m_inputForm.FeaturePrefix = feature_prefix;
            m_inputForm.FeatureDescription = feature_description;
            m_inputForm.FeatureScope = feature_scope;
            m_inputForm.ShowDialog();

            // Retrieve the edited parameters from the input form.
            feature_name = m_inputForm.FeatureName.Replace(" ", "");
            feature_title = m_inputForm.FeatureTitle;
            feature_prefix = m_inputForm.FeaturePrefix.Replace(" ", "_");
            feature_folder = string.IsNullOrEmpty(feature_prefix) ?
                feature_name : feature_prefix + "." + feature_name;
            feature_description = m_inputForm.FeatureDescription;
            feature_scope = m_inputForm.FeatureScope;
            feature_hidden = m_inputForm.Hidden;

            // Add custom parameters that can be referenced from the template.
            replacementsDictionary.Add("$feature_name$", feature_name);
            replacementsDictionary.Add("$feature_title$", feature_title);
            replacementsDictionary.Add("$feature_prefix$", feature_prefix);
            replacementsDictionary.Add("$feature_folder$", feature_folder);
            replacementsDictionary.Add("$feature_description$",
                feature_description);
            replacementsDictionary.Add("$feature_scope$",
                feature_scope.ToString());
            replacementsDictionary.Add("$feature_hidden$",
                feature_hidden ? "TRUE" : "FALSE");

        }
        catch (Exception x)
        {
            MessageBox.Show(x.ToString());
        }
    }

    /// <summary>
    /// Only called for item templates.
```

```
    /// </summary>
    /// <param name="filePath"></param>
    /// <returns></returns>
    bool IWizard.ShouldAddProjectItem(string filePath)
    {
        return true;
    }

#endregion
}
}
```

The Wizard is responsible for presenting the user interface, retrieving the input data, and then placing it into replacement variables that are substituted for text tokens in the files that are included in the project. For example, the following text is used to generate the feature manifest by using replacement variables for the feature ID, title, description, scope, and so on:

```
<?xml version="1.0" encoding="utf-8" ?>
<Feature xmlns="http://schemas.microsoft.com/sharepoint/"
    Id="$guid2$"
    Title="$feature_title$"
    Description="$feature_description$"
    ImageUrl="ECM2007\logo.png"
    Version="1.0.0.0"
    Scope="$feature_scope$"
    Hidden="$feature_hidden$"
    ReceiverAssembly="$safeprojectname$, Version=1.0.0.0, Culture=neutral,
        PublicKeyToken=eb8a6a1622425a15"
    ReceiverClass="$safeprojectname$.FeatureReceiver"
    >
    <ElementManifests>
        <ElementManifest Location="elements.xml"/>
    </ElementManifests>
</Feature>
```

If you decide to modify the included feature template, then all you have to do to install it on your system is run the `InstallProjectTemplate.bat` file that is located in the `ECM2007\VisualStudio` folder. This batch file takes care of copying the `SimpleFeature.zip` file into the Visual Studio installation folder. Then you need to restart Visual Studio in order to start using the template. Listing 2-3 shows the batch file code.

Listing 2-3: InstallProjectTemplate.bat

```
@SET VSTEMPLATE=SimpleFeature
@SET PROJECTTEMPLATEPATH="C:\Program Files\Microsoft Visual Studio
    9.0\Common7\IDE\ProjectTemplates\CSharp\SharePoint\"
@MD %PROJECTTEMPLATEPATH%
@COPY /Y %VSTEMPLATE%\%VSTEMPLATE%.zip %PROJECTTEMPLATEPATH%
@ECHO Installing VS Templates...
DevEnv /installvstemplatess
@SET VSTEMPLATE=
@SET PROJECTTEMPLATEPATH=
```

Chapter 2: Preparing for Records Management Development

If you decide to modify the provided template, then after building the ECM2007 project (which includes the Feature Wizard) and having registered it in the Global Assembly Cache, you need to re-create the SimpleFeature.zip file that contains all of the required elements before executing the InstallProjectTemplate batch file. The only tricky part is creating the zip file. To do that, follow these steps:

1. Navigate to the ECM2007\VisualStudio\SimpleFeature folder from Windows Explorer (*not* from within the ECM2007 Visual Studio project).
2. Delete the SimpleFeature.zip file.
3. Select all of the files in the folder and right-click on the selected files.
4. Choose the Send To->Compressed (zipped) folder command.
5. Rename the new zip file as **SimpleFeature.zip**.

To learn more about creating Visual Studio project templates, visit the Visual Studio 2008 Developer Center at <http://msdn.microsoft.com/en-us/library/6db0hwky.aspx>.

Leveraging XML Schemas for Maximum Flexibility

Solution development on the SharePoint platform can be simplified greatly by taking advantage of XML schemas. Part of the problem when working with SharePoint is that there are so many *touch points* — so many moving parts in any given solution. On the one hand, you have CAML elements that must be specified precisely according to the schema that defines each subsystem. Then you have code components that must coordinate with those CAML declarations. Even within those code components, there are often many pieces that fit together to provide the whole solution. A good example of this might be a list that you create in a feature, and that list works in conjunction with event receivers attached to the list or to a content type associated with the list. You may want to connect everything programmatically, or you may want to divide the task among code and declarative elements. Another example might be a list and content types that are created declaratively and then configured programmatically within the feature receiver. Since the feature receiver is always called after the provisioning of declarative components, you might then use code to examine the content database to ensure that everything needed for the solution is in place and configured properly. There may be other dependent features that must be examined, perhaps to make sure that their lists have not been modified, and the like.

XML schemas can be used as the *glue* to tie all of these elements together. By taking a step back and writing a high-level description of all of the solution components, you can generalize the solution so that it is driven in part by the content of an XML file. This is how many of the built-in components of MOSS are implemented. For instance, the site provisioning engine is used in this way to configure publishing sites based on an XML file that describes the target site topology. In fact, the SharePoint feature provisioning and solution deployment subsystems are all driven using XML files defined by XML schemas.

Later in the book, we'll construct a dynamic file plan based on this approach. We'll use an XML schema to describe the essential elements of the file plan. This allows our file plan definition to grow over time to accommodate all of the aspects needed to support the various code components that make up whatever solution we wish to build. The power of XML schemas comes from the ability to unambiguously describe a set of components and operations on those components using XML tags and attributes. The trick is to leverage the expressive power of XML by integrating it into our standard set of development tools. Fortunately, Visual Studio is already set up to support this development methodology. The Visual Studio

Chapter 2: Preparing for Records Management Development

IDE includes an XML Schema Editor. You can start by simply creating a new XSD file and then begin developing a schema in text mode. Listing 2-4 shows a sample schema that describes an `Employee` object.

Listing 2-4: Employee schema

```
<xs:schema targetNamespace="http://tempuri.org/XMLSchema.xsd"
  elementFormDefault="qualified"
  xmlns="http://tempuri.org/XMLSchema.xsd"
  xmlns:mstns="http://tempuri.org/XMLSchema.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
>
  <xs:element name="Employee">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="FullName" type="xs:string"/>
        <xs:element name="Department" type="xs:string"/>
        <xs:element name="Manager" type="xs:string"/>
        <xs:element name="Email" type="xs:string"/>
        <xs:element name="Phone" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

After the schema has been defined, you can create a sample data file that matches the schema. Again, the Visual Studio IDE helps greatly by binding the schema to the data file to automatically validate its contents during editing. IntelliSense prompts help to ensure that the data has been entered correctly according to the tags and attributes defined in the schema.

Once you have the schema and a sample data file, you have two options for processing the data. The first approach is to read the data using an `XMLDocument` or `XPath` expression and then perform whatever operations they describe. This approach relies on the data being properly formatted with valid values, and so on. A better approach is to take advantage of the strong typing afforded by the .NET compiler (either C# or VB.NET) by generating wrapper classes for the objects specified in the schema and then *deserializing* the XML data into runtime instances of those classes.

This approach offers several benefits. First, the data is automatically validated during the deserialization process, which avoids errors that might show up at a later stage in the processing of the data. Second, changes to the schema are automatically accommodated simply by regenerating the wrapper classes. This effectively turns the schema into a *living specification* that can ultimately translate into dramatic reductions in the amount of time and effort needed to construct the solution. Finally, the generated classes can be extended easily using a code-beside model that is supported by most code generation tools.

The *code-beside model* means that the generated code can be regenerated later without affecting any handwritten code that may be associated with the same class. The tool I use to generate wrapper classes is called `XSD.EXE` and is included with the Visual Studio product. This command-line tool accepts the path to the schema file (with extension `.xsd`) and additional parameters that specify the language (C# or VB.NET) to use when generating the wrappers and whether to generate classes or data sets. Listing 2-5 shows the generated wrapper classes for the `Employee` entity we declared previously.

Listing 2-5: Generated Employee wrapper class

```
namespace ECM2007 {
    using System.Xml.Serialization;

    [System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
    [System.SerializableAttribute()]
    [System.Diagnostics.DebuggerStepThroughAttribute()]
    [System.ComponentModel.DesignerCategoryAttribute("code")]
    [System.Xml.Serialization.XmlTypeAttribute(AnonymousType=true,
        Namespace="http://tempuri.org/XMLSchema.xsd")]
    [System.Xml.Serialization.XmlRootAttribute(
        Namespace="http://tempuri.org/XMLSchema.xsd", IsNullable=false)]
    public partial class Employee {

        private string fullNameField;
        private string departmentField;
        private string managerField;
        private string emailField;
        private string phoneField;

        /// <remarks/>
        public string FullName {
            get {
                return this.fullNameField;
            }
            set {
                this.fullNameField = value;
            }
        }

        /// <remarks/>
        public string Department {
            get {
                return this.departmentField;
            }
            set {
                this.deparmentField = value;
            }
        }

        /// <remarks/>
        public string Manager {
            get {
                return this.managerField;
            }
            set {
                this.managerField = value;
            }
        }

        /// <remarks/>
    }
}
```

```
public string Email {
    get {
        return this.emailField;
    }
    set {
        this.emailField = value;
    }
}

/// <remarks/>
public string Phone {
    get {
        return this.phoneField;
    }
    set {
        this.phoneField = value;
    }
}
}
}
```

The fact that the wrapper class is generated with the `partial` keyword allows us to create additional methods that are defined in a separate file. Thus, if we modify the schema and regenerate the core methods, the additional methods remain intact. This means we can define entire subsystems of functionality easily to extend the component in different ways. As an example, we can extend the `Employee` object to support the automatic construction of an `Employee` instance from data stored in a SharePoint list item by defining a static `CreateFromListItem` method in a separate file as shown in Listing 2-6.

Listing 2-6: Extended `Employee` wrapper class

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Text;
using Microsoft.SharePoint;

namespace ECM2007
{
    public partial class Employee
    {
        /// <summary>
        /// This method constructs an Employee instance from
        /// data stored in a SharePoint list item.
        /// </summary>
        /// <param name="item">the item containing the employee data</param>
        /// <returns>an initialized Employee instance</returns>
        public static Employee CreateFromListItem(SPLListItem item)
        {
            try
            {
```

Continued

Listing 2-6: Extended Employee wrapper class (continued)

```
        return new Employee()
        {
            FullName = item.Title,
            Department = item["Department"].ToString(),
            Email = item["Email"].ToString(),
            Manager = item["Manager"].ToString(),
            Phone = item["Phone"].ToString()
        };
    }
    catch (SPException spex)
    {
        Helpers.HandleException(typeof(Employee), spex);
    }
    return null;
}
}
```

Generating Schema Classes from within Visual Studio

Although XSD.EXE is a powerful tool, running it from the Windows command line or even from a pre-build event can have a negative impact on productivity because of the extra steps involved. We would rather have it integrated directly into the Visual Studio IDE so that whenever a change is made to a schema, the wrapper classes are regenerated automatically.

Visual Studio supports the creation of *custom tools*, which are code modules that are registered on the development machine so that Visual Studio can find them. The purpose of a custom tool is to support code generation, where a source file is post-processed after editing to generate the actual source, which is then compiled into the assembly. This is the same approach used by the built-in editors, such as the Forms Designer, which generate a secondary file containing the code needed to initialize the form.

The downloadable solution files that accompany this book include a project called *XsdClassGenerator*. This project is set up to automatically register itself on the development machine, so all you have to do is open the project in Visual Studio and build it. Once you have done that, then close and reopen the Visual Studio IDE so that the registration entries are reloaded.

The XsdClassGenerator project is based on code that was originally developed by Chris Sells, a long-time builder of useful .NET developer tools. The source code is included in this book with his permission. For more information about Chris Sells and other tools he has created, see www.sellsbrothers.com.

To use the tool, open the Properties window in the Visual Studio IDE. Next open the Solution Explorer window and select the XML schema file (with an .xsd extension) you want to generate classes for. Enter **XsdClassGenerator** as the value for the “Custom Tool” entry in the Properties window as shown in Figure 2-6.

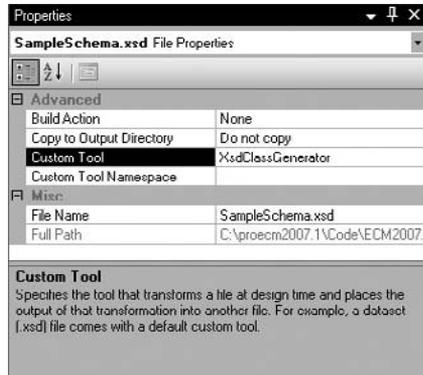


Figure 2-6: Custom Tool property value for an XSD file.

If you wish to reference the generated classes from within a specific namespace, then enter the namespace into the “Custom Tool Namespace” field. Now, whenever the file is saved to disk, a secondary file is created beneath the original file node with the appropriate extension, based on the programming language defined for the current project. Figure 2-7 shows the generated file for a C# project.

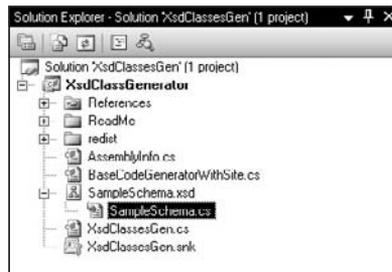


Figure 2-7: Generated C# class file.

Summary

This chapter introduced the essential steps involved in preparing for Records Management solution development on the SharePoint platform. Since regulatory compliance is the primary driver for any RMS development strategy, we first examined some of the key U.S. legislative acts that have been enacted in recent years to gain a better understanding of the primary challenges facing the users of most records management systems.

The chapter then focused on ways to maximize developer productivity by enhancing the Visual Studio development environment with freely downloadable tools and developer resources. These include tools for creating web parts, site and list definitions, solution packages, and other core components of any SharePoint solution. We also looked at the CAML.NET IntelliSense project, which greatly enhances the built-in help provided by the Visual Studio XML Editor to make it more context-sensitive.

Chapter 2: Preparing for Records Management Development

We introduced the ECM2007 foundation class library, which is included in the code download for the book, and showed how to create a set of base classes and interfaces that simplify records management solution development. The chapter showed how to further customize a SharePoint development environment to achieve maximum productivity by creating a custom SharePoint Feature project template that uses a built-in Feature Wizard to gather the information needed to set up a properly constructed SharePoint feature project in Visual Studio.

Finally, the chapter showed how to leverage the power of XML schemas to gain the maximum flexibility when developing components that must work together to deliver an integrated solution. This technique is particularly useful for records management development, wherein many components must be coordinated to provide a complete set of interrelated functionality. The chapter showed how to generate C# or VB.NET wrapper classes from XML schema definitions and introduced the XsdClassGenerator custom tool that enables automatic class regeneration from within the Visual Studio IDE.

3

SharePoint Tools for Managing Records

This chapter introduces the key components for building records management solutions on the SharePoint platform. We'll look at the tools and components that are built into Windows SharePoint Services for building collaborative applications, and we'll also look at the extended functionality provided by MOSS.

It is important to remember that MOSS is essentially a Windows SharePoint Services application. There are additional assemblies that are deployed with MOSS to provide the code behind the MOSS-specific features, but you can really think of MOSS as just a collection of SharePoint features. Thus, when we consider the tools and components needed to build records management solutions, we are really looking at both the WSS fundamentals as well as the extended functionality provided by MOSS.

The core WSS components we examine will include document libraries, which are a specialization of the SharePoint list that provides the ability to store documents and their metadata. It will include content types, which are essential for ECM development and make it much easier to build records management solutions because they provide much of the core functionality needed to associate different kinds of records with the unique processing required to manage them. In this chapter, we'll explore different approaches to creating content types.

Another area we'll explore is the versioning mechanism that is built into WSS. Although versioning is not seen as a primary component for records management, it is a critical part of the content management infrastructure provided by MOSS. Therefore versioning is an important component of any solution strategy that involves collaborative elements. As an example, consider the typical web publishing portal in which minor versions are visible by content authors, while end-users see only major versions. It would be easy to extend this concept into a records management scenario whereby published versions of a document are sent automatically to the Records Center whenever a new major version is created within the publishing portal. Similar functionality might be tied to more complex version metrics, perhaps driven by a workflow. In such scenarios, it is the versioning API and related functionality that can either hamper or facilitate the solution design.

Finally, it is important to understand the mechanisms that are provided by SharePoint to control access to content, so we'll also examine content security. Whatever solution strategy you may wish to pursue, your ability to assign permissions to individuals and groups throughout the content life cycle is determined by the flexibility of the underlying platform for managing permissions. You'll see what the available permissions are, how they are attached to documents and list items, and what tools are available for assigning and managing permissions throughout the content life cycle.

Document Libraries

If you have worked with SharePoint at all, then you already have a general understanding of document libraries. At a very high level you can think of a document library as a special kind of list. In the SharePoint object model, document libraries are, in fact, derived from the `SPLIST`, but unlike standard lists, document libraries are always associated with a document template. It doesn't have to be a Word template — it can be a form template if you're using an InfoPath form library, or it can be a custom document template that has been assigned to the item.

Another aspect of document libraries is that they allow you to manage the transfer of metadata between the library itself, which displays metadata in columns, and each document instance, which may contain additional properties that are not displayed in columns. These additional properties may be in any format. We cannot assume we're always dealing with Office documents, so we have to consider XML documents, PDF files, and other kinds of documents. Therefore, we need a way to map arbitrary document properties between each document and the document library.

Just like lists, document libraries expose a simple folder structure. This is a basic requirement because when you're using the object model to manipulate document libraries, you ultimately need to access the actual file that is stored within the content database. SharePoint deals with this requirement by exposing an object model that is very similar to what most developers are already used to when dealing with the physical file system. It's not actually *structured storage* per se, but we do have the same concepts of folders and files. Folders can contain other folders, and folders may contain files.

Whereas SharePoint 2.0 had separate implementations for lists and document libraries, the SharePoint 3.0 document library inherits most of its functionality from the `SPLIST` object. Thus, there is greater parity between the two concepts. For example, the `SPFolder/SPFile` structure is built into the standard SharePoint list architecture and is available for all lists, not just document libraries.

Since documents are stored as standard list items, it is easy to write custom list management code that manipulates documents in the same ways as any other list item. It is also easy to access and manipulate document metadata. This allows us to build and use a standard set of tools, site columns, field types for those columns, and so on, and it also makes it easy to add custom behavior to items using SharePoint workflows, event receivers, and menu commands without having to code specifically for documents.

Creating Document Libraries

Creating a document library is similar to creating a list, except that with document libraries there is the additional concept of the document template. Lists don't have document templates. Therefore, when creating a document library, one consideration might be the determination of which document template to associate with the library. This will, in turn, determine the format of documents that are based on the template.

This determination also applies to content types that inherit from the document content type, which is the default content type associated with a document library. It is the document content type that adds the document template field to the list item.

To create a document library programmatically, you just create a new list that is based on the document library template type using code like the following:

```
/// <summary>
/// Retrieve or create a document library instance.
/// </summary>
public static SPDocumentLibrary Create(SPWeb web, string title, string
    description)
{
    Guid listid = Guid.Empty;
    SPDocumentLibrary docLib = null;
    try
    {
        docLib = web.Lists[title] as SPDocumentLibrary;
    }
    catch
    {
        /* ignore exception - library does not exist */
    }
    if (docLib == null)
    {
        listid = web.Lists.Add(title, description,
            SPListTemplateType.
                DocumentLibrary);
        docLib = web.Lists[listid] as SPDocumentLibrary;
    }
    return docLib;
}
```

This results in a new object in the content database that includes the necessary support for managing documents in the list. The corresponding class in the SharePoint object model is `SPDocumentLibrary`, which inherits directly from `SPList`. This class extends the `SPList` class by adding methods for getting and setting the document template URL, and for getting the list of checked-out files.

Document Libraries and Property Promotion

One of the more interesting parts of a document library is that it provides an additional layer of support for synchronizing document metadata between the document library and individual documents. This support is provided by a built-in mechanism called *property promotion and demotion*. *Property promotion* refers to the process of copying document properties from a document into the columns of a document library. *Property demotion* refers to the converse process whereby column values are copied from columns in the document library back into the document.

Properties are automatically promoted whenever a document is saved into the document library. So, for example, when you select the *New* command from a document library, a new document opens on the client in whatever application is associated with the document template attached to the library. When the document is saved, any embedded document property values that were changed during

the editing session are copied into the matching columns of the document library. Similarly, whenever you upload a document into a document library, any matching properties are extracted from the document and placed into the appropriate columns of the document library. This works both for the *standard* document properties like `Title`, `Subject`, `Author`, `Category`, and `Keywords`, as well as any *custom* document properties that may have been added to the document.

Properties are demoted whenever you change a property in the document library. For example, if you modify a document's properties via the Edit Properties form in the SharePoint user interface and then open the document for editing or viewing on the client, the new property values appear in the document on the client. When the Edit Properties form is saved, the document parser is loaded and invoked to copy the new property values back into the document.

This functionality is provided by a special component called a *Document Parser*. A *document parser* is a custom COM component that SharePoint calls to handle property promotion and demotion. SharePoint locates the document parser for a given document based on its file extension. This means that although the same document parser may be applied to multiple file types, only one document parser can be associated with any given file type. SharePoint provides out-of-the-box document parsers for the Office file types.

Property promotion and demotion are enabled by default for document libraries in most SharePoint sites, but they are turned off for Records Center sites. This is done by disabling the document parsing functionality at the web-site level. For more information on this topic, see the section entitled, "Property Promotion and Demotion," in Chapter 4.

Content Types

Content types are the primary means by which information is classified and controlled within the SharePoint environment. Every list item stored in the content database is based on a content type. The SharePoint terminology is actually a specialization of a more fundamental concept that is found in many document management systems and is called by different names. The basic idea is the same: A *content type* (or object type) is a named collection of metadata elements that support business processes. In this sense, a content type is very similar to the notion of a class, with properties, methods, and business rules. The business rules are either attached directly to the type using event receivers, or are written separately with reference to those properties and methods. Either way, the goal is to somehow *encapsulate* the behavior and the metadata together so they can be treated as an atomic unit. Figure 3-1 provides a basic illustration of this idea.

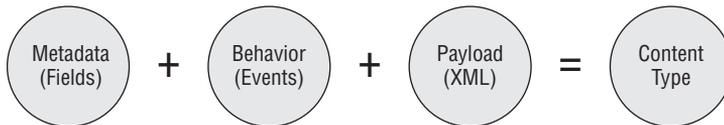


Figure 3-1: Content types in SharePoint.

The current implementation of content types in the SharePoint environment provides only a partial encapsulation of metadata and content-driven behavior. This has ramifications for many "idealistic" design strategies that fail to account for future evolution of the platform. In this book, I've taken the optimistic view that tighter encapsulation will eventually emerge because effective ECM solutions demand it.

Thus, a content type boils down to three primary components — metadata, behavior, and payload. The metadata is the actual columns added to a list whenever a content type is associated with it. I'm using the term *behavior* loosely to describe any custom code that operates on the underlying list item based on information stored within the content type. This includes event receivers, but may also include workflow activities or other custom code that determines what to do based on the content type definition. The term *payload* refers to everything else and may include arbitrary data that is bound to the content type during its lifetime. Figure 3-2 provides an abstract view of the core architecture.

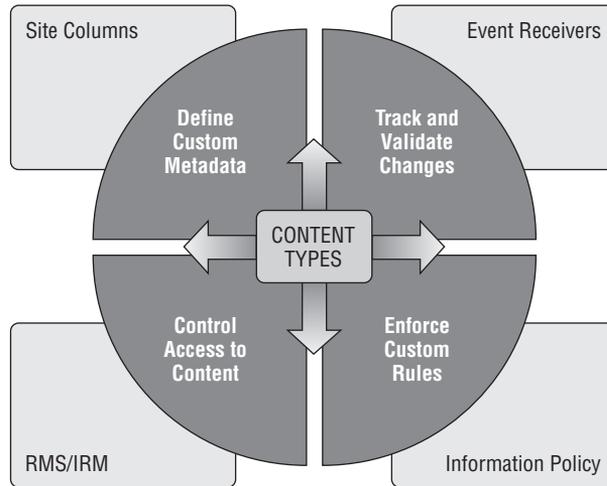


Figure 3-2: Content types and core document management requirements.

Whether columns really need to be bound directly to each list is a question for the SharePoint product team, because it can cause problems with certain types of solutions. In general it works pretty well. For the current implementation, the columns that are defined as part of the content type definition are physically copied into the list instance. It works this way because a list can contain more than one content type. The list acts as a container for the content type templates that are used to create the actual list item fields. You declare and name the columns in much the same way you would if you were adding them directly to a list, but this can cause problems when you're writing a solution that needs to update an existing content type. Since the list keeps a separate copy of the content type definition, the columns from the previous version are still there in the list when you update the master content type template. Even if you check first that the content type is not already being used, if you create it again, you will end up with duplicate columns in a list that the content type was already attached to.

Behavior or code is a key component of a content type. There is no notion of script associated with content types, although that might be useful in many scenarios. Currently the only way to attach behavior to a content type is to write some code and attach it directly to the content type as an event receiver, or call it indirectly through a workflow activity.

It is important to note that for event receivers, even though the code is bound to the content type, the code is invoked in the context of the list and not of the content type. I'm using the term context loosely to mean that event receivers are defined in terms of actions on list items and not in terms of actions on content types.

Chapter 3: SharePoint Tools for Managing Records

Payload is the last part of a content type. Why do we need a content type payload? In short, for extensibility. The content type concept is so fundamental to the way that SharePoint works, it would not be possible to anticipate all of the ways in which a content type definition might be used. So the SharePoint product team built in an extensible mechanism for associating arbitrary data with a content type definition by leveraging the way that data types are defined in XML. XML data types are defined by declaring a unique namespace for a given XML fragment. SharePoint uses the same mechanism to manage a collection of XML documents associated with a content type definition. Any number of XML documents can be added to the content type template, but only one of each *type*, where the type is defined by a unique namespace within the collection, as illustrated in Figure 3-3. We'll come back to this topic when we start to build custom solutions that rely on content types to carry additional information needed by different solution components.

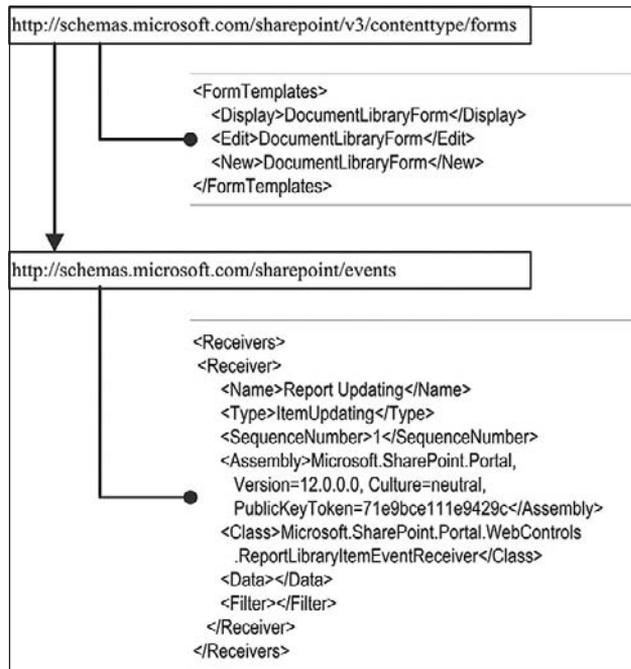


Figure 3-3: Content type payload showing XML documents indexed by namespace.

SharePoint itself uses the XML document collection to store the names of classes and assemblies that implement the event receivers associated with the content type. It is also used to store other information such as the names of the forms that should be used when a particular list item is opened, edited, or created.

Uses for Content Types

SharePoint is such a comprehensive development platform, it is easy to appreciate why many developers often skip the fundamentals and go straight to implementation, focusing too much attention on the

“how” of building a solution, and not enough on the “why”. This section explores a little more carefully the why related to content types, with the not-so-subtle goal of finding a methodology for figuring out when content types are useful and when they are not.

There are many obvious uses for content types. Indeed, the SharePoint architecture is designed around the assumption that many different kinds of solutions will be built around content types. The MOSS publishing features, for example, depend on content types for their implementation. Similarly, workflow in the SharePoint environment relies on content types to manage tasks assigned by workflow activities.

But what if we look beyond the SharePoint environment for a moment and consider the broader uses for a generic “content typing” mechanism? Might this yield a broader definition of ECM than what the SharePoint platform currently supports? To that end, we can identify four primary uses for a such a generic content typing mechanism: organization, classification, validation, and control.

The idea of control is intentionally overbroad to capture both the notions of access control and process control. There may be other kinds of controls that depend on the context in which the content type is used. For the purposes of this discussion, it is good enough to simply distinguish control from classification, organization, and validation.

Organization

The simplest way to use content types is to organize groups of documents by type. This is best used when the organization of the content is not likely to change, or the scenarios in which the content will be used are not well-defined. Using content types to organize documents is often the first step toward developing a more comprehensive content management strategy. In this case, the metadata is not as important as deciding which *bin* to place a document into.

The advantage of using content types instead of simply placing the document into a particular document library is that the organizational scheme can be applied later in the content life cycle across many different sites and site collections without revisiting every document instance.

The disadvantage of using content types merely to organize documents is that it may impose unnecessary constraints on subsequent analysis because of dependencies that tend to diverge from purely organizational goals. For example, you might begin by organizing your documents according to department, distinguishing HR documents from Accounting documents. Later, you may need to define a policy that applies to all Financial documents, regardless of the department they originated from. In this case, using content types for organization would tend to thwart your subsequent functional requirements.

Classification

You can also use content types to define a taxonomy for classifying content based on a specific set of metadata. The content type then becomes a named bundle of metadata that we can work with in much the same way as a traditional class or data structure. However, because we don’t (yet) have a compiler to enforce the classification, problems can arise when we try to build on it. These problems are exacerbated if we try to create very nested hierarchies.

These problems are easy to see when you consider the effect of specialization on a given content-type classification hierarchy. At the top of the hierarchy, you typically have broad categories, like *report* and *statement*. As you move down the hierarchy, more specialization comes in, so you have *bank statement*

Chapter 3: SharePoint Tools for Managing Records

and *policy statement*. These are still broad classifications that are not likely to change very often. But then as you start to move further down the tree, the more specialized the classification becomes and the greater the likelihood that your classification will eventually need to change. So instead of *policy statement*, you now have *document retention policy statement* and *expense reimbursement policy statement*. If new document retention policies have to be implemented because of a change in the law or some other outside influence, then the content type may need to be adjusted, or yet another level would need to be added to the hierarchy.

This is a risky strategy for SharePoint content types because there is no true encapsulation offered by the platform. So there is no way to guarantee that a previously identified content type can be extended without breaking code that depends on that classification. Therefore, any time and effort invested in the creation of deeply nested content type hierarchies are wasted, unless it yields other benefits not directly related to solution development. One such benefit might be the clarity that comes from going through the analysis, especially if knowledge workers are involved as we saw in the section on content modeling in Chapter 1.

Validation

Another way to use content types is to provide a weak encapsulation for the purpose of validating documents in preparation for their involvement in a business process. Content types provide a way to encapsulate metadata and to bind that metadata to a specific set of operations. Those operations could evaluate the metadata to determine whether the current *state* of the document meets the requirements of a particular business process.

I think of this as weak encapsulation because the metadata and the dependent behavior are not linked in any way that can achieve true information-hiding like that provided by a compiled language such as C#. Instead, they are linked through a loose runtime association that allows the metadata to change independently of any code that may depend on it, thus leading to problems if the change is not properly coordinated.

Here again, we are thinking in terms of a *class* with specific *operations*. To implement an effective document validation strategy using content types, we would need to apply even more skill and discipline (discipline because we need the content-type definition to remain fixed throughout the process, so the initial analysis must be as complete as possible).

Control

Since we have adopted a life-cycle model to drive our analysis instead of focusing on the features of a particular platform, it is easy to see that having a generic content typing mechanism with the ability to carry an arbitrary payload can support various operations at various stages of the content life cycle.

Content types can be used to support access control and process control mechanisms by leveraging the payload in various ways. SharePoint and Office use this approach to implement access controls for content after it leaves the server. Information policy information, for instance, is embedded within Office documents and is then used by the Office client applications to control what happens to the document while it is being viewed or edited. We can follow the same model for our own solutions and extend it to control how content is manipulated and transformed throughout its life cycle.

Content Type Definitions

SharePoint content types are defined using an XML template based on Collaborative Application Markup Language (CAML). CAML is an XML derivative or a dialect of XML containing a specialized set of schemas, which SharePoint uses to describe the different components that make up a SharePoint web application. CAML is used throughout the SharePoint platform to describe different SharePoint components. For content types, this includes describing the individual components of the type.

The CAML fragment that describes a content type is called a *content type definition* and is typically read from disk on the SharePoint Server or as part of a SharePoint feature definition. The content type definition is used by SharePoint to create a content type template in the content database. The content type template is then used to create instances of the content type when it is bound to a list. Figure 3-4 shows the relationships between the various components.

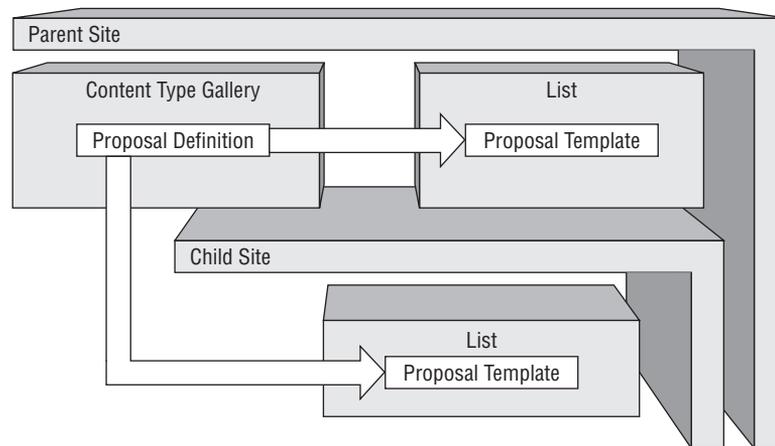


Figure 3-4: Content type definitions, sites, and lists.

The content type definition defines the layout of the metadata, which includes the order, names, and data types of the fields that make up the content type. It also specifies the classes and assemblies for event receivers and which event receiver types have been implemented by those classes.

Creating Content Types

There are two basic ways to create content types: declaratively or programmatically. The traditional approach is to create an XML file containing the required CAML elements and then install the content type as part of a SharePoint feature. This can be somewhat tedious if there are a lot of existing site columns that are referenced from the content type, or if the new content type is based on a deep taxonomy of other content types in its inheritance chain. One way around this is to use the SharePoint user interface on a development server to create the content type and then use a utility to extract the CAML definition from the content database and add it to a feature in Visual Studio. There are tools available on the Internet that simplify the process of extracting the CAML from the content database. Getting the

Chapter 3: SharePoint Tools for Managing Records

CAML is just the first step, however. We would still need to examine the field references individually to make sure they are available in the target site, and since we cannot specify event receivers and custom payloads through the user interface, these would have to be created manually using CAML.

The other approach is to create the content types entirely in code. This has several advantages, including the ability to more easily control changes in the requirements and greater ease in referencing existing content types and site columns. The one major disadvantage is that the SharePoint API does not provide a way to create a content type that is based on an existing content type identifier, but instead always assigns a new identifier when a content type is created using code. This does not work for scenarios that involve deploying several components that must all reference the same content type instance.

Declaring Content Types Using CAML

The standard approach to creating content types is to work directly with CAML. This is usually done by writing the content type definition as part of a SharePoint feature. This approach has the advantage that the CAML content type schema is well-defined and you get IntelliSense support within Visual Studio, so the actual task of writing the XML is not that difficult for simple types. Deploying them is then a simple matter of deploying the feature. The CAML fragment in Listing 3-1 declares a content type for a simple expense report.

Listing 3-1: CAML expense report content type declaration

```
<ContentType ID="0x01010023904EF3F40B244BB2324879A29B3ADC"
  Name="Expense Report" Group="ECM2007"
  Description="A worksheet for recording personal and business expenses."
  Version="13">

<FieldRefs>
  <FieldRef ID="{fa564e0f-0c70-4ab9-b863-0177e6ddd247}" Name="Title"/>
  <FieldRef ID="{6DF9BD52-550E-4a30-BC31-A4366832A87F}" Name="Comment"/>
  <FieldRef ID="{ddd41f57-c23e-43fa-930a-ca2eae38e37e}"
    Name="Project Type" Type="Choice" Required="TRUE">
    <CHOICES>
    <CHOICE>Consulting</CHOICE>
    <CHOICE>Training</CHOICE>
    <CHOICE>Programming</CHOICE>
    <CHOICE>Sales</CHOICE>
    </CHOICES>
  </FieldRef>
</FieldRefs>

</ContentType>
```

The downside of using the declarative approach is that the content type declaration must strictly conform to the content type definition schema. If it does not, then SharePoint will refuse to load it. When you are writing so much CAML code, it's easy to make little mistakes that can be hard to find, so you have to take care when writing it. Also, the field references are defined by GUID, so you have to look them up. This can quickly become a tedious process. To make it a little easier to locate the correct field identifier, you can use the following trick.

Most of the built-in site columns are declared in the file `fieldswss.xml`, which is part of the fields feature located in the `12/TEMPLATES/FEATURES/fields` folder. This file contains about 4,600 lines of CAML code. Instead of searching manually through the file, we'll use an XSLT stylesheet to transform it into a simple HTML table. We'll start by copying the `fieldswss.xml` file to a separate folder, then we'll create a simple XSLT stylesheet like the one shown in Listing 3-2.

Listing 3-2: XSLT stylesheet for field lookups

```
<?xml version="1.0" encoding="utf-8" ?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:wss="http://schemas.microsoft.com/sharepoint/">
  <xsl:output method="html" version="1.0" encoding="utf-8" indent="yes" />

  <xsl:template match="wss:Elements">
    <html>
      <body>
        <h2>SharePoint 3.0 Built-In Fields</h2>
        <table border="0" width="100%" style="font-size:9pt;">
          <tr bgcolor="#9acd32">
            <th align="left">Field</th>
            <th align="left">Group</th>
            <th align="left">Type</th>
            <th align="left">Declaration</th>
          </tr>
          <xsl:apply-templates>
            <xsl:sort select="@Name" />
          </xsl:apply-templates>
        </table>
      </body>
    </html>
  </xsl:template>
  <!------->
  <xsl:template match="wss:Field">
    <tr>
      <td>
        <xsl:value-of select="@Name" />
      </td>
      <td>
        <xsl:value-of select="@Group" />
      </td>
      <td>
        <xsl:value-of select="@Type" />
      </td>
      <td>
        <!-- Emit The Full Declaration -->
        <xsl:element name="FieldRef">
          <xsl:attribute name="ID">
```

Continued

Listing 3-2: XSLT stylesheet for field lookups (continued)

```
<xsl:value-of select="@ID" />
</xsl:attribute>
<xsl:attribute name="Name">
  <xsl:value-of select="@Name" />
</xsl:attribute>
<xsl:attribute name="DisplayName">
  <xsl:value-of select="@DisplayName" />
</xsl:attribute>
</xsl:element>
</td>
</tr>
</xsl:template>
</xsl:stylesheet>
```

Next, we copy the stylesheet into the same folder we copied the fieldswss.xml file into and then open the fieldswss.xml file in Visual Studio. From the Properties tool window, we then select the Stylesheet property and browse to the stylesheet file, as shown in Figure 3-5.

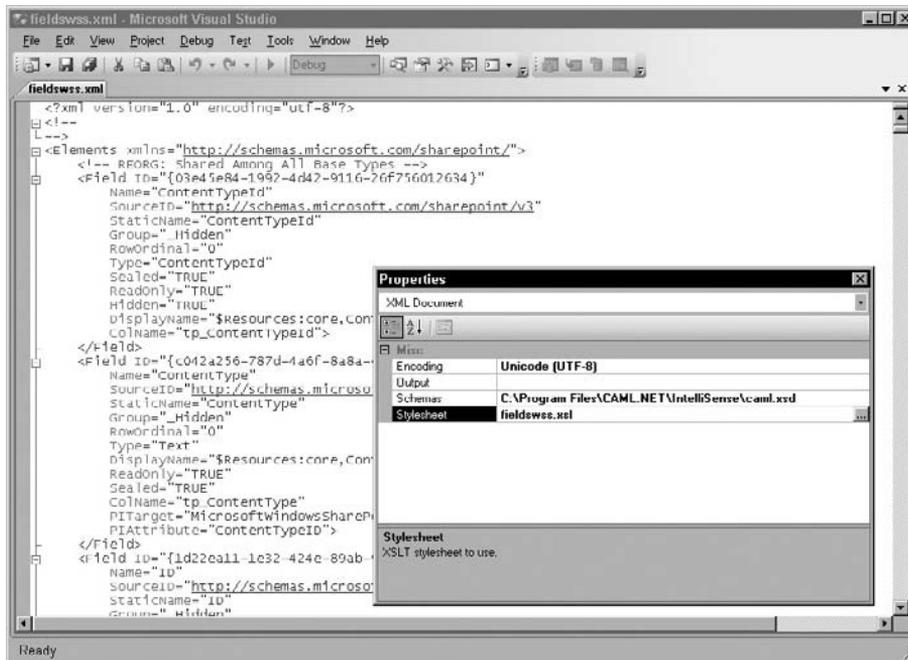


Figure 3-5: Selecting an XSLT stylesheet for the FieldsWSS.xml file.

Now we can click anywhere in the XML text editor for the fieldswss.xml file and select “Show XSLT Output” from the XML menu. The resulting HTML table is shown in Figure 3-6.

Field	Group	Type	Declaration
_Author	\$_Resources.Document_Columns	Text	<FieldRef ID="{246D0907-637C-46b7-9AA0-0BB914DAA832}" Name="_Author" DisplayName="_Resources.core.Author" />
_Category	\$_Resources.Document_Columns	Text	<FieldRef ID="{0F08CACE-C1C3-4654-AE81-B67F2044CA03}" Name="_Category" DisplayName="_Resources.core.Category" />
_CheckinComment	_Hidden	Lookup	<FieldRef ID="{58014f77-5463-437b-ab67-ee79932d667}" Name="_CheckinComment" DisplayName="_Resources.core.Checkin_Comment" />
_Comments	\$_Resources.Document_Columns	Note	<FieldRef ID="{52578FC3-1F01-464d-B016-94CCBCF428CF}" Name="_Comments" DisplayName="_Resources.core.Comments" />
_Contributor	\$_Resources.Document_Columns	Note	<FieldRef ID="{370B7779-0344-4b9f-8F1D-DC1C62EA801}" Name="_Contributor" DisplayName="_Resources.core.Contributor" />
_CopySource	_Hidden	Text	<FieldRef ID="{684e226d-3d88-4a36-8084-a129af52bcdf}" Name="_CopySource" DisplayName="_Resources.core.Copy_Source" />
_Coverage	\$_Resources.Document_Columns	Text	<FieldRef ID="{3B1D59C0-26B1-4d66-ABBD-3EDB4E2CECA}" Name="_Coverage" DisplayName="_Resources.core.Coverage" />
_DCDateCreated	\$_Resources.Document_Columns	DateTime	<FieldRef ID="{9F8B4EE0-84B7-42c6-A094-3CBDE2115EB9}" Name="_DCDateCreated" DisplayName="_Resources.core.DCDateCreated" />
_DCDateModified	\$_Resources.Document_Columns	DateTime	<FieldRef ID="{810BBD02-BBF5-4c67-B1CE-5AD7C5A512B2}" Name="_DCDateModified" DisplayName="_Resources.core.DCDateModified" />
_EditMenuTableEnd	_Hidden	Computed	<FieldRef ID="{2ea78cef-1b69-4019-960a-02c41636cb47}" Name="_EditMenuTableEnd" DisplayName="_Resources.core.Edit_Menu_Table_End" />
_EditMenuTableStart	_Hidden	Computed	<FieldRef ID="{3c6303be-e21f-4366-80d7-d6d0a3b22c7a}" Name="_EditMenuTableStart" DisplayName="_Resources.core.Edit_Menu_Table_Start" />

Figure 3-6: SharePoint field declarations as an HTML table.

Play around with the XSLT code so that it includes all of the detail you need. This example generates the entire field declaration to avoid having to fiddle with the details. Another idea is to include a button that copies the declaration directly to the clipboard.

There are lots of ways to create tools that simplify the process of writing code. Still, once a declarative content type is deployed, the field references are fixed. Because of this, we cannot easily define a higher level of site provisioning code that could, for example, determine dynamically which fields should be included in the content type. Instead, we have to resolve which fields to include well in advance of deployment. In fact, many problems can arise if we decide to change the order or type of declared fields after the fact. From a pure design standpoint, there are many scenarios in which we may not know which fields we need until we've either interacted with the user or consulted an external database, and so on. Going back to our content life-cycle metaphor, these are the kinds of solutions we want to be able to build because these are just the kinds of solutions that map more naturally into the use cases we typically encounter in the field. To address these scenarios, we need a more dynamic approach. The next section describes how to build a utility that makes it easier to declare new content types by browsing through the currently deployed content types in the farm to extract the CAML code from the content database.

A Content Type Browser Utility

To build anything, you need good tools. One point of frustration for many SharePoint developers is the lack of available tools. Although this situation is constantly improving, there is still a gap in the marketplace for developer tools. Once you start diving deeper into the SharePoint API, this gap becomes even more apparent. Many developers are familiar with Lutz Roeder's excellent .NET Reflector (now owned by Red Hat Software), which has become a mainstay for SharePoint developers because of its integrated disassembler and ease of use. For other areas, it often becomes necessary to build your own tools. Working with content types is one of those areas where a custom tool can go a long way toward enhancing your development experience.

Chapter 3: SharePoint Tools for Managing Records

As a first step, let's use Reflector to peek inside the `Microsoft.SharePoint` assembly and examine the classes we have to work with. In particular, we're interested in the `SPContentType` class shown in Figure 3-7.

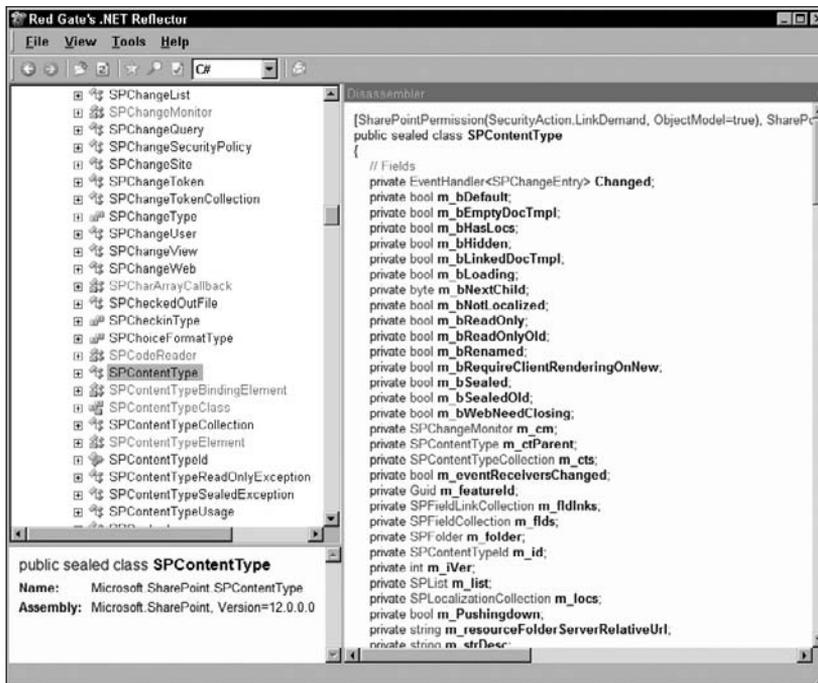


Figure 3-7: The `SPContentType` class in .NET Reflector.

The first thing to point out is that this is a *sealed* class, which means we cannot inherit from this class to implement our own custom specializations. That's probably a good thing, since future versions of the `SPContentType` class are likely to include significant improvements over the current implementation. On the other hand, it would be nice to have a more standard way to build custom content type components.

This is one place where an interface would have been very useful. Having an `ISpContentType` interface that developers could implement and then substitute for the default implementation would have allowed third-party developers and partners to extend the platform quite easily. Hopefully, the next version of SharePoint will address this need.

When developing content types, it helps if you can see what is going on within the content database as you use the content type in various scenarios. This is because the content type payload (XML documents and other properties) carries information that is used throughout a SharePoint deployment. A good example of this is adding an event receiver to an existing content type or applying an information policy. The SharePoint UI does not provide much insight, and a command-line tool, while effective, quickly starts to interfere with developer productivity. It would be nice to have a simple explorer-style forms application just for browsing the content types on the local system.

In the following sections, we'll build a tool that we can then use throughout the book to explore the details of all content types that are deployed on the local server. Then as we develop additional code

that creates or depends on content types, we can easily inspect those details to see how the content type changes and how those changes affect other SharePoint objects. Our Content Type Explorer user interface is shown in Figure 3-8.

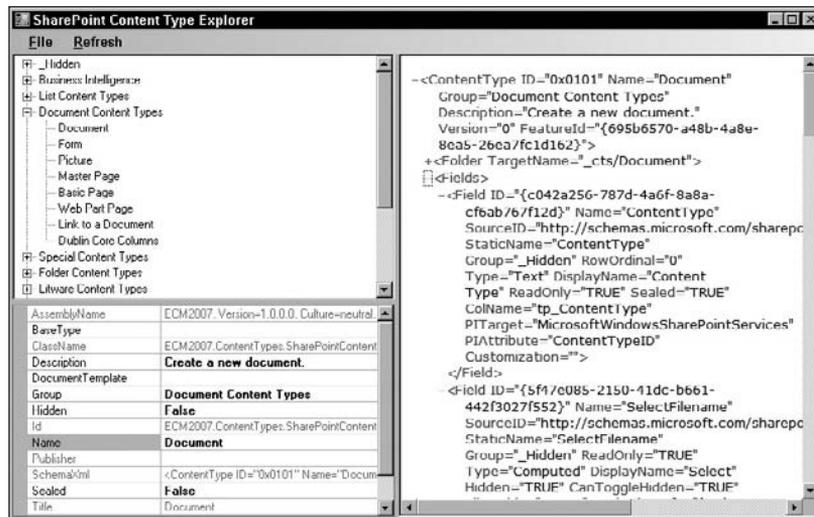


Figure 3-8: A simple Content Type Explorer utility.

The complete source code for the Content Type Explorer is included in the downloadable code for the book.

The utility finds all of the content types on the local farm and then organizes them in a treeview control by group. This list includes all content types in all sites in all site collections in all web applications on the local farm, excluding any duplicates.

What's nice about this is that when you click on an item, the program displays a property grid, but also displays the CAML definition in a separate window. This is the actual schema definition for the content type in its current state. So this gives us a window into the content database that we can use to continually monitor changes that we make either through the SharePoint UI or through our custom code.

This is not a truly live view of the content database, because there are no triggers in place for detecting changes to the content database. One approach to doing that might be to monitor changes to the SharePoint log files and then selectively refresh the display. I'll leave that as an exercise for the reader.

The key component of the content type explorer tool is the code that retrieves all content types in the local farm. A portion of that code is shown in Listing 3-3.

Listing 3-3: Get all ContentTypes

```

/// <summary>
/// Returns a list of all content types that have been defined
/// in the local SharePoint farm.
/// </summary>

```

Continued

Listing 3-3: Get all ContentTypes (continued)

```
/// <returns>the complete list of local content types</returns>
public static List<SPContentType> GetAllContentTypes()
{
    List<SPContentType> allContentTypes = new List<SPContentType>();

    // Create a dictionary to hold the actual instances.
    Dictionary<string, SPContentType> types =
        new Dictionary<string, SPContentType>();

    // Loop over all servers in the farm.
    foreach (SPServer server in SPFarm.Local.Servers)
    {
        // loop over all services running on the server.
        foreach (SPServiceInstance service in server.ServiceInstances)
        {
            // check for the SPWeb service
            if (service.Service is SPWebService)
            {
                SPWebService spws = (SPWebService)service.Service;
                // loop over the IIS web applications controlled by this service.
                foreach (SPWebApplication webapp in spws.WebApplications)
                {
                    // loop over all site collections in the web application.
                    foreach (SPSite siteCollection in webapp.Sites)
                    {
                        // loop over all webs in the site collection.
                        foreach (SPWeb web in siteCollection.AllWebs)
                        {
                            // loop over all available content types
                            foreach (SPContentType ct in web.AvailableContentTypes)
                            {
                                if (!types.ContainsKey(ct.Name))
                                    types[ct.Name] = ct;
                            }
                        }
                    }
                }
            }
        }
    }

    // return a list of just the values
    foreach (SPContentType type in types.Values)
        allContentTypes.Add(type);

    return allContentTypes;
}
```

Starting with the global `SPFarm` object, we get all of the servers in the farm and look for the Web Service instance for each server. Then we drill down, getting the web applications, the `SPSite` objects, and then each `SPWeb`. Finally within the Web, we access the `AvailableContentTypes` collection so we are sure to see all of the content types that are defined within that Web or any of its parent Webs.

Although we could have written the code more narrowly to retrieve only content types defined in a given Web, this is good enough for our purposes.

Once we have a content type, we then add it to a dictionary keyed on the content type name. The rest of the magic happens in the `treeview` control, which then finds the group associated with each content type and puts them in the tree at the appropriate place.

Creating Content Types Programmatically

Obviously, having a collection of reusable components is a good thing. One of my goals with this book is to provide developers with reusable code that can be applied to many different kinds of ECM solutions. This is especially true for something as fundamental as content types. We'll start by creating a new class library called `ECM2007`, and then throughout the book, we'll gradually build it out by adding additional classes and code to deal with each topic as it comes along. By the end, we should have a pretty robust collection of tools that will make your job much easier.

The `ECM2007` class library is included as part of the downloadable companion code for the book.

In the interest of extensibility, we'll start by defining an abstraction for any SharePoint object. This may seem like overkill for creating content types, but bear with me. Remember, we'll be adding other things to our component model as we go along, so having a simple abstraction should quickly become more of a benefit than a hindrance. The main reason we need this is so that we can rely on a common set of properties that all SharePoint objects provide. This will allow us to write generic methods that operate consistently across all objects in the library. It will also allow us to quickly identify those objects that don't fit the model. The `ISharePointObject` interface is shown in Listing 3-4.

Listing 3-4: `ISharePointObject` interface

```
/// <summary>
/// Provides an abstraction for any SharePoint object.
/// </summary>
public interface ISharePointObject
{
    // shared public properties
    string Id { get; }
    string Name { get; }
    string Title { get; }
    string Description { get; }
    string SchemaXml { get; }
    string Publisher { get; }
    string AssemblyName { get; }
    string ClassName { get; }
}
```

The key properties declared in this interfaces are the `SchemaXml` property, which returns the raw CAML that defines an object, and the `AssemblyName` and `ClassName` properties, which are used to locate custom code associated with it.

Now we have two options for developing our library of helper classes. We can create an abstract base class that implements the `ISharePointObject` interface and then derive our helper classes from it. This should suffice for many components that do not have special requirements. But there are other

Chapter 3: SharePoint Tools for Managing Records

classes for which this approach will not work, either because they must inherit from some other class (as is the case for certain SharePoint objects) or because they require special handling (typically during the deployment phase).

Content types are one of those special cases — not so much because of the way that SharePoint defines them, but because of the way we intend to work with them. In short, we want to build a set of tools that yield the same expressive power that CAML provides, but using managed code instead of XML.

One way to accomplish this is through .NET Reflection. By using .NET attribute classes, we can enable users of the class library to mark up their own classes in such a way that the foundation classes in the library can figure out everything that is needed to declare the component and create the corresponding object in the content database.

First, we define an `Attribute` class called `SharePointContentType` that can be applied to any other class. We can refer to the class to which the attribute has been applied as the `holder` class. The attribute itself will include properties that map to those of the `SPContentType` class as defined by the SharePoint object model.

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Diagnostics;
using System.IO;
using System.Reflection;
using System.Runtime.InteropServices;
using System.Text.RegularExpressions;
using System.Xml;
using ECM2007.Utilities;
using Microsoft.SharePoint;
using Microsoft.SharePoint.Administration;

namespace ECM2007.ContentTypes
{
    [AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
    public class SharePointContentType : Attribute, ISharePointObject
    {
    }
}
```

The class inherits from the `System.Attribute` class, and the `AttributeTargets` attribute is added to the class to tell the .NET compiler where the `SharePointContentType` attribute can be placed. Notice that although we are declaring an attribute class, we can also apply other attributes to it, highlighting the power and flexibility of the .NET Reflection architecture for adding custom markup to code.

Next, we implement the `ISharePointObject` interface on our `SharePointContentType` attribute class. This will allow us to reflect over any `holder` classes provided by our callers. We can then rely on our `SharePointObject` abstraction to obtain the properties we need to construct a content type within the content database. The essential parts of the `SharePointContentType` attribute class are shown in Listing 3-5.

Throughout the code examples, you will see references to a `Helpers` class. This is a utility class declared in the `ECM2007` class library that exposes methods for diagnostic logging, creating event receivers, locating attributes, and copying streams. It is implemented in the file `ECM2007/Common/Helpers.cs`.

Listing 3-5: SharePointContentType core properties

```
#region Core Properties

/// <summary>
/// Calculates the identifier automatically based on attributes or type.
/// </summary>
public virtual string Id
{
    get
    {
        // try for the Guid attribute
        GuidAttribute guidAttribute = Helpers.FindAttribute<GuidAttribute>(this);
        if (guidAttribute != null) return guidAttribute.Value;
        // use the type name
        return GetType().FullName;
    }
}

private string m_name;
// The content type name.
public string Name
{
    get
    {
        if (m_ct != null)
            return m_ct.Name;
        if (!string.IsNullOrEmpty(m_name))
            return m_name;
        NameAttribute attribute = Helpers.FindAttribute<NameAttribute>(this);
        if (attribute != null)
            return attribute.Name;
        return GetType().Name;
    }
    set { m_name = value; }
}

// The content type title.
public string Title
{
    get { return m_ct != null ? m_ct.Name : string.Empty; }
}

private bool m_hidden;
// Whether or not the content type is hidden from the user interface.
public bool Hidden
{
    get { return m_ct != null ? m_ct.Hidden : m_hidden; }
    set { m_hidden = value; }
}

private bool m_sealed;
// Whether or not the content type is sealed to prevent modifications.
```

Continued

Listing 3-5: SharePointContentType core properties (continued)

```
public bool Sealed
{
    get { return m_ct != null ? m_ct.Sealed : m_sealed; }
    set { m_sealed = value; }
}

private string m_group;
// The group category with which the content type is associated.
public string Group
{
    get { return m_ct != null ? m_ct.Group : m_group; }
    set { m_group = value; }
}

private string m_description;
// The content type description.
public string Description
{
    get
    {
        if (m_ct != null)
            return m_ct.Description;
        DescriptionAttribute attribute = Helpers.FindAttribute
            <DescriptionAttribute>(this);
        if (attribute != null)
            return attribute.Description;
        if (!string.IsNullOrEmpty(m_description))
            return m_description;
        return string.Empty;
    }
    set { m_description = value; }
}

// Extracts the base type from the raw CAML schema.
private string ParseBaseType(string schemaXml)
{
    return "";
}

private string m_baseType = "Item";
// Specifies the content type on which this one is based.
public string BaseType
{
    get { return m_ct != null ? ParseBaseType(m_ct.SchemaXml) : m_baseType; }
    set { m_baseType = value; }
}

// Retrieves the schema for this object.
public string SchemaXml
{
    get { return m_ct != null ? m_ct.SchemaXml : string.Empty; }
}

// Retrieves the publisher name.
```

```
public virtual string Publisher
{
    get
    {
        PublisherAttribute attribute = Helpers.FindAttribute<PublisherAttribute>
            (this);
        if (attribute != null) return attribute.Name;
        return string.Empty;
    }
}

// Retrieves the fully qualified assembly name for this component.
public virtual string AssemblyName
{
    get
    {
        return GetType().Assembly.GetName().FullName;
    }
}

// Retrieves the fully qualified class name for this component.
public virtual string ClassName
{
    get
    {
        return GetType().FullName;
    }
}

}

#endregion
```

In addition to the core properties of the content type, we also need to declare the individual fields, which are declared using the `FieldRef` element in CAML. When declaring content types in code, we need a similar mechanism to specify the individual columns that make up the content type. The most natural way is to declare the columns in the same way we would declare the public properties of a class. Here again, we can use reflection to declare the properties in the normal way and then mark up selected properties to map them to SharePoint columns. To accomplish this, we can define a second `FieldRef` attribute, as shown in Listing 3-6.

Listing 3-6: `FieldRef` attribute class

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Reflection;
using Microsoft.SharePoint;

namespace ECM2007.ContentTypes
{
    /// <summary>
    /// Use this attribute class to markup public properties
    /// that you want to map to SharePoint fields.

```

Continued

Listing 3-6: FieldRef attribute class (continued)

```
/// </summary>
[AttributeUsage(AttributeTargets.Property, AllowMultiple = false, Inherited =
true)]
public class FieldRef : Attribute
{
    #region Constructors

    public FieldRef()
    {
        InitProperties();
    }

    public FieldRef(string fieldName)
    {
        InitProperties();
        this.FieldName = fieldName;
    }

    void InitProperties()
    {
        this.ShowInDisplayForm = true;
        this.ShowInEditForm = true;
        this.ShowInListSettings = true;
        this.ShowInNewForm = true;
        this.ShowInVersionHistory = true;
        this.ShowInViewForms = true;
        this.DisplaySize = 30;
    }
    #endregion

    #region Properties

    public string FieldName { get; set; }
    public string Description { get; set; }
    public string Group { get; set; }
    public int DisplaySize { get; set; }
    public bool Required { get; set; }
    public bool Hidden { get; set; }
    public bool Indexed { get; set; }
    public bool ReadOnly { get; set; }
    public bool ShowInDisplayForm { get; set; }
    public bool ShowInEditForm { get; set; }
    public bool ShowInListSettings { get; set; }
    public bool ShowInNewForm { get; set; }
    public bool ShowInVersionHistory { get; set; }
    public bool ShowInViewForms { get; set; }

    private string m_displayName = string.Empty;
    public string DisplayName
    {
        get { return string.IsNullOrEmpty(m_displayName) ?
```

```
        this.FieldName : m_displayName; }
        set { m_displayName = value; }
    }

    #endregion
}
}
```

Here, we can take advantage of additional information that the compiler knows about the property, such as its underlying data type and visibility. The following code in the `FieldRefAttribute` class shows how we can extract the field type from the property itself. We can also detect if the data type is an enumeration type and automatically create `CHOICE` fields in the content type.

CHOICE fields are a built-in field type that allows the SharePoint user to select from a list of values, either using a dropdown control or a set of option buttons.

```
/// <summary>
/// Calculates the SPFieldType based on the reflected property info.
/// </summary>
/// <param name="propInfo">describes the property to which this attribute is
/// attached</param>
/// <returns>an SPFieldType that corresponds to the underlying property
/// type</returns>
private SPFieldType GetFieldType(PropertyInfo propInfo)
{
    if (propInfo.PropertyType.IsEnum)
        return SPFieldType.Choice;
    switch (propInfo.PropertyType.Name.ToLower())
    {
        case "int": return SPFieldType.Integer;
        case "decimal": return SPFieldType.Currency;
        case "double": return SPFieldType.Number;
        case "float": return SPFieldType.Number;
        case "datetime": return SPFieldType.DateTime;
        case "guid": return SPFieldType.Guid;
        case "bool": return SPFieldType.Boolean;
    }
    // check the size to determine which type of text field it is
    if (this.DisplaySize > 255)
        return SPFieldType.Note;
    return SPFieldType.Text;
}
```

The final piece of the puzzle is provided by a utility method that adds a given field reference to an existing content type. The big advantage here is that we don't have to look up field identifiers when referencing existing field names using the object model. The SharePoint API looks up the field for us and sets the correct identifier automatically.

```
/// <summary>
/// Adds the field reference to a content type.
/// </summary>
/// <param name="ct"></param>
```

Chapter 3: SharePoint Tools for Managing Records

```
/// <param name="propInfo"></param>
public void AddToContentType(SPContentType ct, PropertyInfo propInfo)
{
    try
    {
        // Locate the field in the web associated with the type.
        SPWeb web = ct.ParentWeb;
        SPField field = null;

        try
        {
            // Perform a special check for changes to the
            // display name of the Title field.
            if (this.FieldName == "Title")
            {
                SPFieldLink fieldLink = ct.FieldLinks["Title"];
                fieldLink.DisplayName = this.DisplayName;
                ct.Update();
                return;
            }

            // check if it's in the available fields
            field = web.AvailableFields[this.DisplayName];
            if (field != null)
            {
                // now check if it's in the actual web
                // if so, then delete it
                field = web.Fields[this.DisplayName];
                if (field != null && field.CanBeDeleted)
                {
                    web.Fields.Delete(this.DisplayName);
                    field = null;
                }
            }
        }
        catch { }

        // Create a new field if none exists.
        if (field == null)
        {
            try
            {
                SPFieldType fieldType = GetFieldType(propInfo);
                string fieldName = web.Fields.Add(this.DisplayName, fieldType,
                    this.Required);
                field = web.Fields[this.DisplayName];
                if (!string.IsNullOrEmpty(this.Description))
                    field.Description = this.Description;
                if (!string.IsNullOrEmpty(this.Group))
                    field.Group = this.Group;
                field.Hidden = this.Hidden;
                field.Indexed = this.Indexed;
            }
            catch { }
        }
    }
}
```

```
field.ReadOnlyField = this.ReadOnly;
field.ShowInDisplayForm = this.ShowInDisplayForm;
field.ShowInEditForm = this.ShowInEditForm;
field.ShowInListSettings = this.ShowInListSettings;
field.ShowInNewForm = this.ShowInNewForm;
field.ShowInVersionHistory = this.ShowInVersionHistory;
field.ShowInViewForms = this.ShowInViewForms;
field.DisplaySize = this.DisplaySize.ToString();

// Add choices for an enum type
if (fieldType == SPFieldType.Choice && propInfo.PropertyType.
    IsEnum)
{
    SPFieldChoice choiceField = field as SPFieldChoice;
    foreach (string s in Enum.GetNames(propInfo.PropertyType))
        choiceField.Choices.Add(s);
    choiceField.Update();
}
}
catch { }
}

if (field != null)
{
    try
    {
        // avoid adding duplicate field links
        bool found = false;
        foreach (SPFieldLink fieldLink in ct.FieldLinks)
            if (fieldLink.Name.Equals(field.Title))
            {
                found = true;
                break;
            }
        if (!found)
            ct.FieldLinks.Add(new SPFieldLink(field));
    }
    catch
    {
    }
}
}
catch
{
}
}
```

With these attributes and components in place, we can now declare content types easily in code. Listing 3-7 shows the declaration of a sample content type.

Listing 3-7: Sample content type created using reflection

```
using System.Diagnostics;
using ECM2007.ContentTypes;
using Microsoft.SharePoint;

namespace SampleContentTypes
{
    [ SharePointContentType(BaseType="Document",
        Description="My Sample Content Type",
        Group="ECM2007",
        Name="MyContentType",
        Sealed=false) ]
    public class SampleContentType : SPItemEventReceiver
    {
        [FieldRef("Dept",
            Description="The name of the department",
            DisplayName="Sample Department")]
        public string Department { get; set; }
        public string Manager { get; set; }

        public override void ItemAdded(SPItemEventProperties properties)
        {
            base.ItemAdded(properties);
            Trace.WriteLine("SampleContentType - ItemAdded");
        }
    }
}
```

Notice that the sample content type shown above inherits the `SPItemEventReceiver` class in order to implement an event receiver. This is made possible by additional code in the attribute class that searches the methods implemented by the class to automatically attach any event receivers it finds. Inheriting from `SPItemEventReceiver` here is similar to what you would do to create any event receiver. The advantage of using Reflection to bind the event receiver to the content type declaration in this way is that the markup is added directly to the code instead of in a separate file, creating a stronger encapsulation between the properties of the content type and its behavior.

Versioning

Versioning is a key feature for content management and provides a foundation for collaboration and approval. Without the ability to track the versions of an item, it would not be possible to establish a baseline for publication from which successive refinements can be made. It would also not be possible to “roll back” or to track the progression of an item from draft to final approval.

The word *version* has a common definition and may lead us to assume that it is just a mechanism for keeping track of which versions there are and that there is some sort of built-in differencing engine being used to detect what has changed between one version and the next. But there are some additional considerations that come into play when dealing with versioning in the context of a SharePoint content

database. One question concerns when versions are created. Another has to do with what changes occur within the content database when versioning is enabled. So we need to dig a little deeper into the advantages and disadvantages of using versioning in the SharePoint environment and then see how versioning affects Records Management on the SharePoint platform.

All of this is built on top of the object model that SharePoint provides for managing content as though it were a file system. The SharePoint virtual file system includes `SPFolder` and `SPFile` objects and exposes a robust layer of functionality that can be leveraged extensively to support custom versioning scenarios. There is also a close relationship in SharePoint between how versioning works and how check-in and check-out work.

SharePoint includes the ability to track changes to sites, site collections, lists, and list items. However, versioning applies only to items stored in lists and document libraries. This is because versions are associated with `SPFile` objects. Versioning does not apply to the other objects, and it does not apply to content types. On the other hand, versioning can be enabled for any kind of list item in any type of list. Document libraries support both major and minor versions, while lists support major versions only. The primary benefits of versioning in SharePoint include:

- ❑ The ability to roll back to a previous version
- ❑ The ability to selectively view or delete individual versions
- ❑ The ability to maintain a history of changes made to a given item

The big advantage of versioning, of course, is that SharePoint monitors file saves so that changes can be rolled back easily along a timeline. It also creates a *version history*, from which you can also selectively review and delete prior snapshots of the document. One big disadvantage, however, is that it is not an incremental snapshot. There is no “differencing engine” that SharePoint uses to keep track of minor changes to a document or list item. Therefore, there is a cost to be paid in terms of storage space for the flexibility that versioning offers. Versioning is especially important in the MOSS environment when you’re designing and building publishing portals.

You can enable versioning for any kind of list item in any kind of list. There is no concept, however, for applying versioning to a content type. Remember that a content type is really just a description of how a list item should be created, with some additional guidance for how instances should behave. When a content type is attached to a list, it influences the columns that are created within the list, and the content type ID controls which columns are displayed for each list item. But it’s the list item, not the content type, that is being instantiated. And so when we talk about versioning, we’re only talking about the list item. Specifically, we’re talking about the `SPFile` object attached to the list item, which is where the content physically resides.

There is an option for an administrator to place a limit on the maximum number of versions that SharePoint will retain. But this is available only for major versions. There is no way to limit the number of minor versions that SharePoint will keep. Thus, storage costs will tend to escalate if a particular type of document requires many iterations before producing a published major version. Again, there is no way to assign these rules to content types, although that would be more natural when designing solutions. Ideally, we would like to specify version throttling according to document type instead of at the list level, especially since each list may contain many different kinds of documents. Currently, there is no way to accomplish this. If you need special throttling for a particular document type, then you have to create a special list that contains only that document type.

Versioning Rules

There are specific rules that govern how SharePoint handles versions. Figure 3-9 illustrates the algorithm used to determine when a version is created or updated within the SharePoint content database.

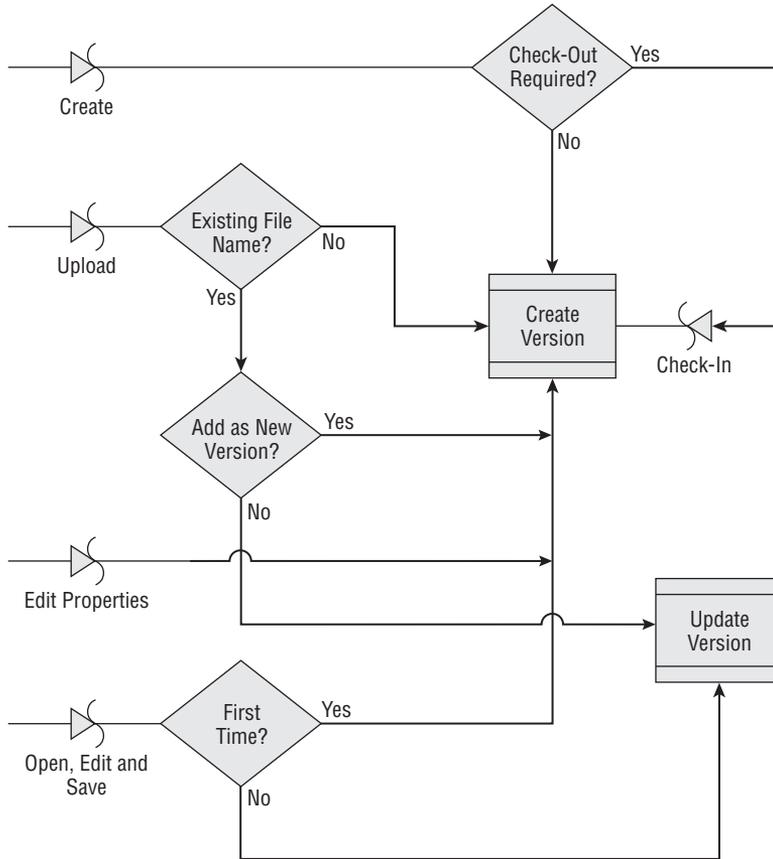


Figure 3-9: SharePoint versioning algorithm.

When you create a document, check-out may or may not be required on the list. Forced check-out is a configuration option that is available for each list. When a new document is created, SharePoint checks the forced check-out flag to determine if check-out is required. If it is, then it first performs the check-out procedure, which will eventually lead to a subsequent point along the timeline at which a check-in must occur. Not until the file is checked in will a new version be created for the file. On the other hand, if forced check-out is not enabled for the list, then a new version is created immediately.

The upload sequence is slightly different from creating a document. When a document is uploaded, there is the possibility that the document being uploaded may overwrite an existing document with the same filename. If that is not the case and there is no naming conflict, then SharePoint just creates the new version. If there is a naming conflict, then the user is presented with the option to either add the new document as a new version or overwrite the old one. You could also go into the document and edit its properties. With versioning enabled, a new version is created immediately.

A different sequence is followed when just opening the document, for example, in Word 2007 and then editing and saving it. The first time the document is saved, a new version is created. For subsequent saves in the same *session*, it simply updates the current version. This means that there is no additional storage cost for saving the same document multiple times per editing session. If a new editing session is started, or if the cookie times out, then a new version is created the next time the document is saved. Figure 3-10 shows what happens when editing a Word 2007 document in a document library with major versions enabled.

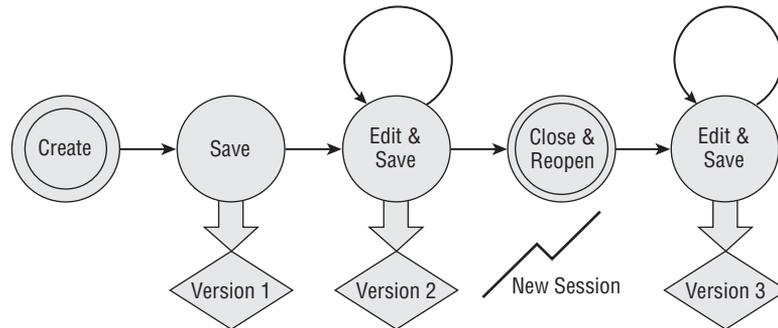


Figure 3-10: Document versioning sequence.

This algorithm carries some implications for users who rely on the ability to revert to a prior version after saving the file. If they perform a long series of edits, keeping the document checked out for long periods of time, they will still overwrite the current version each time the document is saved, because SharePoint sees this as a single editing session. It is important, therefore, for users to understand that for long editing sessions, it is better to check out the document, save it locally, and then check it back in when the edits are completed.

Check-Out, Check-In, and Versioning

Check-out and check-in provide a standardized “locking” mechanism for documents that works the way most people understand. The primary characteristics are as follows:

- ❑ The person who checks out the document *owns* the item.
- ❑ The owner (or a delegate) can check in or undo the check-out.
- ❑ Undo discards the new versions.

The SharePoint check-out and check-in mechanisms can alter the versioning sequence and also depend on whether minor versioning is enabled. There is also the option in SharePoint to force the user to check out a document before editing it. This also has an impact on the versioning sequence. Figure 3-11 illustrates what happens when the user interface is used to check out a document with minor versioning enabled. After the initial save, the document is assigned a minor version. Then, if the document is closed and then checked out on the server, immediately a new version (0.2) is created on the server. Discarding the check-out at that point reverts back to version 0.1. Checking in the document (whether changes were made or not), the user is presented with a choice of whether to update the current version, create a new minor version (0.2), or publish a new major version (1.0).

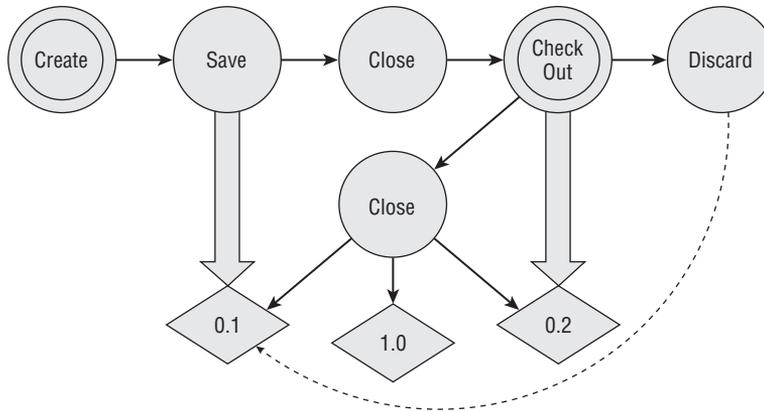


Figure 3-11: Check-out versioning sequence.

One problem with forced check-out is that there is no way to do it when you're using content types. In the real world, we would really like to force check-out and have that setting applied to the item based on the content type. Currently, the only way to force check-out is to configure the document library directly. The result is that even though the library may support multiple content types, check-outs will apply to every document in the library. And since there is no event that fires when the user begins the editing process, there is no way to programmatically force a check-out on the fly for a given content type. Other approaches for solving the problem might include writing extensive JavaScript code that essentially hijacks the entire process of editing a document or implementing a custom interface that enforces custom business rules when retrieving the files. Both of these approaches would be costly to implement.

Accessing Document Versions in the Browser

SharePoint provides several shortcuts to access information through the browser using *virtual paths*, which are specially recognized tokens that can be added to a URL to retrieve data from the content database. To retrieve a particular version of a document, all that is needed is the version number and the relative path to the document.

The form of the URL is `http://www.mydomain.com/_vti_history/###/Shared%20Documents/mydocument.docx`. The virtual pathname is `_vti_history` followed by a forward slash and a number we can refer to as the *version selector*.

SharePoint applies a simple algorithm to calculate the version selector that is based on a modulus of the number 512. Thus, major versions are a multiple of 512, and minor versions are the remainder after dividing by 512.

As an example, to retrieve version 2.0 of "a large document.doc" from the Versioned Documents library, we could enter the following URL: `http://www.mydomain.com/_vti_history/1024/Versioned%20Documents/A%20Large%20Document.doc`. Here, the value of 1024 is computed by multiplying 512 by the major version of 2. To get version 2.1, we would enter `http://www.mydomain.com/_vti_history/1025/Versioned%20Documents/A%20Large%20Document.doc`. In this case, the value of 1025 is computed by multiplying 512 by the major version of 2 to get 1,024, and then adding the minor version of 1.

Versioning Pitfalls

The ability to track versions comes at a cost. The most obvious cost is measured in the resulting expansion of the content database. Each new version of a document copies the entire content of the previous version. This cost is greater for documents than for list items. The point is that there is no “differencing engine” available to store only the changed bits, so the decision to enable versioning can have a significant impact on the overall storage profile of the site collection in which the library resides.

One way to mitigate this effect is to place a limit on the number of versions allowed for a given document library. When the limit is reached, new versions replace the oldest versions in round-robin fashion. While this works for major versions, it does not work when minor versioning is enabled, because SharePoint does not provide a way to limit the number of minor versions for a document.

Another pitfall associated with versioning is the introduction of user adoption headaches and frustration because of the way that versioning is implemented. One thing that users find annoying is the way that SharePoint numbers each version. Major and minor versions are numbered differently. Major versions start with 1, while minor versions start with 0. So the first major version of a document is version 1.0, but if minor versions are enabled, then the first version is 0.1. This can be confusing for many users.

Another point of frustration for users is the fact that if a version limit has been applied, then older versions are replaced with new versions once the limit has been reached. Often, users are not aware of such limits that have been imposed by administrators. Furthermore, the limits can be changed at any time. This can cause problems, especially in highly collaborative environments, where some users may rely on the ability to roll back to an older version only to find that the prior content has been lost. So if you’ve set it to keep five versions, users may not realize that by creating a new version, they are automatically deleting the oldest version once they get up to five. That may mean that you have to educate the users about how many versions they actually have available to them. And you may have to take some additional care balancing the need to control the growth of the content database versus controlling how the users interact with it. When the round-robin effect happens, it’s a permanent deletion. SharePoint does not move the document into the Recycle Bin, which would have been useful in those cases in which the user does not know about the deletion and then creates a new version, thereby losing the oldest version.

Also, with the version limits, even if you start from scratch and you set the limit, if that limit is lower than the versions that already exist for an item, the versioning behavior becomes non-intuitive. Although the limit is set to save, say, three documents, if there were already five versions in existence when you set that limit, SharePoint will keep those five versions until you create the sixth. At that time, it will delete the oldest versions so that the total number of versions (excluding the current version) equals the new version limit.

Another issue is related to content approval. If approval is enabled, then moderation is also enabled, and that forces a new version to be created immediately for pending items whether or not they have been approved or rejected. As an example, let’s configure a document library to enable versioning and adjust the default view so that the version number is displayed. Then we’ll turn on content approval so that the approval status is also displayed. Figure 3-12 shows an existing document at version 1.0.

If we then edit the item, the approval status changes to Pending and a new version is created, as shown in Figure 3-13. This makes sense because SharePoint needs to retain the ability to roll back the item if it is ultimately rejected. It also relieves the SharePoint code from having to deal with differences between the proposed changes and the original bits. The problem is it doesn’t actually perform the rollback.

Chapter 3: SharePoint Tools for Managing Records

Notice what happens if we make changes to the document and then reject the item. The version number stays at 2.0, and the content does as well. It does not revert back to the previous version. Consequently, we end up with a duplicate copy of the document in its prior state.

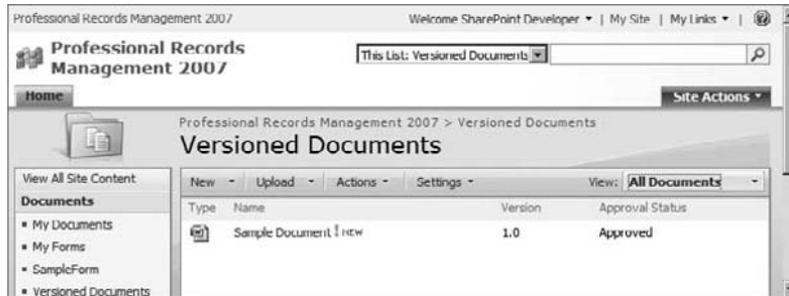


Figure 3-12: Approved list item.

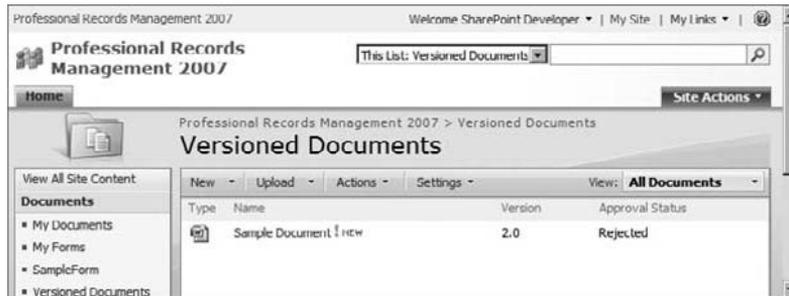


Figure 3-13: Item rejected but version remains.

The absence of a differencing engine can cause other problems, because SharePoint has no way to determine which bits are “good” and which bits are expendable. Consider the simple case of opening the Edit view of a document without actually making any changes and then pressing the OK button anyway. SharePoint dutifully increments the version (and copies the bits) whenever the OK button is clicked on the Edit page without actually comparing the bits to determine if any changes were made. Figure 3-14 shows the version history for a rather large document after simply clicking the OK button on the Edit page without actually making any changes. In this case, each exploratory click cost more than half a megabyte in storage space.

This is where the object model comes into play. We can use the Versioning API to clean up these kinds of issues where too many versions exist, the content database starts to explode, and performance starts to degrade. This can be difficult code to write, however, because again we have the problem of needing to examine the bits of each item to determine the proper course of action. At a minimum, we can provide an administrator with discovery tools so that informed choices can be made.

Professional Records Management 2007 > Versioned Documents > A Large Document > Version History

Versions saved for A Large Document.docx

All versions of this document are listed below with the new value of any changed properties.

Delete All Versions | Delete Draft Versions

No. ↓	Modified	Modified by	Size	Comments
8.0	7/29/2009 5:58 AM	SharePoint Developer	521.6 KB	
7.0	7/29/2009 5:58 AM	SharePoint Developer	521.6 KB	
6.0	7/29/2009 5:58 AM	SharePoint Developer	521.6 KB	
5.0	7/29/2009 5:58 AM	SharePoint Developer	521.6 KB	
4.0	7/29/2009 5:58 AM	SharePoint Developer	521.6 KB	
3.0	7/29/2009 5:58 AM	SharePoint Developer	521.6 KB	
2.0	7/29/2009 5:58 AM	SharePoint Developer	521.6 KB	
1.0	7/29/2009 5:57 AM	SharePoint Developer	521.6 KB	

Approval Status Pending

Figure 3-14: Version history after null edits.

Programmatic Versioning

Uses for programmatic versioning include:

- Fine-tuning the content database
- Managing large collections of documents
- Fixing versioning problems

So how do we deal with these issues? There are a lot of different scenarios where it may become necessary to programmatically adjust the version history. We might need to delete versions that were created unnecessarily. Of course, we can do that piecemeal by painstakingly examining each document and then deciding which version to keep. Or we could create a set of versioning tools that help to identify duplicate items (by comparing bits) and then automatically archive or eliminate them. We could have another set of routines that search for list items or documents that match a given versioning profile.

As an example, we might create a routine that calculates the average number of versions that exist for all documents in a document library. We could use this to create a web part that shows a list of the document libraries in a site along with the average number of versions being kept for documents in that library. Yet another idea might be to retrieve a list of all documents that have a greater than average version count. The basic concept is to build a versioning component library for calculating versioning metrics so we can use them to construct web parts and utilities that deal with version-specific issues in the content database. Based on these metrics, we can make more informed choices about setting the correct versioning limits, determining whether minor versions are really required, and so on.

In order to build a versioning component library, we need to define a versioning API. In this section, we'll develop the API and then build a set of versioning components and add them to our ECM2007 class library. These components will focus on calculating versioning metrics for lists and list items. To make our components available to any code that deals with versioning, we'll declare them as *extension methods*.

C# 3.0 Extension Methods

Version 3.0 of the C# language introduced a new feature called *extension methods*, which are static methods that can be invoked using instance method syntax. This allows new functionality to be added to existing classes without modifying the assembly in which those classes are defined. In order to use extension methods in your code, you need to install Microsoft .NET Framework 3.5.

As an example, the following code adds a `ToStringEx` method to the `SPListItem` class:

```
namespace ECM2007.Extensions
{
    using Microsoft.SharePoint;

    public static class SPListItemEx
    {
        public static string ToStringEx(this SPListItem item)
        {
            return string.Format("Extended string for item: {0}",
                item.ToString());
        }
    }
}
```

Extension methods are declared by specifying the `this` keyword as the first parameter of the method. To call the method, we add a `using` statement to reference the containing namespace and then simply invoke it as though it were a part of the `SPListItem` class, like so:

```
public static void DumpItems(SPList list)
{
    foreach (SPListItem item in list.Items)
        Console.WriteLine(item.ToStringEx());
}
```

Calculating Version Metrics

It is useful to create a set of routines that find list items or documents that match a given versioning profile. This makes it possible to focus in on the versioning metrics and decide further actions to take for a given set of list items. As an example, when a document library has been up-and-running for a while, it may not be obvious how it is being used. Having a set of routines that calculate, for instance, the average number of versions for all documents in the library makes it possible to produce a report showing a list of the items that have an unusually high number of major or minor versions. Such a report could then be used to either trim the document library or move those documents to a different one that is perhaps designed to handle the higher collaboration frequency.

This set of routines will be applied to list items instead of files so they can be used with the `SPListItemCollection` class. This class is a good choice to extend because it can be retrieved either directly from a list or indirectly as the return value of a CAML query, making it easy to incorporate versioning metrics into web parts and other controls. Since they are dealing with list items, the first thing to do is extend the `SPListItemVersion` class so that it's not necessary to parse the version label for every version instance referenced. This class is located in the `Versioning` folder of the `ECM2007` project.

```
public static class SPListItemVersionEx
{
    /// <summary>
```

```
/// Returns the major version number as an integer.
/// </summary>
public static int GetMajorVersionNumber(this SPListItemVersion version)
{
    return Int32.Parse(version.VersionLabel.Split('.')[0]);
}

/// <summary>
/// Returns the minor version number as an integer.
/// </summary>
public static int GetMinorVersionNumber(this SPListItemVersion version)
{
    return Int32.Parse(version.VersionLabel.Split('.')[1]);
}
}
```

There is a separate class called `SPFileVersion` that is used to retrieve version information for the files attached to a list item. Whereas the `SPListItemVersion` class includes the metadata fields associated with each item, the `SPFileVersion` class includes the array of bytes associated with the file. This is not just the new bytes (or delta) associated with a particular version, but a full copy of the original file, including both the old and the new bytes.

Next, a couple of methods are needed for the `SPListItem` class to count versions and to determine the sizes of each copy of the files associated with each version. Additional routines can then be constructed using these as a base.

```
public static class SPListItemEx
{
    /// <summary>
    /// Returns the highest version number for the item.
    /// </summary>
    /// <param name="item">the list item to be tested</param>
    /// <param name="level">optional version level to test</param>
    /// <returns>the number of matching versions</returns>
    public static int GetVersionCount(
        this SPListItem item, params SPFileLevel[] level)
    {
        int result = 0;
        if (item.File != null && item.File.Versions.Count > 0)
        {
            SPFileLevel targetLevel = SPFileLevel.Published;

            if (level != null && level.Length > 0)
                targetLevel = level[0];

            foreach (SPListItemVersion version in item.Versions)
            {
                int versionNumber = 0;
                if (version.Level == targetLevel)
                {
                    switch (targetLevel)
                    {
                        case SPFileLevel.Published:
                            versionNumber = version.GetMajorVersionNumber();
                            break;
                    }
                }
            }
        }
    }
}
```

```
        case SPFileLevel.Draft:
            versionNumber = version.GetMinorVersionNumber();
            break;
    }
}
if (versionNumber > result)
    result = versionNumber;
}
}
return result;
}

/// <summary>
/// Returns the size in bytes by version for the item.
/// </summary>
/// <param name="item">the list item to be tested</param>
/// <param name="level">optional version level to test</param>
public static long GetByteCount(
    this SPListItem item, params SPFileLevel[] level)
{
    long result = 0;
    if (item.File != null)
    {
        if (level == null || level.Length == 0)
            result = item.File.Length;
        else
        {
            foreach (SPFileVersion version in item.File.Versions)
            {
                if (version.Level == level[0])
                    result += version.File.Length;
            }
        }
    }
    return result;
}
}
```

To make it easier to call these routines, we use the `params` keyword for the `SPFileLevel` array so that the `level` parameter is optional. Although it is declared as an array, the code only references the first element if provided to match versions at that level.

Listing 3-8 shows a set of routines that extend the `SPListItemCollection` class to provide versioning metrics. This class file is also located in the `Versioning` folder of the `ECM2007` project.

Listing 3-8: `SPListItemCollectionEx.cs`

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

```
using Microsoft.SharePoint;

namespace ECM2007.Versioning
{
    /// <summary>
    /// Extension methods for SPListItemCollection.
    /// </summary>
    public static class SPListItemCollectionEx
    {
        /// <summary>
        /// Determines the highest version number of all items in the collection.
        /// </summary>
        /// <param name="items">a collection of list items</param>
        /// <param name="level">optional level of items to test</param>
        /// <returns>the maximum number of versions in the collection</returns>
        public static int GetHighestVersion(
            this SPListItemCollection items, SPFileLevel level)
        {
            int result = 0;
            foreach (SPListItem item in items)
            {
                int count = item.GetVersionCount(level);
                if (count > result)
                    result=count;
            }
            return result;
        }

        /// <summary>
        /// Computes the average number of versions in the collection.
        /// </summary>
        /// <param name="items">a collection of list items to be tested</param>
        /// <param name="level">optional level of items to test</param>
        /// <returns>the average number of versions in the collection</returns>
        public static double GetAverageVersionCount(
            this SPListItemCollection items, SPFileLevel level)
        {
            double result = 0.0;
            int count = items.Count;
            if (count > 0)
            {
                foreach (SPListItem item in items)
                    result += item.GetVersionCount(level);
                result /= count;
            }
            return result;
        }

        /// <summary>
        /// Computes the total number of versions contained in the collection.
        /// </summary>
        /// <param name="items">list containing items to be tallied</param>
        /// <param name="level">optional level of items to test</param>
        /// <returns>the total number of versions in all items of the
```

Continued

Listing 3-8: SPListItemCollectionEx.cs (continued)

```
list</returns>
public static int GetVersionCount(
    this SPListItemCollection items, SPFileLevel level)
{
    int result = 0;
    foreach (SPListItem item in items)
        result += item.GetVersionCount(level);
    return result;
}

/// <summary>
/// Computes the size of the largest version in the collection.
/// </summary>
/// <param name="items">a collection of list items to be tested</param>
/// <param name="level">optional level of items to test</param>
/// <returns>the size of the largest version in the collection</returns>
public static long GetMaxSize(
    this SPListItemCollection items, SPFileLevel level)
{
    long result = 0;
    foreach (SPListItem item in items)
    {
        long size = item.GetByteCount(level);
        if (size > result)
            result = size;
    }
    return result;
}

/// <summary>
/// Computes the size of the smallest version in the collection.
/// </summary>
/// <param name="items">a collection of list items to be tested</param>
/// <param name="level">optional level of items to test</param>
/// <returns>the size of the smallest version in the collection</returns>
public static long GetMinSize(
    this SPListItemCollection items, SPFileLevel level)
{
    long result = items.Count > 0 ? long.MaxValue : 0;
    foreach (SPListItem item in items)
    {
        long size = item.GetByteCount(level);
        if (size > 0 && size < result)
            result = size;
    }
    return result;
}

/// <summary>
/// Computes the total bytes used by all versions in the collection.
/// </summary>
/// <param name="items">list containing items to be tested</param>
```

Chapter 3: SharePoint Tools for Managing Records

```
/// <param name="level">optional level of items to test</param>
/// <returns>the total bytes used by matching versions of all
/// items in the collection</returns>
public static long GetTotalSize(
    this SPLListItemCollection items, SPFileLevel level)
{
    long result = 0;
    foreach (SPLListItem item in items)
        result += item.GetByteCount(level);
    return result;
}

/// <summary>
/// Computes the average number of bytes used by all versions in the
/// collection.
/// </summary>
/// <param name="items">list containing items to be tested</param>
/// <param name="level">optional level of items to test</param>
/// <returns>the average bytes used by matching versions of all items
/// in the collection</returns>
public static double GetAverageSize(
    this SPLListItemCollection items, SPFileLevel level)
{
    double result = 0;
    int count = items.GetVersionCount(level);
    if (count > 0)
    {
        foreach (SPLListItem item in items)
            result += item.GetByteCount(level);
        result /= count;
    }
    return result;
}

/// <summary>
/// Returns a collection of all items with an above average number of
/// versions.
/// </summary>
/// <param name="items">list containing items to be retrieved</param>
/// <param name="level">optional level of items to include</param>
/// <returns>collection of items with matching version counts
/// above the average for the list</returns>
public static List<SPLListItem> GetItemsByVersionAboveAverage(
    this SPLListItemCollection items, SPFileLevel level)
{
    List<SPLListItem> result = new List<SPLListItem>();
    double averageVersions = items.GetAverageVersionCount(level);
    foreach (SPLListItem item in items)
    {
        if (item.GetVersionCount(level) > averageVersions)
            result.Add(item);
    }
    return result;
}
```

Continued

Listing 3-8: SPListItemCollectionEx.cs (continued)

```
    }

    /// <summary>
    /// Returns all items with a below average number of versions.
    /// </summary>
    /// <param name="items">list containing items to be retrieved</param>
    /// <param name="level">optional level of items to include</param>
    /// <returns>collection of items with matching version counts
    /// below the average for the list</returns>
    public static List<SPListItem> GetItemsByVersionBelowAverage(
        this SPListItemCollection items, SPFileLevel level)
    {
        List<SPListItem> result = new List<SPListItem>();
        double averageVersions = items.GetAverageVersionCount(level);
        foreach (SPListItem item in items)
            if (item.GetVersionCount(level) < averageVersions)
                result.Add(item);
        return result;
    }

    /// <summary>
    /// Returns all items with an above average number of bytes.
    /// </summary>
    /// <param name="items">list containing items to be retrieved</param>
    /// <param name="level">optional level of items to include</param>
    /// <returns>collection of items with matching version sizes
    /// above the average for the list</returns>
    public static List<SPListItem> GetItemsBySizeAboveAverage(
        this SPListItemCollection items, SPFileLevel level)
    {
        List<SPListItem> result = new List<SPListItem>();
        double averageSize = items.GetAverageSize(level);
        foreach (SPListItem item in items)
            if (item.GetByteCount(level) > averageSize)
                result.Add(item);
        return result;
    }

    /// <summary>
    /// Returns all items with a below average number of bytes.
    /// </summary>
    /// <param name="items">list containing items to be retrieved</param>
    /// <param name="level">optional level of items to include</param>
    /// <returns>collection of items with matching version sizes
    /// below the average for the list</returns>
    public static List<SPListItem> GetItemsBySizeBelowAverage(
        this SPListItemCollection items, SPFileLevel level)
    {
        List<SPListItem> result = new List<SPListItem>();
        double averageSize = items.GetAverageSize(level);
        foreach (SPListItem item in items)
            if (item.GetByteCount(level) < averageSize)
                result.Add(item);
    }
}
```

```
        return result;
    }

    /// <summary>
    /// Returns all items with a version count above a specified number.
    /// </summary>
    public static List<SPLListItem> GetItemsByVersionAboveTarget(
        this SPLListItemCollection items, SPFileLevel level, int target)
    {
        List<SPLListItem> result = new List<SPLListItem>();
        foreach (SPLListItem item in items)
            if (item.GetVersionCount(level) > target)
                result.Add(item);
        return result;
    }

    /// <summary>
    /// Returns all items with a version count below a specified number.
    /// </summary>
    public static List<SPLListItem> GetItemsByVersionBelowTarget(
        this SPLListItemCollection items, SPFileLevel level, int target)
    {
        List<SPLListItem> result = new List<SPLListItem>();
        foreach (SPLListItem item in items)
            if (item.GetVersionCount(level) < target)
                result.Add(item);
        return result;
    }
}
}
```

Listing 3-9 shows how the extension methods are used. All of the methods are declared in the `ECM2007.Versioning` namespace. Using them requires the appropriate `using` statement and a `SPLListItemCollection` instance. The sample program shown in Listing 3-9 accepts the address of a SharePoint site or list and then calls the `ProcessList` routine for each matching list instance.

This program is included in the downloadable code for the book in the project `ECM2007.VersioningTestConsole`. To use it on your own system, you will need to change the command-line argument in the project properties so that it references a site on your local SharePoint farm.

Listing 3-9: VersioningTestConsole.cs

```
using System;
using System.Collections.Generic;
using System.Reflection;
using System.Linq;
using System.Text;
using Microsoft.SharePoint;
using ECM2007.Versioning;

namespace ECM2007.VersioningTestConsole
{
    class Program
```

Continued

Listing 3-9: VersioningTestConsole.cs (continued)

```
{
    /// <summary>
    /// Returns the program usage banner.
    /// </summary>
    static string Usage
    {
        get
        {
            string name = Assembly.GetExecutingAssembly().GetName().Name;
            Console.WriteLine("Usage: {0} -url <url>", name);
            Console.WriteLine("");
            Console.WriteLine("-url\t\tthe url of a SharePoint site or list");
            Console.WriteLine("");
            return "Invalid number of arguments";
        }
    }

    /// <summary>
    /// Extracts arguments from the command line.
    /// </summary>
    static bool GetArgs(string[] args, ref string url)
    {
        url = null;
        StringComparison comparisonType =
            StringComparison.InvariantCultureIgnoreCase;

        for (int i = 0; i < args.Length - 1; i++)
        {
            if (args[i].Equals("-url", comparisonType))
                url = args[i + 1];
        }

        return !string.IsNullOrEmpty(url);
    }

    /// <summary>
    /// Entry Point
    /// </summary>
    /// <param name="args"></param>
    static void Main(string[] args)
    {
        string url = "http://localhost:108";

        // Extract arguments and abort if invalid.
        if (!GetArgs(args, ref url))
            throw new System.Exception(Usage);

        // Open the SharePoint site collection...
        using (SPSite siteCollection = new SPSite(url))
        {
            Console.WriteLine("Opening site: {0} ", siteCollection.Url);

            // Get the root web...
```

Chapter 3: SharePoint Tools for Managing Records

```
using (SPWeb site = siteCollection.OpenWeb())
{
    Console.WriteLine("Opening web: {0}", site.Title);

    // Get the target list...
    SPList targetList = site.GetList(url);
    if (targetList != null)
        ProcessList(targetList);
    else foreach (SPList list in site.Lists)
        ProcessList(list);
}

Console.WriteLine("");
Console.WriteLine("Press any key...");
Console.ReadKey();
}

/// <summary>
/// Processes an individual list.
/// </summary>
/// <param name="list"></param>
static void ProcessList(SPList list)
{
    Console.WriteLine("Processing list: {0}", list.Title);

    Console.WriteLine("Highest major version = {0}",
        list.Items.GetHighestVersion(SPFileLevel.Published));
    Console.WriteLine("Highest minor version = {0}",
        list.Items.GetHighestVersion(SPFileLevel.Draft));

    Console.WriteLine("Number of major versions = {0}",
        list.Items.GetVersionCount(SPFileLevel.Published));
    Console.WriteLine("Number of minor versions = {0}",
        list.Items.GetVersionCount(SPFileLevel.Draft));

    Console.WriteLine("Average major versions = {0}",
        list.Items.GetAverageVersionCount(SPFileLevel.Published));
    Console.WriteLine("Average minor versions = {0}",
        list.Items.GetAverageVersionCount(SPFileLevel.Draft));

    Console.WriteLine("Maximum major version size = {0}",
        list.Items.GetMaxSize(SPFileLevel.Published));
    Console.WriteLine("Maximum minor version size = {0}",
        list.Items.GetMaxSize(SPFileLevel.Draft));

    Console.WriteLine("Minimum major version size = {0}",
        list.Items.GetMinSize(SPFileLevel.Published));
    Console.WriteLine("Minimum minor version size = {0}",
        list.Items.GetMinSize(SPFileLevel.Draft));

    Console.WriteLine("Total size of all major versions = {0}",
        list.Items.GetTotalSize(SPFileLevel.Published));
}
```

Continued

Listing 3-9: VersioningTestConsole.cs (continued)

```
Console.WriteLine("Totals size of all minor versions = {0}",
    list.Items.GetTotalSize(SPFileLevel.Draft));

Console.WriteLine("Average size of major versions = {0}",
    list.Items.GetAverageSize(SPFileLevel.Published));
Console.WriteLine("Average size of minor versions = {0}",
    list.Items.GetAverageSize(SPFileLevel.Draft));

List<SPListItem> items =
    list.Items.GetItemsByVersionAboveAverage(SPFileLevel.Published);
Console.WriteLine(
    "{0} items containing greater than average major versions",
    items.Count);

items = list.Items.GetItemsByVersionBelowAverage
    (SPFileLevel.Published);
Console.WriteLine(
    "{0} items containing fewer than average major versions",
    items.Count);
    }
}
}
```

Figure 3-15 shows the collection of documents in a document library with different versions created.



Figure 3-15: A library of versioned documents.

Content Security

SharePoint permissions control everything that can be done to content within the SharePoint environment. In this section, we'll explore SharePoint permissions and permission levels and see how they can be applied from a content life-cycle perspective, which puts additional demands on the granularity of permission sets. First, we need a basic understanding of how SharePoint permissions work and the available API for manipulating them.

SharePoint Permissions

There are 33 individual permissions that can be attached to objects in SharePoint. These permissions are divided into the following three categories and define what actions can be taken for any given object:

- ❑ Personal Permissions (3)
- ❑ List Permissions (12)
- ❑ Site Permissions (18)

Permissions can be grouped into named collections called *permission levels* as a way to assign them to what are essentially roles within the SharePoint object model. Permission levels can be assigned to users, groups, sites, lists, or list items using an `SPRoleAssignment` object. SharePoint optimizes the allocation of permission masks by allowing them to be inherited. By default, inheritance is enabled for each item. The following code illustrates the steps involved in assigning unique permissions to an object:

```
public static class RoleAssignmentSample
{
    public static void AssignPermissionLevel(SPListItem item,
        string groupName, string roleDefinitionName)
    {
        // Get the group whose permissions are to be modified for the item.
        SPGroup group = item.Web.SiteGroups[groupName];

        // Create a new role assignment object for the group.
        SPSRoleAssignment roleAssignment =
            new SPSRoleAssignment(group as SPPrincipal);

        // Break permission inheritance for the item, since
        // it will now have its own permissions. Pass 'true'
        // to copy any existing role assignments.
        item.BreakRoleInheritance(true);

        // Bind the role assignment to the new permission level
        roleAssignment.RoleDefinitionBindings.Add(
            item.Web.RoleDefinitions[roleDefinitionName]);

        // apply the new role assignment to the item
        item.RoleAssignments.Add(roleAssignment);
    }
}
```

Chapter 3: SharePoint Tools for Managing Records

As you can see in the code, in order to attach unique permissions to an item, permission inheritance must be *broken*. This is done by an explicit call that tells SharePoint to attach the permissions directly to the object instead of determining the permissions based on the object's parent.

There are nine default permission levels as shown in the following table, and there are four default SharePoint groups: Owner, Member, Visitor, and Viewer.

Whenever a site collection is created through the SharePoint user interface, a special method named `SPWeb.CreateDefaultAssociatedGroups` is called on the root web of the site collection. This method takes care of creating the default groups for the web site. Note that this method is not called when creating a site collection using the STSADM command-line utility.

Level	Description
Full Control	This permission level has all permissions enabled and is assigned to the Owners group by default.
Design	This allows users to create lists and document libraries, edit pages, and apply thematic elements like borders and stylesheets using tools like SharePoint Designer 2007.
Contribute	This allows items to be added, edited, and deleted.
Approve	This enables users to edit and approve pages, documents, and list items.
Manage Hierarchy	This enables users to create sites and to edit pages, documents, and list items.
Read	This provides read-only access to the site, which allows users or groups to view items and pages, open items, and view documents.
Restricted Read	This permission allows users to view the content of pages and documents, but not prior versions. This is intended to allow a wide audience of users to see only the current version of special content like company policies.
View Only	This permission is more restrictive than Read permission and controls whether the user can open the file in a client application if a server-side viewer is available. If so, then the user cannot open the file in the native application.
Limited Access	The purpose of this permission level is to grant the minimal set of permissions possible for a specific object without granting access to everything in the site. It is therefore not possible to assign it directly to users or groups. Instead, it is assigned automatically by SharePoint when you grant access to an object that inherits permissions from another object that the user or group does not have access to.

Next, we will examine more closely the three categories of permissions.

Personal Permissions

The three personal permissions are focused on the ability to keep private information and the ability to show private web parts on a Web Part page. These permissions control each user's ability to interact with the section of the content database that is reserved for personal data. As an example, a web part developer can mark individual properties as *personalizable*, which causes the web part manager object on the page to reserve space in the database for a copy of each property value to be stored uniquely for each user. Thus, a web part that displays a personal zip code would display a different zip code for every user visiting the page. Removing the `SPBasePermissions.UpdatePersonalWebParts` would disallow this capability.

List Permissions and Groups

SharePoint groups are defined in the site collection and can be assigned to multiple permission levels. Whenever a new site collection is created, then the default set of groups is created automatically. Figuring out which permission levels are associated with each group can be confusing at first. Figure 3-16 shows how the 12 list permissions are mapped to the default permission levels and groups.

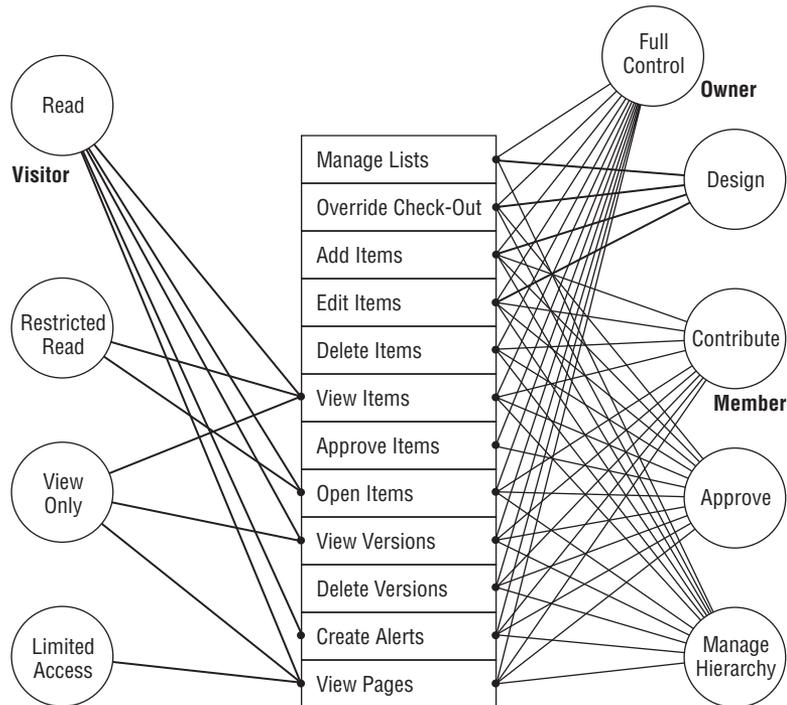


Figure 3-16: List permissions and groups.

Site Permissions and Groups

There is a similar mapping for the 18 site permissions, as shown in Figure 3-17.

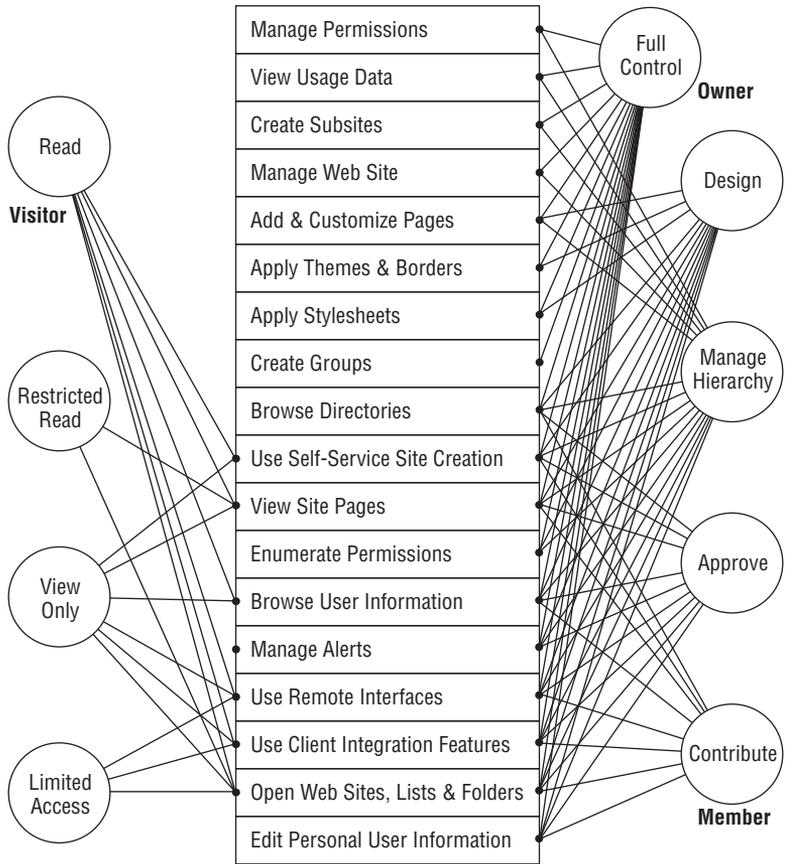


Figure 3-17: Site permissions and groups.

Permission Dependencies

There's one more thing to think about related to permissions. It is sometimes necessary to create a custom role definition for a particular user or group. When doing so, it is important to understand permission dependencies, because that will determine the effective permissions being granted. With the exception of the Open permission, which grants the ability to open web sites, lists, and folders to access their contents, every SharePoint permission depends on one or more of the other permissions. This means, for example, that if you grant the "Manage Alerts" site permission, you are also automatically granting the "Create Item Alerts," "View List Items," and "Open List Items" list permissions as well as the "View Site Pages" and "Open" site permissions. Figure 3-18 illustrates the interdependencies between each of the 33 permissions.

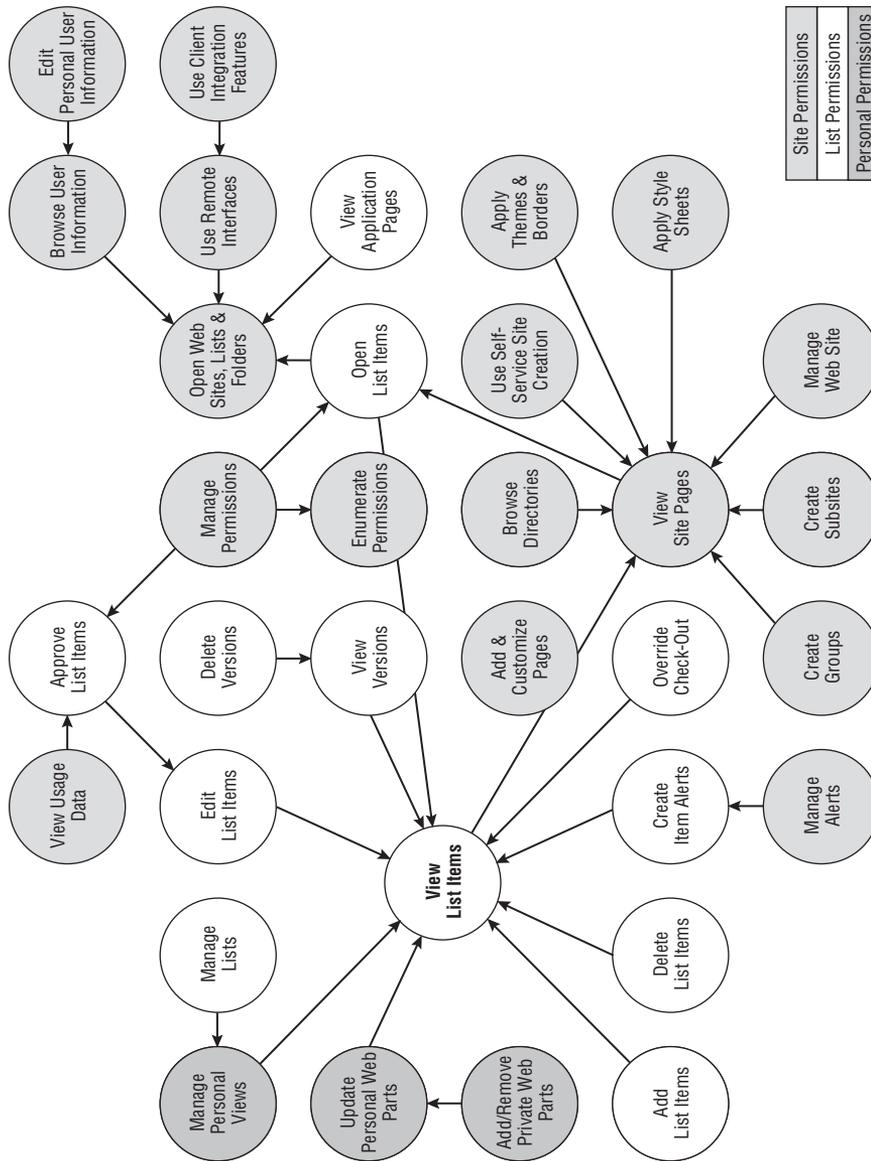


Figure 3-18: Permission dependencies.

Strategies for Controlling Access to Content

There are several strategies available for dealing with access control in the context of ECM. Ideally, we'd like an end-to-end life-cycle strategy that is governed primarily by the roles involved and for what purposes a given content element will be used. Unfortunately, there are technology limitations that force design concessions, and the best we can hope for is a compromise.

Strategy	Description
Role-Driven	Identify roles and create groups. Create and assign fixed permission levels. Move content between libraries to attach permissions to list items.
Content-Driven	Associate roles with content types. Assign item-level permissions using an event receiver or workflow activity.
Life-Cycle-Driven	Identify permission levels at each life-cycle stage for each role. Create a separate document library for each life-cycle stage. Attach a <i>permission manifest</i> to each document library. Assign item-level permissions in a content type event receiver.

In order to apply a role-driven strategy whereby content is moving through different document libraries at different stages in its life cycle, there would have to be a mechanism for attaching custom permission manifests to the document library so they could be retrieved programmatically. A content-driven strategy would go one step further and assign permissions to each item by reinterpreting the manifest at each stage. A life-cycle strategy would likely require a state-machine workflow to determine which document libraries were needed, and when and what permissions were required for the items currently residing in those document libraries.

There is a hidden limit of about 1,800 unique permissions per securable object, which is any object that implements the `Microsoft.SharePoint.ISecurableObject` interface. This interface exposes methods for determining the roles that a given object is assigned to, as well as the permissions of the object. Permissions are assigned to securable objects using Access Control Lists (ACLs) that are managed by the operating system. Since there is a finite number of ACLs available in the system, it means there is a limit to the number of unique permissions.

Role-Driven Strategy

If we're pursuing a role-driven strategy using artifacts from a role/activity modeling exercise, we can easily identify the roles, define the groups, and determine the fixed permission levels. In order to assign the permission levels correctly, our strategy would be to move content between libraries with different fixed permission levels as the content moves through its life cycle. This might be a bit non-intuitive for certain users who would then have to go to a different library in order to interact

with the content, but it would solve the problem of being able to easily assign different permissions at different stages of the life cycle, perhaps by using a workflow. However, this is a compromise. It is a work-around for the limitation in the number of unique permissions that can be assigned to a given document or list item. Typically, the limit is not reached because in most scenarios, you don't need to use unique permissions. Instead, you are using groups or permissions associated with a library or permissions that are inherited from the parent object.

Content-Driven Strategy

Ideally when using a life-cycle model, you want to assign unique permissions so that the permissions are driven at least in part by the needs of the content, and not by the current location in which that content is stored. In this way, you could pursue a purely content-driven strategy for assigning permissions. A content-driven strategy would simply associate the roles with the content type, break permission inheritance, and then assign item-level permissions in an event receiver associated with the content type at each stage of the content life cycle. This might work for scenarios involving less than 1,800 users. But once you get to larger deployments, it again breaks because you hit the operating system limit on ACLs that can be attached to the objects required to implement the strategy.

Life-Cycle Access Control Strategy

The goal here is to move a document through different stages and change its required permissions at each stage. Ideally, you want to handle this at the content-type level, but you don't necessarily want to change the content type of the document from one stage to the next. One enhancement to this approach might be instead to assign a special property to each document library that contains a permission manifest and then read the manifest in the content-type event receiver. Here again, you may run into the hard limits imposed by the operating system because SharePoint will begin to throw exceptions as it runs out of ACLs to assign.

Although these limits throw a monkey wrench into the life-cycle access control strategy, it may still be possible to optimize the content life cycle by looking for ways to share permission sets and then applying the unique permissions to a special list or document library that is used just to store those items. Although it adds a layer of complexity, it may preserve the life-cycle strategy until the ACL problem is fixed or a better solution is found.

Figure 3-19 illustrates the life-cycle strategy using a "Statement of Work" document that must support multiple roles. During the creation phase, the sales team needs special permissions related to creating the document and any associated resources. Once it moves into the review-and-edit phase, the sales permissions are modified to support review. At the same time, the engineering team needs its own review-and-edit permissions. For approval and publication, the management team is given access, and the permissions for the other teams are removed. Finally, the document is archived by the original sales team, which takes final control of its disposition.

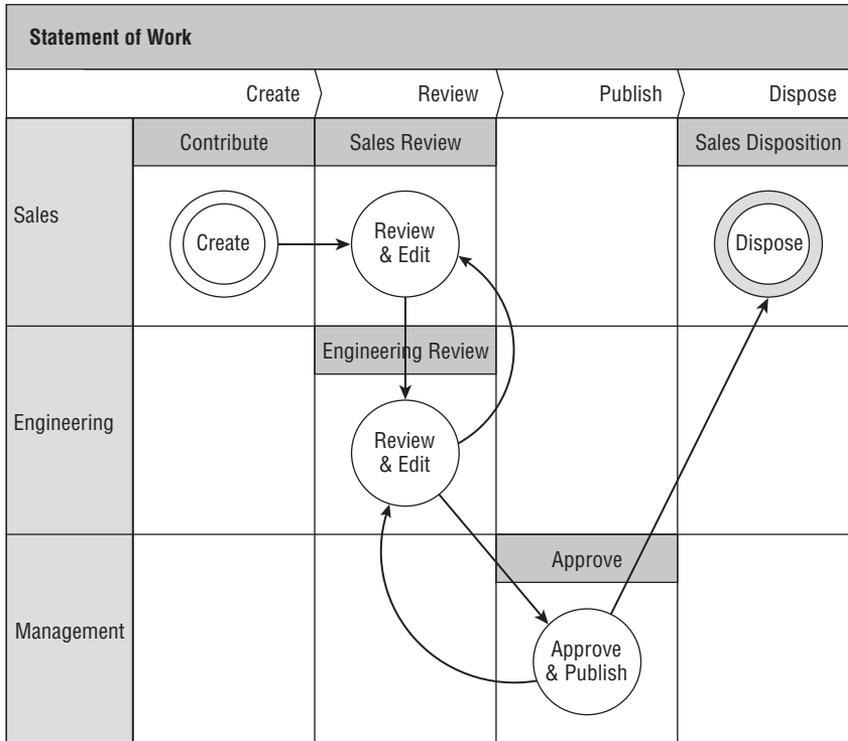


Figure 3-19: Role-based permission cycles.

Ultimately, we can envision a methodology based on an executable schema that identifies every content element that may be required for each cell of the matrix in Figure 3-19 and then provides prescriptive guidance or even rules that govern the transformation of such content as it moves from one cell to another.

Information Rights Management

To round out our discussion of content security, some attention must be given to the problem of controlling what happens to content after it leaves the server. It's one thing to control content access while it is stored electronically, but what about when it moves into client applications?

Information Rights Management (IRM) and Rights Management Services (RMS) are fully integrated into the SharePoint platform. Although digital rights management affects all types of content, including official records, a full treatment of IRM/RMS is beyond the scope of this book. For a very detailed overview of how to configure and use RMS/IRM in the SharePoint Environment, I highly recommend Jason Medero's chapter, "Using Information Rights Management," in Real World SharePoint 2007: Indispensable Experiences from 16 MOSS and WSS MVPs (ed. Scot Hillier; Wrox Press, 2007).

In a Microsoft-only world, where everyone uses Microsoft Office products to create and edit their documents, it would be a simple matter of building in the appropriate hooks that would prevent users from violating any content management policies that may govern a particular document instance. Fortunately (or unfortunately, depending on your point of view), we don't live in such a world. Consequently, a more open set of tools is needed to manage the security of information once it leaves the confines of the central repository.

Microsoft has provided an extensible platform for managing digital assets in a distributed environment. It's called *Rights Management Services* (RMS). It runs on the server, providing a central point from which client applications running on the same network can request a license (EUL) for any digital asset. RMS can then correlate that license at a later time with other applications that need to determine what rights to grant to users accessing the protected resource. The license file that is retrieved from the RMS server is then used to encrypt and decrypt the content so that it can be safely transmitted across the network. Once encrypted, the contents cannot be opened without contacting the RMS server and supplying the correct credentials. This means that the RMS server acts as a central certifying authority that controls what happens to the content wherever it may happen to end up.

While RMS provides the basic ability to grant permissions and validate license files, an additional layer is needed to complete the Information Rights Management architecture within the SharePoint environment. The ultimate goal is to enable the construction of protected document libraries that allow companies to set up special repositories for sensitive documents and to control what happens to those documents when they are checked out of the library. Whether using an Office client application or an application from some other vendor, it should not be possible for unauthorized users to view or print protected documents. It should also be possible for administrators to modify the permissions associated with a given user without touching the document itself, or the document library in which it is stored.

The basic principles of IRM in the SharePoint environment are as follows:

- ❑ Files are not stored in an encrypted form, but are encrypted “on demand” when downloaded by using a public/private key pair obtained from the RMS server.
- ❑ SharePoint limits the set of users and/or applications that can encrypt or decrypt files.
- ❑ The RMS server controls the rights of users to manipulate documents.
- ❑ SharePoint attaches a textual *policy statement* to documents so that end-users can be informed about the policies that control their access to the protected content.

These basic principles ensure that information rights can be managed effectively in a distributed environment. The fact that they are also referenced by a written policy statement attached to the document further reinforces the user's awareness of their responsibilities related to the content and also eliminates the ignorance defense if the policy is broken.

The Records Management Object Model and API

Before we can build custom records management components to automate the management of official records, we need to understand the tools provided as part of the SharePoint API for dealing with records and related data structures. The components we need are located in the `Microsoft.Office.RecordsManagement` namespace, which is implemented in the `Microsoft.Office.Policy` assembly, located in the ISAPI folder of the 12 Hive. Figure 3-20 shows the primary components provided.

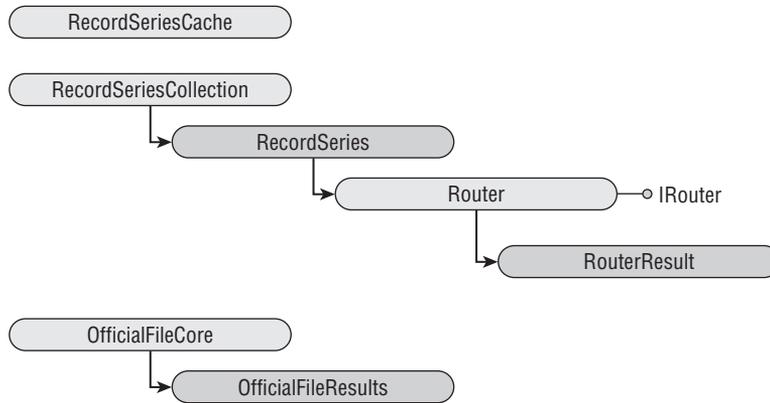


Figure 3-20: Records Management API core components.

The core API can be accessed either directly, by calling the public methods of the `OfficialFileCore` class, or indirectly using the Official File Web Service. The obvious advantage of using the Web Service is that it enables both manual and automated file submission from client applications that may or may not be running on a Windows platform. The not-so-obvious downside is that there are several assumptions made by the server-side API that require special attention in the client. In particular, as we saw earlier in the chapter, the Records Center implementation relies on the submission of additional information along with each incoming record so that it can properly handle things like routing, auditing, and property promotion. Although it is possible to submit records without this additional information, doing so may lead to inconsistent results. This is especially true if the same repository is set up to receive manual submissions from SharePoint document libraries as well as automated submissions directly from client applications.

Figure 3-21 shows the general processing sequence that occurs when a file is submitted. In the simple case, where we are not supplying additional metadata, we can simply call the `OfficialFileCore.SubmitFile` method. This is demonstrated in Listing 3-10 in the following section, “The Official File Web Service.”

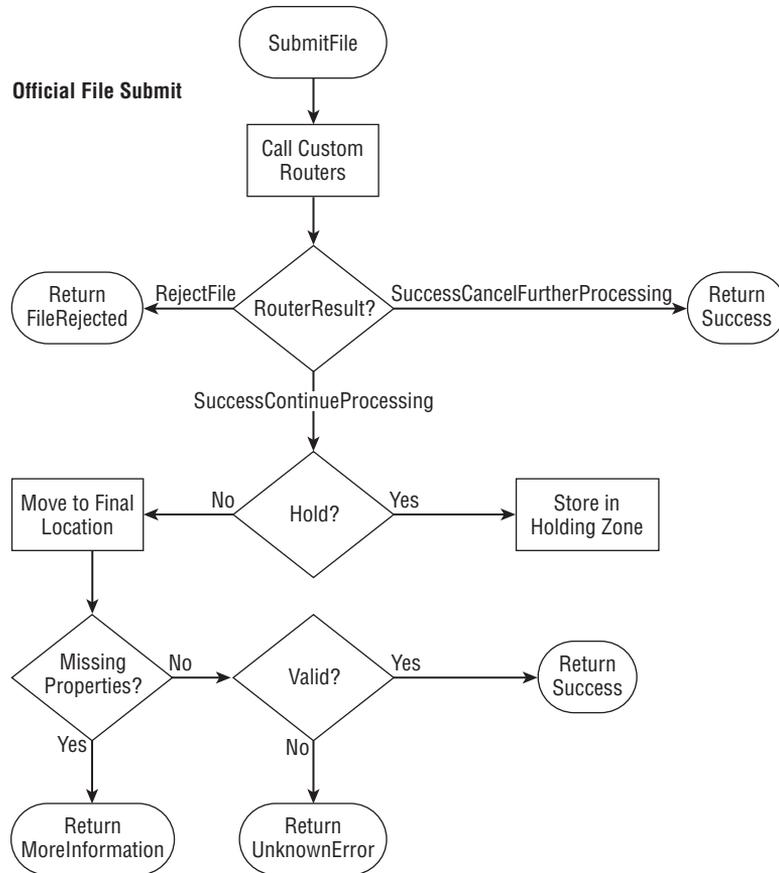


Figure 3-21: Official File submission sequence.

The Official File Web Service

The Official File Web Service is another tool we can use to work with official records. It is bundled automatically with any Records Center site. It provides a standardized mechanism for managing a Records Center remotely, and it is the same mechanism that is used by SharePoint itself to submit files to the repository. You can see the Web Service definition by entering a URL based on the following pattern into a web browser: `http://<site url>/_vti_bin/officialfile.asmx`.

Figure 3-22 shows the resulting page displayed in the browser.

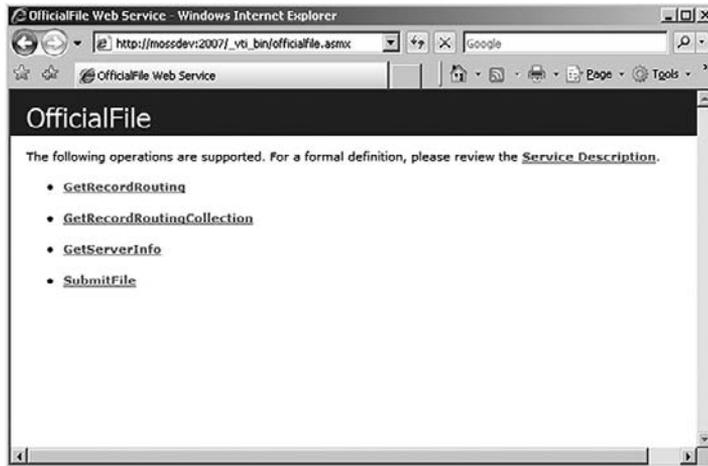


Figure 3-22: Official File Web Service methods.

Using the methods of the Official File Web Service, you can implement applications that run either on the client or the server to submit documents to a repository or to determine what record types it currently supports. Connecting to the Web Service requires a web reference, which you can add to your project by specifying the appropriate server and portal names as shown in Figure 3-23.

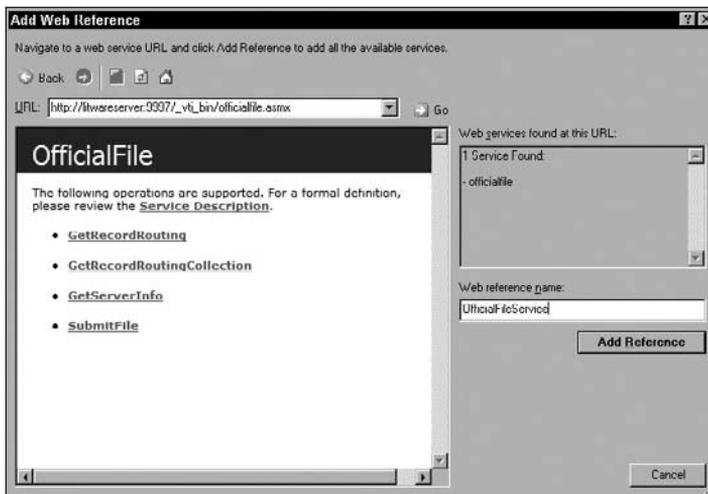


Figure 3-23: Adding an Official File Web Service reference.

When you add a web reference in this way, the Web Service address is hardcoded to the URL you enter into the dialog, but you can change it later before invoking the web methods to target a different repository, using code like the following:

```
OfficialFileService.RecordsRepository repository =
    new OfficialFileService.RecordsRepository();
repository.Credentials = System.Net.CredentialCache.DefaultCredentials;
repository.Url=GetRepositoryUrlFromUser(); // user-defined function to get the url
```

The `GetRecordRoutingCollection` web method returns an XML string containing the list of available record routing types in the Records Center. The easiest way to parse the string is to use an `XPathNodeIterator`. Typically, you will use code like the following to search through the list of available routing types for a match with the type of the document you are sending to the repository:

```
string rTypes = repository.GetRecordRoutingCollection();
string expr = "/RecordRoutingCollection/RecordRouting/Name";

XPathDocument doc = new XPathDocument(new StringReader(rTypes));
XPathNavigator nav = doc.CreateNavigator();
XPathNodeIterator iter = nav.Select(expr);

while (iter.MoveNext())
{
    if (nameToMatch == iter.Current.Value)
    {
        /* prepare the file for submission */
    }
}
```

To submit a file to the repository using the Web Service, you must perform the following steps:

1. Obtain the array of bytes that represent the file contents.
2. Create an array of `RecordsRepositoryProperty` objects, and populate them with the name, type, and value of properties you want to be stored along with the file.
3. Send the data and the properties to the repository, and parse the result string to determine if the submission was successful.

The code in Listing 3-10 illustrates the process.

Listing 3-10: Submitting a file using the Web Service

```
/// <summary>
/// Submits the file using the specified routing type.
/// </summary>
public string Submit(OfficialFileService.RecordsRepository repository,
                    RoutingType routingType, string filePath)
{
```

Continued

Listing 3-10: Submitting a file using the Web Service *(continued)*

```
string result = "Not submitted";
try
{
    // Retrieve the file data.
    byte[] data = File.ReadAllBytes(filePath);

    // Create a property array.
    OfficialFileService.RecordsRepositoryProperty[] props
        = new OfficialFileService.RecordsRepositoryProperty[1];

    // Specify some property data.
    props[0] = new OfficialFileWebServiceClient.OfficialFileService
        .RecordsRepositoryProperty();
    props[0].Name = "OriginalPath";
    props[0].Type = "String";
    props[0].Value = filePath;

    // Parse the xml result.
    string result = repository.SubmitFile( data, props,
        routingType.Name, filePath,
        WindowsIdentity.GetCurrent().Name);

    XPathNavigator nav =
        new XPathDocument(
            new StringReader(
                string.Format("<OFS>{0}</OFS>", result)
            )).CreateNavigator();

    XPathNodeIterator iter = nav.Select("/OFS/ResultCode");
    result = iter.MoveNext() ?
        iter.Current.Value : "Error retrieving result code.";
}
catch (Exception x)
{
    result = x.Message;
}
return result;
}
```

Summary

This chapter introduced the key components provided by SharePoint for building records management solutions. We looked at the tools and components that are built into Windows SharePoint Services for building collaborative applications as well as extended functionality provided by MOSS.

Document libraries are a specialization of `SPList` that add document templates and document property promotion. Document libraries and lists support the same standard file I/O operations, providing

a consistent API based on folders, subfolders, and files. Document templates can be bound to server-side (.ASPX) or client-side (.DOTX, etc.) files, enabling a variety of methods for ensuring that documents are created properly.

Content types are one of the most important components in the SharePoint platform for creating content management solutions. The three primary components of content types are metadata (fields), behavior (event receivers), and an extensible payload (XML documents). Together, they provide the foundation for ECM on the SharePoint platform. Although content types support simple metadata inheritance, they do not support behavioral inheritance, limiting their usefulness for building deep taxonomies. A single document can be associated with only one content type at a time, which can affect the extensibility of life-cycle-based solutions. Nevertheless, metadata drives all stages of the content life cycle, and content types provide a way to control metadata throughout the content life cycle.

Versioning and check-in/check-out are key features that enable rollback, selective viewing, deletion, and item-level tracking. Windows SharePoint Services follows a specific set of rules that determine when a new version is created or an existing version is updated. Versioning can significantly increase the size of the content database and can also cause problems for users who are not accustomed to controlled collaboration. The Windows SharePoint Services Versioning API enables many different kinds of reporting and version management tools and web parts. Versioning complements check-in and check-out by managing rollback for client-side editing.

SharePoint permissions and permission levels provide role-based security for sites, lists, document libraries, folders, and list items. The role-based security model can be manipulated through code attached to lists and to content types. Using event receivers, the role-based security model can be extended to support content-driven and life-cycle-driven content security strategies. The static nature of SharePoint permission levels means that life-cycle strategies must rely on external code to adjust user/group permissions at each stage for a given item instance.

The MOSS Records Management API provides the high-level components and methods needed to manipulate documents and list items as official records. The key component is the `OfficialFileCore` implementation, which includes the functionality needed to submit records to the repository. Working in conjunction with the core API, the Official File Web Service enables access to the same functionality through web methods that can be called from either server or client applications.

4

The MOSS 2007 Records Center

Although it is possible to use portions of the MOSS Records Management functionality, such as information policy, separately from a Records Center site, most of the records management functionality provided by MOSS is tied in some way to the Records Center site definition. As with most of the extensions that MOSS introduces into the core Windows SharePoint Services environment, the Records Center site definition is activated as a feature. This provides a great window into the underlying functionality. Analyzing the built-in Records Management feature provides valuable insights into the way the records management components are organized and how they are intended to work together.

The Records Management Feature

Within the 12/TEMPLATE/FEATURES folder, there is a single feature named *Records Management* — shown in Listing 4-1 — that is responsible for setting up the custom actions and information policies needed for records management.

Listing 4-1: Records Management feature

```
<Feature Id="6d127338-5e7d-4391-8f62-a11e43b1d404"  
  Title="$Resources:RecordsManagementFeatureTitle"  
  Description="$Resources:RecordsManagementFeatureDescription"  
  DefaultResourceFile="dlccore"  
  Version="12.0.0.0"  
  Hidden="TRUE"  
  Scope="Farm"
```

Continued

Listing 4-1: Records Management feature (continued)

```
ReceiverAssembly="Microsoft.Office.Policy, Version=12.0.0.0, Culture=neutral,
  PublicKeyToken=71e9bce111e9429c"
ReceiverClass="Microsoft.Office.RecordsManagement.Internal.
  RecordsManagementFeatureReceiver"
xmlns="http://schemas.microsoft.com/sharepoint/"
<ElementManifests>
  <ElementManifest Location="Links.xml" />
</ElementManifests>
</Feature>
```

While not specific to the Records Center site definition, this feature must be activated in order for much of the Records Center functionality to work.

Note that the Records Management feature is a hidden, farm-scoped feature bound to an implementation of `SPFeatureReceiver` that overrides the `FeatureActivated` method to provision not only the built-in policy features (e.g., the Expiration, Audit, Barcode, and Label policy features), but also the entire Information Policy subsystem, including the timer jobs used for policy updates, expiration, and holds.

For more information about these timer jobs and how they operate, see Chapter 8.

The feature also installs the five custom actions shown in Listing 4-2 that enable administrators to access the Information Policy configuration pages and to configure audit settings for the site collection. These links show up on the Site Settings menu and the List Settings, Content Type Settings, and Content Type Template Settings pages. In addition, two custom actions are added to the Operations page of Central Administration to support the configuration of Information Policy reports and Information Policy security.

Listing 4-2: Records Management custom actions

```
<Elements
  xmlns="http://schemas.microsoft.com/sharepoint/"
  <!-- _Layouts/Settings -->
  <CustomAction
    Id="PolicyTemplate"
    GroupId="SiteCollectionAdmin"
    Location="Microsoft.SharePoint.SiteSettings"
    RequireSiteAdministrator="true"
    Sequence="90"
    Title="$Resources:dlccore, PolicyLinks_SiteSettings_PolicyTemplates;">
    <UrlAction
      Url="_layouts/Policylist.aspx" />
  </CustomAction>
  <CustomAction
    Id="ListPolicySettings"
    GroupId="Permissions"
    Location="Microsoft.SharePoint.ListEdit"
    Rights="ManageLists"
```

```

        Sequence="100"
        Title="$Resources:dlccore, PolicyLinks_ContentTypeSettings_PolicySettings;">
        <UrlAction Url="_layouts/policy.aspx?List={ListId}" />
    </CustomAction>
</CustomAction
    Id="ContentTypePolicySettings"
    GroupId="General"
    Location="Microsoft.SharePoint.ContentTypeSettings"
    Sequence="100"
    Title="$Resources:dlccore, PolicyLinks_ContentTypeSettings_PolicySettings;">
    <UrlAction Url="_layouts/policy.aspx"/>
</CustomAction>
</CustomAction
    Id="ContentTypeTemplatePolicySettings"
    GroupId="General"
    Location="Microsoft.SharePoint.ContentTypeTemplateSettings"
    Sequence="100"
    Title="$Resources:dlccore, PolicyLinks_ContentTypeSettings_PolicySettings;">
    <UrlAction Url="_layouts/policy.aspx"/>
</CustomAction>
</CustomAction
    Id="AuditSettings"
    GroupId="SiteCollectionAdmin"
    Location="Microsoft.SharePoint.SiteSettings"
    RequireSiteAdministrator="true"
    Sequence="70"
    Title="$Resources:Reporting_AuditSettingsFeatureTitle;">
    <UrlAction
        Url="_layouts/AuditSettings.aspx" />
</CustomAction>
<!-- _Admin/Operations -->
</CustomAction
    Id="PolicyRptConfiguration"
    GroupId="LoggingAndReporting"
    Location="Microsoft.SharePoint.Administration.Operations"
    Sequence="40"
    Title="$Resources:dlccore, PolicyLinks_Operations_PolicyRptConfiguration;">
    <UrlAction
        Url="/_admin/PolicyRptConfig.aspx" />
</CustomAction>
</CustomAction
    Id="PolicyFeaturesConfig"
    GroupId="Security"
    Location="Microsoft.SharePoint.Administration.Operations"
    Sequence="50"
    Title="$Resources:dlccore, PolicyLinks_Operations_PolicyFeaturesConfig;">
    <UrlAction
        Url="/_admin/Policyfeatures.aspx" />
</CustomAction>
</Elements>

```

The Records Center Site Definition

SharePoint Server 2007 includes a special site definition located at 12/TEMPLATE/SiteTemplates/offfile that implements the default Records Repository. Figure 4-1 shows the basic structure of this site definition, which declares the core components of the MOSS Records Management infrastructure.

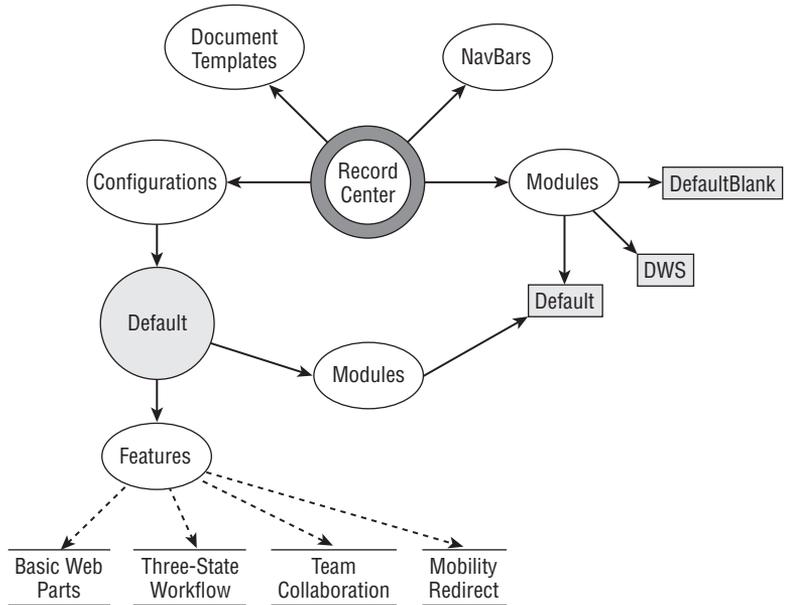


Figure 4-1: Records Center site definition.

By itself, the site definition looks just like a normal team site. Although you might expect to see special feature dependencies within the site definition that create the lists and other components that appear whenever a site based on this definition is created, they are not there. Instead, the site definition only includes the standard set of WSS features that enable web parts, team collaboration, mobile devices, and the three-state workflow. In fact, there is only one configuration that references a single default module.

So how are the components created? The answer is in the WEBTEMPOFFFILE.XML file that specifies how the site definition is presented on the Create Site page in the SharePoint user interface. A portion of the XML code for this template is shown below:

```
<Templates xmlns:ows="Microsoft SharePoint">
  <Template Name="OFFFILE" ID="14483">

    <Configuration
      ID="1"
      Title="Records Center"
      Hidden="FALSE"
      ImageUrl="/_layouts/images/officialfile.jpg"
      Description="This template creates a site designed for records management.
        Records managers can configure the routing table to direct incoming
```

```
files to specific locations. The site prevents records from being
modified after they are added to the repository."
ProvisionAssembly="Microsoft.Office.Policy, Version=12.0.0.0,
Culture=neutral, PublicKeyToken=71e9bce11e9429c"
ProvisionClass="Microsoft.Office.RecordsManagement.RecordsRepository.Setup"
ProvisionData=""
DisplayCategory="Enterprise"
VisibilityFeatureDependency="9E56487C-795A-4077-9425-54A1ECB84282"
>
</Configuration>

</Template>
</Templates>
```

Notice the `ProvisionAssembly` and `ProvisionClass` attributes in the `Configuration` node with `ID="1"`. The `Setup` class declared in the `Microsoft.Office.RecordsManagement.RecordsRepository` namespace supplies an implementation of `SPWebProvisioningProvider` that SharePoint calls whenever an instance of the site is created.

For more information about using the `SPWebProvisioningProvider` class to customize the creation of SharePoint sites, see David Mann's article, "Customizing SharePoint Site Creation" (www.sharepointbriefing.com/spcode/article.php/3792601/Customizing-SharePoint-Site-Creation.htm).

The `Setup` class performs the following operations:

- Applies the default configuration of the OFFFILE site definition to the new web site.
- Sets up the Records Repository Users Group.
- Sets up the global `RecordSeriesCollection`.
- Sets up the holding zone for records that are missing required metadata.
- Sets up the holds processing layer.
- Prepares the workflow provisioning layer.
- Sets up the default record series for unclassified records.
- Sets up a list to receive submitted records.
- Prepares the default expiration actions.
- Sets up the default links.
- Sets up the default validators and web parts.

The following sections describe how these components are configured.

Records Center Components

The Records Center site definition installs several auxiliary components that come into play at various stages in the processing of records. The following sections describe each component in detail.

The Records Repository Users Group

The Records Repository Users Group identifies users who can submit records to the Records Center using Web Services. If this group does not exist in the web site, it is created, and its group identifier is stored in the web site properties array under the key `_dlc_RepositoryUsersGroup`. If the key already exists in the web site properties, it means that a previously provisioned site group may have been removed, in which case the group is also re-created, and the new group identifier is stored in place of the old one. This logic ensures that the group exists after the provisioning process is complete.

The Record Routing Table

The Record Routing table is a generic list with the columns shown in the following table:

Column	Type	Description
Title	Single line of text	The record series name
Description	Multiple lines of text	Description of the series
Location	Single line of text	The name of the document library in which matching records will be stored
Aliases	Multiple lines of text	The names of additional content types that match this record series
Default	Yes/No	Whether this record series should be used for records that do not match any record series

After creating the Record Routing table, the provisioning code adds an additional hidden field that specifies the custom router to use for incoming records. Then the default view is created containing the visible columns and a ListView web part is added to the *left* zone. The Records Repository User Group is granted Reader privileges on the list, and an event receiver is added for the `ItemAdding`, `ItemUpdating`, and `ItemDeleting` events. Finally, the list is set up to prevent the removal of items by setting the `AllowDeletion` flag to `false`, and special properties are added to the root folder to support holds processing.

The Holding Zone

When records are submitted to the repository that are missing required metadata, they are placed into a special document library that is referred to as the *holding zone* in the SharePoint SDK. The title of this library is "Records Pending Submission." The holding zone columns that are added to this library are listed in the following table:

Column	Type	Description
Title	Single line of text	The record title
Source Url	Single line of text	The source of the record

Column	Type	Description
Record Routing	Single line of text	The record routing type to be used for processing the record
User Name	Single line of text	The name of the user who submitted the record

In addition to the column definitions, two folders are added to the list to provide a place to store the record properties and audit history. This list is protected from deletion, and the Records Repository Users Group is granted *contributor* permissions for the list.

The Holds List

During the provisioning process, the Holds Processing layer is initialized, and the Holds List is created. The Holds List is a generic list that contains individual items called *Holds* that are used to suspend the normal expiration processing for submitted records. The Holds List columns that are added to this list are shown in the following table:

Column	Type	Description
Title	Single line of text	The title of the hold
Description	Multiple lines of text	Text describing the purpose of the hold
Managed By	Person or Group	Identifies the person or group responsible for managing the hold

There are some additional hidden columns that are added to the Holds List to manage the hold count, hold status, and last reporting date. For more information about how the Holds Processing layer is initialized within the Records Center, see Chapter 11.

Workflow Provisioning

The workflow provisioning layer ensures that the Workflow Task and History Lists are created. The Task List that is used is the standard Task List that appears in the Quick Launch navigation bar. The History List is created as a standard workflow history list, which does not appear in the content area but does appear on the association page for the Disposition Approval workflow. This workflow manages document expiration and retention for records in the repository.

Default Record Series

Records that have not been assigned to a record series are routed to the Default Record Series, which is set up by default to be associated with the Unclassified Records document library. The provisioning code creates this library and clears the `AllowDeletion` flag so that it cannot be removed. An *Unclassified Records* entry is also added to the master `RecordSeriesCollection` associated with the Web and shows up as an item in the Record Routing List.

Default Expiration Actions

There is only one default expiration action, which is to remove a record from the repository when the record expires. This functionality is provided by a custom policy resource that is linked to the Expiration policy by the provisioning code.

Records Center File Processing

The main event for the Records Center is the file submission process. Whether sent via the Official File Web Service or the Records Management API, the `OfficialFileCore.SubmitFile` method is ultimately called to process incoming records. The following sections describe the core routing mechanism and explain how the other components contribute to the overall processing sequence.

The Core Record Routing Mechanism

Incoming records are processed via the `SubmitFile` method, which accepts the parameters listed in the following table:

Parameter	Type	Description
<code>Web</code>	<code>SPWeb</code>	The web site in which to store the document
<code>Bytes</code>	<code>byte[]</code>	The content of the document being submitted
<code>Properties</code>	<code>RecordsRepositoryProperty[]</code>	The collection of properties for the document being submitted
<code>recordSeriesName</code>	<code>String</code>	The name of the record routing type for the document being submitted
<code>sourceUrl</code>	<code>String</code>	The source URL of the document being submitted
<code>userName</code>	<code>String</code>	The name of the user who submitted the document
<code>sentViaSMTP</code>	<code>bool</code>	True if the file is being sent through SMTP
<code>resultDetails</code>	<code>out String</code>	A string that will contain the results of the call; for example, the reason why the document was rejected

Figure 4-2 shows the basic processing sequence that is implemented by this routine. The first thing that happens is a check to see if the current user is a member of the Records Center Users Group. If not, then the call returns with a result of `OfficialFileResult.InvalidArgument`. Next, a check is done to ensure that the record series specified corresponds to a valid routing type and that the file being submitted has actual contents. Then if a custom router is associated with the routing type, it is loaded, and the record is processed through the router.

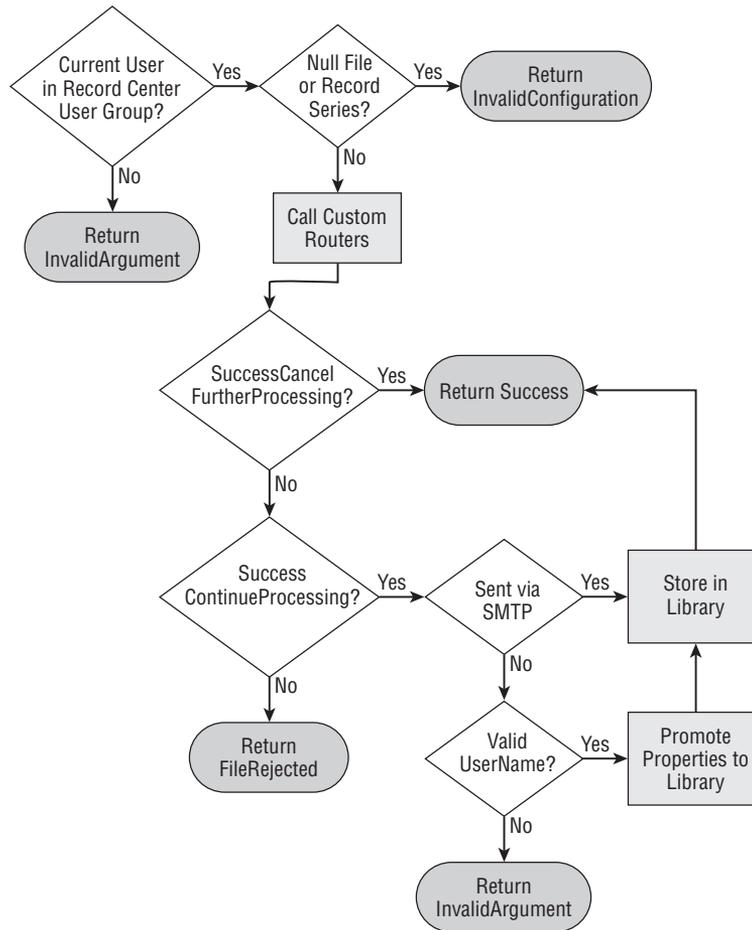


Figure 4-2: Processing sequence for submitted records.

Depending on the result returned from the custom router, if any, additional processing is either abandoned (in the case of `RouterResult.SuccessCancelFurtherProcessing`) or continued. If continued, then the Boolean flag is checked to see if the file was submitted via electronic mail. If so, then no attempt is made to promote the document properties into the destination library because special handling is needed for e-mail records. Instead, the document is simply placed into the destination library, which will be the *Records Pending Submission* library for e-mail records. Otherwise, a final check is made to determine if the username that was supplied to the routine is valid.

The submitted username may not be the same as the authenticated user under whose account the SubmitFile method is being called. This can happen if the file is submitted from outside the SharePoint farm, as in the case of a client application or external web application, or from a site collection in a web application separate from the Records Center that uses a different authentication method.

If the submitted username is valid, then the properties submitted with the document are processed, and any matching properties are copied into the appropriate columns of the document library. Finally, the document is placed into the destination library.

Custom Record Routing

The custom router processing has an interesting undocumented feature that is not obvious. The signature of the `IRouter.OnSubmitFile` method declares the `location` parameter, which is of type `SPList`, as a `ref` parameter as shown below:

```
RouterResult OnSubmitFile(  
    string recordSeries,  
    string sourceUrl,  
    string userName,  
    ref byte[] fileToSubmit,  
    ref RecordsRepositoryProperties[] properties,  
    ref SPList destination,  
    ref string resultDetails  
)
```

This parameter declaration suggests that the implementer of a custom router can change the location into which the document will be stored by simply specifying a different document library from the one specified by the routing type. Unfortunately, this value is ignored, and the original location specified by the routing type is used.

Property Storage

When a document is submitted to the repository using either the Records Management API or the Official File Web Service, its metadata properties are passed in an array. Two processing phases are applied to the array when the record is processed by the Records Center. These processing steps occur after a Record routing type has been found that matches the incoming document.

In the first processing phase, each property is compared to the columns that are declared on the document library associated with the Record routing type used to route the document to its final destination. Matching columns are then populated with the data in the associated properties. Having the document property values in document library columns allows standard search mechanisms to work as expected. However, there may be many additional properties that do not need to be placed into column values. This is where the second processing phase comes in.

In the second phase, the entire collection of properties is written to an XML document and stored in a subfolder within the primary folder associated with the new record. This preserves all of the properties so that advanced search utilities, reporting tools, and other components can locate and work with them.

To see how this all works, start by creating or uploading a document to a team site and adding some column values for testing. Figure 4-3 shows a sample document with some column values added.

Professional Records Management > Sample Documents > Lorem Ipsum > Edit Item

Sample Documents: Lorem Ipsum

OK Cancel

X Delete Item | Spelling... * indicates a required field

Name * Lorem Ipsum .doc

Title Extending the MOSS Record Routing Framework

Subject SharePoint

Category Records Management

Article Date 1/10/2009

Author John Holliday
The primary author

Byline Everything You Need To Know About Records Management

City Fernandina Beach

Company John Holliday & Associates

Department IT

State/Province Florida

Created at 8/3/2009 6:21 AM by SharePoint Master
Last modified at 8/3/2009 6:27 AM by SharePoint Master

OK Cancel

Figure 4-3: Sample document prior to submission.

Send the document to the Records Center using the “Send To” command on the Edit Control Block dropdown, and then navigate to the Unclassified Records library in the Records Center and open the folder corresponding to today’s date. Inside that folder, you will find a document having the name of the document you submitted with a unique suffix added, along with a subfolder named *Properties* as shown in Figure 4-4.

Professional Records Management > Unclassified Records > 2009-08-03T01-29-59Z

Unclassified Records

This document library is an example that can be used to store records submitted to the Records Center that do not match any other Record Routing entry.

New Upload Actions Settings View: All Documents

Type	Name	Modified	Modified By	Hold Status	Required Data	Starting Date	Ending Date
Folder	Audit History	8/3/2009 6:30 AM	System Account				
Folder	Properties	8/3/2009 6:30 AM	System Account				
Document	Lorem Ipsum_VW4ASH1.NEW	8/3/2009 6:30 AM	System Account		This was required.	1/10/2009	1/10/2010

Figure 4-4: Sample document Properties folder.

Inside the Properties folder, you will find an XML file with the same name as the document in the parent folder. This file contains all of the properties that were submitted along with the record. If you click on the filename, it should open in the browser, and you can see all of the properties and their values, as shown in Figure 4-5.

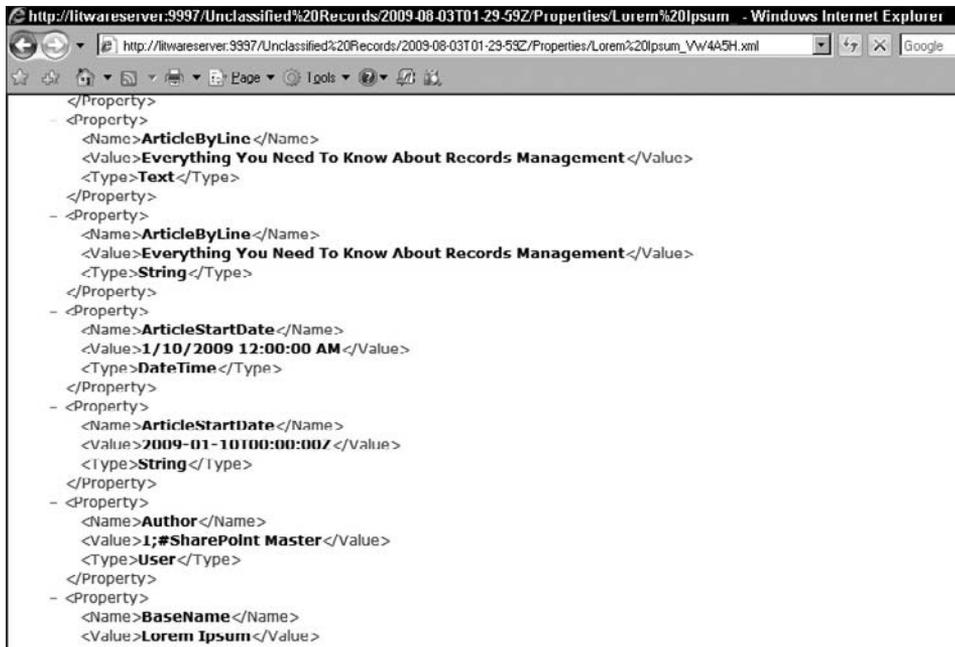


Figure 4-5: Sample document properties.

Audit History

A special property is used by SharePoint to submit the audit history for a document, if auditing is enabled for the record being submitted and there are audit records associated with the document. The name of the special property is `AuditHistory`, and it is stored in the Property array as an XML string.

When a record arrives for processing, the Records Center separates out the audit history from the other properties and copies the individual audit entries into a string that is ultimately written to the destination library in its own folder. This process removes the audit history from the properties that are submitted with the document and ensures a clean separation between the previous audit history and any subsequent audit records that are created through interactions with the record in the repository.

To see how this works, repeat the exercise from the previous section but with auditing enabled in the source web site. To enable auditing, navigate to the Site Settings page of the source web site and choose the "Site collection audit settings" link in the Site Collection Administration section. For this exercise, just enable the first two events in the "Documents and Items" section, as shown in Figure 4-6. Press OK to save your settings.

Now go back to the sample document library and edit some properties. For example, change the name of the document from the previous exercise. Save your changes, and submit the file again to the repository.

This time, when you navigate to the storage location inside the Unclassified Records library, you will find two documents and an additional folder named *Audit History*, as shown in Figure 4-7.

Professional Records Management > Site Settings > Configure Audit Settings

Configure Audit Settings

This Web application is configured to enable anonymous access. The actions of anonymous users will be audited, but their identities will not be recorded.

Documents and Items
Specify the events that should be audited for documents and items within this site collection.

Specify the events to audit:

- Opening or downloading documents, viewing items in lists, or viewing item properties
- Editing items
- Checking out or checking in items
- Moving or copying items to another location in the site
- Deleting or restoring items

Lists, Libraries, and Sites
Specify the events that should be audited for lists, libraries, and sites within this site collection.

Specify the events to audit:

- Filtering content types and columns
- Searching site content
- Editing users and permissions

OK Cancel

Figure 4-6: Configure Audit Settings.

Professional Records Management > Unclassified Records > 2009-08-03T01:29:59Z

Unclassified Records

This document library is an example that can be used to store records submitted to the Records Center that do not match any other Record Routing entry.

New Upload Actions Settings View: All Documents

Type	Name	Modified	Modified By	Hold Status	Required Data	Starting Date	Ending Date
Folder	Audit History	8/3/2009 6:30 AM	System Account				
Folder	Properties	8/3/2009 6:30 AM	System Account				
Document	Lorem Ipsum_VW4ASH NEW	8/3/2009 6:30 AM	System Account		This was required.	1/10/2009	1/10/2010

Figure 4-7: Audit History folder created.

Inside the Audit History folder, as for the Properties folder, there will be an XML file whose name matches the newly submitted filename. Instead of property values, this file contains audit record entries as shown in Figure 4-8. Scroll through the list to verify that the change you made to the document was captured in the Audit History file.

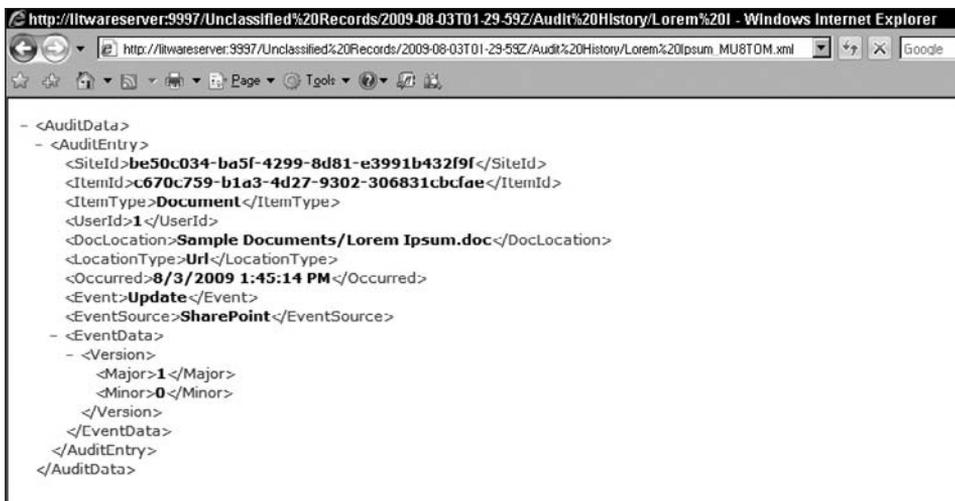


Figure 4-8: Audit History entries.

Property Promotion and Demotion

SharePoint includes a document parser that can extract document properties and write them into the appropriate columns in the document library where the document is stored. This process is called *property promotion* and happens automatically for any document libraries contained in an *SPWeb* object whose `ParserEnabled` property is set to `true`. The reverse is called *demotion*, in which the property values stored in the document library columns are written back into the document. If the `ParserEnabled` property is set to `false`, then the built-in document parser is disabled, and there will not be any property promotion or demotion for any document libraries in the entire Web. This is important for the Records Repository, because with the document parser enabled, it means that changes to the column values in the document library will automatically cause the parser to *demote* the property back into the document, resulting in a modification of the record, which is not allowed.

Unfortunately, it is not possible to turn off property demotion without also turning off property promotion. The result is that if a record needs to be changed at some point after its submission, perhaps to update incorrect metadata, then additional care must be taken to ensure that any changes made to the document are propagated back to the corresponding column in the document library in which the record is stored; otherwise, the library metadata and the document metadata will be out of sync.

To see the difference in property handling between a document library in a web site with parsing enabled and one without, perform the following test. First, create a Microsoft Office Word document, and enter values for the built-in `Subject` and `Category` properties, as shown in Figure 4-9.

Next, create a document library in an ordinary team site, and add two columns, one for the `Subject` and another for the `Category`. Figure 4-10 shows the result of adding the standard site columns for `Subject` and `Category`.



Figure 4-9: Document properties in a Word document.

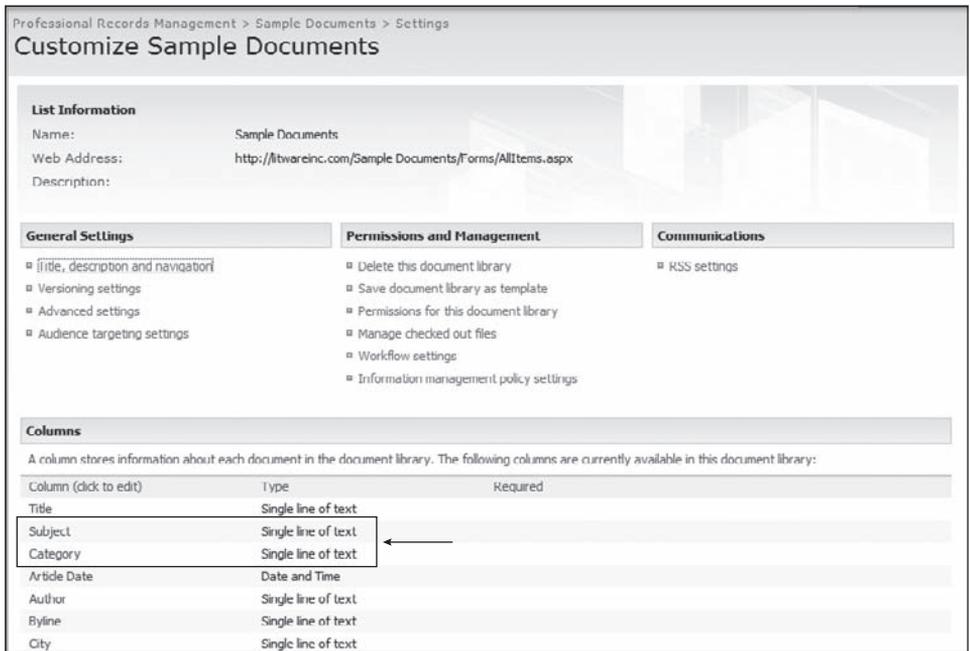


Figure 4-10: Document library with columns added.

Chapter 4: The MOSS 2007 Records Center

Upload the sample document, and note that the property values are automatically promoted into the appropriate columns as shown in Figure 4-11.

Professional Records Management > Sample Documents > Lorem Ipsum > Edit Item

Sample Documents: Lorem Ipsum

OK Cancel

Delete Item | Spelling... * indicates a required field

Name *	Lorem Ipsum.doc
Title	Extending the MOSS Record Routing Framework
Subject	SharePoint
Category	Records Management
Article Date	1/10/2009
Author	John Holliday The primary author
Byline	Everything You Need To Know About Records Management
City	Fernandina Beach
Company	John Holliday & Associates
Department	IT
State/Province	Florida

Created at 8/3/2009 6:21 AM by SharePoint Master
Last modified at 8/3/2009 6:45 AM by SharePoint Master

OK Cancel

Figure 4-11: Document properties promoted.

Now create a second document library in a Records Center site, and repeat the steps taken above, adding the Subject and Category columns and then uploading the same document to the library. Figure 4-12 shows the resulting library with the values not promoted.

Professional Records Management > Sample Documents (in Rec Center) > Lorem Ipsum > Edit Item

Sample Documents (in Rec Center): Lorem Ipsum

The document was uploaded successfully. Use this form to update the properties of the document.

OK Cancel

Delete Item | Spelling... * indicates a required field

Name *	Lorem Ipsum.doc
Title	
Subject	
Category	

Created at 8/3/2009 6:59 AM by System Account
Last modified at 8/3/2009 6:59 AM by System Account

OK Cancel

Figure 4-12: Document properties not promoted.

It is important to realize that disabling the document parser is an all-or-nothing proposition for the Records Center web site. It means that the default Records Center disables the document parser for *all* document libraries in the site. Thus, any solution that creates document libraries that are not used for record storage will have to handle property promotion explicitly for those libraries.

Summary

This chapter provided a detailed look at the Records Center site definition to understand its architecture and to see how the various components of the MOSS Records Management infrastructure work together to provide a foundation for building records management solutions.

We examined the individual SharePoint Features used to install the Records Center components and then looked at how the record routing table, the Holds List, and the unclassified records document library are set up according to basic assumptions about how a Records Repository will be used. To understand how all of these components fit together, we explored the core record processing sequence used by the Records Center to handle incoming records, including how metadata properties are submitted and stored, how audit entries are processed, and how properties are promoted to columns in the destination document library.

5

Building and Configuring a Records Repository

In Chapter 4, we examined the structure of the Records Center site definition and explored its architecture and the components that together provide the core records processing functionality. That functionality depends on the proper configuration of the Records Center site. In this chapter, we'll look into the steps you should follow in order to achieve that goal, whether doing it manually or programmatically.

Creating the Records Center Site

It is a common requirement to set up a Records Center site, and there is a lot of information available for doing that using the SharePoint UI. We'll review those steps in this chapter as well, but then we'll take a slightly different approach toward building a semi-automated process based on a *dynamic file plan*. This process will allow us to set up a Records Center so that it supports a specific set of record types. Later in the book, we'll add more features so that the dynamic file plan becomes our central point of focus for managing official records throughout their life cycle. I like to think of this as a *semi-automated* approach because it still requires the creation of a document (the actual file plan) that describes the types of records that are being configured. It's also important to note that the *automated* part may not happen immediately when the Records Center site is created, but later on, after the site is up and running.

I've taken this approach because there are really two stages to setting up a Records Center site. The first part has to do with creating the site itself based on the Records Center site definition and then activating the features needed to make everything work. That includes installing any custom components like routers, workflows, and master pages. The second part has to do with creating the appropriate record routing types for incoming documents, associating them with document libraries, and identifying the appropriate metadata that will be promoted to columns, and the like. In short, the first part has more to do with standard Microsoft Office SharePoint Server (MOSS) site provisioning, while the second part is more focused on the records management infrastructure.

Chapter 5: Building and Configuring a Records Repository

Ideally, we'd like to automate both parts. For example, we could use a technique called *feature stapling* to attach our custom infrastructure components to the out-of-the-box Records Center site definition.

Feature stapling is a technique whereby a SharePoint Feature is bound (stapled) to a site definition so that the feature is activated automatically whenever a site is created based on the site definition. You can use this technique to add features to out-of-the-box site definitions like the Records Center. For more information about feature stapling and how it works, see <http://msdn.microsoft.com/en-us/library/bb861862.aspx>.

Using the feature stapling approach, we could then easily ensure that every Records Center site has the necessary components installed without having to build our own custom Records Center site definition. Once the new Records Center is created, we would then activate a second set of tools that any records administrator could use to extend the Records Center site to enable it to handle a given set of document types, which have been identified in the file plan. We would activate the document type-specific tools using a separate set of SharePoint features so as to consolidate the individual steps that would normally be required to configure a particular document type.

Now, let's go step-by-step through the process of manually creating and configuring a Records Center site using the SharePoint UI. Later in the chapter, we will refer back to this site as we explore other ways to set up and manage a Records Center site programmatically.

In order to create a Records Center site, the MOSS enterprise features must be activated at both the site and site-collection levels.

Creating a Records Center Manually

Our Records Repository will provide controlled storage for the legal and accounting departments to archive contracts and legal documents related to ongoing litigation. It will also be used to house annual reports produced by the accounting department.

The first step is to set up a new site collection for official records. In order to facilitate greater administrative control over the contents of the repository, we will create the site collection in a separate web application.

Using a separate web application makes it easy to partition official records from other types of content. The web application you create will have its own content database and will be used exclusively to store and manage official documents.

1. Open a web browser and navigate to the Central Administration web site.
2. When the page opens, click on the Application Management tab.
3. From the Application Management page, click "Create or extend Web application" in the "SharePoint Web Application Management" section.
4. On the Create or Extend Web Application page, click "Create a new Web application." In the IIS Web Site section, select the default "Create a new IIS web site." Enter **SharePoint - Records Center** into the *Description* field. Enter a number into the *Port* field or accept the default. Leave the *Host Header* field blank. See Figure 5-1.



Figure 5-1: Creating a Records Center web application.

Port 9995 just happens to be the port number I chose to use when creating the Records Center web application in this example. Using a specific port number can make it easier to locate the web application during development because you'll tend to use the same number in each of your virtual machines. For example, I've gotten into the habit of using port 9999 for the central administration site. You can use any port or URL you like, as may be appropriate for your system or network configuration.

5. Scroll down to the Application Pool section. Select "Create new application pool" and enter **SharePoint - Records Center** as the Application Pool name. Select Configurable under "Select a security account for this application pool." Enter the administrator username and password, making sure to specify the fully qualified name including the user's domain, and then select the "Restart IIS Automatically" radio button as shown in Figure 5-2.

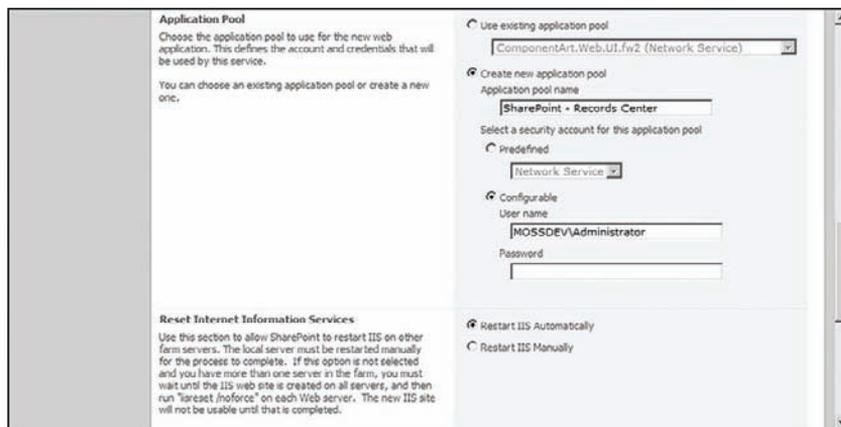


Figure 5-2: The Records Center Application Pool.

6. Scroll down to the "Database Name and Authentication" section. Enter **WSS_Records** into the *Database Name* field as shown in Figure 5-3. Click OK to create the new web application.

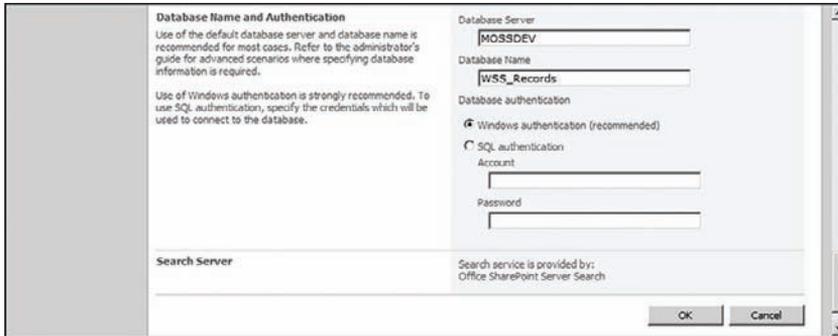


Figure 5-3: The Records Center database name.

7. On the Application Created page shown in Figure 5-4, click on the “Create Site Collection” link.

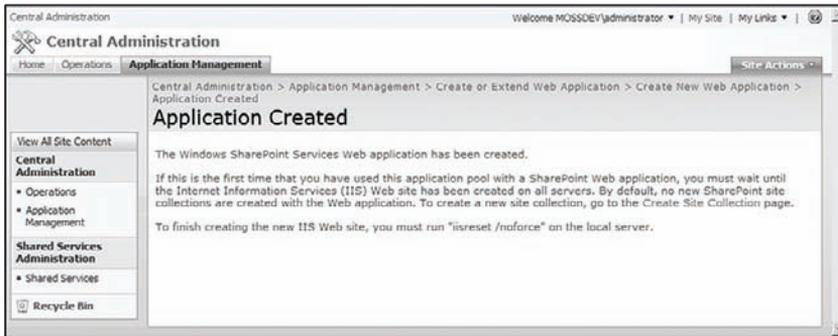


Figure 5-4: Records Center site creation confirmation.

8. On the Create Site Collection page, ensure that the correct application and port are visible in the Web Application dropdown shown in the “Web Application” section of Figure 5-5. In this example, it is *http://mossdev:9995/*.

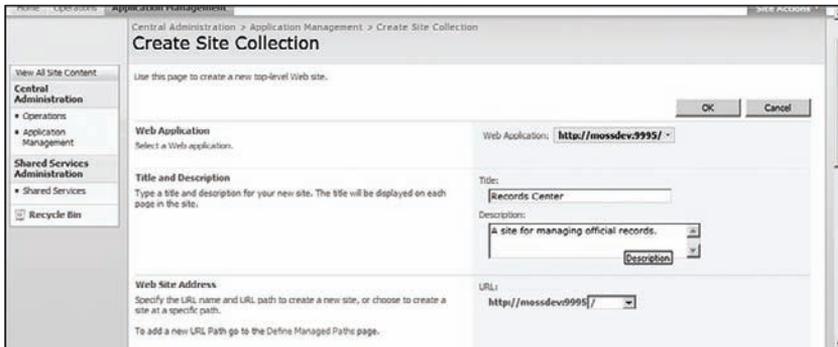


Figure 5-5: Create site collection page.

9. Enter **Records Center** in the *Title* field, and any description you like.
10. Scroll down to the “Template Selection” section. Click on the Enterprise tab and select the Records Center site template as shown in Figure 5-6.

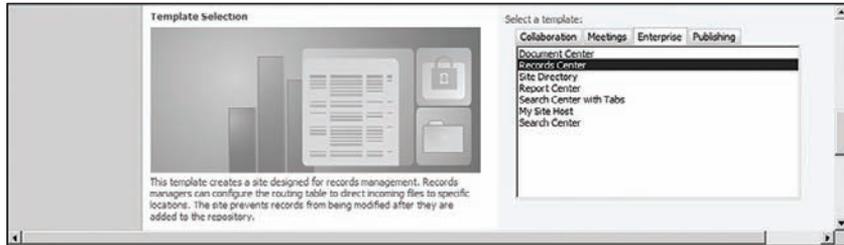


Figure 5-6: Records Center Template Selection.

11. In the “Primary Site Collection Administrator” section, enter the name of the site collection administrator and press [Ctrl]+K to resolve the name. Optionally, select another user as the Secondary Site Collection Administrator and press [Ctrl]+K again to resolve the name.

Pressing [Ctrl]+K in the people picker control is the same as pressing the icon to the right of the textbox that shows a user with a checkmark next to it, which causes the control to check the names you entered against the actual user accounts that are currently registered in the domain in which the system is running. (In my system, that results in the primary site administrator being MOSSDEV\Administrator.)

12. Finally, press OK to create the top-level site, and then from the Top-Level Site Successfully Created page, click the link to open the site and verify the site URL, which should display a new Records Center site as shown in Figure 5-7.

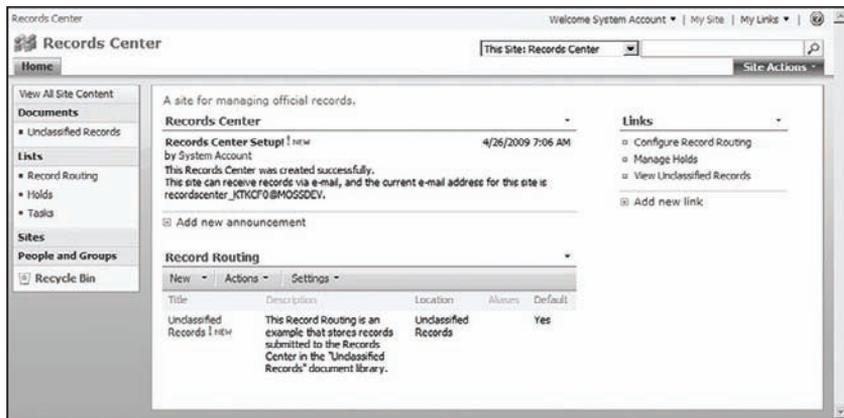


Figure 5-7: The Records Center home page.

We have now created a Records Center site using the SharePoint user interface. In the next section, we will look at how to do the same thing programmatically. Later, we will configure the repository to accept official records that are submitted from other SharePoint sites or from external applications.

Creating a Records Center Programmatically

Another approach to building a Records Center is to write a SharePoint feature, console application or PowerShell script that calls the SharePoint API to build the site. This, of course, gives you more flexibility to control how the site is structured and can come in handy when dealing with the second part of configuring the site, that is, setting up routing types and other components. Listing 5-1 shows the code needed to create a Records Center as the root web of a site collection in its own web application.

Listing 5-1: Creating a Records Center programmatically

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Diagnostics;
using System.DirectoryServices;
using System.Security;
using Microsoft.SharePoint;
using Microsoft.SharePoint.Administration;

namespace ECM2007.RecordCenterBuilder
{
    /// <summary>
    /// This program creates a records center site in its
    /// own web application as the root web of a site
    /// collection.
    /// </summary>
    class Program
    {
        int targetPort = 9995;
        string appName = "ECM2007_RecordCenter";
        string appTitle = "ECM2007 Records center";
        string dbName = "WSS_Content_ECM2007_RecordCenter";
        string appDescription = "A site for managing official records.";
        string appTemplate = "OFFILE#1";
        string appPoolName = "ECM2007_RecordCenter";
        string adminEmail = "admin@ecm2007.com";
        string adminLogin = "Administrator";
        string adminPassword = "password1";
        string adminDisplayName = "Records Administrator";

        static void Main(string[] args)
        {
            new Program(args).Run();
        }

        public Program(string[] args)
        {
        }

        public void Run()
        {
        }
    }
}
```

```
{
    try
    {
        // Get the local farm object.
        SPFarm farm = SPFarm.Local;
        SPWebApplicationBuilder builder = new
            SPWebApplicationBuilder(farm);

        // Check if the web application exists.
        string appUrl = string.Format("http://localhost:{0}", targetPort);
        SPWebApplication app = SPWebApplication.Lookup(new Uri(appUrl));

        if (app == null)
        {
            // Use the SPWebApplicationBuilder to create the app.
            builder.Port = targetPort;
            builder.CreateNewDatabase = true;
            builder.UseNTLMExclusively = true;
            builder.AllowAnonymousAccess = false;
            builder.UseSecureSocketsLayer = false;
            builder.ApplicationPoolId = appPoolName;
            builder.IdentityType = IdentityType.SpecificUser;
            builder.ApplicationPoolUsername = adminLogin;
            builder.ApplicationPoolPassword = new SecureString();
            foreach (Char c in adminPassword.ToCharArray())
                builder.ApplicationPoolPassword.AppendChar(c);

            // Create the app with a new app pool.
            Log("Creating web application at {0}", appUrl);
            app = builder.Create();
            app.Name = appName;
            app.Update();

            Log("Provisioning content database: {0}",
                builder.DatabaseName);
            app.Provision();

            // Add the new application to central admin.
            Log("Adding application to central admin console.");
            SPWebService.AdministrationService.WebApplications.Add(app);

            // Recycle the app pool.
            using (DirectoryEntry pool = new DirectoryEntry(
                string.Format("IIS://localhost/w3svc/apppools/{0}",
                    appPoolName)))
            {
                Log("Recycling app pool: {0}", appPoolName);
                pool.Invoke("Recycle");
            }

            // Create the site collection.

```

Continued

Listing 5-1: Creating a Records Center programmatically (continued)

```
        Log("Creating records center site collection...");
        SPSite siteCollection = app.Sites.Add("/", appTitle,
        appDescription,
            1033, appTemplate, adminLogin, adminDisplayName,
            adminEmail);
        Log("Records center created successfully.");
    }
    else
    {
        Log("Records center already exists at url: {0}", appUrl);
    }

    Log("Redirecting...");
    Process.Start(appUrl);
}
catch (Exception x)
{
    HandleException(x);
}
}

void Log(string format, params object[] args)
{
    string message = string.Format(format,args);
    Trace.WriteLine(message,"Records Center Builder");
    Debug.WriteLine(message,"Records Center Builder");
    Console.WriteLine(message);
}

void HandleException(Exception x)
{
    Log("Exception occured: {0}\n{1}", x.Message, x.ToString());
}
}
}
```

This code starts by creating an instance of the `SPWebApplicationBuilder` class. This class is a great way to set up new web applications because it allows you to specify all of the properties needed by IIS and then call a single `Create` method to build the web site. The `SPWebApplicationBuilder` object then handles all of the low-level communication with IIS so you don't have to.

The result of calling the `SPWebApplicationBuilder.Create` method is a new instance of the `SPWebApplication` class. This class encapsulates the steps needed to provision a SharePoint content database using its `Provision` method. Then we must add the new web application to the Central Administration console so that it shows up in the list of web applications for the administrator.

Finally, we recycle the application pool and then create a new site collection based on the OFFILE#1 site template. This specifies the site definition and configuration needed to create a new Records Center site at the root.

Working with Traditional File Plans

Chapter 1 introduced the core elements of the traditional file plan that is familiar to records managers and content administrators. These are typically created as worksheets and are used to describe the kinds of documents that will eventually become “official records” and how they should be handled at the enterprise level. In the following sections, we will use this approach to set up the content types, document libraries, and record routing types needed to process incoming records.

Creating Document Libraries and Content Types

Whether we create the Records Center site manually through the user interface or programmatically using code like that shown above, it must first be configured to accept specific kinds of documents before we can begin submitting records. In this example, we will set up a document library to hold legal documents like contracts and service agreements, as well as litigation support documents like motions and other court filings. We will also configure a separate document library for the accounting department that will hold financial statements and the like.

From the Quick Launch navigation bar on the left side of the Records Center home page, click on the Documents link. From the All Site Content page, click Create. On the Create page, click “Document Library.” Enter **Legal Documents** as the library name, along with a description, and click Create. Repeat this process to create a document library called *Financial Documents*.

Next, we need to add the metadata that we wish to associate with incoming documents. This is the point where the file plan influences the configuration of the Records Center.

When you are setting up the document library, you can either specify the record metadata directly on the library itself, or you can use content types. If your file plan does not distinguish between different document types within the same category, then you can attach the appropriate columns directly to the library. On the other hand, if you anticipate that the Records Center will receive different kinds of documents within the same category, then it is better to create a content type for each one so that you can more easily manage the metadata. After creating the required content types, you can then add them to any library you choose.

Let’s assume we have a simple file plan that identifies the three record types `Brief`, `Motion`, and `Statement` as shown in the following table. The `Brief` and `Motion` record types belong to the Legal Documents category, and the `Statement` record type belongs to the Financial Documents category.

Record Type	Description	Media	Category	Retention	Disposition
Brief	A legal brief for use during litigation	Electronic	Legal Documents	2 years	Archive
Motion	A motion that was filed with the court	Electronic, Printed	Legal Documents	5 years	Archive
Statement	A statement of account, such as a bank statement	Electronic	Financial Documents	5 years	Archive

Chapter 5: Building and Configuring a Records Repository

Let's assume further that our file plan specifies certain metadata fields that must be tracked for each record type as shown in the following table.

To simplify the table layout, we'll use an asterisk to indicate required fields. Also, the data types shown are the internal type names, and not the display names that appear in the dropdown lists of the SharePoint user interface.

Record Type	Property	Description	Type
Legal Document	Matter	A unique number that identifies the matter the document refers to	Text*
	Attorney	The name of the attorney who is responsible for the document	Text
Brief	Case Number	A unique case number that identifies the case to which the brief applies	Text*
	Plaintiffs	The names of the plaintiffs in a case	Note
	Defendants	The names of the defendants in a case	Note
Motion	Docket Number	The docket number that was assigned to the case	Text*
	Subject	The subject of the motion	Text*
Statement	Balance	The current statement balance	Currency*
	Period Starting Date	The date on which the current period started	DateTime*
	Period Ending Date	The date on which the current period ends	DateTime*

Using this information, we can set up the Legal Documents document library so that there is a prescribed location for incoming documents that match the profile of records described in the file plan. The best way to do this is to create a separate content type for each record type.

Navigate to the Site Settings page of the Records Center site, and then click on the "Site content types" link to open the Site Content Type Gallery page. Click on the Create button to create a new content type. First, we'll create a content type for each of the parent categories, and then we'll create content types for the individual record types.

Enter **Legal Document** into the *Name* field, and then enter **Any kind of legal document** for the description. In the "Select parent content type from" dropdown, select "Document Content Types" and then select Document for the "Parent Content Type." In the Group section, select the "New group" option and enter **Record Types** into the textbox field as shown in Figure 5-8.

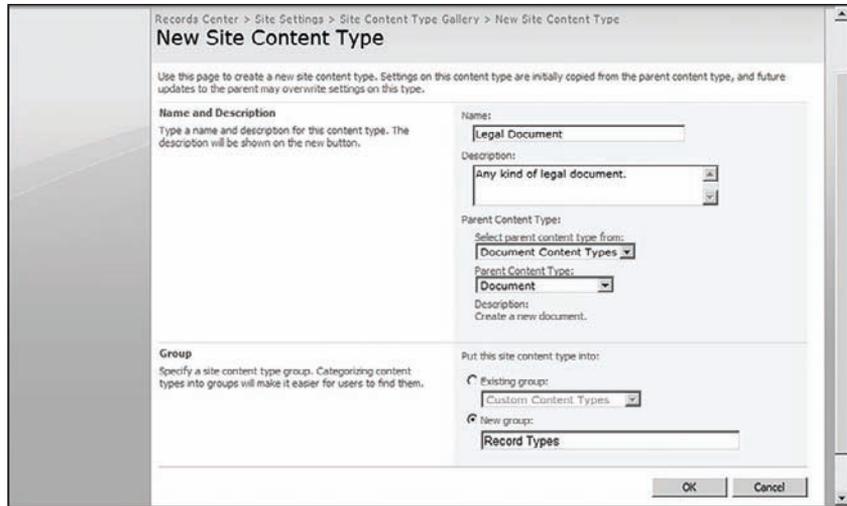


Figure 5-8: Legal Document Content Type.

Using content types for general record categories like this makes it easier to capture properties for file plans that are created early in the development cycle and may lack the detail needed to define the record types more precisely. As metadata properties and behaviors are discovered, these content types will provide a convenient place to attach them before performing further analysis.

Repeat this process to create a second high-level content type called *Financial Document* with an appropriate description. Assign this content type to the same Record Types group.

Now, we'll create a separate content type for each record type described in the file plan. Create another content type and enter **Brief** into the *Name* field. Copy the description from the file plan shown in the first table above, and select *Legal Document* as the parent content type. Repeat this process for each of the content types described in the file plan. When you are finished, you should have five content types declared as shown in Figure 5-9.

Record Types		
Brief	Legal Document	Records Center
Financial Document	Document	Records Center
Legal Document	Document	Records Center
Motion	Legal Document	Records Center
Statement	Financial Document	Records Center

Figure 5-9: Record types in the Content Type gallery.

Next, we need to add columns so that we can capture the metadata for each type. From the Site Content Type Gallery page, click on the link for the *Legal Document* content type you just created. Since the *Matter* and *Attorney* columns are not among the built-in site columns, we'll have to create them. Click on the "Add from new site column" link to open the New Site Column page. Enter **Matter** into the *Column name* field and choose "Single line of text" as the data type. Select "New group" under "Put this site column into," enter **Legal Document Columns**, and press OK to create the column.

Chapter 5: Building and Configuring a Records Repository

Although this is a required column, you cannot specify it on this page because you are only defining the column within the site column gallery at this point. To specify that the column is required, navigate back to the Content Type Settings page for the Legal Document content type, and click the link for the `Matter` column to open the Change Site Content Type Column: Legal Document page. From here you can specify that the column is required, as shown in Figure 5-10.

The screenshot shows a web browser window with the address bar displaying the path: Home > Records Center > Site Settings > Site Content Type Gallery > Site Content Type > Change Content Type Column. The main heading is "Change Site Content Type Column: Legal Document". Below the heading, there is a sub-heading "Use this page to edit a column on this content type." The page is divided into three main sections: "Site Column Information", "Column Settings", and "Update List and Site Content Types". The "Site Column Information" section contains text about the column and a link to edit the site or list column it refers to. The "Column Settings" section has a sub-heading "Specify settings for this content type column." and three radio buttons: "Required (Must contain information)" (selected), "Optional (May contain information)", and "Hidden (Will not appear in forms)". The "Update List and Site Content Types" section has a sub-heading "Specify whether all child site and list content types using this type should be updated with the settings on this page. This operation can take a long time, and any customizations made to these values on the child site and list content types will be lost." and two radio buttons: "Yes" (selected) and "No". At the bottom right, there are three buttons: "Remove", "OK", and "Cancel".

Figure 5-10: Editing site content type to make it required.

Repeat this process for each of the columns specified in the file plan from the first table above. As you create each column, it is a good idea to check first to determine whether there is already a site column in the gallery that you can use. For instance, the `Subject`, `Start Date`, and `End Date` columns are available out-of-the-box.

Although the `Start Date` and `End Date` column names do not match the `Period Starting Date` and `Period Ending Date` column names exactly, they are close enough to capture the information we need.

With the content types defined, we can now complete the configuration of our two document libraries. Navigate to the Document Library Settings page for the Legal Documents document library and click on the “Advanced settings” link to open the Document Library Advanced Settings page. Select the `Yes` option under “Allow management of content types” and click `OK`. Click the “Add from existing site content types” link, and add the `Brief`, `Motion`, and `Legal Document` content types to the library.

To keep things nice and tidy, it is good practice to remove the default `Document` content type from the list of content types for the library. This avoids unnecessary confusion for records managers who may not know what to do if the content type of a given item does not match the record types specified in the file plan. It is also possible to exclude the `Legal Document` content type, since both `Brief` and `Motion` inherit from it. Including it will allow the records manager to route other kinds of legal documents into the library and have them automatically assigned to the `Legal Document` type by default.

Since the Legal Documents library will only contain legal documents, you can remove the Document content type and set the default content type to Legal Document. To do this, click on the link for the Document content type and select “Delete this list content type” from the list of options on the page. Click on the “Change new button order and default content type” link to specify Legal Document as the first content type in the list so that it becomes the default.

Repeat this process to configure the Financial Documents document library such that the default content type is Financial Document and the Statement content type is also declared for the library. Now we are ready to configure the Record Routing Table that tells the SharePoint records processing infrastructure how incoming documents should be processed.

Setting Up the Record Routing Table

For this file plan, we need to create two routing types — one for legal documents and another for financial documents. Using these routing types, the Records Center can identify incoming records from each category and route them to the appropriate document library. The content types we defined for the record types within each category will be mapped to the list of aliases declared for each routing type.

From the home page of the Records Center site, click on the New button on the toolbar of the Record Routing list. On the Record Routing: New Item page, enter the details shown in Figure 5-11. This creates a new routing type that will match the three content types Legal Document, Brief, and Motion for any incoming record. When matched, the incoming record will be placed into the Legal Documents document library.

Records Center > Record Routing > New Item

Record Routing: New Item

Attach File | Spelling... * indicates a required field

OK Cancel

Title * Legal Document

Description All kinds of legal documents, including briefs, motions

Location * Legal Documents
The title of the library where records matching this record routing item should be stored. Libraries used to store submitted records cannot be deleted.

Aliases Brief/Motion
A '/' delimited list of alternative names that represent this record routing entry.

Default
If checked, this routing item will be used for submitted records that do not match the title or aliases of any other record routing item.

OK Cancel

Figure 5-11: Creating a Record Routing Table entry.

Repeat the process to create a routing table entry for financial documents. When you are done, you will have a routing table that resembles the one shown in Figure 5-12.

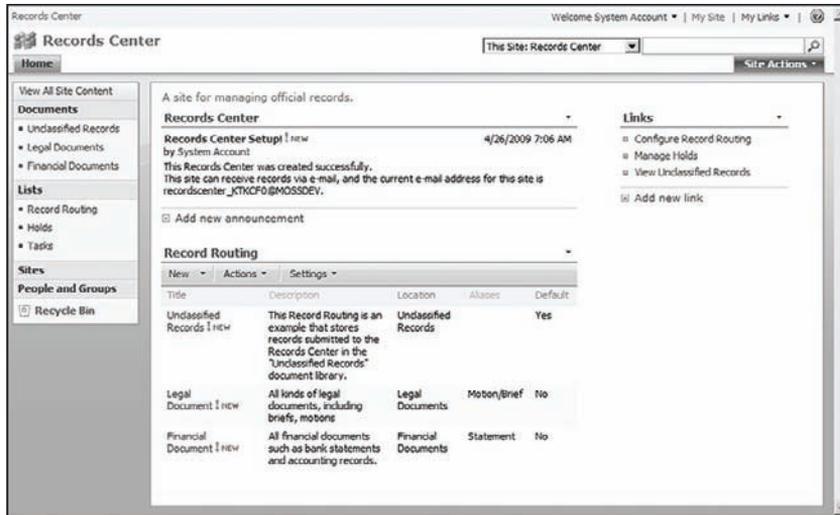


Figure 5-12: Configured Record Routing Table.

At this point, the Records Center is set up and ready to process incoming records. As you saw from the preceding discussion, there are a lot of steps involved in configuring each record type. This can become an increasingly time-consuming and tedious process as the number of record types, categories, and metadata properties continues to grow. In Chapter 6, we'll look at the steps needed to submit files to the Records Center. But first, let's consider an alternate approach to configuring the Records Center that does not require as many manual steps.

Building and Using Dynamic File Plans

Now, we'd like to extend the traditional notion of a file plan to include the ability to *execute* a file plan so that the record definitions and other rules contained within it are applied automatically to an existing Records Center site. The idea is to create a file plan document using XML instead of a table so that it includes all of the information that would normally be entered into a traditional file planning worksheet, but it can also include additional information that is interpreted by our records management components to set up the Records Center site automatically so that it can process records of any type described in the plan. Such a *dynamic* file plan could be loaded by a custom SharePoint feature to automatically configure the Records Center site to which it is attached. It could also be extended to set up holds and to account for other types of special conditions as they are discovered so that the Records Center can be adjusted directly from rules contained within the plan without having to perform those steps manually. This approach can dramatically reduce the risk of omitting critical steps and making errors that would otherwise compromise the integrity of the Records Center site.

Since our main concern is configuring a specific Records Center site that already exists, we'll start by creating a site-scoped feature that can be activated from any Records Center site. This design fits nicely with the MOSS *one records center per farm* model, but also supports Records Center topologies that include multiple Records Center sites, as long as they reside within the same site collection.

The functionality of our feature will be as follows:

1. Create a File Plan gallery to hold the file plan documents.
2. Enable administrators to create or upload file plans into the gallery.
3. Enable administrators to *execute* a file plan within the Records Center site.

Figure 5-13 shows the File Plan feature project structure within Visual Studio.

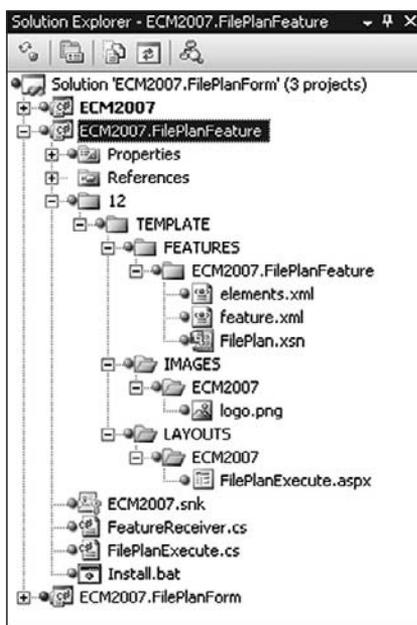


Figure 5-13: File Plan feature project.

The File Plan gallery will reside at the site-collection level so that when a file plan is executed for a given Web, our code will have access to the site column and site content type galleries that are shared by all Webs within the site collection. When executing a file plan, the administrator will be presented with a list of the record types that can be activated. If there are optional settings for a given file plan, these can be set on the File Plan Execution page. When the administrator presses the Execute button, the selected record specifications will be used to configure the Records Center site automatically.

Before we can build the feature, we need a way to capture the essential elements of the file plan. We'll do this using InfoPath 2007 driven by an XML schema. In the sections that follow, we'll design the schema and create an InfoPath form for entering the record specifications.

One of the primary benefits of developing a schema separately from the form is the ability to reuse the same schema in different layers of the solution. This approach lends flexibility to the solution architecture so it can be easily adapted to changing roles and evolving requirements.

XML schemas can also simplify both the design and development of supporting code related to data elements that conform to the schema. By using an .xsd file as the primary data source for the form, our

schema can be used to generate wrapper classes that understand how to serialize and de-serialize any data files created from the form template. We can then extend those generated classes in various ways to perform other operations on the underlying data.

Designing the File Plan Schema

We'll start by designing a simple schema that describes a file plan. Our schema will include the traditional file plan components we identified in Chapter 1, but we'll extend it now to include some additional information that will be useful for configuring a Records Center site. For instance, we can include things like the name of the content type to associate with a particular record (or we can derive it from the record name), the name of the document library to create, the name of the custom router to use, and which information policy features to enable, and so on. Listing 5-2 shows the portion of the `FilePlan` schema that reveals its overall structure.

Listing 5-2: FilePlan schema

```
<xs:annotation>
  <xs:documentation>
    Schema for declaring a dynamic file plan that describes
    how to convert one or more documents into "official records",
    including instructions for processing each record type
    when it is submitted to a SharePoint records repository.
  </xs:documentation>
</xs:annotation>

<xs:element name="FilePlan">
  <xs:annotation>
    <xs:documentation>
      This is the top-level element that defines the file plan
      and includes general information such as the plan title and
      description.
    </xs:documentation>
  </xs:annotation>
  <xs:complexType>
    <xs:all>
      <xs:element name="Title" type="xs:string"/>
      <xs:element name="Description" type="xs:string"/>
      <xs:element name="Author" type="xs:string"/>
      <xs:element name="Records" type="RecordSet" maxOccurs="1"/>
    </xs:all>
  </xs:complexType>
</xs:element>
```

Listing 5-3 shows the additional details needed to describe an individual record specification.

Listing 5-3: RecordSpecification schema

```
<xs:complexType name="RecordSpecification">
  <xs:annotation>
    <xs:documentation>
```

Chapter 5: Building and Configuring a Records Repository

This element describes a single record type and is used to control how documents matching a given type are processed when submitted to a records center site.

```
</xs:documentation>
</xs:annotation>
<xs:all>
  <xs:element name="Sources" type="RecordSources" minOccurs="0"/>
  <xs:element name="Media" type="xs:string" minOccurs="0">
    <xs:annotation>
      <xs:documentation>
        Describes the media type for the document (such as web page,
        spreadsheet or presentation).
      </xs:documentation>
    </xs:annotation>
  </xs:element>

  <xs:element name="Aliases" type="AliasList" minOccurs="0" maxOccurs="1"/>
  <xs:element name="Location" type="xs:string" minOccurs="0" maxOccurs="1">
    <xs:annotation>
      <xs:documentation>
        Specifies the location in the records center in which records of this
        type will be stored. If the location does not already exist, one is
        created automatically when the file plan is executed.
      </xs:documentation>
    </xs:annotation>
  </xs:element>

  <xs:element name="Router" type="xs:string" minOccurs="0">
    <xs:annotation>
      <xs:documentation>
        Specifies the name of the custom router to associate with
        this record type. The specified router must be installed separately
        into the records center by an administrator.
      </xs:documentation>
    </xs:annotation>
  </xs:element>

  <xs:element name="AssignedTo" type="People" minOccurs="0">
    <xs:annotation>
      <xs:documentation>
        Specifies the people who are responsible for managing this
        record type. The users in this list may be notified when the
        record series is created, and tasks can be created automatically
        to remind them to perform related tasks.
      </xs:documentation>
    </xs:annotation>
  </xs:element>

  <xs:element name="Description" type="xs:string" minOccurs="0">
    <xs:annotation>
      <xs:documentation>
        A brief description of the purpose of the record series.
      </xs:documentation>
    </xs:annotation>
  </xs:element>
</xs:all>
```

Continued

Listing 5-3: RecordSpecification schema (continued)

```
</xs:element>
</xs:all>
<xs:attribute name="Name">
  <xs:annotation>
    <xs:documentation>The name of the record series.</xs:documentation>
  </xs:annotation>
</xs:attribute>
<xs:attribute name="Type" type="RecordType" default="Electronic"/>
</xs:complexType>
```

Note that we have provided elements in the schema to capture the source locations of each record specification. In later chapters, we will extend the file plan schema to include additional information where necessary to support other operations that occur over the course of the content life cycle such as finding and moving documents into the repository, setting up auditing, and configuring information policies.

With the schema defined, we can now use it to build our InfoPath form. Before describing the steps in detail, a brief introduction to the InfoPath forms design process will help to clarify the importance of starting with a schema. As you will see in the following sections, InfoPath 2007 provides strong support for schema-driven component building.

A Quick Introduction to InfoPath 2007

In this section, we'll explore ways to use InfoPath effectively. This ties into the notion of schematizing metadata in order to drive different phases of the content life cycle for a given solution, and it requires that we consider InfoPath in a broader context than as a stand-alone Forms Designer. Like the other programs that make up the Microsoft Office 2007 client suite, InfoPath 2007 can be viewed as one component of an enterprise-wide content management system. *InfoPath* provides a controlled *path* for information to flow through the system in ways that allow us to extend and use it to drive other parts of the system.

In addition to providing a rich end-user experience for entering data into forms, the primary goal of InfoPath 2007 is to capture schematized XML data and to do it in a way that conforms to widely accepted standards. Consequently, InfoPath uses XSL stylesheets to present views of the data and uses the same APIs to read and write the data that we will be using in our own code. By adhering to the same schema for both the gathering and processing of data, we can build different components that interoperate on that data at different stages of the content life cycle. An important secondary goal of InfoPath is to establish and maintain tight integration with SharePoint so that data can be easily consumed through SharePoint sites.

InfoPath 2007 is the second version of the product. If you had any experience using InfoPath 2003, you will immediately see that the new version provides a significantly improved design experience, although it is still somewhat labor-intensive from a developer perspective.

In my talks with developers who use InfoPath 2007 regularly, I've found that they tend to take little shortcuts on the design side, ultimately relinquishing their forms to page designers who are more familiar with design tools. The Visual Studio developers I know lack the patience required to produce very sophisticated designs. Similarly, herein we will focus on making sure that the required elements are in place and that the schema is complete, and we won't spend a lot of time with the design tools.

One of the major enhancements to InfoPath 2007 is that it now offers the ability to separate pure design activities from the steps required to deploy a form. It includes several tools and wizards that assist in the deployment process for various deployment targets, including the file system, SharePoint sites, and document libraries. This helps to eliminate concerns about deployment issues that don't really affect the design process. Also, the tighter integration with SharePoint really takes advantage of its new functionality. For example, you can publish a form to SharePoint directly from the InfoPath designer as a new content type that is bound to the form template and then use that content type in multiple libraries.

Another big requirement that came from users of InfoPath 2003 was the need to extend the reach of InfoPath forms beyond the client machine and to access electronic forms from a web browser so that users without the InfoPath client could still interact with the form in the same robust manner and with a similar rich user experience as that presented to users of the client-side product. The term *extending the reach* is the politically correct way to say you don't really get full fidelity when interacting with a form through the browser. There are some limitations that come in because of the fact that you're publishing into a shared environment. Nevertheless, the ability to enable interaction with a form without having the InfoPath client installed enables the deployment of enterprise-wide content management solutions and fits well with the life-cycle approach we are taking here.

Using the InfoPath Forms Designer

Let's take a closer look at the InfoPath Forms Designer. When you open InfoPath, you get various options for creating a form. You can start by importing or designing a form using a blank template or by customizing one of the included sample forms such as the Asset Tracking form shown in Figure 5-14.

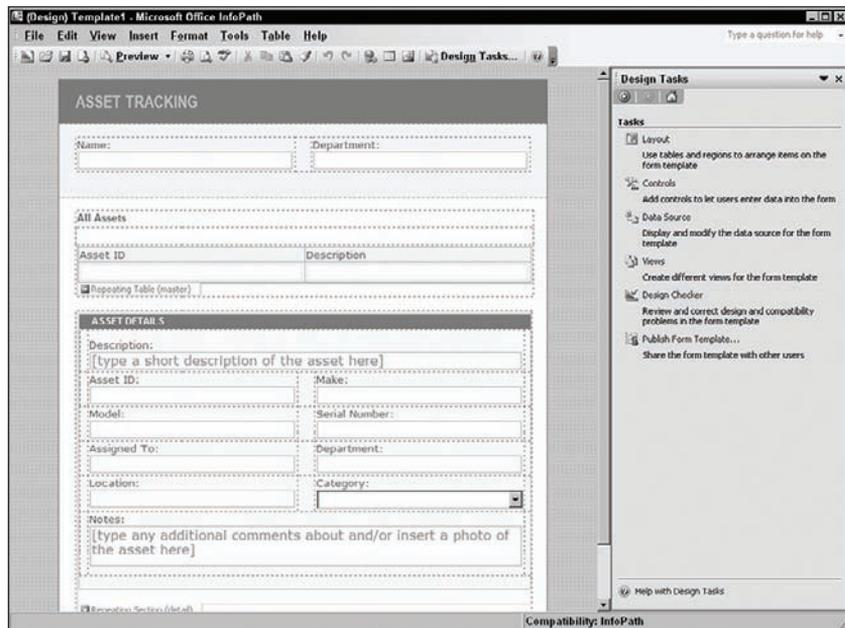


Figure 5-14: An Asset Tracking form.

Chapter 5: Building and Configuring a Records Repository

The design process involves dragging items onto the Design surface to layout controls, which are used by the end-user to manipulate data that comes from one of the embedded data sources. Figure 5-15 shows the main data source that describes the asset tracking information. To create a control on the form, you simply drag the data element you want onto the form. InfoPath then allows you to select a user-interface control that can display that type of data, and then it automatically binds the data element to the control you have chosen.

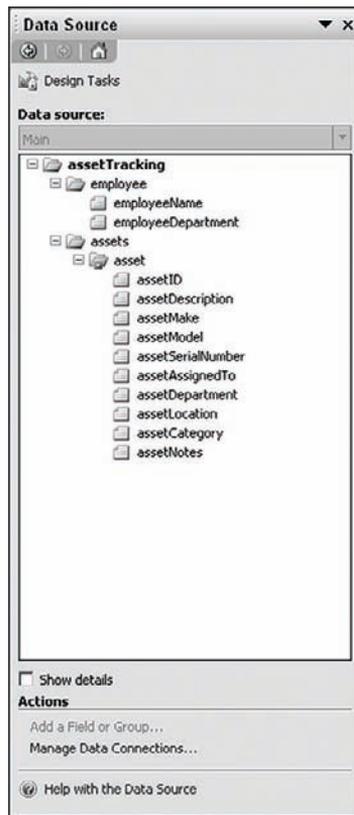


Figure 5-15: Asset Tracking data source.

Not only does InfoPath provide a Design surface onto which you can drag controls or data elements, but it also includes an integrated validation framework so that if you make mistakes while binding data elements to controls, InfoPath can warn you that something is wrong with that element. This makes it much easier to build forms, and InfoPath does a pretty good job of validation. InfoPath 2003 did not provide as much support, which made it possible to spend a lot of time designing a form only to end up with errors that were difficult to find and fix.

Chapter 5: Building and Configuring a Records Repository

While designing a form, you can click on the Preview button to see what the form will look like to the end-user. You can also set up data connections that allow your form to store data to or retrieve data from external data sources.

Another important area to pay attention to when designing a form is the set of options available for enabling the user to *submit* the form after filling it out. *Submitting* a form is different from *saving* a form. The user can save the form at any time and come back to it later to continue filling it out. When the user is ready to submit the form, a special operation comes into play and may have special rules attached. InfoPath lets you enable or disable the Submit button and define the rules that govern what happens when the user presses it.

If you turn on the ability to submit the form, there are many options for doing so. You could specify that when the user submits the form, code is executed that automatically sends the form data to various locations. The actual locations might depend on other factors related to the environment in which the form was deployed. You might also perform custom actions and create rules that are executed when the form is submitted. InfoPath lets you change the text that is displayed on the Submit menu item title, or you can hide it completely. You also get additional features such as the ability to customize the messages that are displayed on the success or failure of the submission, and you can control what happens after submission, such as close the form, leave it open for continued editing, or create another instance of the form. Since the submit processing has been separated out, you don't have to write special code to accomplish many of the standard submit operations. This simplifies the whole process and allows you to add code only when absolutely necessary.

Designing the File Plan Form

To simplify the creation of file plans, we will use InfoPath 2007 to design a simple form based on the file plan schema. By using InfoPath, we can avoid having to build a user interface for editing file plan documents. Figure 5-16 shows the completed form in the InfoPath Forms Designer. This isn't the most elegantly designed form, but it should be good enough for our purposes.

The screenshot shows a web form titled "File Plan" with the following sections and fields:

- Title:** Sample Plan
- Description:** This is a sample file plan that describes a set of record types for processing in the record center.
- Author:** John Holliday
- Records** section:
 - Record Type:** Legal Document
 - Description:** Any kind of legal document, such as contracts, briefs, motions, pleadings, etc.
 - Type:** Radio buttons for Electronic (selected), Bar Coded, and Labeled.
 - Location:** Legal Documents
 - Router:** (empty)
 - Media:** (empty)
 - Aliases:** List with radio buttons: Contract, Pleading, Brief, Motion, and Add Alias.
 - Assigned to:** List with radio buttons: John Holliday and Add User.
 - Sources:** Path: (empty), File Mask: (empty)
 - Auditing:** (empty)
 - Expiration:** Click here to configure an expiration policy for (empty)

Figure 5-16: File Plan form.

Chapter 5: Building and Configuring a Records Repository

We start by creating a new InfoPath Form Template project in Visual Studio and giving it the name **ECM2007.FilePlanForm**, as shown in Figure 5-17.

VSTA is an initiative that Microsoft created to allow managed-code assemblies to be associated with any kind of application, whether managed or unmanaged. VSTA is a very powerful platform, that when integrated into an application provides a pared-down version of the Visual Studio IDE embedded within the application itself. InfoPath 2007 includes VSTA and thus embeds the Visual Studio IDE into the Forms Designer. In contrast, VSTO is a subset of VSTA that is geared more toward building office applications using the .NET Framework. Instead of embedding the Visual Studio IDE into the Forms Designer, it embeds the Forms Designer into Visual Studio, giving the developer the same development experience as for other types of solutions. The screenshots in this book that involve the creation of custom InfoPath forms will generally show the VSTO user interface.

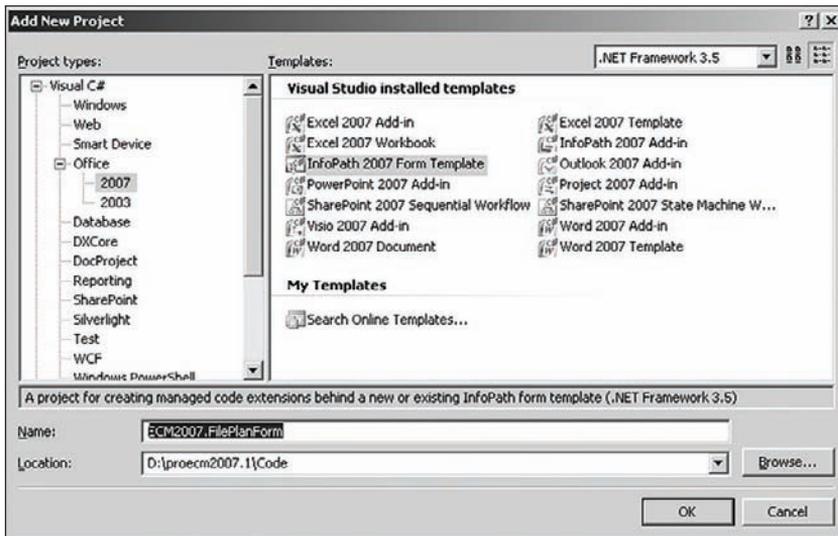


Figure 5-17: Creating a new VSTO InfoPath project

InfoPath provides several ways to build the data source that a form will be based on. We can extract a schema directly from a Web Service or a SQL database, or we can base the data source on an existing XML schema. In the next dialog, we select “Form Template” based on “XML or Schema” as shown in Figure 5-18. This allows us to automatically define the data source for the form based on the sales proposal schema we have already created.

From here, we can simply browse to the FilePlan.xsd file we created previously, as seen in Figure 5-19.

When we press OK, InfoPath generates the form template and then opens it in the Visual Studio InfoPath Form Designer.

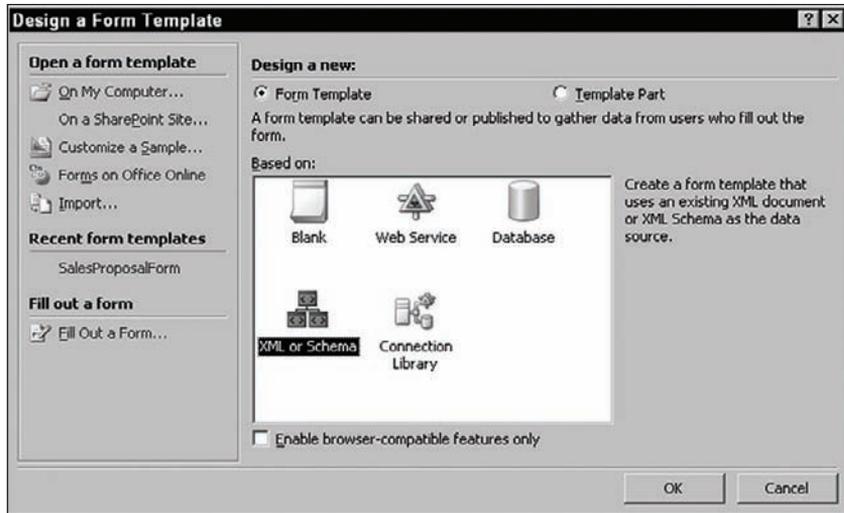


Figure 5-18: Choosing a form template.

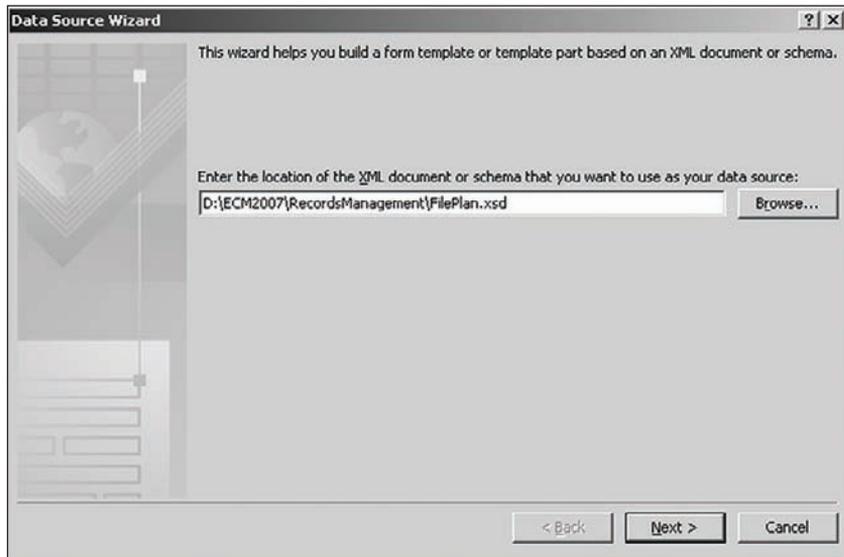


Figure 5-19: InfoPath Data Source Wizard.

When creating an InfoPath form template using VSTO, the standard InfoPath Form Designer is integrated directly into the Visual Studio IDE. The same options are available, but they are placed in different locations within the IDE. This can be somewhat confusing at first, but you just have to learn where the commands are located within each environment.

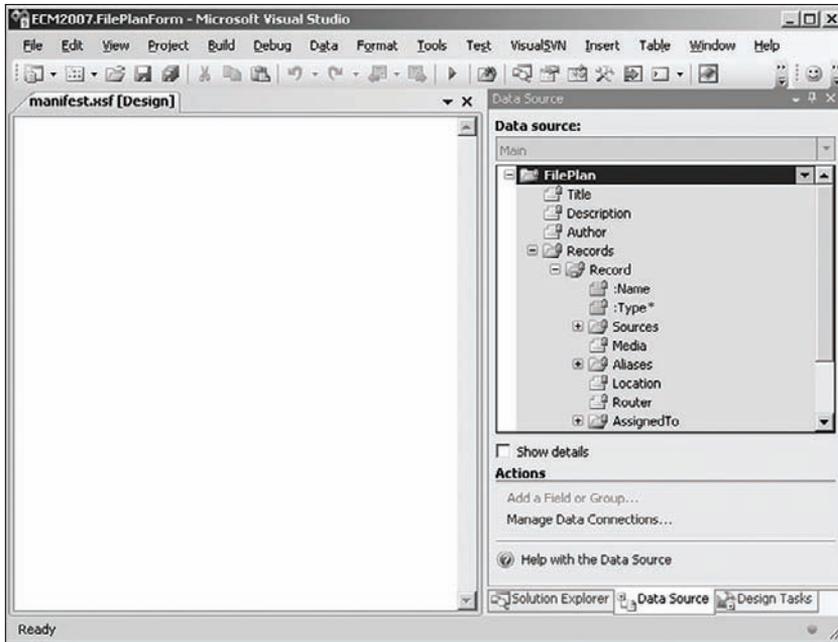


Figure 5-20: File Plan Form data source.

From the Design Tasks window, we can begin designing our form. Click on the “Data Source” link to examine the `FilePlan` data source definition that InfoPath has created. Note in Figure 5-20 that the data source is *locked* to prevent inadvertent modifications of the schema.

By locking data sources based on external schemas, InfoPath ensures that the schema will not be changed when controls are dragged onto the Form Designer surface.

At this point, we can start dragging data elements onto the form. If you are following along in your own copy of InfoPath, use the illustration in Figure 5-21 as a guide to drag-and-drop controls onto the form. The most important point is to ensure that we have included all of the data elements we need. One way to do that is to simply drag the entire data source onto the Design surface and then rearrange the controls. Another approach (used here) is to create the basic layout using layout tables and then drag individual data elements or element groups into the layout table cells.

When we finish designing our form, we are ready to test it. To do this, we can right-click on the `manifest.xsf` file in Solution Explorer and select “Open With” from the context menu. Choose “Microsoft Office InfoPath 2007” and press OK. This will open a new instance of InfoPath with our form ready for editing. Enter some data into the form fields.

After entering our sample data, we can then save the file to disk and open it for editing within Visual Studio, as shown in Figure 5-22.

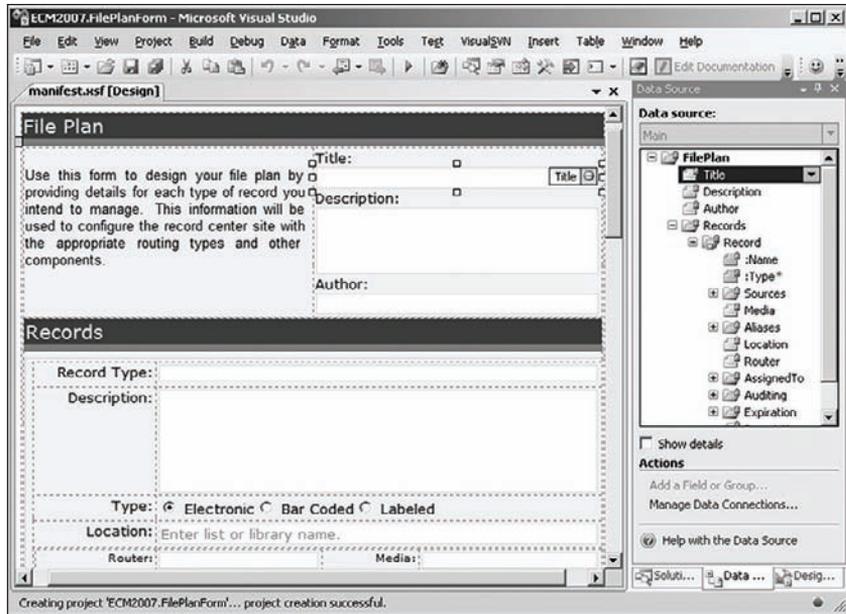


Figure 5-21: File Plan Form design.

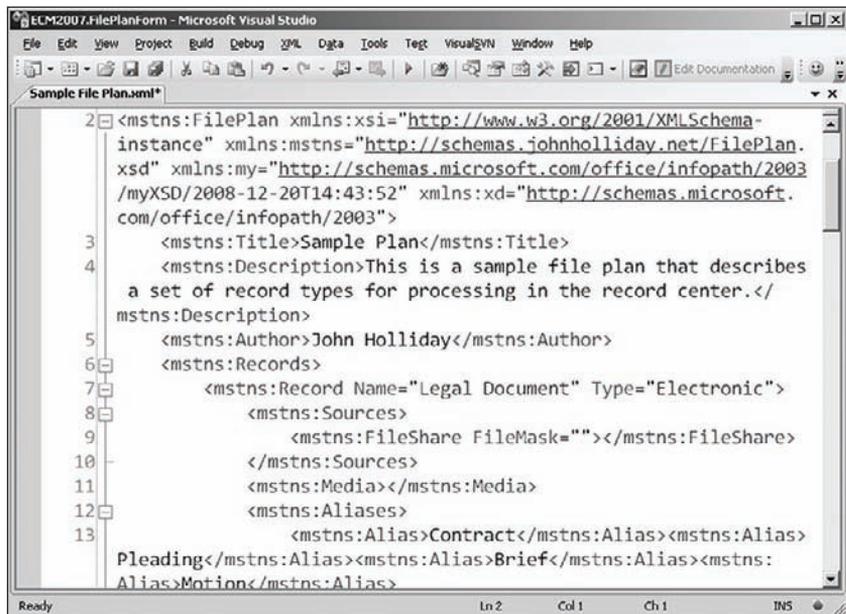


Figure 5-22: File plan data in the Visual Studio IDE.

Chapter 5: Building and Configuring a Records Repository

At this point, we have a File Plan form template that can be loaded into InfoPath to gather XML data that is guaranteed to conform to our file plan schema. Next, we need to build a File Plan API that will allow us to serialize and de-serialize the data into C# wrapper classes. We will use these classes to simplify the code we must write in order to interpret the instructions contained within a given file plan to programmatically configure the Records Center site to which they are applied.

Generating Serialization Classes

In this step, we will generate C# classes for each of the elements of the file plan schema defined previously. To do this, we will use the `XsdClassGenerator` custom tool described in Chapter 2 that acts as a wrapper for the `XSD.EXE` command-line utility that comes with Visual Studio.

Make sure the Properties window is open in Visual Studio, and then right-click on the `FilePlan.xsd` file in the Solution Explorer window. Enter `XsdClassGenerator` in the 'Custom Tool' property, and then press the [Enter] key to generate the wrapper classes. You can also specify a custom namespace that the tool will use to enclose the generated classes.

A new node appears beneath the `FilePlan.xsd` file named `FilePlan.cs` that contains the serialization classes we need. (The top-level `FilePlan` component is shown in Listing 5-4.) In later steps, we will add more code in separate files to extend the functionality of the generated classes. At this point, however, we can go ahead and build the project to create a compiled assembly. This will ultimately become a custom File Plan API that can be incorporated into any solution that requires file plan functionality.

Listing 5-4: FilePlan serialization wrapper classes

```
//-----  
// <auto-generated>  
//     This code was generated by a tool.  
//     Runtime Version:2.0.50727.3031  
//  
//     Changes to this file may cause incorrect behavior and will be lost if  
//     the code is regenerated.  
// </auto-generated>  
//-----  
  
//  
// This source code was auto-generated by xsd, Version=2.0.50727.42.  
//  
namespace ECM2007.RecordsManagement {  
    using System.Xml.Serialization;  
  
    /// <remarks/>  
    [System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]  
    [System.SerializableAttribute()]  
    [System.Diagnostics.DebuggerStepThroughAttribute()]  
    [System.ComponentModel.DesignerCategoryAttribute("code")]  
    [System.Xml.Serialization.XmlTypeAttribute(AnonymousType=true,  
        Namespace="http://schemas.johnholliday.net/FilePlan.xsd")]  
    [System.Xml.Serialization.XmlRootAttribute()
```

Chapter 5: Building and Configuring a Records Repository

```
        Namespace="http://schemas.johnholliday.net/FilePlan.xsd",
        IsNullable=false)]
public partial class FilePlan {

    private string titleField;
    private string descriptionField;
    private string authorField;

    private RecordSpecification[] recordsField;

    /// <remarks/>
    public string Title {
        get {
            return this.titleField;
        }
        set {
            this.titleField = value;
        }
    }

    /// <remarks/>
    public string Description {
        get {
            return this.descriptionField;
        }
        set {
            this.descriptionField = value;
        }
    }

    /// <remarks/>
    public string Author {
        get {
            return this.authorField;
        }
        set {
            this.authorField = value;
        }
    }

    /// <remarks/>
    [System.Xml.Serialization.XmlArrayItemAttribute("Record",
    IsNullable=false)]
    public RecordSpecification[] Records {
        get {
            return this.recordsField;
        }
        set {
            this.recordsField = value;
        }
    }
}
}
```

Chapter 5: Building and Configuring a Records Repository

In practice, it's a good idea to remove the `DebuggerStepThroughAttribute` from the generated wrapper classes because this will prevent you from stepping through the debugger as you write code to extend them. Although a bit tedious because you have to do this every time the classes are regenerated, it can save a lot of time while developing the custom API. The source code for the `XsdClassGenerator` tool is provided with the code downloads for the book, so it's possible to add some additional methods to remove these lines after the file is generated, but it has never seemed worth the hassle.

To handle the serialization and de-serialization of XML data into the classes, we use the `System.Xml.Serialization.XmlSerializer` class. To make it easier to load `FilePlan` objects into memory, we will extend the generated wrapper classes by adding three versions of a static `Load` method. These methods will accept either a path to a file containing XML file plan data, an `SFFile` object, or an `SPLListItem` from which an `SFFile` object can be obtained. We will place this code into a separate file called `FilePlanEx.cs` so that it is not overwritten when the file plan schema changes and the wrapper classes are regenerated. Listing 5-5 shows the de-serialization code.

Listing 5-5: FilePlan de-serialization methods

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Xml;
using System.Xml.Serialization;
using Microsoft.SharePoint;

namespace ECM2007.RecordsManagement
{
    /// <summary>
    /// Extended implementation of the generated FilePlan
    /// class to support Record Center configuration.
    /// </summary>
    public partial class FilePlan
    {
        public const string FILEPLAN_NAMESPACE =
            "http://schemas.johnholliday.net/FilePlan.xsd";
        private List<ListDescriptor> m_documentLibraries = null;
        private List<ListDescriptor> m_lists = null;

        #region Deserialization Methods
        /// <summary>
        /// Loads a file plan object from a file.
        /// </summary>
        /// <param name="filePath">fully qualified path to the file</param>
        /// <returns>a FilePlan component</returns>
        public static FilePlan Load(string filePath)
        {
            try
            {
                XmlSerializer ser = new XmlSerializer(typeof(FilePlan),
                    FILEPLAN_NAMESPACE);
                using (FileStream fs = File.Open(filePath, FileMode.Open))
                    return (FilePlan)ser.Deserialize(fs);
            }
        }
    }
}
```

```
    }
    catch (FileNotFoundException fx)
    {
        Helpers.HandleException("ECM2007.FilePlan", fx);
    }
    catch (XmlException x)
    {
        Helpers.HandleException("ECM2007.FilePlan", x);
    }
    return null;
}

/// <summary>
/// Loads a file plan object from a SharePoint file.
/// </summary>
/// <param name="file">the file object containing the file plan</param>
/// <returns>a FilePlan component</returns>
public static FilePlan Load(SPFile file)
{
    try
    {
        XmlSerializer ser = new XmlSerializer(typeof(FilePlan),
            FILEPLAN_NAMESPACE);
        return (FilePlan)ser.Deserialize(file.OpenBinaryStream());
    }
    catch (SPException spx)
    {
        Helpers.HandleException("ECM2007.FilePlan", spx);
    }
    return null;
}

/// <summary>
/// Loads a file plan object from a SharePoint list item.
/// </summary>
/// <param name="item">the list item containing the file plan
    document</param>
/// <returns>a FilePlan component</returns>
public static FilePlan Load(SPListItem item)
{
    if (item.File != null)
        return Load(item.File);
    Helpers.Log("Cannot load file plan. List item '{0}' has no file",
        item.Title);
    return null;
}
#endregion
}
}
```

In addition to the de-serialization methods, we'll need some code for configuring the Records Center site. Listing 5-6 shows the configuration code, starting with the `ConfigureRecordCenter` method that iterates through all of the `RecordSpecification` objects declared within the file plan and then delegates to the `RecordSpecification.Execute` method to perform the actual configuration step.

Chapter 5: Building and Configuring a Records Repository

We will extend the RecordSpecification class later in this section in the same way we extended the FilePlan class, namely, by creating a separate file and declaring a partial class containing the additional methods we need.

Listing 5-6: Records Center configuration from a file plan

```
#region Record Center Configuration

/// <summary>
/// Configures an existing record center site with the
/// content types, lists, document libraries and record
/// routing entries needed to implement a file plan.
/// </summary>
/// <param name="recordCenter">a record center site</param>
public bool ConfigureRecordCenter(SPWeb recordCenter)
{
    try
    {
        bool success = true;
        foreach (RecordSpecification record in this.Records)
            success = success && record.Execute(this, recordCenter);
        return success;
    }
    catch (Exception x)
    {
        Helpers.HandleException(this, x,
            "Failed to configure record center: {0}",
            recordCenter.Title);
    }
    return false;
}

public RecordSpecification Find(string recordName)
{
    foreach (RecordSpecification record in this.Records)
        if (record.Name.Equals(recordName))
            return record;
    return null;
}

/// <summary>
/// Gets the list of unique document libraries as
/// specified by records defined in the file plan.
/// </summary>
[XmlIgnore]
public List<ListDescriptor> DocumentLibraries
{
    get
    {
        if (m_documentLibraries == null)
        {
            Dictionary<string, ListDescriptor> libraries
                = new Dictionary<string, ListDescriptor>();

```

Chapter 5: Building and Configuring a Records Repository

```
        foreach (RecordSpecification record in this.Records)
        {
            if (record.Type == RecordType.Electronic)
            {
                if (libraries.ContainsKey(record.Location))
                    libraries[record.Location].Update(record);
                else
                    libraries.Add(record.Location, new ListDescriptor(record));
            }
        }
        m_documentLibraries = libraries.Values.ToList<ListDescriptor>();
    }
    return m_documentLibraries;
}

}

/// <summary>
/// Gets ths list of unique lists needed for managing
/// physical records as specified by records in the
/// file plan.
/// </summary>
[XmlIgnore]
public List<ListDescriptor> PhysicalRecordLists
{
    get
    {
        if (m_lists == null)
        {
            Dictionary<string, ListDescriptor> lists
                = new Dictionary<string, ListDescriptor>();

            foreach (RecordSpecification record in this.Records)
            {
                if (record.Type != RecordType.Electronic)
                {
                    if (lists.ContainsKey(record.Location))
                        lists[record.Location].Update(record);
                    else
                        lists.Add(record.Location, new ListDescriptor(record));
                }
            }
            m_lists = lists.Values.ToList<ListDescriptor>();
        }
        return m_lists;
    }
}

/// <summary>
/// Creates a task list that is used to notify
/// record managers of outstanding tasks for records
/// that were created by this plan.
/// </summary>
/// <param name="recordCenter">a record center site</param>
```

Continued

Listing 5-6: Records Center configuration from a file plan *(continued)*

```
/// <returns>the associated task list</returns>
public SPList CreateTaskList(SPWeb recordCenter)
{
    string listName = string.Format("{0} Tasks", this.Title);
    string listDescription = string.Format(
        "Records management tasks for file plan '{0}'", this.Title);
    return SharePointList.Create(recordCenter,
        SPListTemplateType.Tasks, listName, listDescription);
}

/// <summary>
/// Creates the content types as specified by records
/// defined in the file plan. A separate content type
/// is created for each record type.
/// </summary>
public void CreateContentTypes(SPWeb recordCenter)
{
    foreach (RecordSpecification record in this.Records)
        record.CreateContentTypes(this, recordCenter);
}

/// <summary>
/// Creates custom lists needed by the records described
/// in the file plan.
/// </summary>
private void CreatePhysicalRepositories(SPWeb recordCenter)
{
    foreach (RecordSpecification record in this.Records)
        record.CreatePhysicalRepository(this, recordCenter);
}

/// <summary>
/// Creates the document libraries specified by records
/// defined in the file plan.
/// </summary>
private void CreateDocumentLibraries(SPWeb recordCenter)
{
    foreach (RecordSpecification record in this.Records)
        record.CreateDocumentLibrary(this, recordCenter);
}

/// <summary>
/// Creates the record routing types identified by this
/// file plan. Each routing type is configured using properties
/// and options contained within the file plan.
/// </summary>
private void CreateRecordSeries(SPWeb recordCenter)
{
    foreach (RecordSpecification record in this.Records)
        record.CreateRecordSeries(this, recordCenter);
}
#endregion
```

Most of the work of executing the file plan is delegated to the `RecordSpecification` object, which is one of the XSD-generated wrapper classes declared in the file plan schema. We can apply the same technique to extend this class to support the additional operations we need. Listing 5-7 shows the `RecordSpecification.Execute` method and the steps involved in configuring the Records Center for a given record type.

Listing 5-7: RecordSpecification.Execute method

```
public bool Execute(FilePlan filePlan, SPWeb recordCenter)
{
    try
    {
        // create or locate the required content types
        CreateContentTypes(filePlan, recordCenter);
        // create lists for physical record types
        CreatePhysicalRepository(filePlan, recordCenter);
        // create libraries for electronic record types
        CreateDocumentLibrary(filePlan, recordCenter);
        // populate the record series table
        CreateRecordSeries(filePlan, recordCenter);
        // return success indicator
        return true;
    }
    catch (Exception x)
    {
        Helpers.HandleException(this, x);
    }
    return false;
}
```

We also need some utility methods to help ensure that the strings we are using are valid for the record series name, the target location, and the name of each content type declared within the file plan. The following routines are declared within the `RecordSpecification` class to check for missing characters and return reasonable default values in case any of the file plan data is invalid:

```
/// <summary>
/// Returns a safe name for the record specification.
/// </summary>
[XmlIgnore]
public string SafeName
{
    get
    {
        return this.Name.Trim().Replace(" ", "_")
            .Replace(";","");
    }
}

/// <summary>
/// Returns a safe location for the record specification.
/// </summary>
[XmlIgnore]
```

```
public string SafeLocation
{
    get
    {
        if (string.IsNullOrEmpty(this.Location))
            return "Unclassified records";
        return this.Location;
    }
}

/// <summary>
/// Calculates the name to be used for content types.
/// </summary>
public string GetContentTypeName(FilePlan filePlan)
{
    return String.Format("{0} {1}", filePlan.Title, this.Name);
}
```

At a minimum, executing a record specification means that a storage location must be provided for incoming records of that type. But the incoming record can also have metadata associated with it that must be promoted to columns in the target document library. The best way to handle incoming metadata is to create a content type for each record specification so that we can add the required metadata columns later in the life cycle. We can also use the content type to attach policies to the incoming record and to execute custom code when the record arrives.

Our strategy for creating the content type is to place it into a group having the name of the file plan and to select the parent content type based on whether the record is an electronic document or a physical one. Physical records will inherit the generic `Item` content type, and electronic records will inherit from `Document`. We create the content types first, because we will later need to bind the content type to the document library or list used to store the associated records. Listing 5-8 shows the `RecordSpecification.CreateContentTypes` method that handles this process.

Listing 5-8: CreateContentTypes method

```
public void CreateContentTypes(FilePlan filePlan, SPWeb recordCenter)
{
    try
    {
        string typeName = GetContentTypeName(filePlan);
        SPContentType type = SharePointContentType.Find(
            recordCenter, typeName);

        // create the type if necessary
        if (type == null)
        {
            string groupName = String.IsNullOrEmpty(filePlan.Title) ?
                "File Plan" : string.Format("File Plan - {0}", filePlan.Title);

            string baseType = this.Type == RecordType.Electronic ?
                "Document" : "Item";

            type = SharePointContentType.Create(recordCenter, typeName,
```

```
        this.Description, groupName, baseType);
    }

    // ensure that the type exists
    if (type == null)
        throw new SPEException(string.Format(
            "Failed to create content type for record: '{0}'",
            this.Name));

    // setup event receivers for physical records so that
    // record managers are notified when records are added to
    // the content database
    if (type != null)
    {
        SPEventReceiverDefinition receiver =
            Helpers.FindOrCreateEventReceiver(
                GetType(), type, SPEventReceiverType.ItemAdded);

        SharePointContentType.AddXmlDocumentString(type,
            FilePlan.FILEPLAN_NAMESPACE,
            CreateReceiverData(filePlan, recordCenter));
    }
}
catch (Exception x)
{
    Helpers.HandleException(this, x);
    throw x;
}
}
```

Physical records are not associated with routing types but can still be tracked using the functions provided by the Records Center. Figure 5-23 shows a sample file plan input form being used to declare a physical record by selecting **Bar Coded** or **Labeled** as the record type.

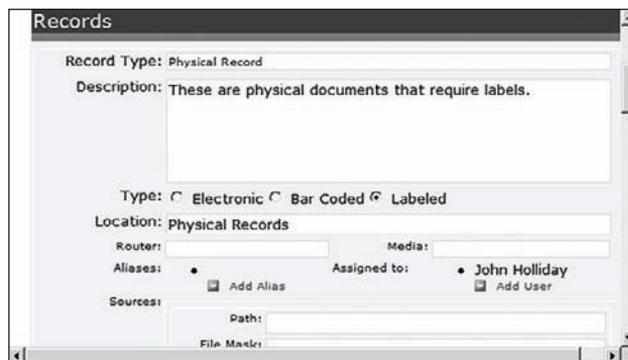


Figure 5-23: Entering a physical record specification.

When the content type is created, we also add an `ItemAdded` event receiver so we can respond to records that arrive in the repository. Since we'll be reusing the same event receiver method for all

Chapter 5: Building and Configuring a Records Repository

record types, we have to prepare some contextual information that can be used by the event receiver method to determine which record specification controls the disposition of the record. We'll create a custom XML fragment to hold the context data, and we'll store it in the content type itself using the `XmlDocuments` collection. Listing 5-9 shows the code that creates the context data, which we add to the content type using our `SharePointContentType` utility class.

Listing 5-9: Contextual data for a record specification

```
private string CreateReceiverData(FilePlan filePlan, SPWeb recordCenter)
{
    XmlDocument xmlDoc = new XmlDocument();
    string ns = FilePlan.FILEPLAN_NAMESPACE;
    new XmlNamespaceManager(xmlDoc.NameTable).AddNamespace("fp", ns);

    XmlElement rootNode = xmlDoc.CreateElement("fp", "FilePlanRecord", ns);
    xmlDoc.AppendChild(rootNode);

    XmlElement node;
    rootNode.AppendChild(node = xmlDoc.CreateElement("fp", "Name", ns));
    node.InnerText = this.SafeName;

    rootNode.AppendChild(node = xmlDoc.CreateElement("fp", "Type", ns));
    node.InnerText = this.Type.ToString();

    rootNode.AppendChild(node = xmlDoc.CreateElement("fp", "TaskListId", ns));
    SPList taskList = filePlan.CreateTaskList(recordCenter);
    node.InnerText = taskList.ID.ToString();

    rootNode.AppendChild(node = xmlDoc.CreateElement("fp", "Users", ns));
    foreach (string user in this.AssignedTo)
    {
        XmlElement userNode = xmlDoc.CreateElement("fp", "User", ns);
        userNode.InnerText = user;
        node.AppendChild(userNode);
    }
    return xmlDoc.OuterXml;
}
```

Now that the content type is created, we can proceed to create the document library for electronic records or the custom list for physical records. After creating the library, we'll configure it with the required content types and set the appropriate flag to ensure that it appears on the Quick Launch navigation bar.

```
/// <summary>
/// Creates a document library for the record type.
/// </summary>
/// <param name="filePlan">a file plan objects</param>
/// <param name="recordCenter">a records center site</param>
public void CreateDocumentLibrary(FilePlan filePlan, SPWeb recordCenter)
{
    if (this.Type == RecordType.Electronic)
    {
        string listName = this.SafeLocation;
```

Chapter 5: Building and Configuring a Records Repository

```
if (String.IsNullOrEmpty(listName))
    listName = this.SafeName;

SPList list = SharePointList.Create(recordCenter,
    SPListTemplateType.DocumentLibrary, listName,
    this.Description);

SPDocumentLibrary library = list == null ?
    null : list as SPDocumentLibrary;

if (library == null)
{
    // either found a list that was not a library, or
    // failed to create a new library
    Helpers.Log(this,
        "Failed to create document library '{0}' in records center '{1}'",
        listName, recordCenter.Title);
}
else
{
    // configure the library with the required content types
    library.ContentTypesEnabled = true;
    library.OnQuickLaunch = true;
    SharePointList.AddContentType(library, GetContentTypeName(filePlan));
    SharePointList.RemoveContentType(library, "Document");
    library.Update();
}
}
}
```

To configure the document library, we enable content types, add the content type we created previously, and remove the default Document content type. Similarly, when creating a custom list for physical records, we remove the default Item content type.

```
/// <summary>
/// Creates a custom list which is used to track physical records.
/// Creates a custom task list for use by the file plan.
/// </summary>
/// <param name="filePlan">a file plan object</param>
/// <param name="recordCenter">a records center site</param>
public void CreatePhysicalRepository(FilePlan filePlan, SPWeb recordCenter)
{
    if (this.Type != RecordType.Electronic)
    {
        string listName = this.SafeLocation;
        if (String.IsNullOrEmpty(listName))
            listName = this.SafeName;
        SPList list = SharePointList.Create(recordCenter,
            SPListTemplateType.GenericList, listName, this.Description);
        if (list == null)
            Helpers.Log(this,
                "Failed to create physical repository for record '{0}'",
                this.Name);
    }
    else
```

```
{
    // configure the list with the record content type
    list.ContentTypesEnabled = true;
    SharePointList.AddContentType(list, GetContentTypeName(filePlan));

    // remove the default "Item" content type
    SharePointList.RemoveContentType(list, "Item");
    list.Update();

    // create a task list for this record
    // (shared by all record types in the plan)
    filePlan.CreateTaskList(recordCenter);
}
}
```

For physical records, we take one additional step. This is where the `ItemAdded` event receiver will play a role. Physical records are not *submitted* to the repository. Instead, they are added directly to the designated list. Whenever this happens, we will automatically create a task that informs the persons responsible for the record type that they must perform a manual task related to that record. For example, it may be necessary to generate a label or a barcode and physically attach it to the record in question. This can be handled any number of ways. In this example, we'll simply create a task and assign it to the first user listed in the record specification. To make it easy to track and locate the tasks that belong to this file plan, we create a task list for each plan as shown in the code above. When a physical record is added to any list using one of the content types created by the file plan, the `ItemAdded` event receiver shown in Listing 5-10 is called.

Listing 5-10: `ItemAdded` event receiver

```
/// <summary>
/// Handles the addition of a physical record entry.
/// </summary>
/// <param name="properties"></param>
public override void ItemAdded(SPItemEventProperties properties)
{
    // Create a new task for the record manager associated with physical
    // record types so that they are reminded to perform tasks outside
    // of SharePoint, such as attaching a barcode or a label.

    DisableEventFiring();
    try
    {
        SPList taskList = null;
        RecordType recordType;
        string recordName;
        SPUser recordManager;

        Helpers.Log(this, "ItemAdded '{0}'", properties.ListItem.Title);
        if (ParseReceiverData(properties, out taskList,
            out recordType, out recordName, out recordManager))
        {
            string title;
```

Chapter 5: Building and Configuring a Records Repository

```
SPListItem item = taskList.Items.Add();
switch (recordType)
{
    case RecordType.Barcoded:
        title = "Add barcode to physical record: ";
        break;
    case RecordType.Labeled:
        title = "Add label to physical record: ";
        break;
    default:
        title = "Record added: ";
        break;
}
item["Title"] = string.Format("{0}{1}", title,
    properties.ListItem.Title);
string description = @"A physical record of type '{0}' " +
    "has been created. You now need to generate a {1} " +
    "and attach it to the document so it can be tracked.";
item["Body"] = string.Format(description, recordName,
    recordType == RecordType.Barcoded ? "BARCODE" : "LABEL");
if (recordManager != null)
    item["AssignedTo"] = recordManager;
item.SystemUpdate();
}
}
catch (SPException x)
{
    Helpers.HandleException(this, x);
}
EnableEventFiring();
}
```

The following method parses the receiver data that we created earlier to extract the key elements needed to determine the type of record, the ID of the target task list, and the login name of the user who is responsible for managing records of this type:

```
private bool ParseReceiverData(SPItemEventProperties properties,
    out SPList taskList, out RecordType recordType,
    out string recordName, out SPUser recordManager)
{
    taskList = null;
    recordType = RecordType.Labeled;
    recordManager = null;
    recordName = string.Empty;
    SPWeb recordCenter = properties.ListItem.Web;
    SPContentType ct = properties.ListItem.ContentType;

    try
    {
        string xmlData = SharePointContentType.GetXmlDocumentString(
            ct, FilePlan.FILEPLAN_NAMESPACE);
        XmlDocument xmlDoc = new XmlDocument();
        xmlDoc.LoadXml(xmlData);
        XmlNamespaceManager nsmgr = new XmlNamespaceManager(xmlDoc.NameTable);
```

Chapter 5: Building and Configuring a Records Repository

```
nsmgr.AddNamespace("fp", FilePlan.FILEPLAN_NAMESPACE);
XmlNode node = xmlDoc.SelectSingleNode("fp:FilePlanRecord/fp:Name", nsmgr);
recordName = node.InnerText;
node = xmlDoc.SelectSingleNode("fp:FilePlanRecord/fp:Type", nsmgr);
recordType = (RecordType)Enum.Parse(typeof(RecordType), node.InnerText);
node = xmlDoc.SelectSingleNode("fp:FilePlanRecord/fp:TaskListId", nsmgr);
taskList = recordCenter.Lists[new Guid(node.InnerText)];
XmlNodeList nodes = xmlDoc.SelectNodes("fp:FilePlanRecord/fp:Users/
fp:User", nsmgr);
if (nodes != null && nodes.Count > 0)
{
    string userName = nodes[0].InnerText;
    foreach (SPUser user in recordCenter.SiteUsers)
        if (user.LoginName.ToLower().Contains(userName.ToLower()))
        {
            recordManager = user;
            break;
        }
}
return true;
}
catch (Exception x)
{
    Helpers.HandleException(this, x,
        "Failed to parse receiver data for item '{0}'",
        properties.ListItem.Title);
}
return false;
}
```

Incoming documents will now be routed to the correct location as specified in the plan, and physical documents can be tracked easily using the built-in task management functionality as illustrated in Figure 5-24.



The screenshot displays a web interface for task management. At the top, the breadcrumb path is 'ECM2007 Record Center > Sample Plan Tasks > Add barcode to physical record: Archival Policies for 2009'. The main heading is 'Sample Plan Tasks: Add barcode to physical record: Archival Policies for 2009'. Below the heading is a toolbar with buttons for 'New Item', 'Edit Item', 'Delete Item', 'Manage Permissions', and 'Alert Me', along with a 'Close' button. The task details are presented in a table-like format:

Title	Add barcode to physical record: Archival Policies for 2009
Priority	(2) Normal
Status	Not Started
% Complete	
Assigned To	MOSSDEV\john.holliday
Description	A physical record of type 'Physical_Record' has been created. You now need to generate a BARCODE and attach it to the document so it can be tracked.
Start Date	12/23/2008
Due Date	

At the bottom, it shows the creation and modification history: 'Created at: 12/23/2008 12:15 AM by System Account' and 'Last modified at: 12/23/2008 12:15 AM by System Account', with a 'Close' button.

Figure 5-24: File plan task detail.

Finally, if the record specification describes an electronic record type, we can set up the record series table entry as shown in Listing 5-11. For this to work, the document library must already exist. We grab the list of aliases for the record, as well as the name of any custom router that should be associated with the record series.

Listing 5-11: Creating the record series table entry

```
/// <summary>
/// Creates the record series table entry for this record type.
/// </summary>
/// <param name="filePlan"></param>
/// <param name="recordCenter"></param>
public void CreateRecordSeries(FilePlan filePlan, SPWeb recordCenter)
{
    if (this.Type == RecordType.Electronic)
    {
        // access the routing table for the records center
        RecordSeriesCollection routingTable =
            new RecordSeriesCollection(recordCenter);

        routingTable.Add(this.Name, this.SafeLocation, this.Description,
            this.Aliases, this.Router, false);
    }
}
```

At this point, we have an InfoPath form that produces XML data that can be serialized and de-serialized using our custom File Plan API. We can also create sample data easily using the rich InfoPath 2007 user interface that will ensure that the data always conforms to our file plan schema. In the next step, we will build a SharePoint feature to publish the form into a special location within the Records Center site so it can be used by a records manager to create executable file plans.

Building the File Plan Gallery

Now that we have a file plan schema and a simple form for creating and editing file plan documents, we can proceed to create a feature that can be activated within any Records Center site. When the feature is activated, it will create a File Plan gallery as a form library and then attach custom actions that enable the administrator to execute custom commands on a given file plan document.

First, we need a content type that describes the file plan itself. Creating a content type will allow us to associate the `Execute` command with individual file plan documents using a second custom action that will appear on the Edit Control Block (ECB) dropdown menu. The following CAML code declares the content type and associates it with the `FilePlan.xsn` file as the document template:

```
<!-- Declare a 'FilePlan' content type that specifies the InfoPath form as its
document template. -->
<ContentType
    ID="0x01010100D7412981EC6441FA8F805FBA212C7E05"
```

Chapter 5: Building and Configuring a Records Repository

```
Name="File Plan"
Description="Describes how to store official records"
Version="0"
Group="ECM2007"
>
  <DocumentTemplate TargetName="FilePlan.xsn" />
</ContentType>
```

Since the content type references the FilePlan.xsn file, we need to provision the file to the appropriate place so that it will load properly when a new file plan document is created. To do this, we add a *module* element that places the file in the proper location.

```
<!-- Declare a module to provision the document template for the content type. -->
<Module Name="FilePlanResources" Url="_cts/File Plan" RootWebOnly="TRUE">
  <File Url="FilePlan.xsn" Type="Ghostable"/>
</Module>
```

I generally prefer to create content types in code rather than declaratively as we've done here because it's faster and I don't have to worry about field identifiers and typos that inevitably happen when you're working with XML. In this case, however, we need the content type identifier in order to declare the custom action, and there is no programmatic way to create content types with a specific identifier, nor is there a way to programmatically create custom actions, so we are forced to use CAML.

With the content type declared, we can add a custom action to its ECB that points to a File Plan Execution page that we will create shortly.

```
<!--
Add a custom command to the FilePlan ECB that enables a plan
administrator to execute the plan against the current site.
-->
<CustomAction
  Id="FilePlan.Execute"
  Location="EditControlBlock"
  RegistrationType="ContentType"
  RegistrationId="0x01010100D7412981EC6441FA8F805FBA212C7E05"
  ImageUrl="/_layouts/images/CheckValues.gif"
  Title="Execute File Plan"
  Sequence="120"
  Description="Applies the rules contained in the selected file plan to the
  current records center site.">
  <UrlAction Url="/_layouts/ECM2007/FilePlanExecute.aspx?ItemId=
  {ItemId}&ListId={ListId}"/>
</CustomAction>
```

We'll create the gallery itself using a `ListInstance` element, but we won't bind it to the content type in the declarative CAML code. Instead, we'll write some code in the feature receiver to handle that and take care of a few additional loose ends, like setting up the document template that will point to our custom InfoPath form. Listing 5-12 shows the CAML code needed for the `ListInstance` declaration.

Listing 5-12: ListInstance declaration

```
<!--
Create a form library to hold the file plan forms, which
are InfoPath 2007 forms based on the file plan schema.
-->
<ListInstance
  FeatureId="00BFEA71-1E1D-4562-B56A-F05371BB0115"
  TemplateType="115"
  Description="Use the file plan gallery to create file plans for managing
  official records. You can execute a file plan to create the record
  series, lists and document libraries associated with the document types
  identified in the plan."
  OnQuickLaunch="True"
  Title="File Plans"
  Url="FilePlan" />
```

Listing 5-13 shows the implementation of the `FeatureReceiver` class that takes care of configuring the File Plan gallery list instance after the feature is activated.

Listing 5-13: File plan FeatureReceiver

```
using System;
using System.Diagnostics;
using ECM2007.ContentTypes;
using Microsoft.SharePoint;

namespace ECM2007.FilePlanFeature
{
    /// <summary>
    /// Handles events during feature installation and activation.
    /// </summary>
    public class FeatureReceiver : SPFeatureReceiver
    {
        const string TRACE_CATEGORY = "ECM2007.FilePlanFeature";
        const string FILEPLANTYPE = "File Plan";
        const string FILEPLANTEMPLATE = "FilePlan.xsn";
        const string FILEPLANGALLERY = "File Plans";

        /// <summary>
        /// Configure the site collection on feature activation.
        /// </summary>
        /// <param name="properties"></param>
        public override void FeatureActivated(SPFeatureReceiverProperties
            properties)
        {
            SPSite siteCollection = properties.Feature.Parent as SPSite;
            try
            {
                // Configure the file plan gallery.
            }
        }
    }
}
```

Continued

Listing 5-13: File plan FeatureReceiver (continued)

```
using (SPWeb site = siteCollection.RootWeb)
{
    // Locate the Form content type.
    SPContentType formType = SharePointContentType.Find(site,
        "Form");
    if (formType == null)
        Fail("Form content type not found.");

    // Locate the FilePlan content type.
    SPContentType filePlan = SharePointContentType.Find(site,
        FILEPLANTYPE);
    if (filePlan == null)
        Fail("Content Type Not Found: '{0}'", FILEPLANTYPE);

    // Locate the gallery.
    Log("Configuring the file plan gallery");
    SPDocumentLibrary gallery = SharePointList.Find(site,
        FILEPLANGALLERY)
        as SPDocumentLibrary;
    if (gallery == null)
        Fail("File Plan Gallery not found");

    // Enable content types and add the file plan type to the list.
    gallery.ContentTypesEnabled = true;
    gallery.ContentTypes.Add(filePlan);

    // Remove the default "Form" from the content types in
    // the gallery.
    gallery.ContentTypes.Delete(formType.Id);

    // Setup the remaining properties.
    gallery.EnableVersioning = true;
    gallery.EnableFolderCreation = false;
    gallery.ForceCheckout = true;
    gallery.Update();
}
}
catch (Exception x)
{
    HandleException("FeatureActivated", x);
}
}

/// <summary>
/// For testing purposes, try to remove the file plan gallery upon
/// deactivation,
/// but only if the list is empty.
/// </summary>
/// <param name="properties"></param>
public override void FeatureDeactivating(SPFeatureReceiverProperties
    properties)
{
```

Chapter 5: Building and Configuring a Records Repository

```
SPSite siteCollection = properties.Feature.Parent as SPSite;
try
{
    #if DEBUG
        SPList list = SharePointList.Find(siteCollection.RootWeb,
            FILEPLANGALLERY);
        if (list != null && list.Items.Count == 0)
            SharePointList.Delete(siteCollection.RootWeb, FILEPLANGALLERY);
    #endif
}
catch (Exception x)
{
    HandleException("FeatureDeactivating", x);
}
}

public override void FeatureInstalled(SPFeatureReceiverProperties
    properties)
{
}

public override void FeatureUninstalling(SPFeatureReceiverProperties
    properties)
{
}

/// <summary>
/// Fails with an error message.
/// </summary>
/// <param name="format"></param>
/// <param name="args"></param>
void Fail(string format, params object[] args)
{
    throw new Exception(String.Format(format, args));
}

/// <summary>
/// Logs a diagnostic message.
/// </summary>
void Log(string format, params object[] args)
{
    Trace.WriteLine(String.Format(format, args), TRACE_CATEGORY);
}

/// <summary>
/// Generic exception handler.
/// </summary>
void HandleException(string scope, Exception x)
{
    Log("Exception occurred: {0}\n{1}", scope, x.ToString());
}
}
}
```

Chapter 5: Building and Configuring a Records Repository

To allow the user to navigate to the File Plan gallery, we'll add a link to the Site Settings page in the appropriate section as shown in Figure 5-25. Listing 5-14 shows the CAML code needed to set this up.

Listing 5-14: CustomAction link to the File Plan gallery

```
<!-- Add a command to the 'Galleries' section of the site settings page to
  enable navigation to the file plan gallery. -->
<CustomAction
  Id="FilePlanGallery"
  Location="Microsoft.SharePoint.SiteSettings"
  GroupId="Galleries"
  Title="File plans"
  Description="File Plan Gallery"
  Sequence="100"
>
  <UrlAction Url="/FilePlan/forms/allitems.aspx"/>
</CustomAction>
```

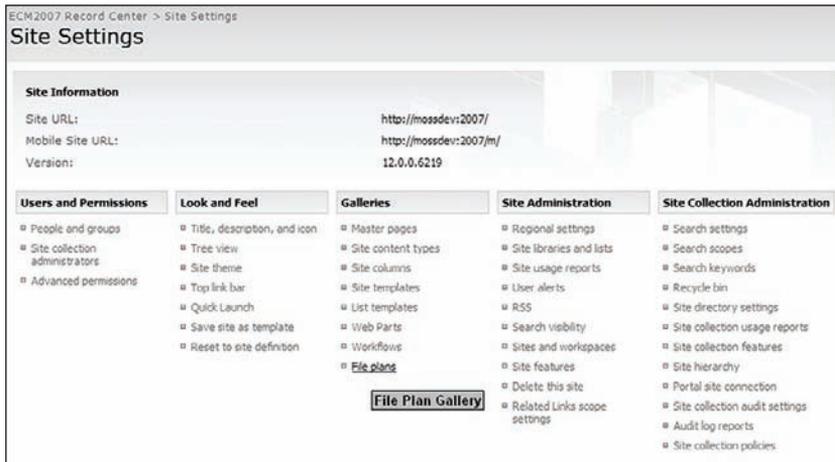


Figure 5-25: File Plan gallery link.

As with any SharePoint feature that declares a feature receiver, after building the assembly we must ensure that it is installed properly in the Global Assembly Cache and that the feature XML is copied into the SharePoint 12 hive. This must be done before we can activate the feature. Listing 5-15 shows the install.bat file that is invoked from the post-build event to handle the required feature deployment steps.

This script uses an APPPOOL environment variable to specify the application pool to reset after the feature is installed. If you are building the code supplied with the book, then remember to set this variable to the appropriate application pool name.

Listing 5-15: Post-build event installation batch script

```
@SET SPROOT="C:\Program Files\Common Files\Microsoft Shared\Web Server  
Extensions\12"  
@SET STSADM="C:\Program Files\Common Files\Microsoft Shared\Web Server  
Extensions\12\BIN\STSADM"  
@SET GACUTIL="C:\Program Files\Microsoft SDKs\Windows\v6.0A\bin\gacutil.exe"  
@SET IISAPP="C:\Windows\System32\iisapp.vbs"  
  
@SET APPPOOL="ECM2007_RecordCenter"  
  
@Echo Installing %1 Assembly in the GAC  
%GACUTIL% -if %2%  
  
@Echo Copying files to the 12 Hive  
XCOPY /e /y 12\* %SPROOT%  
  
@Echo Installing Feature "ECM2007.FilePlanFeature"  
%STSADM% -o installfeature -name ECM2007.FilePlanFeature -force  
  
@Echo Recycling SharePoint Application Pools  
CSCRIPT %IISAPP% /a %APPPool% /r  
  
@Echo Installation Complete  
@SET APPPOOL=
```

Once activated, the new file plan content type is created and attached to the File Plan gallery as shown in Figure 5-26.

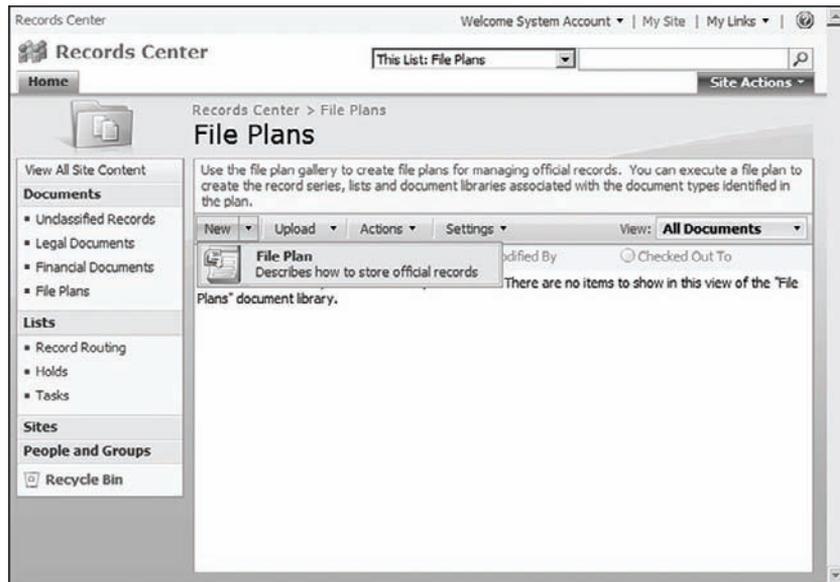


Figure 5-26: File Plan gallery.

Executing the File Plan

To execute a file plan, we'll need a custom ASPX page with code-behind. To invoke the operation, the plan administrator will use the ECB associated with each file plan instance, as shown in Figure 5-27.

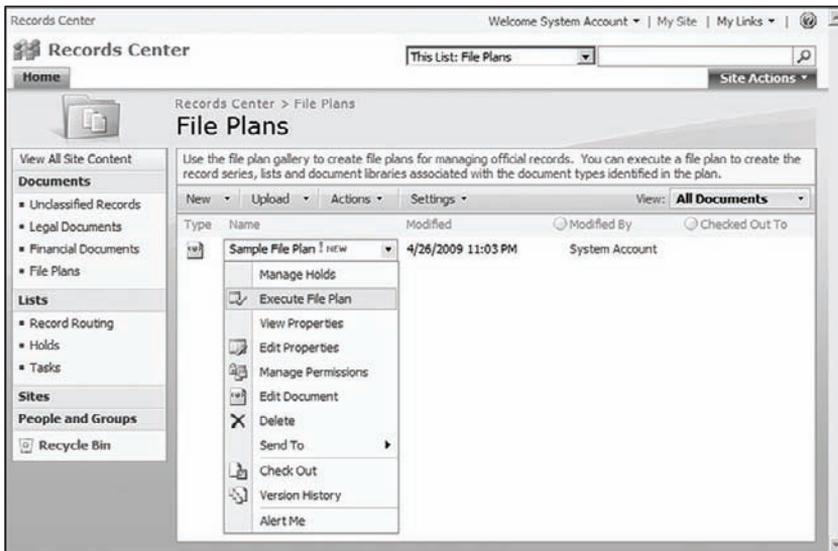


Figure 5-27: Execute File Plan ECB command.

We will perform the execution in two steps so that we can allow the plan administrator to review the plan details before committing to the execution process. We can also display a list of the record types that are affected by the plan and let the administrator select which ones to process. Listing 5-16 shows the markup for the Plan Execution page.

Listing 5-16: File Plan Execute page

```
<%@ Assembly Name="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
  PublicKeyToken=71e9bce111e9429c" %>
<%@ Assembly Name="ECM2007.FilePlanFeature, Version=1.0.0.0, Culture=neutral,
  PublicKeyToken=eb8a6a1622425a15" %>

<%@ Page Language="C#" MasterPageFile="~/_layouts/application.master"
  Inherits="ECM2007.FilePlanFeature.FilePlanExecute"
  EnableViewState="true" EnableViewStateMac="true" %>
<!------->
<%@ Register TagPrefix="SharePoint"
  Namespace="Microsoft.SharePoint.WebControls"
  Assembly="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
  PublicKeyToken=71e9bce111e9429c" %>

<%@ Register TagPrefix="Utilities"
  Namespace="Microsoft.SharePoint.Utilities" Assembly="Microsoft.SharePoint,
  Version=12.0.0.0, Culture=neutral, PublicKeyToken=71e9bce111e9429c" %>
```

Chapter 5: Building and Configuring a Records Repository

```
<!------->
<%@ Register TagPrefix="wssuc"
    TagName="InputFormSection"
    Src="~/_controltemplates/InputFormSection.ascx" %>

<%@ Register TagPrefix="wssuc"
    TagName="InputFormControl"
    Src="~/_controltemplates/InputFormControl.ascx" %>

<%@ Register TagPrefix="wssuc"
    TagName="ButtonSection"
    Src="~/_controltemplates/ButtonSection.ascx" %>
<!------->
<asp:Content ID="PageTitle" runat="server"
    ContentPlaceHolderID="PlaceHolderPageTitle">
    Execute File Plan
</asp:Content>
<!------->
<asp:Content ID="PageTitleInTitleArea" runat="server"
    ContentPlaceHolderID="PlaceHolderPageTitleInTitleArea">
    Execute File Plan '<asp:Label ID="FilePlanPageTitle" runat="server" />'
</asp:Content>

<asp:Content ID="Main" runat="server"
    ContentPlaceHolderID="PlaceHolderMain">
    <p>
        Executing a file plan automatically creates the content types,
        lists, document libraries and information policies needed
        to manage the record types identified by the plan.
        After these items are created, the routing table is
        populated with matching record series entries.
    </p>
    <table border="0" cellspacing="0" cellpadding="0" width="100%">
    <tr>
        <td>
            <!-- Plan Details -->
            <wssuc:InputFormSection runat="server"
                Title="Plan Details">
                <Template_InputFormControls>
                <tr>
                    <td>
                        <asp:Label ID="PlanAuthor" runat="server" /><br />
                        <asp:Label ID="PlanDescription" runat="server" /><br />
                    </td>
                </tr>
                </Template_InputFormControls>
            </wssuc:InputFormSection>

            <!-- Record Type List -->
            <wssuc:InputFormSection runat="server"
                Title="Official Records"
                Description="Select the records you wish to process using the
                    checkboxes provided.">
                <Template_InputFormControls>
```

Continued

Listing 5-16: File Plan Execute page (continued)

```
<tr>
  <td>
    <asp:CheckBoxList runat="server"
      ID="chkboxlistRecordTypes" />
  </td>
</tr>
</Template_InputFormControls>
</wssuc:InputFormSection>

<!-- Buttons -->
<wssuc:ButtonSection runat="server">
  <Template_Buttons>
    <asp:Button UseSubmitBehavior="false" runat="server"
      class="ms-ButtonHeightWidth" Text="Execute"
      id="btnExecute" Enabled="true" />
  </Template_Buttons>
</wssuc:ButtonSection>
</td>
</tr>
</table>
</asp:Content>
```

To minimize the amount of work we have to do, we can apply the standard application.master master page and create a code-behind page class that inherits from `LayoutsPageBase` as shown in Listing 5-17. We can also take advantage of some SharePoint-provided user controls to display the plan details and the buttons used to start the plan execution. These are brought into the page from the `_controltemplates` folder using the `wssuc` tag prefix. The controls we need are the `InputFormSection`, `InputFormControl`, and `ButtonSection` controls shown in the listing. These are template controls that contain the actual server controls we'll populate from the code-behind.

Listing 5-17: File plan execution

```
public class FilePlanExecute : LayoutsPageBase
{
    protected Label FilePlanPageTitle;
    protected Label PlanDescription;
    protected Label PlanAuthor;
    protected CheckBoxList chkboxlistRecordTypes;
    protected Button btnExecute;
    private FilePlan _filePlan;

    protected override void OnLoad(EventArgs e)
    {
        // get the current site and web
        SPWeb site = this.Web;

        // get the source list
        string listId = Request.QueryString["ListId"];
        SPList list = site.Lists[new Guid(listId)];

        // get the current list item
```

```
string itemId = Request.QueryString["ItemId"];
SPListItem item = list.Items.GetItemById(Convert.ToInt32(itemId));

// install the click handler
btnExecute.Click += new EventHandler(btnExecute_Click);

// load the file plan from the item.
_filePlan = FilePlan.Load(item);

FilePlanPageTitle.Text = _filePlan.Title;
PlanAuthor.Text = "Plan author: " + _filePlan.Author;
if (!string.IsNullOrEmpty(_filePlan.Description))
    PlanDescription.Text = "Plan description: " + _filePlan.Description;

// populate the check box list with the names of
// the records identified by the plan
foreach (RecordSpecification record in _filePlan.Records)
    chkboxlistRecordTypes.Items.Add(new ListItem(record.Name));
}
```

When the page is loaded, we retrieve the source list and list item from the request parameters supplied by our custom action. Then we wire up the button `click` event and load the file plan from the item.

The `FilePlan.Load` method is a static utility method we added to the `FilePlan` object that we generated from the file plan schema using the `XsdClassGenerator` custom tool. This method handles the de-serialization of a `FilePlan` object from the `SPFile` object associated with a list item.

Next, we retrieve the plan title and description and store them in the appropriate controls. Finally, we iterate through the list of `RecordSpecification` objects in the plan and populate the checkbox list so that the plan administrator can select which ones to process. Figure 5-28 shows the resulting page within the SharePoint environment.



Figure 5-28: File Plan Execution page.

Chapter 5: Building and Configuring a Records Repository

When the user clicks on the Execute button, we want two things to happen. First, we want the page to go into a holding pattern similar to all the other pages within SharePoint that inform the user to wait while the operation is in progress. We need this because we don't know how long it will take to process all the records in the file plan. Since we'll potentially create lots of content types, lists, and other objects for a given record specification, we anticipate that this could take a long time. Second, we want to redirect the user back to the home page when the execution is completed. If errors occur, we also want the option to redirect the user to some other page containing diagnostic information.

Fortunately, there is an easy way to accomplish this. The SharePoint API provides a utility class called `SPLongOperation` that can be invoked on an application page to inform the user that a potentially long operation is in progress. It automatically changes the page image to an animated rotating icon and handles the threading needed to set up background processing for our code. Listing 5-18 shows the `btnExecute_Click` method using this technique.

Listing 5-18: File plan execution method

```
/// <summary>
/// Handles the execute button click.
/// </summary>
/// <param name="sender"></param>
/// <param name="e"></param>
void btnExecute_Click(object sender, EventArgs e)
{
    Log("Executing file plan: {0}", _filePlan.Title);
    using (SPLongOperation operation = new SPLongOperation(this.Page))
    {
        bool failed = false;
        string message = "File plan execution failed.";
        List<string> recordsToProcess = new List<string>();

        // process only the selected record types
        foreach (ListItem item in chkboxlistRecordTypes.Items)
            if (item.Selected)
                recordsToProcess.Add(item.Text);

        operation.Begin();

        try
        {
            foreach (string recordName in recordsToProcess)
            {
                RecordSpecification record = _filePlan.Find(recordName);
                if (record != null)
                    failed = failed || !record.Execute(_filePlan, this.Web);
            }
        }
        catch (Exception x)
        {
            failed = true;
            message = String.Format(
                "Exception occurred during file plan execution: {0}",
                x.ToString());
        }
    }
}
```

```

    }

    if (failed)
        Log(message);

    operation.End(this.Web.Url);
}
}

```

First, we create a list of the record names the plan administrator has selected for processing, and then we start the operation, keeping track of whether each `RecordSpecification` object succeeds or fails execution against the current `SPWeb`.

In this example, we simply display a message on failure. Alternatively, we could redirect the user to a diagnostic page by passing the URL to the `SPLongOperation.End` method.

After all is said and done, we should end up with a fully populated Record Routing Table along with the content types and document libraries or lists needed to manage the records according to the file plan. Figure 5-29 shows the resulting routing table. The big advantage here is that we are able to coordinate all the required steps by driving the creation of the individual components from a single schematized data file. This will allow us to extend the solution further in later chapters when we enhance the file plan definition to include support for additional operations such as auditing and document retention.

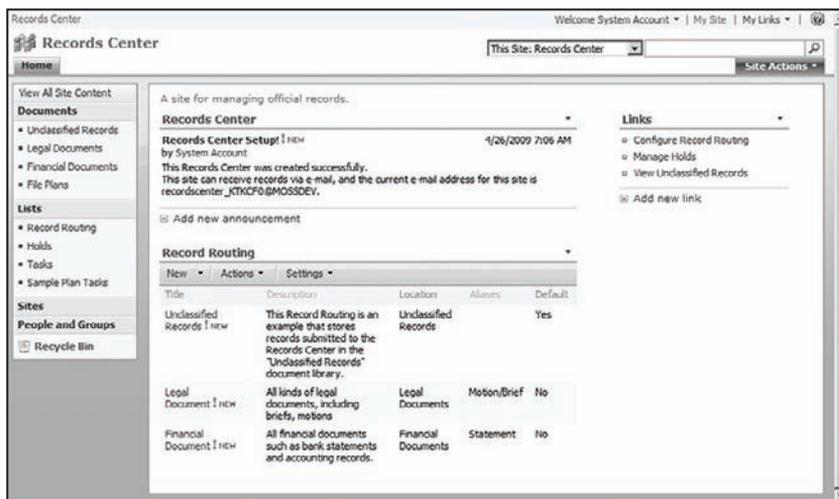


Figure 5-29: Fully configured file plan routing table.

Summary

This chapter described the process of creating a Records Center site using both the SharePoint user interface and the Records Management API. Building a Records Center site involves creating a SharePoint site based on the Records Center site template, but before records can be submitted to

Chapter 5: Building and Configuring a Records Repository

the site, additional configuration steps must be performed. These include creating the appropriate content types, document libraries, and Record Routing Table entries that describe each record type that the Records Center will recognize and process.

Configuring a Records Center site requires that the information contained in a file plan worksheet must be translated into the corresponding components within the SharePoint environment. Because this information typically includes not only the high-level descriptions of record types, but the low-level descriptions of required metadata, the steps required to configure the Records Center can be tedious and prone to error.

InfoPath 2007 provides many options for creating electronic forms, including the ability to design a form based on a predefined XML schema. One strategy for simplifying the steps needed to set up a Records Center properly is to create a dynamic file plan that is based on an XML schema that describes the type and structure of data used at each processing stage for a given record type. This is essential for building code components that can operate on that content in a consistent manner.

Using a dynamic file plan offers several other advantages, including the ability to capture the essential elements of the file plan in a form that can be automated using the Records Management API, and capturing additional information that can drive external processes such as workflow activities and scheduled processes.

The chapter demonstrated the construction of a SharePoint feature that extends the Site Settings page to include a link to a File Plan gallery that enables a plan administrator to create a file plan using InfoPath 2007 and execute it within the Records Center site to create the required components automatically.

6

Populating the Records Repository

In Chapter 4, we examined the structure of the Records Center and looked at the records processing architecture. In Chapter 5, we learned how to set up and configure a Records Center site. In this chapter, we will explore the different ways in which official records can be submitted. This will include two scenarios:

- ❑ Submitting records one at a time from within a document library
- ❑ Submitting records in a batch using a custom application page

Submitting Individual Records

SharePoint enables individual users to select a document in a library and send it to a Records Repository via the Edit Control Block (ECB) menu attached to the library. The `Send To` command displays the name of the Records Center site. Selecting this command sends the selected document to the repository, displaying the result (success or failure) in a separate page. In order for this process to work, the farm must first be configured by the farm administrator to communicate with the Records Center site, and the Records Center site itself must be configured to give the user permission to do so.

Configuring the Farm for Manual Submission

In Chapter 5, we created our Record Center site and configured the record routing types. Now we can complete the configuration of the SharePoint Farm so that users can send documents to the repository easily from the SharePoint UI.

Open the Central Administration web site. Navigate to the Application Management page and select “Records center” in the “External Service Connections” section. On the Configure

Chapter 6: Populating the Records Repository

Connection to Records Center page, click “Connect to a Records Center.” Enter the URL of the Official File Web Service, using the URL of the Records Center site we created in Chapter 5. For example, if the Records Center is located at `http://mossdev:9995/`, then the Official File Web Service URL would be `http://mossdev:9995/_vti_bin/officialfile.asmx`. Enter the appropriate URL and give the Records Center a title, such as *Records Center*. This is the name that will appear as a location under the *Send To* menu item on the ECB for all document items in all document libraries in the farm.

Notice that we are entering the full path to the Web Service endpoint that includes the web application and port number we configured earlier. To verify that you have entered the URL correctly, simply copy and paste it into the address bar of the web browser. You should see the Web Service Definition Language (WSDL) page as shown in Figure 6-1.

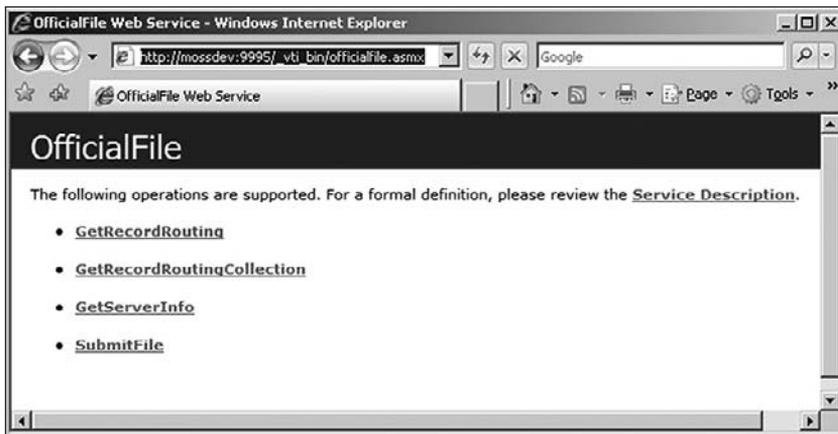


Figure 6-1: Official File Web Service page.

One limitation of this architecture is that there is only one Records Center URL we can define for the entire farm. So if we had multiple Records Center sites, we would have to use a custom action of our own in order to extend the ECB dropdown menu to allow users to select from the list of different Records Center sites. The problem with that approach is that all of the additional processing that SharePoint performs prior to sending the file would then have to be duplicated in our code.

SharePoint delegates the task of preparing a file for submission to a special ASPX page located in the LAYOUTS folder called *SendToOfficialFile.aspx*. The parameters that are passed to this page include the source URL of the file to be submitted. The page then retrieves the referenced file and then, in turn, delegates its processing to the `SPFile.SendToOfficialFile` method. This method extracts the content type from the file and then invokes the Official File Web Service indirectly using the internal class `Microsoft.SharePoint.OfficialFile`. We'll look at how this works a bit later when we examine how to submit files programmatically. First, we'll complete the configuration of the Records Center site and then test it to make sure the farm is properly set up to enable manual file submission.

Granting Users Permission to Submit Records

SharePoint requires that users must be granted special permission before they are allowed to submit files to a Records Center site through the Official File Web Service. This additional layer of

protection is incorporated directly into the records processing infrastructure that is part of the Records Management API.

During the record processing sequence, the Records Center checks whether the login name supplied with the incoming file matches the name of a user who belongs to the Records Center Web Service Submitters group. To add a user to this group, use the “People and Groups” link on the Records Center home page, and then click the More link in the Groups section of the navigation bar on the left side of the page, as shown in Figure 6-2.

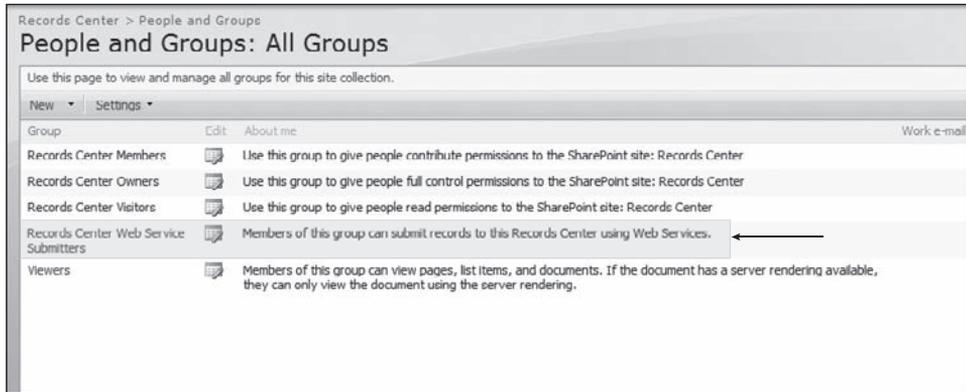


Figure 6-2: Records Center Web Service Submitters group.

Testing the Farm Configuration

Now we can test our work by submitting a few documents from various places to our Records Repository.

1. Open the browser and navigate to the local farm. Select “Site Settings” from the Site Actions menu and open the Content Types gallery for the site.
2. Create the following content types:

Type	Parent	Group
Brief	Document	Legal Record Types
Contract	Document	Legal Record Types
Motion	Document	Legal Record Types
Annual Report	Document	Financial Record Types
Financial Statement	Document	Financial Record Types

The Content Type gallery should now resemble the one shown in Figure 6-3.

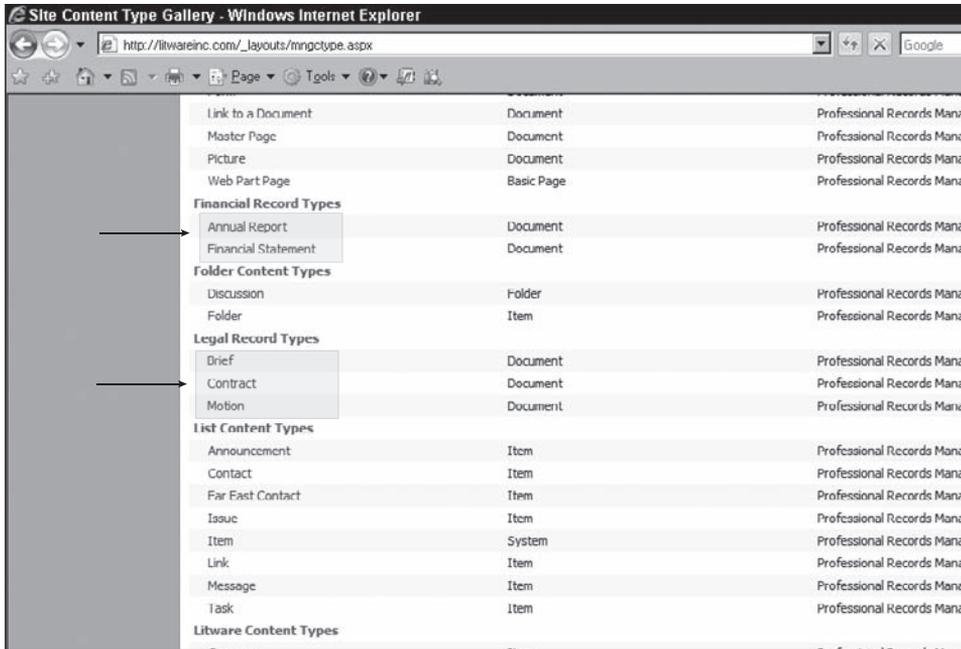


Figure 6-3: Content Type gallery.

3. Click Site Actions ⇄ Create and create a new document library called *Shared Documents* if it does not already exist.
4. Open the Document Library Settings page and click on the “Advanced Settings” link. Select “Allow management of content types” and click OK.
5. On the Customize Shared Documents page, click “Add from existing site content types” in the “Content Types” section.
6. On the Add Content Types: Shared Documents page, select the content types you just created from the list and click on the Add button.
7. Now you can navigate back to the Shared Documents page and upload some documents according to the type of document it is. Figure 6-4 shows a sampling of documents.



Figure 6-4: Shared documents.

8. Select any document and then choose Send To ⇨ Records Center from the context menu.
9. When the operation has completed, navigate back to the Records Center site home page. You should now see the document in the appropriate library, depending on its content type. For example, if the Records Center includes the routing type shown in Figure 6-5 and a document with a content type of Annual Report is submitted, then the Financial Documents document library in the Records Center would receive the record, as shown in Figure 6-6.

Figure 6-5: Financial Documents routing type.

Type	Name	Modified	Modified By	Hold Status
	2009-04-29T11-41-02Z	4/29/2009 7:41 AM	System Account	

Figure 6-6: Financial Documents library contents.

Submitted records are not moved from one document library to another. They are copied. This means that official records that are stored in the Records Center represent only a snapshot of the document at the time it was submitted.

Let's go back to the Records Center and take a closer look at what happens on the receiving end whenever a document is submitted. First of all, notice that the document library is organized by folder, using the current day and time to name the folder.

Chapter 6: Populating the Records Repository

The date format used here is the `xsd:dateTime` format, which is defined in Chapter 5.4 of ISO 8601, the international standard for date and time representations. For more information, see <http://www.iso.org>. This format is a combination of a date and a time and has the following layout: `[-] CCYY-MM-DDThh:mm:ss [Z | (+ | -) hh:mm]`, where the optional `Z` suffix indicates the time zone.

If we drill into this folder, we see the actual item as shown in Figure 6-7, with some additional characters appended so that it can be tracked as a unique object. If you click on it, it will open the file for viewing or editing.

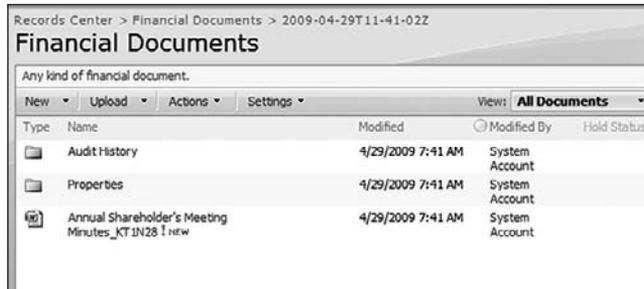


Figure 6-7: Contents of folder containing submitted files.

Notice also that there is a subfolder called *Properties*. There may also be an audit history folder if auditing information was provided along with the submitted file. Let's click on the *Properties* folder to take a closer look inside.

Figure 6-8 shows the XML document that describes the record's metadata properties after it has been submitted. The filename matches that of the submitted file, but the file type is XML. Clicking on the link opens the file on the client machine.



Figure 6-8: XML file containing record properties.

Figure 6-9 shows an example of the contents of a property file. If you scroll to the bottom of this file, you can see that the record routing type name is included, along with the source URL and the user who submitted the file. The other properties are all of the columns and column values that were associated with the file within the source document library.

```

- <Property>
  <Name>_Comments</Name>
  <Value />
  <Type>String</Type>
</Property>
- <Property>
  <Name>ContentType</Name>
  <Value>Annual Report</Value>
  <Type>String</Type>
</Property>
</Properties>
<RecordRouting>Annual Report</RecordRouting>
<SourceUrl>http://mossdev:108/Shared Documents/Annual Shareholder's Meeting
Minutes.docx</SourceUrl>
<UserLoginName>SHAREPOINT\system</UserLoginName>
</RecordsRepositorySubmission>

```

Figure 6-9: Record properties.

Notice that the RecordRouting type name in this example is Annual Report, and not Financial Document. The type name is the value that was supplied when the file was submitted, and not the name of the record routing list item in the Record Routing Table.

The document properties were extracted from the document and copied into this file. This system allows the original metadata to be processed independently of the document and supports post-processing of the metadata from external code, including workflow activities and timer jobs. We can also surface selected properties automatically into the target document library by defining matching columns either in the library itself or on the content type associated with the incoming record. SharePoint will automatically promote these properties into the library.

This type of property promotion works for simple types, such as Text, Integer, and DateTime. It does not work for complex types such as User and Lookup without custom coding.

In a typical records management scenario, we're talking about thousands of documents that may be coming into the Repository. They won't be coming in all at once, as people are continually interacting with documents, and various workflows are completing their life cycles, and so on. Therefore, compliance officers may need additional collaboration and search tools for finding and performing planned operations on different kinds of records. However, as solution developers, we're mainly concerned with getting documents into the Repository as efficiently as possible. In the following sections, we'll look at the tools we have for accomplishing this.

Submitting Records Programmatically

In Chapter 3, we saw how to submit files to a remote repository using the Official File Web Service. You would typically use that kind of code when writing applications that run on client machines where SharePoint is not also running. You might also use the same approach when writing SharePoint features that include commands that enable users to submit records. However, the server-side SharePoint API provides a shortcut that makes the process of submitting records more direct and straightforward.

Chapter 6: Populating the Records Repository

Listing 6-1 shows the code needed to submit a file located in a document library to the Records Center site that is currently configured for the farm. Unlike the raw Official File Web Service approach you would have to use from a client application, where the Records Center site could be located anywhere, when using the server-side API, you don't have to know the URL of the Records Center site, and you don't have to include a web reference directly in your code. Instead, you can simply tell the `SPFile` object to send itself to the Records Center that is configured in the farm. It will then automatically retrieve the information it needs through the SharePoint Administrative API.

Listing 6-1: Submitting a file stored in a document library

```
using Microsoft.SharePoint;

namespace ECM2007.RecordsManagement
{
    public static class OfficialFileSend
    {
        /// <summary>
        /// Submits a file to a records center site using the
        /// records management API.
        /// </summary>
        /// <param name="documentUrl">the address of the file to be
        /// sent</param>
        /// <param name="resultDetails">the result of submitting the
        /// file</param>
        /// <returns>the OfficialFileResult from submitting the
        /// file</returns>
        public static OfficialFileResult Submit(string documentUrl, ref
            string resultDetails)
        {
            string additionalInfo = string.Empty;
            OfficialFileResult result = OfficialFileResult.UnknownError;

            using (SPSite site = new SPSite(documentUrl))
            {
                using (SPWeb web = site.OpenWeb())
                {
                    SPFile file = web.GetFile(documentUrl);
                    if (file != null)
                    {
                        result = file.SendToOfficialFile(out additionalInfo);
                    }
                }

                resultDetails = additionalInfo;
                return result;
            }
        }
    }
}
```

Submitting Multiple Records

One of the common pain points for SharePoint users who are tasked with sending documents to a Records Repository is the fact that they can only send them one at a time. There are many scenarios where this is very inefficient. As an example, consider a situation in which existing records must be categorized and moved into a new SharePoint portal environment. In such cases, the original files are not active in the sense that they are involved in any kind of collaborative workflow. Instead, they are typically uploaded in bulk, and someone must then go through the list to determine if they have the required metadata and are properly classified before being sent to the Repository.

In this section, we will create an application page that displays all of the documents in the site collection and allows the user to select which ones to send to the Records Center by marking a checkbox next to each item. The results of each file submission will be displayed next to the item after processing has completed. We will install the application page using a web-scoped feature, which we will create using the ECM2007 Feature Wizard. Figure 6-10 shows the Wizard dialog with the appropriate field values entered.

The screenshot shows the 'Enter Feature Details' dialog box. It includes a logo on the left and several input fields. The 'Scope' section has radio buttons for 'Farm', 'Web Application', 'Site', and 'Web', with 'Web' selected. The 'Name' field is split into two boxes: 'ECM2007' and 'OfficialFileSend', with a 'Hidden' checkbox to the right. The 'Title' field contains 'ECM2007 - Official File Send'. The 'Description' field contains the text: 'This feature adds the ability to send multiple documents to the records center in a single batch operation. The results of each file submission are stored for later review.' A 'Create Feature Project' button is located at the bottom right.

Figure 6-10: Feature Wizard.

The Feature Wizard generates the standard layout for our feature project, which includes the 12\TEMPLATE folder with subfolders for FEATURES and IMAGES. We will deploy our custom application page to a subfolder of the LAYOUTS folder, so we need to add it as shown in Figure 6-11.

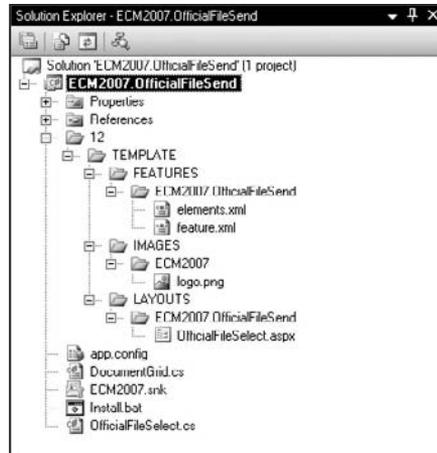


Figure 6-11: ECM2007.OfficialFileSend project structure.

Listings 6-2 and 6-3 show the feature and elements manifests, respectively. These files instruct SharePoint on how to install the feature and declare the custom action that will appear on the Actions menu of any document library in the web site on which the feature is activated.

Listing 6-2: feature.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<Feature xmlns="http://schemas.microsoft.com/sharepoint/"
  Id="345fca5b-a92c-4f1e-8f5d-3cb28348877c"
  Title="ECM2007 - Official File Send"
  Description="This feature adds the ability to send multiple
  documents to the records center in a single batch operation. The
  results of each file submission are stored for later review."
  ImageUrl="ECM2007\logo.png"
  Version="1.0.0.0"
  Scope="Web"
  Hidden="FALSE"
  >
  <ElementManifests>
    <ElementManifest Location="elements.xml" />
  </ElementManifests>
</Feature>
```

Listing 6-3: elements.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<Elements xmlns="http://schemas.microsoft.com/sharepoint/">
  <CustomAction
    Id="OfficialFileSelect"
    GroupId="ActionsMenu"
    Location="Microsoft.SharePoint.StandardMenu"
    RegistrationType="List">
```

```

RegistrationId="101"
Sequence="2000"
Title="Official File Submit"
Description="Select one or more documents to send to the records
center." >
  <UrlAction Url="/_layouts/ECM2007.OfficialFileSend/
  OfficialFileSelect.aspx?ListId={ListId}"/>
</CustomAction>
</Elements>

```

Figure 6-12 shows the custom command as it appears on the Actions menu of a document library. Choosing this command navigates to the OfficialFileSelect.aspx page that will display the documents contained in the library and enables the user to choose which ones to send to the configured repository.

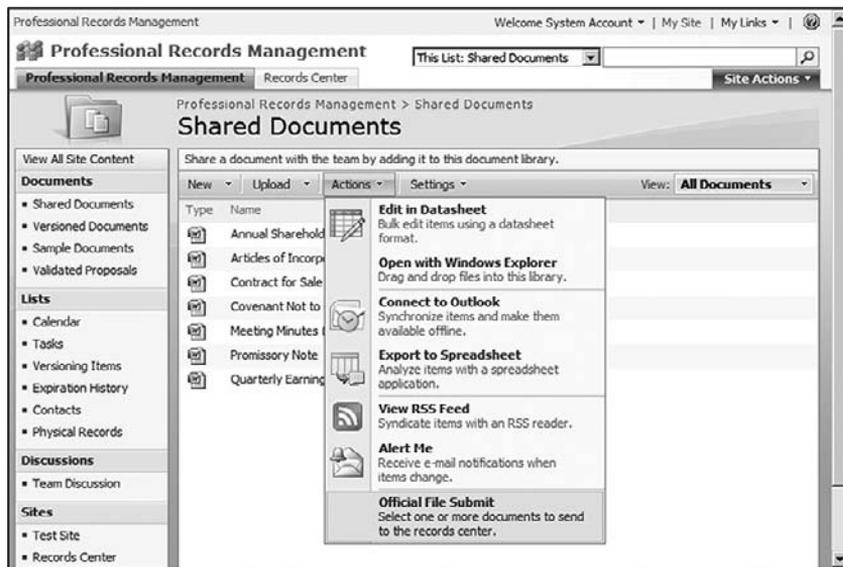


Figure 6-12: Official File Send custom command.

We will use a custom grid control to display the documents and allow the user to select which ones to process. We also need to display the results of each attempt to process an item. To handle this, we will inherit from the SPGridView control and add the following columns:

Column	Description
ID	The document identifier
FileLeafRef	A link to the actual document item
Select	A checkbox that the user will use to specify whether to process the item
Record Type	The content type of the document
Processing Results	A label control that displays the results of processing the item

Chapter 6: Populating the Records Repository

To add the `Select` and `Processing Results` columns, we will use `Template` controls that are added when the grid is initialized. Listing 6-4 shows the `DocumentGrid` class implementation.

Listing 6-4: DocumentGrid.cs

```
using System;
using System.Collections.Generic;
using System.Data;
using System.Diagnostics;
using System.Web.UI;
using System.Web.UI.WebControls;
using Microsoft.SharePoint;
using Microsoft.SharePoint.WebControls;

namespace ECM2007.OfficialFileSend
{
    /// <summary>
    /// This class extends the SPGridView control to display
    /// a list of documents with support for multiple selection.
    /// </summary>
    public class DocumentGrid : SPGridView
    {
        public const string ID_ITEMID = "ItemId";
        public const string ID_ITEMSTATE = "ItemState";
        public const string ID_CHECKBOX = "ItemSelected";
        public const string ID_RESULT = "SendToOfficialFileResult";
        public const string NO_RESULTS = "N/A";
        public SPDocumentLibrary m_library = null;
        protected DataTable m_dataTable = null;

        /// <summary>
        /// The document library associated with the grid.
        /// </summary>
        SPDocumentLibrary Library
        {
            get
            {
                return ((OfficialFileSelect)this.Page).Library;
            }
        }

        /// <summary>
        /// Data table containing list items to be displayed in the grid.
        /// </summary>
        DataTable DataTable
        {
            get
            {
                if (m_dataTable == null)
                    m_dataTable = this.Library.Items.GetDataTable();
                return m_dataTable;
            }
            set
        }
    }
}
```

```
        {
            m_dataTable = value;
        }
    }

    /// <summary>
    /// Configure the grid.
    /// </summary>
    protected override void OnInit(EventArgs args)
    {
        try
        {
            this.EnableViewState = true;
            this.DataTable = this.Library.Items.GetDataTable();

            // trap the data binding event
            this.RowDataBound += new
            GridViewRowEventHandler(DocumentGrid_RowDataBound);

            Log("Setting grid properties");
            this.AutoGenerateColumns = false;
            this.AllowFiltering = true;
            this.HeaderStyle.Font.Bold = true;
            this.AlternatingRowStyle.CssClass = "ms-alternating";

            BoundField id = new BoundField();
            id.DataField = "ID";
            id.HeaderText = "ID";
            id.Visible = false;
            Columns.Add(id);

            BoundField docName = new BoundField();
            docName.DataField = "FileLeafRef";
            docName.HeaderText = "Document";
            Columns.Add(docName);

            TemplateField checkbox = new TemplateField();
            checkbox.HeaderText = "Select";
            checkbox.ItemTemplate = new CheckBoxTemplate();
            Columns.Add(checkbox);

            BoundField contentType = new BoundField();
            contentType.DataField = "ContentType";
            contentType.HeaderText = "Record Type";
            Columns.Add(contentType);

            TemplateField result = new TemplateField();
            result.HeaderText = "Processing Results";
            result.ItemTemplate = new ResultTemplate();
            Columns.Add(result);

            // setup for paging
            this.PageSize = 20;
```

Continued

Listing 6-4: DocumentGrid.cs (continued)

```
        this.AllowPaging = true;
        this.PageIndexChanging += new
        GridViewPageEventHandler(SPDocumentGrid_PageIndexChanging);
        this.PagerTemplate = null;
        this.PagerSettings.Mode = PagerButtons.NextPreviousFirstLast;

        this.PagerSettings.NextPageImageUrl =
        "/_layouts/images/ewr020.gif";
        this.PagerSettings.FirstPageImageUrl =
        "/_layouts/images/ewr018.gif";
        this.PagerSettings.PreviousPageImageUrl =
        "/_layouts/images/ewr019.gif";
        this.PagerSettings.LastPageImageUrl =
        "/_layouts/images/ewr021.gif";

        // bind the data to the grid
        Refresh();
    }
    catch (Exception x)
    {
        HandleException(x);
    }
}

/// <summary>
/// Retrieve the list of selected items.
/// </summary>
public List<SPLListItem> GetSelectedItems()
{
    List<SPLListItem> items = new List<SPLListItem>();
    for (int i = 0; i < Rows.Count; i++)
    {
        GridViewRow row = Rows[i];
        if (row.RowType == DataControlRowType.DataRow)
        {
            CheckBox checkBox = row.FindControl(ID_CHECKBOX) as
            CheckBox;
            if (checkBox.Checked == true)
            {
                HiddenField itemId = row.FindControl(ID_ITEMID) as
                HiddenField;

                items.Add(this.Library.Items.GetItemById(
                    Convert.ToInt32(itemId.Value)));
            }
        }
    }
    return items;
}

/// <summary>
```

```
    /// Handles the paging logic.
    /// </summary>
    void SPDocumentGrid_PageIndexChanging(object sender,
    GridViewPageEventArgs e)
    {
        this.PageIndex = e.NewPageIndex;
        this.DataBind();
    }

    /// <summary>
    /// Diagnostic output routine.
    /// </summary>
    /// <param name="format"></param>
    /// <param name="args"></param>
    protected void Log(string format, params object[] args)
    {
        string category = GetType().Name;
        Trace.WriteLine(string.Format(format, args) + "...", category);
    }

    /// <summary>
    /// Generic exception handler.
    /// </summary>
    /// <param name="x"></param>
    protected void HandleException(Exception x)
    {
        Log("Exception occurred: {0}", x.ToString());
    }

    /// <summary>
    /// Helper method to refresh the grid.
    /// </summary>
    private void Refresh()
    {
        try
        {
            // Bind the data table to the grid.
            Log("Binding the data table to the grid");
            this.DataSource = this.DataTable.DefaultView;
            this.DataBind();
        }
        catch (Exception x)
        {
            HandleException(x);
        }
    }
}
}
```

The `OnInit` routine creates the grid columns, sets up the paging controls, and performs the initial data binding against the list item collection associated with the library. An event handler is created for the `RowDataBound` event so that custom fixup code can be executed as each row is bound to the grid.

Chapter 6: Populating the Records Repository

The `OnInit` routine also sets the `EnableViewState` member to `true` so that the checkbox state is maintained during postback events. Notice that the `checkbox` and `result` fields are created as instances of `TemplateField` and that the item templates are created as instances of `CheckBoxTemplate` and `ResultTemplate`. These are custom classes that provide support for the `Select` and `Processing Results` columns. Listing 6-5 shows the implementation for the `CheckBoxTemplate` class.

Listing 6-5: CheckBoxTemplate

```
/// <summary>
/// Custom item template for selecting documents to process.
/// </summary>
class CheckBoxTemplate : ITemplate
{
    /// <summary>
    /// Called by the page framework to instantiate an item.
    /// </summary>
    void ITemplate.InstantiateIn(Control container)
    {
        // Add the checkbox for selecting the item
        CheckBox box = new CheckBox();
        box.ID = DocumentGrid.ID_CHECKBOX;
        box.Visible = true;
        box.EnableViewState = true;
        box.CausesValidation = true;
        container.Controls.Add(box);

        // add a hidden field to hold the item identifier
        HiddenField itemId = new HiddenField();
        itemId.ID = DocumentGrid.ID_ITEMID;
        itemId.Visible = false;
        container.Controls.Add(itemId);
    }
}
```

When the template is instantiated for a given row, it creates a new `CheckBox` control that is displayed to the user, and a `HiddenField` control that is not displayed, but holds the list item identifier that is associated with the row. This identifier will be retrieved when the `Submit` button is pressed to locate the list item for processing. Listing 6-6 shows the `ResultTemplate` implementation.

Listing 6-6: ResultTemplate

```
/// <summary>
/// Custom item template for displaying processing results.
/// </summary>
class ResultTemplate : ITemplate
{
    /// <summary>
    /// Called when the item is instantiated.
    /// </summary>
    void ITemplate.InstantiateIn(Control container)
    {
```

```
        // Add a label to display the records processing
        // result string, or "not sent" if results unavailable
        Label results = new Label();
        results.ID = DocumentGrid.ID_RESULT;
        results.Visible = true;
        results.Width = Unit.Pixel(200);
        container.Controls.Add(results);
    }
}
```

This class simply creates a `Label` control that is used to display the current processing results for the item. Since the user can navigate to the `OfficialFileSend.aspx` page at any time, the processing results are stored in the property bag for each item. This means that the page will always display some processing result string, whether the item has been sent to the Repository or not. The item will contain either a null value for the property or a string that indicates the most recent processing result for that item.

Both the item identifier and the current processing result string for each item are set in the `DocumentGrid_RowDataBound` event handler, which is called once for each row. The following code shows the implementation of this method:

```
/// <summary>
/// Called when a row is bound to data. Stores the list item
/// identifier into the hidden field used to locate the selected
/// item for records processing.
/// </summary>
void DocumentGrid_RowDataBound(object sender, GridViewRowEventArgs e)
{
    // check for a data row
    if (e.Row.RowType == DataControlRowType.DataRow)
    {
        // set the item identifier
        HiddenField itemId = e.Row.FindControl(ID_ITEMID) as HiddenField;
        if (itemId != null)
        {
            DataRowView data = e.Row.DataItem as DataRowView;
            itemId.Value = data["ID"].ToString();
        }

        // set the item result
        Label resultLabel = e.Row.FindControl(ID_RESULT) as Label;
        if (resultLabel != null)
        {
            // default to no results
            string results = DocumentGrid.NO_RESULTS;

            if (itemId != null)
            {
                // get the item to retrieve most recent results
                SPListItem item = null;
                try
                {
                    item = this.Library.Items.GetItemById(
                        Convert.ToInt32(itemId.Value));
                }
            }
        }
    }
}
```

```
    }
    catch (SPException spx)
    {
        HandleException(spx);
    }

    if (item != null)
    {
        try
        {
            results = item.Properties[DocumentGrid.ID_RESULT].ToString();
        }
        catch (Exception x)
        {
            HandleException(x);
        }
    }
    // set the label to the most recent results
    resultLabel.Text = results;
}
}
```

First, we set the hidden field value to the item identifier. Next, we locate the result label control and set its text to either a default *N/A* string or to the value of the most recent processing result.

Listing 6-7 shows the markup that declares the `OfficialFileSelect.aspx` page. It includes a `Register` element for the `ECM2007.OfficialFileSend` assembly that contains the page implementation and declares an instance of the `ECM2007.DocumentGrid` class shown above.

Listing 6-7: OfficialFileSelect.aspx

```
<%@ Assembly Name="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
    PublicKeyToken=71e9bce111e9429c" %>

<%@ Assembly Name="ECM2007.OfficialFileSend, Version=1.0.0.0,
    Culture=neutral, PublicKeyToken=eb8a6a1622425a15" %>

<%@ Page Language="C#" MasterPageFile="~/_layouts/application.master"
    Inherits="ECM2007.OfficialFileSend.OfficialFileSelect"
    EnableViewState="true" EnableViewStateMac="true" %>

<%@ Register TagPrefix="SharePoint"
    Namespace="Microsoft.SharePoint.WebControls"
    Assembly="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
    PublicKeyToken=71e9bce111e9429c" %>

<%@ Register TagPrefix="Utilities" Namespace="Microsoft.SharePoint.Utilities"
    Assembly="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
    PublicKeyToken=71e9bce111e9429c" %>

<%@ Register TagPrefix="wssuc" TagName="ButtonSection"
```

```
Src="~/_controltemplates/ButtonSection.ascx" %>

<%@ Register TagPrefix="ECM2007" Namespace="ECM2007.OfficialFileSend"
Assembly="ECM2007.OfficialFileSend, Version=1.0.0.0, Culture=neutral,
PublicKeyToken=eb8a6a1622425a15" %>

<asp:Content ID="PageTitle" runat="server"
ContentPlaceHolderID="PlaceHolderPageTitle">
Records Center File Submit
</asp:Content>

<asp:Content ID="PageTitleInTitleArea" runat="server"
ContentPlaceHolderID="PlaceHolderPageTitleInTitleArea">
Send Files to <asp:Label ID="RecordCenterTitle" runat="server" /> from
'<asp:Label ID="LibraryTitle" runat="server" />'
</asp:Content>

<asp:Content ID="Main" runat="server" ContentPlaceHolderID="PlaceHolderMain">
<p>
The following list displays all of the documents contained in this
list. Mark
the items you wish to send to the records center and then click the
'Submit' button
to begin the operation. The results will be displayed next to each
item after the
files have been submitted.
</p>

<!-- Records list -->
<ECM2007:DocumentGrid runat="server" id="RecordsGrid"
EnableViewState="true"
AutoGenerateColumns="false"
/>

<!-- Buttons -->
<wssuc:ButtonSection runat="server">
<template_buttons>
<asp:Button UseSubmitBehavior="false" runat="server"
class="ms-ButtonHeightWidth" Text="Submit"
id="btnSubmit" Enabled="true"/>
</template_buttons>
</wssuc:ButtonSection>
</asp:Content>
```

Listing 6-8 shows the actual implementation of the `OfficialFileSelect` application page, which inherits from `LayoutsPageBase` to give it the standard SharePoint look and feel. The `btn_Submit_Click` method is called when the user clicks on the Submit button. This method retrieves the list of selected list items from the grid and then starts an `SPLongOperation` that automatically displays a rotating progress icon while each record is processed and sent to the Repository. The result string from each record submission attempt is stored in the property bag for each item. If an exception occurs, the exception details are stored into the property instead. When the operation completes, the user is redirected to the same page so that the results are refreshed.

Listing 6-8: OfficialFileSelect.cs

```
using System;
using System.Collections.Generic;
using System.Web.UI.WebControls;
using Microsoft.SharePoint;
using Microsoft.SharePoint.Administration;
using Microsoft.SharePoint.WebControls;

namespace ECM2007.OfficialFileSend
{
    public class OfficialFileSelect : LayoutsPageBase
    {
        protected Label LibraryTitle;
        protected Label RecordCenterTitle;
        protected OfficialFileSend.DocumentGrid RecordsGrid;
        protected Button btnSubmit;

        /// <summary>
        /// Represents the library associated with the page.
        /// </summary>
        public SPDocumentLibrary Library
        {
            get
            {
                SPDocumentLibrary library =
                    ViewState["Library"] as SPDocumentLibrary;

                if (library == null)
                {
                    Guid listId = new Guid(Request.Params["ListId"]);
                    library = this.Web.Lists[listId] as SPDocumentLibrary;
                }

                return library;
            }
            set
            {
                ViewState["Library"] = value;
            }
        }

        /// <summary>
        /// Override to initialize the controls on the page.
        /// </summary>
        /// <param name="e"></param>
        protected override void OnInit(EventArgs e)
        {
            // retrieve the document library instance
            LibraryTitle.Text = this.Library.Title;

            // get the name of the configured records center site
            RecordCenterTitle.Text = "Records Center";
            SPWebApplication app = SPAdministrationWebApplication.Local;
        }
    }
}
```

```
using (SPSite recordsCenter = new
SPSite(app.OfficialFileUrl.ToString()))
{
    if (recordsCenter != null)
        RecordCenterTitle.Text = recordsCenter.RootWeb.Title;
}

// install the click handler for the submit button
btnSubmit.Click += new EventHandler(btnSubmit_Click);
}

/// <summary>
/// Diagnostic helper routine.
/// </summary>
void Log(string format, params object[] args)
{
    System.Diagnostics.Trace.WriteLine(
        string.Format(format, args), "OfficialFileSelect");
}

/// <summary>
/// Handles the submit button click.
/// </summary>
void btnSubmit_Click(object sender, EventArgs e)
{
    Log("Processing Records");

    // Start a long operation so we can display a rotating
    // icon to the user while the files are submitted to the
    // records center.
    using (SPLongOperation operation = new
        SPLongOperation(this.Page))
    {
        string message = "Record processing failed.";

        // find the selected items
        List<SPLListItem> recordsToProcess =
            RecordsGrid.GetSelectedItems();

        // start the operation
        operation.Begin();

        try
        {
            foreach (SPLListItem item in recordsToProcess)
            {
                Log(String.Format("Processing record '{0}'",
                    item.Name));
                string resultString = string.Empty;
                try
                {
                    // submit the file and retrieve the result
                    OfficialFileResult result =
                        item.File.SendToOfficialFile(out
```

Continued

Listing 6-8: OfficialFileSelect.cs (continued)

```
        resultString);

// save the result string as a custom property
if (string.IsNullOrEmpty(resultString))
{
    switch (result)
    {
        case OfficialFileResult.FileCheckedOut:
            resultString = "The file is checked
            out.";
            break;
        case OfficialFileResult.FileRejected:
            resultString = "The file was
            rejected.";
            break;
        case
        OfficialFileResult.InvalidConfiguration:
            resultString = "Invalid
            configuration.";
            break;
        case OfficialFileResult.MoreInformation:
            resultString = "Additional
            information needed.";
            break;
        case OfficialFileResult.NotFound:
            resultString = "The current user was
            not found in the submitter's group.";
            break;
        case OfficialFileResult.Success:
            resultString = "The file was
            processed successfully.";
            break;
        case OfficialFileResult.UnknownError:
            resultString = "An unknown error
            occurred.";
            break;
    }
}
}
catch (Exception x)
{
    resultString = String.Format(
        "An exception occurred during record
        processing: {0}",
        x.ToString());
    Log(message);
}

// store the item result
item.Properties[DocumentGrid.ID_RESULT] =
resultString;
item.Update();
```


Chapter 6: Populating the Records Repository

The chapter then described the steps needed to submit records programmatically without having to invoke the Web Service methods directly.

Finally, the chapter presented a SharePoint feature project that installs a custom application page that enables users to select a group of documents in a document library and then submit them in a single batch operation to the Records Repository.

7

Information Management Policy

When talking about information policy, I often ask the question, “What is the difference between a policy and a rule?” This invariably leads the discussion toward notions of corporate governance and regulatory compliance. Somehow the idea of *breaking a rule* doesn’t carry the same weight as *violating a policy*. There can be a *rule* that says you shall capitalize all occurrences of the word *COMPANY* in all legal agreements, failing which a given document might still be valid, but no additional guidance is available to interpret the application of the rule. On the other hand, if we declare a *policy* that contains the same requirement, we tend to look to the policy itself to determine the appropriate context within which to analyze the appropriate next steps.

Information policy in SharePoint is similar, in that the policy becomes a sort of repository of regulatory information in the form of policy items that implement the collection of rules. At the heart of information policy is the ability to make explicit the rules that govern the creation, disposition, and use of content. At the very least, it gives us a place to record our thoughts about the different scenarios in which a given type of content will be used. The important thing here is that we must maintain a clear separation between a given piece of content and any policies we wish to associate with it.

Separating the actual content from any policy statements that refer to it is important for several reasons. First, it makes it possible to modify the policy without referring back to the original content that is currently being affected by it. This should be obvious when we start to look at automating content management using policies because it allows us to modify the policy code without modifying the content. But it works both ways. It also allows us to modify the content definition without referring back to the policy. The interpretation of the policy is really only relevant when it is attached to the content, and its relevance may be further constrained by the content life cycle.

Another reason for maintaining a clean separation between policy statements and the affected content is the ability to associate multiple policies with the same piece of content. Although the SharePoint information management policy framework only allows a single policy to be defined for a given content type or list at a time, we can still use policy features to capture the essence of our policies and effectively apply multiple rules to each content item by enabling multiple policy features.

Information Policy Architecture

So, what is information management policy in the SharePoint Server 2007 environment? The information policy framework provides a non-invasive way to apply a prescribed set of rules to any list item. Whether attached to the list that contains the item, or associated with a content type, a *policy* is simply a block of text that describes the policy (for administrators to understand its application), a second block of text that *states* the policy (for users to understand its purpose), and a collection of policy *items* that carry additional information that SharePoint uses to locate the code needed to apply the policy to the list item. Figure 7-1 shows the default Policy configuration page that content administrators use to configure policy settings.

Edit Policy: Document	
Name and Administrative Description The name and administrative description are shown to list managers when configuring policies on a list or content type.	Name: <input type="text" value="Document"/> Administrative Description: <input type="text" value="This is a description for list managers."/>
Policy Statement The policy statement is displayed to end users when they open items subject to this policy. The policy statement can explain which policies apply to the content or indicate any special handling or information that users need to be aware of.	Policy Statement: <input type="text" value="This is the policy statement for end users."/>
Labels You can add a label to a document to ensure that important information about the document is included when it is printed. To specify the label, type the text you want to use in the "Label format" box. You can use any combination of fixed text or document properties, except calculated or built-in properties such as GUID or CreatedBy. To start a new line, use the \n character sequence.	<input type="checkbox"/> Enable Labels
Auditing Specify the events that should be audited for documents and items subject to this policy.	<input type="checkbox"/> Enable Auditing
Expiration Schedule content disposition by specifying its retention period and the action to take when it reaches its expiration date.	<input type="checkbox"/> Enable Expiration
Barcodes Assigns a barcode to each document or item. Optionally, Microsoft Office applications can require users to insert these barcodes into documents.	<input type="checkbox"/> Enable Barcodes

Figure 7-1: Information Policy configuration page.

Once a policy is defined, it is added to a global catalog of policy objects and is used to manage subcollections of policy items. Each policy item represents a separate component that can be bound to custom code that further defines how the policy is applied to a given list item. Applying a policy to a list item may involve any number of operations, such as changing or filtering the item content or modifying the item metadata. The actual interpretation of a given policy is the responsibility of the developer, who writes code that SharePoint calls whenever the policy is applied.

Policies can be created at three levels:

- ❑ In the Site Collection Policies list in the root web of a site collection
- ❑ Attached to a site content type
- ❑ Attached to a list or document library

Adding a policy to the Site Collection Policies list defines it as a global policy. Global policies can be exported to a file and then imported into other site collections, thus making it easier to maintain consistency between policies at the enterprise level. Once a global policy has been created, it can then be attached to content types and lists. SharePoint provides additional support for ensuring consistency by preventing users from editing global policies at the content type or list level.

Attaching a policy to a site content type limits its reusability because there is no way to export or import the policy, but it can be associated with more than one list. When a policy is attached to a content type, the XML policy definition is propagated automatically to child content types that inherit from the content type to which the policy is attached. SharePoint also prevents users from changing the policy settings for child content types, as shown in Figure 7-2. If the policy is removed from the parent, users can then attach a policy to the child.

Subsequently attaching another policy to the parent overrides any policies already attached to the child, thereby removing them. If the second policy is then removed from the parent, the original policy that was attached to the child will be gone.

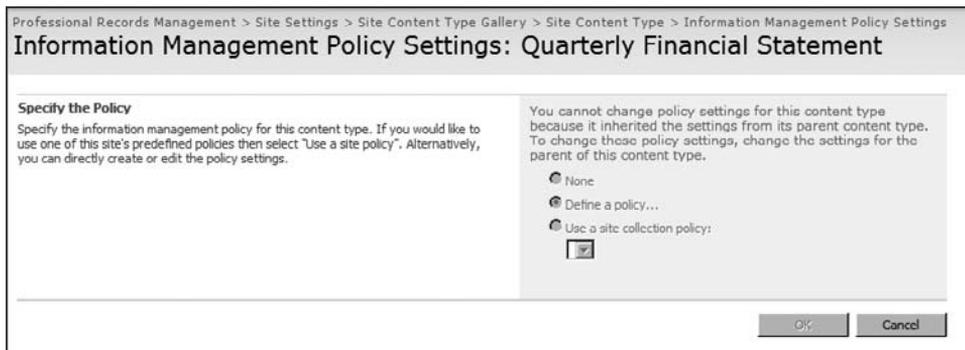


Figure 7-2: Policy locked for child content type.

Attaching a policy directly to a list or document library is the least flexible approach because you cannot reuse the policy definition. Instead, you must re-create the policy for each list separately. Also, you can only attach a policy directly to lists and document libraries that are associated with a single content type. If multiple content types are enabled, then the policy associated with each content type is used.

Site collection administrators can selectively disable certain policy features from being added to policies at the content type or list level, effectively forcing users to work with global policies for the disabled policy features. This is done from the Security Configuration section of the Operations page in the Central Administration web site.

Limitations of Information Policy in SharePoint Server 2007

While the information policy framework is very powerful, the current architecture is limited by the fact that it allows only one policy to be applied at any given time to a list or to a content type. It is further limited by the way in which policy definitions for content types are tightly bound to the content type definition, essentially adding the policy statement to the content type itself. This may not seem to be a severe limitation at first because every policy can support multiple policy features, as you'll see in a moment. Using a single policy, for example, a content administrator can bind multiple processing components to every list item, and this is good enough for many scenarios. However, to move beyond simple processing logic to handle life-cycle-driven scenarios, where members of different roles interact with the same content element in different ways, using a single policy falls short.

Consider the common requirement to manage document retention and content auditing at the same time. With the current architecture, we have to enable both the Expiration Policy Feature and the auditing policy feature as part of the same policy. Consequently, if we were to suspend the policy, it would simultaneously suspend *all* features, which is not always what we want. This is part of the reason why the *exempt from policy* functionality applies only to the Expiration Policy Feature and not to the policy as a whole. Suspending other policies might be useful for many solutions, but unfortunately, it is only available for that single policy feature.

Policies, Policy Features, and Policy Resources

Information management policy is implemented in the `Microsoft.Office.Policy` assembly, which is located in the ISAPI folder of the 12 hive. A quick look at the namespaces it contains reveals a tightly coupled collection of components that provide a global policy catalog tied to a site collection. Thus, every site collection maintains a separate list of policy objects, which you access by instantiating a policy catalog using the public constructor. From the catalog, you can then access the individual policies that are currently defined for the site collection.

```
using (SPSite site = new SPSite("http://localhost")) {
    PolicyCatalog catalog = new PolicyCatalog(site);
}
```

Each policy is represented by an instance of the `Microsoft.Office.RecordsManagement.InformationPolicy.Policy` class, which acts as a wrapper for the XML policy definition, adding a few methods for manipulating the policy template. Figure 7-3 shows the basic architecture of the information policy framework.

The `Policy` object is described by a schema that is used to validate the XML policy definition whenever a new policy object is created. The actual schema is buried inside the `Microsoft.Office.Policy` assembly in the `ValidateManifest` method of the `Policy` object implementation. The complete schema definition is shown in Listing 7-1.

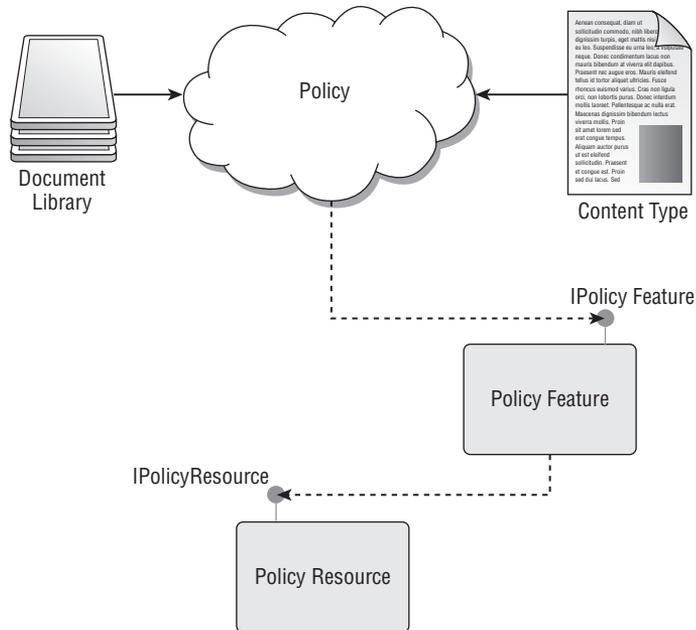


Figure 7-3: Information policy architecture.

Listing 7-1: Policy definition schema

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'
  xmlns='office.server.policy'
  elementFormDefault='qualified'
  targetNamespace='office.server.policy'>

  <xsd:element name='Policy' type='policy' />
<!------->
<xsd:complexType name='policy'>
  <xsd:sequence>
    <xsd:element name='Name' type='xsd:string' />
    <xsd:element name='Description' type='xsd:string' minOccurs='0' />
    <xsd:element name='Statement' type='xsd:string' minOccurs='0' />
    <xsd:element name='PolicyItems' type='policyItems' minOccurs='0' />
  </xsd:sequence>
  <xsd:attribute name='id' type='xsd:string' use='required' />
  <xsd:attribute name='local' type='xsd:boolean' />
</xsd:complexType>
```

Continued

Listing 7-1: Policy definition schema (continued)

```
<!------->
<xsd:complexType name='policyItems'>
  <xsd:sequence>
    <xsd:element name='PolicyItem' type='policyItem'
      minOccurs='0' maxOccurs='unbounded' />
  </xsd:sequence>
</xsd:complexType>
<!------->
<xsd:complexType name='policyItem'>
  <xsd:sequence>
    <xsd:element name='Name' type='xsd:string' />
    <xsd:element name='Statement' type='xsd:string' minOccurs='0' />
    <xsd:element name='Description' type='xsd:string' minOccurs='0' />
    <xsd:element name='CustomData' type='customData' minOccurs='0' />
  </xsd:sequence>
  <xsd:attribute name='featureId' type='xsd:string' use='required' />
  <xsd:attribute name='BlockPreview' type='xsd:boolean' />
</xsd:complexType>
<!------->
<xsd:complexType name='customData'>
  <xsd:sequence>
    <xsd:any processContents='skip' minOccurs='0' maxOccurs='unbounded' />
  </xsd:sequence>
</xsd:complexType>

</xsd:schema>
```

This schema defines a kind of policy envelope that can support a heterogeneous collection of policy items. The `customData` element is declared so that each policy item can define its own custom data, which is also stored as an XML string, but which the policy item can interpret in any way that it likes. Policy features and policy resources are both packaged into the policy as policy items at this level. Later in the chapter, we'll look at how each of these policy items is implemented, but first we need to understand how policies are applied to list items by examining the information policy life cycle.

The Information Policy Life Cycle

SharePoint determines when to apply a policy to a given list item based on a number of factors. Such factors include whether the settings for individual policy features have changed since the item was added to the list, whether items were already in the list when the policy was applied, and whether a policy was in force when new list items were created. Figure 7-4 illustrates the normal event processing sequence.

Policy features implement the `IPolicyFeature` interface, which is also declared in the `Microsoft.Office.RecordsManagement.InformationPolicy` namespace. The interface definition is shown in Listing 7-2.

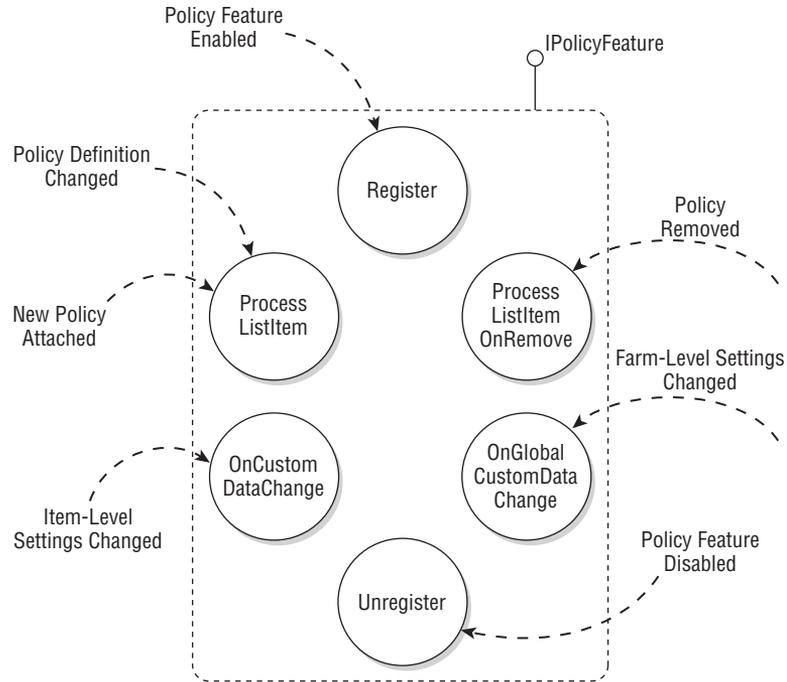


Figure 7-4: Information policy processing of list items.

Listing 7-2: IPolicyFeature interface

```

using Microsoft.SharePoint;
using System;

namespace Microsoft.Office.RecordsManagement.InformationPolicy
{
    public interface IPolicyFeature
    {
        void OnCustomDataChange(PolicyItem policyItem, SPContentType ct);
        void OnGlobalCustomDataChange(PolicyFeature feature);
        bool ProcessListItem(SPSite site, PolicyItem policyItem, SPLListItem
            listItem);
        bool ProcessListItemOnRemove(SPSite site, SPLListItem listItem);
        void Register(SPContentType ct);
        void UnRegister(SPContentType ct);
    }
}

```

Chapter 7: Information Management Policy

When a new policy is attached to a list or to a content type associated with an item in a list, the information policy framework calls the `IPolicyFeature.Register` method for any policy features that were enabled for the policy. This method is also called if a policy is modified to enable a new policy feature. Similarly, the `IPolicyFeature.Unregister` method is called whenever a policy feature is disabled in the UI.

Enabling or disabling a policy feature is performed by the content manager using the checkbox next to the policy feature name displayed on the Policy configuration page.

After the policy feature is registered, the `IPolicyFeature.ProcessListItem` method is then called for any list items now being managed by the policy that contains the policy feature. This method is also called if the policy definition changes or if a new policy is attached to the list or content type. It is also called if a new item is created while the policy is in force.

The `IPolicyFeature.ProcessListItemOnRemove` method is called when a policy is removed from the list or content type or when the policy feature is disabled. The thinking behind this is that any content elements that are governed by a policy may require special processing whenever the policy is enabled or disabled. Calling this method gives the policy feature an opportunity to perform any required actions, such as adding or removing administrative metadata to the item, attaching event receivers to the item, and so on.

Policy features may also require custom configuration data at the farm level for global settings and at the user level for settings that apply to each policy instance. SharePoint provides a separate UI for each set of settings and also provides abstract server controls we can use to build a custom user interface for users and administrators. Since we must implement our own custom controls derived from these abstract server controls, SharePoint can easily detect when changes have been made to the base control properties. It then calls the appropriate method in the policy feature whenever this occurs. `IPolicyFeature.OnGlobalCustomDataChange` is called when changes are made through the farm-level UI (Central Administration), and `IPolicyFeature.OnCustomDataChange` is called for changes made via the standard SharePoint UI.

One approach to building policy features and policy resources programmatically is to first generate wrapper classes using the published schema and then extend them with custom methods that create and manipulate the policy objects within SharePoint. For most of the information policy components, this turns out to be more trouble than it's worth because the default XML serialization includes more information than is typically needed and can cause the validation method to fail. A more direct approach is to simply create an abstract base class that generates the appropriate XML on demand by pulling the required element values from virtual methods we can implement in a derived class. I will use this technique throughout the chapter to construct custom policy items.

Policy resources are code components that participate in the application of a policy feature to a given policy. They enable developers to define a sort of mini-framework of cooperating components that together provide the required functionality. Policy features and their associated policy resources are typically implemented together and share a common set of principles and/or data structures and interfaces. SharePoint imposes no restrictions on how policy features and their policy resources communicate. The only requirement is that a given policy resource must accurately identify the policy feature with which it is associated. It does this by providing SharePoint with the appropriate policy feature identifier, which must be known to the developer implementing the policy resource.

Building Custom Policy Features

SharePoint's information policy features enable content administrators to apply different kinds of controls to various documents without creating a separate SharePoint feature for every document type. The interface-based loose coupling between policies, policy features, and policy resources enables developers to provide the reusable layers of functionality needed to make this work.

In the preceding chapters, we began implementing a set of custom components as part of a foundation class library for working with SharePoint objects. In this section, we'll extend our ECM2007 foundation class library to include specialized components for building information policies, policy features, and policy resources.

Designing for Extensibility

One of the limitations of the standard SharePoint information management policy architecture is that it requires the developer to anticipate the different ways in which different types of content will be used. This is often unrealistic because the policy framework is itself intended to allow developers to delegate many of the configuration details to an administrator or business analyst. The ability to build custom policy features partially addresses this by providing a way for the developer to gather configuration settings from the administrator and then design the policy feature so that it is driven by those settings.

However, it is also possible to extend the framework further by leveraging policy resources and publishing well-known interfaces that developers can implement easily without having to deal with unnecessary complexity. The expiration policy framework is a good example of this. By implementing the `IExpirationFormula` and `IExpirationAction` interfaces on custom policy resource classes, you can extend and customize the Expiration Policy Feature without having to address any of the other issues related to content expiration. Like that, it's a good idea to identify similar patterns and expose them as well-defined interfaces when designing your own policy features.

Creating Reusable Policy Components

When working with policy features, it is useful to have a library of components that can be reused in several projects. In this exercise, you will use a set of utility classes that greatly simplifies the creation of custom policies, policy features, and policy resources. Let's declare another interface that captures the essential elements that will be merged into any serialized policy object:

```
using System;

namespace ECM2007.InformationPolicy
{
    /// <summary>
    /// Describes an information policy.
    /// </summary>
    interface ISharePointPolicy
    {
        /// <summary>
        /// The policy identifier.
        /// </summary>
        Guid PolicyId { get; set; }
    }
}
```

```
    /// The name of the policy.
    /// </summary>
    string PolicyName { get; set; }
    /// <summary>
    /// A brief description of the policy.
    /// </summary>
    string PolicyDescription { get; set; }
    /// <summary>
    /// The policy statement presented to end users.
    /// </summary>
    string PolicyStatement { get; set; }
    /// <summary>
    /// The XML manifest that describes the policy.
    /// </summary>
    string PolicyManifest { get; }
}
}
```

Note that we have excluded the collection of policy items from the interface definition. We don't really need them when creating policy objects. SharePoint will create the policy items automatically when we add policy features to the policy. If we later decide to build a reporting or administrative layer, then it might be useful to add the policy item collection to our interface. In that case, we might also need additional methods for finding and extracting existing policies.

Next, we declare a class that implements the interface and provides an abstract base from which to derive specific policy objects. We can again leverage our `SharePointObject` base class to add support for using the `InstallUtil` utility for registering the policy directly from the command line without activating a feature.

Feature activation via SharePoint solution packages is the best practice for deploying SharePoint components. However, there are situations in which a command-line approach might also be useful. Here, we are designing for both scenarios.

```
using System;
using System.Text;
using System.Diagnostics;
using System.Runtime.InteropServices;
using Microsoft.Office.RecordsManagement.InformationPolicy;
using Microsoft.SharePoint;
using ECM2007.Utilities;

namespace ECM2007.InformationPolicy
{
    /// <summary>
    /// Base class for declaring information policies.
    /// </summary>
    public abstract class SharePointPolicy : SharePointObject, ISharePointPolicy
    {
    }
}
```

To enable the creation of policies, we declare a couple of static factory methods — one to create a policy in a site collection and another to associate a policy with a content type.

The return value from these factory methods is an instance of the `SharePoint Policy` class, which we can then use to perform other operations on the resulting policy object.

The first method creates a policy for a site collection based on information contained in a separate class. That class will represent the actual policy, and we will use a combination of attributes and data members to provide the information needed to declare the policy within SharePoint. To retrieve that information, we use the `ISharePointPolicy` interface declared previously.

```
public static Policy Create(SPSite site, Type policyType)
{
    ISharePointPolicy policyInstance =
        Activator.CreateInstance(policyType) as ISharePointPolicy;
    if (policyInstance != null)
    {
        try
        {
            PolicyCollection.Add(site, policyInstance.PolicyManifest);
        }
        catch
        {
            PolicyCollection.Delete(site, policyInstance.PolicyId.ToString());
            PolicyCollection.Add(site, policyInstance.PolicyManifest);
        }
    }
    return FindPolicy(site, policyType);
}
```

After creating an instance of the referenced type, we call its implementation of the `PolicyManifest` property to add the policy definition to the policy catalog for the site collection. If it already exists, we handle the `InvalidOperationException` exception and then attempt to delete the policy before trying a second time to add it to the catalog. If we are successful, then we return the result of finding the new policy in the catalog.

The next method creates a policy and associates it with a content type.

```
public static Policy Create(SPContentType ct, Type policyType)
{
    Policy policy = null;
    try
    {
        policy = FindPolicy(ct.ParentWeb.Site, policyType);
        if (policy == null)
            policy = Create(ct.ParentWeb.Site, policyType);
        if (policy != null && !policy.IsLocal)
            Policy.CreatePolicy(ct, policy);
    }
    catch (Exception x)
    {
        Trace.WriteLine(string.Format(
            "Failed to create policy of type '{0}' for content type '{1}' - {2}",
            policyType.Name, ct.Name, x.Message));
    }
    return policy;
}
```

Chapter 7: Information Management Policy

Often, it is necessary to locate an existing policy, so we can add a utility method for that purpose, which we can call while creating a policy to ensure that a duplicate policy is not created inadvertently.

```
/// <summary>
/// Locates a policy in a site collection.
/// </summary>
/// <param name="site"></param>
/// <param name="policyType"></param>
/// <returns></returns>
public static Policy FindPolicy(SPSite site, Type policyType)
{
    ISharePointPolicy policyInstance =
    Activator.CreateInstance(policyType) as ISharePointPolicy;
    if (policyInstance != null)
    {
        PolicyCatalog catalog = new PolicyCatalog(site);
        foreach (Policy policy in catalog.PolicyList)
            if (policy.Id.Equals(policyInstance.PolicyId.ToString()))
                return policy;
    }
    return null;
}
```

Finally, we declare a virtual property that constructs a properly formatted XML fragment that SharePoint recognizes as a valid policy definition. This method retrieves the individual properties from the derived class to fill out the XML fragment.

```
/// <summary>
/// Returns the schema xml that describes the policy.
/// </summary>
public override string SchemaXml
{
    get
    {
        StringBuilder sb = new StringBuilder(
            "<Policy xmlns='urn:schemas-microsoft-com:office:server:policy' ");
        sb.AppendFormat("id = '{0}' ", this.Id);
        sb.AppendFormat("Local = '{0}'>", this.IsLocal ? "TRUE" : "FALSE");
        sb.AppendFormat(@"<Name>{0}</Name>", this.PolicyName);
        sb.AppendFormat(@"<Description>{0}</Description>", this.PolicyDescription);
        sb.AppendFormat(@"<Statement>{0}</Statement>", this.PolicyStatement);
        sb.Append("</Policy>");
        return sb.ToString();
    }
}
```

With this component in our library, declaring a new policy is a simple matter of deriving a new class from the abstract base and then either decorating it with attributes or overriding the properties we wish to customize.

Policy Features

Policy features are also defined within SharePoint by a schema as shown in Listing 7-3.

Listing 7-3: Policy feature schema

```

<?xml version="1.0" encoding="utf-8"?>
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'
            xmlns:p='urn:schemas-microsoft-com:office:server:policy'
            elementFormDefault='qualified'
            targetNamespace='urn:schemas-microsoft-com:office:server:policy'>

  <xsd:element name='PolicyFeature' type='p:policyFeature' />
<!------->
  <xsd:complexType name='policyFeature'>
    <xsd:sequence>
      <xsd:element name='LocalizationResources' type='xsd:string' minOccurs='0' />
      <xsd:element name='Name'>
        <xsd:simpleType>
          <xsd:restriction base='xsd:normalizedString'>
            <xsd:whiteSpace value='collapse' />
            <xsd:pattern value='^\s*(.*)*' />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name='Description' type='xsd:string' minOccurs='0' />
      <xsd:element name='Publisher' type='xsd:string' minOccurs='0' />
      <xsd:element name='ConfigPage' type='xsd:string' minOccurs='0' />
      <xsd:element name='ConfigPageInstructions' type='xsd:string' minOccurs='0' />
      <xsd:element name='DefaultCustomData' type='p:customData' minOccurs='0' />
      <xsd:element name='GlobalConfigPage' type='xsd:string' minOccurs='0' />
      <xsd:element name='GlobalCustomData' type='p:customData' minOccurs='0' />
      <xsd:element name='AssemblyName' type='xsd:string' minOccurs='0' />
      <xsd:element name='ClassName'>
        <xsd:simpleType>
          <xsd:restriction base='xsd:string'>
            <xsd:pattern value='([\s])+' />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name='ResourceTypes' type='p:resourceTypes' minOccurs='0' />
    </xsd:sequence>
    <xsd:attribute name='id' type='xsd:string' use='required' />
    <xsd:attribute name='group' type='xsd:string' />
  </xsd:complexType>
<!------->
  <xsd:complexType name='customData'>
    <xsd:sequence>
      <xsd:any processContents='skip' minOccurs='0' maxOccurs='unbounded' />
    </xsd:sequence>
  </xsd:complexType>
<!------->
  <xsd:complexType name='resourceTypes'>
    <xsd:sequence>
      <xsd:element name='ResourceType' type='p:resourceType' minOccurs='0'
                  maxOccurs='unbounded' />
    </xsd:sequence>
  </xsd:complexType>

```

Continued

Listing 7-3: Policy feature schema (continued)

```
<!------->
<xsd:complexType name='resourceType'>
  <xsd:sequence>
    <xsd:element name='Name' type='xsd:string' />
  </xsd:sequence>
  <xsd:attribute name='id' type='xsd:string' use='required' />
  <xsd:attribute name='required' type='xsd:boolean' use='required' />
</xsd:complexType>

</xsd:schema>
```

We start by declaring an interface we can use to retrieve the essential elements of a policy feature. This includes the XML manifest as well as the configuration page and instructions that will appear on the configuration page for administrators.

```
using System;

namespace ECM2007.InformationPolicy
{
    /// <summary>
    /// Describes an information policy feature.
    /// </summary>
    interface ISharePointPolicyFeature
    {
        /// <summary>
        /// The unique policy feature identifier.
        /// </summary>
        string Id { get; }
        /// <summary>
        /// The name of the feature.
        /// </summary>
        string Name { get; set; }
        /// <summary>
        /// A brief description of what the feature does.
        /// </summary>
        string Description { get; set; }
        /// <summary>
        /// The name of the user control that provides a user interface
        /// for configuring the policy feature.
        /// </summary>
        string ConfigPage { get; set; }
        /// <summary>
        /// Additional instructions that are added to the configuration
        /// user interface for the policy feature.
        /// </summary>
        string ConfigPageInstructions { get; set; }
        /// <summary>
        /// Retrieves the XML manifest for the feature.
        /// </summary>
        string Manifest { get; }
    }
}
```

The `ISharePointPolicyFeature` implementation will call back through a set of virtual methods that can be overridden by derived classes. We can also use this pattern to redirect calls from the `IPolicyFeature` implementation as well. We then end up with a default implementation of the core methods as well as providing an easy way for developers to further extend the component.

```
#region Virtual Properties
public virtual string ConfigPage { get; set; }
public virtual string ConfigPageInstructions { get; set; }
#endregion

#region Virtual Overrides
/* Virtual methods that can be overridden by derived classes. */
public virtual void OnCustomDataChange(PolicyItem policyItem, SPContentType ct) { }
public virtual void OnGlobalCustomDataChange(PolicyFeature feature) { }
public virtual bool ProcessListItem(SPSite site, PolicyItem policyItem,
    SPListItem listItem) { return false; }
public virtual bool ProcessListItemOnRemove(SPSite site, SPListItem listItem)
    { return false; }
public virtual void Register(SPContentType ct) { }
public virtual void UnRegister(SPContentType ct) { }
#endregion
```

The `SchemaXml` override builds the CAML needed to describe the policy feature to SharePoint by retrieving the required values from the derived class implementation.

```
/// <summary>
/// Returns the schema xml that describes the policy.
/// </summary>
public override string SchemaXml
{
    get
    {
        StringBuilder sb = new StringBuilder(
            "<PolicyFeature xmlns='urn:schemas-microsoft-com:
            office:server:policy' ");
        sb.AppendFormat("id = '{0}'>", this.Id);
        sb.AppendFormat("<Name>{0}</Name>", this.Name);
        sb.AppendFormat("<Description>{0}</Description>", this.Description);
        sb.AppendFormat("<Publisher>{0}</Publisher>", this.Publisher);
        sb.AppendFormat("<ConfigPage>{0}</ConfigPage>", this.ConfigPage);
        sb.AppendFormat("<ConfigPageInstructions>{0}</ConfigPageInstructions>",
            this.ConfigPageInstructions);
        sb.AppendFormat("<AssemblyName>{0}</AssemblyName>", this.AssemblyName);
        sb.AppendFormat("<ClassName>{0}</ClassName>", this.ClassName);
        sb.Append("</PolicyFeature>");
        return sb.ToString();
    }
}
```

With the virtual methods and properties in place, we can simply delegate the remaining methods as needed.

```
#region IPolicyFeature Members

void IPolicyFeature.OnCustomDataChange(PolicyItem policyItem, SPContentType ct)
```

Chapter 7: Information Management Policy

```
{
    this.OnCustomDataChange(policyItem, ct);
}

void IPolicyFeature.OnGlobalCustomDataChange(PolicyFeature feature)
{
    this.OnGlobalCustomDataChange(feature);
}

bool IPolicyFeature.ProcessListItem(SPSite site, PolicyItem policyItem, SPListItem
    listItem)
{
    return this.ProcessListItem(site, policyItem, listItem);
}

bool IPolicyFeature.ProcessListItemOnRemove(SPSite site, SPListItem listItem)
{
    return this.ProcessListItemOnRemove(site, listItem);
}

void IPolicyFeature.Register(SPContentType ct)
{
    this.Register(ct);
}

void IPolicyFeature.UnRegister(SPContentType ct)
{
    this.UnRegister(ct);
}

#endregion
```

The Register and Unregister methods are called when the feature is registered for a content type and when the policy feature is deactivated, respectively.

```
/// <summary>
/// This method is called when the feature is registered for a content type.
/// Adds a "TrustedPrinters" field to the content type.
/// </summary>
/// <param name="ct"></param>
public override void Register(SPContentType ct)
{
    base.Register(ct);
    Log("Registering print control for content type: " + ct.Name);

    // Setup the item event receiver for the content type.
    ItemEventReceiver.Create(ct, typeof(PrinterPolicyEventReceiver));

    // Add the "TrustedPrinters" field to the content type.
    SPFieldLink fieldRef = SharePointContentType.AddOrCreateFieldReference(ct,
        PrinterPolicyFeature.TrustedPrintersFieldName, SPFieldType.Text, false,
        true, true, true, true);
}
```

```
}

/// <summary>
/// This method is called when the policy feature is removed.
/// Uninstalls the event receiver from the content type.
/// </summary>
/// <param name="ct"></param>
public override void UnRegister(SPContentType ct)
{
    base.UnRegister(ct);
    Log("Unregistering PrintControl for content type: " + ct.Name);
    ItemEventReceiver.Remove(ct, typeof(PrinterPolicyEventReceiver));
}
}
```

The `ProcessListItem` method is called by the policy framework for list items that were created before the policy feature was applied.

```
/// <summary>
/// This method is called by the policy framework for list items that were
/// created before the policy was applied to the list.
/// </summary>
/// <param name="site"></param>
/// <param name="policyItem"></param>
/// <param name="listItem"></param>
/// <returns></returns>
public override bool ProcessListItem(SPSite site,
    Microsoft.Office.RecordsManagement.InformationPolicy.PolicyItem policyItem,
    SPLListItem listItem)
{
    Log("Processing list item: " + listItem.Title);
    bool result = base.ProcessListItem(site, policyItem, listItem);

    // Add the policy item custom data to the TrustedPrinters field.
    try
    {
        // Get the policy data from the item payload.
        XmlDocument xmlDoc = new XmlDocument();
        xmlDoc.LoadXml(policyItem.CustomData);

        XmlNamespaceManager nsmgr = new XmlNamespaceManager(xmlDoc.NameTable);
        nsmgr.AddNamespace("p", PrinterPolicyFeature.PrintControlPolicyNamespace);
        XmlNode node = xmlDoc.SelectSingleNode("p:data/p:printers", nsmgr);

        // Store the list of trusted printers into the list item.
        listItem[PrinterPolicyFeature.TrustedPrintersFieldName] = node.InnerText;
        listItem.Update();
    }
    catch (Exception x)
    {
        Log("ProcessListItem failed for item: " + listItem.Title + " - " +
            x.ToString());
    }
    return result;
}
}
```

Chapter 7: Information Management Policy

The `OnCustomDataChange` event fires whenever the custom data for a policy item changes. In this case, we will load the custom data from the policy item, which we will store as an XML document. We can structure the XML fragment any way we want. We just have to ensure that both the custom control and the policy feature handle the data in the same way. In this case, we just need to store and retrieve the list of trusted printers as shown in Listing 7-4.

Listing 7-4: `OnCustomDataChange` method implementation

```
/// <summary>
/// This method is called when the custom data for a policy item changes.
/// </summary>
/// <param name="policyItem"></param>
/// <param name="ct"></param>
public override void OnCustomDataChange(PolicyItem policyItem, SPContentType ct)
{
    base.OnCustomDataChange(policyItem, ct);
    Log("OnCustomDataChange for policy item: " + policyItem.Name
        + " and content type " + ct.Name + " where policy item custom data = "
        + policyItem.CustomData);

    try
    {
        // Get the custom data from the policy item.
        XmlDocument xmlDoc = new XmlDocument();
        xmlDoc.LoadXml(policyItem.CustomData);
        XmlNamespaceManager nsmgr = new XmlNamespaceManager(xmlDoc.NameTable);
        nsmgr.AddNamespace("p", PrinterPolicyFeature.PrintControlPolicyNamespace);

        // Add the custom data to the content type payload.
        ct.XmlDocuments.Delete(PrinterPolicyFeature.PrintControlPolicyNamespace);
        ct.XmlDocuments.Add(xmlDoc);
        ct.Update();
    }
    catch (Exception x)
    {
        Log("OnCustomDataChange failed for item: " + policyItem.Name + " - " +
            x.ToString());
    }
}
```

As mentioned earlier, policy features and policy resources work together to provide a complete package of functionality for an information policy. Since policy resources are also packaged as policy items when added to a policy, we can take a similar approach and define an abstract interface and base class implementation to make it easier to create custom policy resource components.

Policy Resources

We start by defining a simple abstraction for our generic policy resource component. We do this by declaring the `ISharePointPolicyResource` interface shown in Listing 7-5.

Listing 7-5: ISharePointPolicyResource interface

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

namespace ECM2007.InformationPolicy
{
    public interface ISharePointPolicyResource
    {
        string FeatureId { get; }
        string Publisher { get; }
    }
}
```

As we have done for other components in our ECM2007 foundation class library throughout the book, we will include a special `SchemaXml` property that constructs the CAML code needed to describe the resource to SharePoint. It does this by retrieving the resource name, description, publisher, assembly, and class name from the derived class that will actually implement the policy resources we will eventually build. The CAML code must conform to the policy resource schema shown in Listing 7-6.

Listing 7-6: Policy resource schema

```
<?xml version="1.0" encoding="utf-8"?>
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'
            xmlns='urn:schemas-microsoft-com:office:server:policy'
            elementFormDefault='qualified'
            targetNamespace='urn:schemas-microsoft-com:office:server:policy'>

    <xsd:element name='PolicyResource' type='policyResource' />
<!------->
<xsd:complexType name='policyResource'>
    <xsd:sequence>
        <xsd:element name='LocalizationResources' type='xsd:string' minOccurs='0' />
        <xsd:element name='Name'>
            <xsd:simpleType>
                <xsd:restriction base='xsd:normalizedString'>
                    <xsd:whiteSpace value='collapse' />
                    <xsd:pattern value='^[\\s](.)*' />
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:element>
        <xsd:element name='Description' type='xsd:string' minOccurs='0' />
        <xsd:element name='Publisher' type='xsd:string' minOccurs='0' />
        <xsd:element name='GlobalConfigPage' type='xsd:string' minOccurs='0' />
        <xsd:element name='GlobalCustomData' type='customData' minOccurs='0' />
        <xsd:element name='AssemblyName' type='xsd:string' minOccurs='0' />
        <xsd:element name='ClassName'>
            <xsd:simpleType>
                <xsd:restriction base='xsd:string'>
                    <xsd:pattern value='([\\s])+ ' />
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>
```

Continued

Listing 7-6: Policy resource schema (continued)

```
        </xsd:simpleType>
    </xsd:element>
</xsd:sequence>
<xsd:attribute name='id' type='xsd:string' use='required' />
<xsd:attribute name='featureId' type='xsd:string' use='required' />
<xsd:attribute name='type' type='xsd:string' use='required' />
</xsd:complexType>
<!------->
<xsd:complexType name='customData'>
    <xsd:sequence>
        <xsd:any processContents='skip' minOccurs='0' maxOccurs='unbounded' />
    </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

Listing 7-7 shows the declaration of our abstract base class with the `SchemaXml` method that constructs the required CAML code. Notice the two protected virtual properties, `FeatureId` and `ResourceType`. These values will provide the appropriate context for the policy resource when the component is loaded into the SharePoint environment. The `FeatureId` value tells SharePoint which feature to associate the resource with. It then loads the component and makes it available to the policy feature when it runs. The `ResourceType` property is provided so that the feature can distinguish among the different types of resources that may be loaded. You'll see in a moment how this value is used to distinguish expiration formulas from expiration actions.

Listing 7-7: SharePointPolicyResource class

```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel;
using System.Configuration.Install;
using Microsoft.Office.RecordsManagement.InformationPolicy;
using Microsoft.Office.RecordsManagement.PolicyFeatures;

namespace ECM2007.InformationPolicy
{
    /// <summary>
    /// Base class for policy resource implementations.
    /// </summary>
    public abstract class SharePointPolicyResource : SharePointObject,
        ISharePointPolicyResource
    {
        /// <summary>
        /// Override this property to specify the correct ID
        /// of the policy feature this resource belongs to.
        /// </summary>
        protected virtual string FeatureId
        {
            get { return string.Empty; }
        }
    }
}
```

```

    }

    /// <summary>
    /// Override this property to specify the policy resource type.
    /// </summary>
    protected virtual string ResourceType
    {
        get { return this.Name; }
    }

    /// <summary>
    /// Override to construct the required manifest for this component type.
    /// </summary>
    public override string SchemaXml
    {
        get
        {
            StringBuilder sb = new StringBuilder(
                "<PolicyResource xmlns='urn:schemas-microsoft-com:
                office:server:policy' ");
            sb.AppendFormat("id = '{0}' ", this.Id);
            sb.AppendFormat("featureId = '{0}' ", this.FeatureId);
            sb.AppendFormat("type = '{0}'>", this.ResourceType);
            sb.AppendFormat("<Name>{0}</Name>", this.Name);
            sb.AppendFormat("<Description>{0}</Description>", this.Description);
            sb.AppendFormat("<Publisher>{0}</Publisher>", this.Publisher);
            sb.AppendFormat("<AssemblyName>{0}</AssemblyName>", this.AssemblyName);
            sb.AppendFormat("<ClassName>{0}</ClassName>", this.ClassName);
            sb.Append("</PolicyResource>");
            return sb.ToString();
        }
    }
}
}
}

```

Leveraging the `InstallUtil` utility is a great way to deploy policy resources. We'll implement our base class so that it provides a default implementation of the `Install` and `Uninstall` methods that are inherited from the `System.Configuration.Installer` component. These methods can then be enabled or disabled from the declaration of any derived classes by including or excluding the `RunInstaller` attribute from the class definition.

```

    /// <summary>
    /// Installs the policy resource into the local farm.
    /// </summary>
    public override void Install(System.Collections.IDictionary stateSaver)
    {
        base.Install(stateSaver);
        string manifest = this.SchemaXml;
        Console.WriteLine("Installing policy resource: {0}", this.Name);
        try
        {
            Console.WriteLine("Attempting to delete using id: {0}", this.Id);
            PolicyResourceCollection.Delete(this.Id);
        }
    }
}

```

```
        catch (Exception x1)
        {
            Console.WriteLine("Delete failed: {0}", x1.Message);
        }
    try
    {
        Console.WriteLine("Validating manifest:\n{0}", manifest);
        PolicyResource.ValidateManifest(manifest);
        PolicyResourceCollection.Add(manifest);
    }
    catch (Exception x2)
    {
        Console.WriteLine("Error in manifest: {0}", x2.Message);
    }
}

/// <summary>
/// Removes the policy resource from the local farm.
/// </summary>
public override void Uninstall(System.Collections.IDictionary savedState)
{
    base.Uninstall(savedState);
    Console.WriteLine("Uninstalling policy resource with id: {0}", this.Id);
    PolicyResourceCollection.Delete(this.Id);
}
```

We can now use our base set of information policy components to build solutions more quickly and easily.

In the following sections, we'll develop a custom information policy feature that controls the distribution of printed material by limiting the printing of certain documents to a specific set of secure printers. Our policy feature will contain a list of trusted printers and will work in conjunction with a Visual Studio Tools for Office (VSTO) add-in that will trap the `print` event to determine whether a trusted printer is being used. If not, then a dialog will be displayed to the user, and the print job will be canceled.

Creating a Printer Control Policy Feature

This is an implementation of a use case suggested by Microsoft in the white paper, "Compliance Features in the 2007 Microsoft Office System" (<http://www.microsoft.com/downloads/details.aspx?familyid=d64dfb49-aa29-4a4b-8f5a-32c922e850ca&displaylang=en>), which was published in November 2006. I decided to implement this use case as a way to explore the information policy framework, but also to take a closer look at the inherent dependency between server-side and client-side policy components.

We'll start by creating a new SharePoint Feature project named *ECM2007.PrinterControlPolicyFeature*. Since Information Policy Features are installed globally, our feature could be scoped to any level. For convenience and in anticipation of adding additional components such as content types and fields to our solution later, we'll set the feature scope to *Site*, as shown in Figure 7-5.

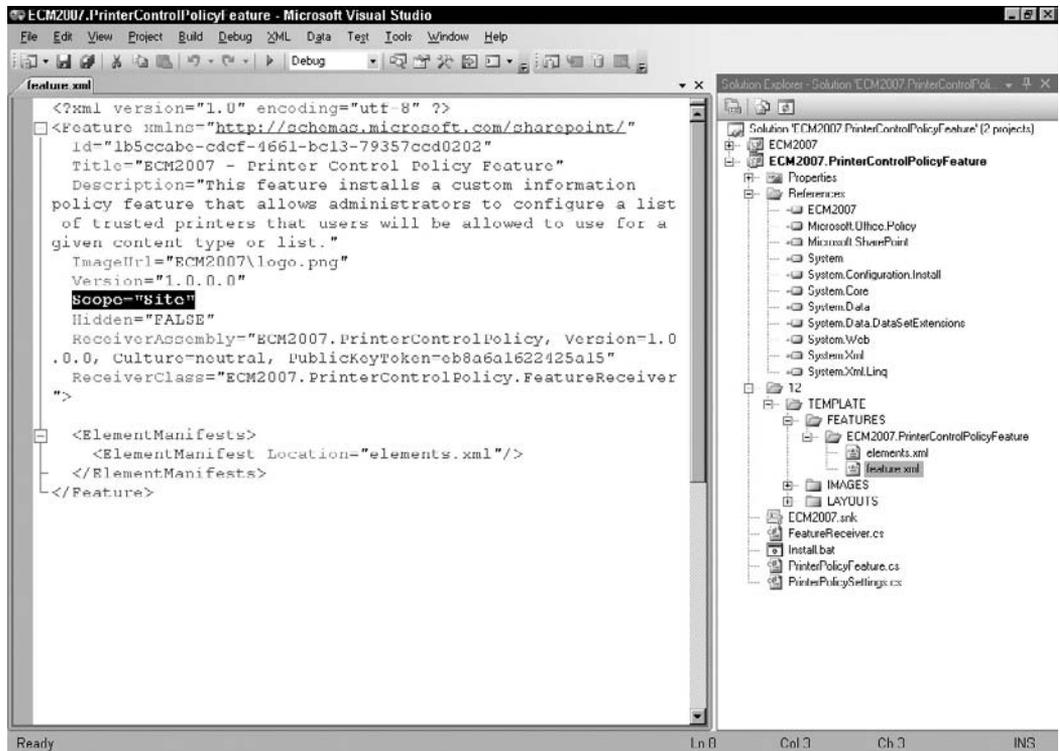


Figure 7-5: Printer control policy feature project details.

In order to use the Records Management features, we need to add a reference to the `Microsoft.Office.Policy` assembly, which is located in the `12\ISAPI` folder. We'll also need to add a reference to the `ECM2007` component library that contains the Information Policy components we built earlier in the chapter.

The `ECM2007` project includes some additional references we need to add in order for our project to build. First, we'll add a reference to the `System.Configuration.Install` assembly from the `.NET` tab so that we can use the `InstallUtil` command-line utility to install our components into the SharePoint environment. We will also need the `System.Web` assembly.

Creating the Policy Feature Class

Policy features allow administrators to select which parts of a policy should be applied to different items. Here, we will create a custom policy feature that maintains a list of *trusted* printers that users are allowed to send output to. To support the policy feature implementation on the client machine, we will create a separate VSTO add-in that will use this information to prevent the end-user from printing to an untrusted printer.

Chapter 7: Information Management Policy

We add a class to the project called `PrinterPolicyFeature` and then add the following using statements at the top of the file.

```
using System;
using System.ComponentModel;
using System.Diagnostics;
using System.Xml;
using Microsoft.Office.RecordsManagement.InformationPolicy;
using Microsoft.SharePoint;

using ECM2007.ContentTypes;
using ECM2007.InformationPolicy;
```

We'll delete the generated class declaration and replace it with the following code:

```
[Name("Document Print Controller")]
[Description("Maintains a list of 'trusted' printers so that administrators "
    + "can control where documents are printed.")]
[Publisher("John F. Holliday")]
public class PrinterPolicyFeature : SharePointPolicyFeature
{
    public const string TrustedPrintersFieldName = "TrustedPrinters";
    public const string PrintControlPolicyNamespace =
        "urn:ecm401:policy:printcontrol";

    public PrinterPolicyFeature()
    {
    }
}
```

Our policy feature class inherits from the `ECM2007.InformationPolicy.SharePointPolicyFeature` abstract base class. This class provides a default implementation of the `IPolicyFeature` interface, which must be implemented to enable SharePoint to communicate with the policy feature. Using an abstract base allows us to implement only the methods we require.

The base class also uses attributes to generate the CAML code needed to register our component in the SharePoint environment. The `TrustedPrintersFieldName` and `PrintControlPolicyNamespace` constants will be used to set up the content types and list items affected by the policy.

First, we'll create a static method for writing trace messages during development:

```
/// <summary>
/// Helper method for logging trace messages.
/// </summary>
public static void Log(string message)
{
    Trace.WriteLine(message, "ECM2007.PrinterPolicyFeature");
}
```

Policy features work by hooking into the event receiver mechanism for the lists and content types to which they are attached. When the printer control policy feature is registered for a content type, it will set up event receivers for the `ItemAdded` and `ItemUpdated` events so that it can copy the list of trusted printers into a special column on the target list item. To make it easier to register and unregister event

receivers, the ECM2007 utility library includes a class called `ItemEventReceiver`. In the next step, we will inherit from this class to declare event receiver methods for these two event types.

We add a new *nested class* declaration inside the `PrinterPolicyFeature` class named `PrinterPolicyEventReceiver` that inherits from `ItemEventReceiver`:

```
/// <summary>
/// This class implements the event receivers for the policy feature.
/// </summary>
public class PrinterPolicyEventReceiver : ItemEventReceiver
{
    /// <summary>
    /// Handles the item added event to set the list of trusted printers.
    /// </summary>
    /// <param name="properties"></param>
    public override void ItemAdded(SPIItemEventProperties properties)
    {
        base.ItemAdded(properties);
        Log("PrinterPolicyFeature.ItemAdded - " + properties.ListTitle);
        AddPrintersToListItem(properties.ListItem);
    }

    /// <summary>
    /// Handles the item updated event to set the list of trusted printers.
    /// </summary>
    /// <param name="properties"></param>
    public override void ItemUpdated(SPIItemEventProperties properties)
    {
        base.ItemUpdated(properties);
        Log("PrinterPolicyFeature.ItemUpdated - " + properties.ListTitle);
        AddPrintersToListItem(properties.ListItem);
    }
}
```

Both of these routines delegate the job of adding the list of printers to the list item whenever a new item is added or an existing item is updated. The list of printers is extracted from the content type payload (a custom `XMLDocument` added by the policy feature) and placed into the *TrustedPrinters* field of the list item. We add the following code to the `PrinterPolicyEventReceiver` class definition:

```
/// <summary>
/// Gets the list of trusted printers from the content type associated
/// with the item and places it into the appropriate field.
/// </summary>
/// <param name="item"></param>
private void AddPrintersToListItem(SPLListItem item)
{
    if (item == null || item.ContentType == null)
        return;
    Log("AddPrintersToListItem '" + item.Title + "'");
    string xml = item.ContentType.XmlDocuments[PrinterPolicyFeature.
        PrintControlPolicyNamespace];
    if (string.IsNullOrEmpty(xml))
    {
        Log("-- content type payload is empty");
    }
}
```

Chapter 7: Information Management Policy

```
    }
    else
    {
        Log("-- loading content type payload");
        XmlDocument xmlDoc = new XmlDocument();
        xmlDoc.LoadXml(xml);
        XmlNamespaceManager nsmgr = new XmlNamespaceManager(xmlDoc.NameTable);
        nsmgr.AddNamespace("p", PrinterPolicyFeature.PrintControlPolicyNamespace);
        XmlNode node = xmlDoc.SelectSingleNode("p:data/p:printers", nsmgr);
        Log("-- '" + node.InnerText + "'");
        SPField field = item.Fields[PrinterPolicyFeature.TrustedPrintersFieldName];

        try
        {
            item[field.Id] = node.InnerText;
            item.SystemUpdate();
            Log("-- UPDATED!");
        }
        catch (SPException x)
        {
            Log(string.Format("-- FAILED to update item - {0}", x.Message));
        }
    }
}
```

Now we are ready to override selected methods of the `SharePointPolicyFeature` base class. We add a region to the `PrinterPolicyFeature` class named `SharePointPolicyFeature Overrides` and insert the following code inside the region:

```
/// <summary>
/// This method is called when the feature is registered for a content type.
/// Adds a "TrustedPrinters" field to the content type.
/// </summary>
/// <param name="ct"></param>
public override void Register(SPContentType ct)
{
    base.Register(ct);
    Log("Registering print control for content type: " + ct.Name);

    // Setup the item event receiver for the content type.
    ItemEventReceiver.Create(ct, typeof(PrinterPolicyEventReceiver));

    // Add the "TrustedPrinters" field to the content type.
    SPFieldLink fieldRef = SharePointContentType.AddOrCreateFieldReference(ct,
        PrinterPolicyFeature.TrustedPrintersFieldName, SPFieldType.Text, false,
        true, true,true,true);
}

/// <summary>
/// This method is called when the policy feature is removed.
/// Uninstalls the event receiver from the content type.
/// </summary>
/// <param name="ct"></param>
public override void UnRegister(SPContentType ct)
```

```
{
    base.UnRegister(ct);
    Log("Unregistering PrintControl for content type: " + ct.Name);
    ItemEventReceiver.Remove(ct, typeof(PrinterPolicyEventReceiver));
}
```

These routines handle the registration and un-registration of the feature for a given content type. The code first creates an `ItemEventReceiver` for the content type using the nested class we just created, and then it adds a `TrustedPrinters` field to the content type.

Next, we will override the `OnCustomDataChange` method, which is called whenever the custom data associated with the policy item changes. This happens when the policy administrator enters a new value into the custom UI we will provide for specifying the list of trusted printer names. We add the following code to the class definition:

```
/// <summary>
/// This method is called when the custom data for a policy item changes.
/// </summary>
/// <param name="policyItem"></param>
/// <param name="ct"></param>
public override void OnCustomDataChange(PolicyItem policyItem, SPContentType ct)
{
    base.OnCustomDataChange(policyItem, ct);
    Log("OnCustomDataChange for policy item: " + policyItem.Name
        + " and content type " + ct.Name + " where policy item custom data = "
        + policyItem.CustomData);

    try
    {
        // Get the custom data from the policy item.
        XmlDocument xmlDoc = new XmlDocument();
        xmlDoc.LoadXml(policyItem.CustomData);
        XmlNamespaceManager nsmgr = new XmlNamespaceManager(xmlDoc.NameTable);
        nsmgr.AddNamespace("p", PrinterPolicyFeature.PrintControlPolicyNamespace);

        // Add the custom data to the content type payload.
        ct.XmlDocuments.Delete(PrinterPolicyFeature.PrintControlPolicyNamespace);
        ct.XmlDocuments.Add(xmlDoc);
        ct.Update();
    }
    catch (Exception x)
    {
        Log("OnCustomDataChange failed for item: " + policyItem.Name + " - " +
            x.ToString());
    }
}
```

SharePoint monitors the life cycle of items affected by a policy and calls the `ProcessListItems` method when updates are needed for a given item. Next, we will override this method to update the `TrustedPrinters` field with the updated list from the policy item. We add the following code to the class definition:

```
/// <summary>
/// This method is called by the policy framework for list items that were
/// created before the policy was applied to the list.
```

Chapter 7: Information Management Policy

```
/// </summary>
/// <param name="site"></param>
/// <param name="policyItem"></param>
/// <param name="listItem"></param>
/// <returns></returns>
public override bool ProcessListItem(SPSite site, Microsoft.Office.
    RecordsManagement.InformationPolicy.PolicyItem policyItem, SPLListItem
    listItem)
{
    Log("Processing list item: " + listItem.Title);
    bool result = base.ProcessListItem(site, policyItem, listItem);

    // Add the policy item custom data to the TrustedPrinters field.
    try
    {
        // Get the policy data from the item payload.
        XmlDocument xmlDoc = new XmlDocument();
        xmlDoc.LoadXml(policyItem.CustomData);

        XmlNamespaceManager nsmgr = new XmlNamespaceManager(xmlDoc.NameTable);
        nsmgr.AddNamespace("p", PrinterPolicyFeature.PrintControlPolicyNamespace);
        XmlNode node = xmlDoc.SelectSingleNode("p:data/p:printers", nsmgr);

        // Store the list of trusted printers into the list item.
        listItem[PrinterPolicyFeature.TrustedPrintersFieldName] = node.InnerText;
        listItem.Update();
    }
    catch (Exception x)
    {
        Log("ProcessListItem failed for item: " + listItem.Title + " - " +
            x.ToString());
    }
    return result;
}
```

In order to test the policy feature, it must be registered in the site collection when the feature is activated. In the `FeatureReceiver.cs` file, we replace the `FeatureActivated` method with the following code:

```
/// <summary>
/// Override to install the printer control policy feature.
/// </summary>
/// <param name="properties"></param>
public override void FeatureActivated(SPFeatureReceiverProperties properties)
{
    SPSite site = properties.Feature.Parent as SPSite;
    if (!SharePointPolicyFeature.Install(typeof(PrinterPolicyFeature)))
        Trace.WriteLine("Printer Policy Feature installation failed.");
}
```

This method gets the `SPSite` object from the feature receiver properties and uses the static `Install` method of the `SharePointPolicyFeature` base class to register the policy feature within the SharePoint environment.

Similarly, we replace the `FeatureDeactivating` method with the code shown below:

```
/// <summary>
/// Override to remove the printer control policy feature.
/// </summary>
/// <param name="properties"></param>
public override void FeatureDeactivating(SPFeatureReceiverProperties properties)
{
    SPSPSite site = properties.Feature.Parent as SPSPSite;
    if (!SharePointPolicyFeature.Uninstall(typeof(PrinterPolicyFeature)))
        Trace.WriteLine("Printer Policy Feature removal failed.");
}
}
```

Now we are ready to test our work so far. Building the project and navigating to any SharePoint site, from the Site Settings page, we can select “Site Collection Features” and see the feature listed.

After activating the feature, we can navigate to any document library and then from the List Settings page, select the “Information management policy settings” link. If the document library has content types enabled, then we will see a list of content types to choose from. Clicking one of the links will take us to the Policy Settings page. If content types are not enabled for the list, we will go directly to that page. From the Policy Settings page, select the “Define a policy” link and click OK. Now, we should see our policy feature displayed at the bottom of the list of available policy features to be enabled, as shown in Figure 7-6.

Policy Statement The policy statement is displayed to end users when they open items subject to this policy. The policy statement can explain which policies apply to the content or indicate any special handling or information that users need to be aware of.	Policy Statement: <input type="text" value="This is the policy statement for end users."/>
Labels You can add a label to a document to ensure that important information about the document is included when it is printed. To specify the label, type the text you want to use in the "Label format" box. You can use any combination of fixed text or document properties, except calculated or built-in properties such as GUID or CreatedBy. To start a new line, use the '\n' character sequence.	<input type="checkbox"/> Enable Labels
Auditing Specify the events that should be audited for documents and items subject to this policy.	<input type="checkbox"/> Enable Auditing
Expiration Schedule content disposition by specifying its retention period and the action to take when it reaches its expiration date.	<input type="checkbox"/> Enable Expiration
Barcodes Assigns a barcode to each document or item. Optionally, Microsoft Office applications can require users to insert these barcodes into documents.	<input type="checkbox"/> Enable Barcodes
Document Print Controller Add the names of trusted printers here, separated by semicolons. This will prevent users from printing documents to which this policy is applied on unsecure printers. NOTE: In order for this to work, you must also deploy the PrintController add-on for Microsoft Office to all client machines.	<input type="checkbox"/> Enable Document Print Controller

OK Cancel Delete

Figure 7-6: Printer Control policy feature configuration.

If we enable the feature, we will get a post-back, but nothing is displayed beneath the checkbox. In the next step, we will add a custom user interface to enable administrators to specify the list of trusted printers.


```
<asp:TextBox ID="TextBoxPrinters" runat="server"
    TextMode="MultiLine"
    Rows="5" MaxLength="1024" Columns="40" class="ms-input"
    ToolTip="Enter the list of trusted printers here." />
<asp:RequiredFieldValidator ID="RequiredValidatorPrinters"
    ControlToValidate="TextBoxPrinters"
    ErrorMessage="At least one printer name is required."
    Text="Please enter a semicolon-delimited list of printer
        names."
    EnableClientScript="false" runat="server" />
</td>
</tr>
</table>
</p>
```

This template declares a `Label`, a `TextBox`, and an associated `FieldValidator` control. It inherits from a code-behind class we will now implement in a separate file.

We add a new class to the project named `PrinterPolicySettings` and replace the generated class with the following code:

```
/// <summary>
/// This class implements a custom Information Policy settings control
/// that enables an administrator to enter the list of trusted printers.
/// </summary>
public class PrinterPolicySettings : CustomSettingsControl
{
    // holds the custom data associated with the control
    string m_customData = string.Empty;

    // automatically bound to the TextBox control in the template
    protected TextBox TextBoxPrinters;
}
```

We'll need to pull in some additional code, so we add the following `using` statements at the top of the file:

```
using System.Web.UI.WebControls;
using System.Xml;
using Microsoft.Office.RecordsManagement.InformationPolicy;
using Microsoft.SharePoint;
```

The `m_customData` string will hold the custom data being transferred to and from the `TextBox` control. The protected `TextBoxPrinters` control is automatically bound to the matching control in the template by ASP.NET.

Our `PrinterPolicySettings` class inherits from a base class that is provided by the SharePoint Information Policy framework for implementing policy feature user interfaces. The `CustomSettingsControl` class is an abstract class that declares several abstract methods. Because they are abstract methods, we must provide an implementation for all of them in our derived class. Listing 7-8 shows the custom settings control methods we are responsible for implementing.

Listing 7-8: Information Policy custom settings control

```
using Microsoft.SharePoint;
using System;
using System.Collections.Specialized;
using System.Web.UI;

namespace Microsoft.Office.RecordsManagement.InformationPolicy
{
    public abstract class CustomSettingsControl : UserControl, IPostBackDataHandler
    {
        protected CustomSettingsControl();

        public abstract SPContentType ContentType { get; set; }
        public abstract string CustomData { get; set; }
        public abstract SPList List { get; set; }

        public abstract bool LoadPostData(string postDataKey,
            NameValueCollection values);
        public abstract void RaisePostDataChangedEvent();
    }
}
```

Several methods will not be used in this solution, so we can enter stubs for the overrides as follows:

```
/// <summary>
/// Not used - must be implemented because base class is abstract.
/// </summary>
public override void RaisePostDataChangedEvent(){}

/// <summary>
/// Not used - must be implemented because base class is abstract.
/// </summary>
public override SPList List { get; set; }

/// <summary>
/// Not used - must be implemented because base class is abstract.
/// </summary>
public override SPContentType ContentType { get; set; }
```

The custom data will consist of a list of printer names that we will embed into an XML fragment so it can be added to the payload of a content type. Add the following method to the class definition:

```
/// <summary>
/// This accessor is called to get or set the custom data
/// associated with the control.
/// </summary>
public override string CustomData
{
    get
```

```
{
    XmlDocument xmlDoc = new XmlDocument();
    string ns = PrinterPolicyFeature.PrintControlPolicyNamespace;
    new XmlNamespaceManager(xmlDoc.NameTable).AddNamespace("p", ns);
    XmlElement rootNode = xmlDoc.CreateElement("p", "data", ns);
    xmlDoc.AppendChild(rootNode);
    XmlElement printersNode = xmlDoc.CreateElement("p", "printers", ns);
    rootNode.AppendChild(printersNode);
    printersNode.InnerText = TextBoxPrinters.Text;
    m_customData = xmlDoc.InnerXml;
    return m_customData;
}
set
{
    m_customData = value;
}
}
```

The `LoadPostData` method is called to set the custom data string after the text in the control is modified.

```
/// <summary>
/// This method updates the custom data associated with the control.
/// </summary>
/// <param name="postDataKey"></param>
/// <param name="values"></param>
/// <returns></returns>
public override bool LoadPostData(string postDataKey, System.Collections.
    Specialized.NameValueCollection values)
{
    string oldData = this.CustomData;
    string newData = values[postDataKey];
    if (oldData != newData)
    {
        this.CustomData = newData;
        return true;
    }
    return false;
}
```

Finally, we need to initialize the `TextBox` with the initial data when the page is loaded.

```
/// <summary>
/// Loads the custom data string into the textbox whenever the control is loaded.
/// </summary>
/// <param name="e"></param>
protected override void OnLoad(EventArgs e)
{
    base.OnLoad(e);
    if (!(base.IsPostBack || string.IsNullOrEmpty(m_customData)))
```

```
{
    try
    {
        XmlDocument xmlDoc = new XmlDocument();
        xmlDoc.LoadXml(m_customData);
        XmlNamespaceManager nsmgr = new XmlNamespaceManager(xmlDoc.NameTable);
        nsmgr.AddNamespace("p", PrinterPolicyFeature.
            PrintControlPolicyNamespace);
        XmlNode node = xmlDoc.SelectSingleNode("p:data/p:printers", nsmgr);
        TextBoxPrinters.Text = node.InnerText;
    }
    catch (Exception x)
    {
        System.Diagnostics.Trace.WriteLine("Failed to load printer settings - "
            + x.Message, GetType().Name);
    }
}
}
```

Now that we have created the control template and implemented its code-behind class, we need to tell SharePoint where it is located on the server. We also need to provide some instructions for the administrator describing the purpose of the feature and how it should be used. Time to reopen the `PrinterPolicyFeature.cs` file for editing:

Because the base `SharePointPolicyFeature` class inherits from the `Installer` class, Visual Studio treats the class file as a component, causing the IDE to attempt to load a designer for it. To go directly to the code editor, we must right-click on the file and choose “View Code.”

Now we can add the following lines to the `PrinterPolicyFeature` constructor:

```
this.ConfigPage = "ECM2007.PrinterControlPolicyFeature/PrinterPolicySettings.ascx";
this.ConfigPageInstructions = @"Add the names of trusted printers "
    + "here, separated by semicolons. This will prevent users from printing "
    + "documents to which this policy is applied on unsecure printers. "
    + "NOTE: In order for this to work, you must also deploy the "
    + "PrintController add-on for Microsoft Office to all client machines.";
```

After building the project again, we can navigate back to the Policy Settings page for the policy we created earlier. Now, our custom instructions appear beneath the “Document Print Controller” caption. When we click on the checkbox next to the “Enable Document Print Controller” policy feature, we should see a labeled `TextBox` control where we can enter the list of trusted printers, as illustrated in Figure 7-7.

To test our work, we can enter some names into the `TextBox` control and save the policy. Whenever we add or update an item to which the policy is attached, we should see the list of printers in the “Trusted Printers” column of the item.

It is not necessary to copy the list of trusted printers into a column of each list item. This was done here only to illustrate the steps that are typically required when writing policy features that need to modify the list item fields or place special metadata into columns. For the purposes of this particular scenario (managing trusted printers), we can access the policy definition along with its custom data directly from the content type payload. In the next section, we will use this approach to prevent users from printing to untrusted printers.

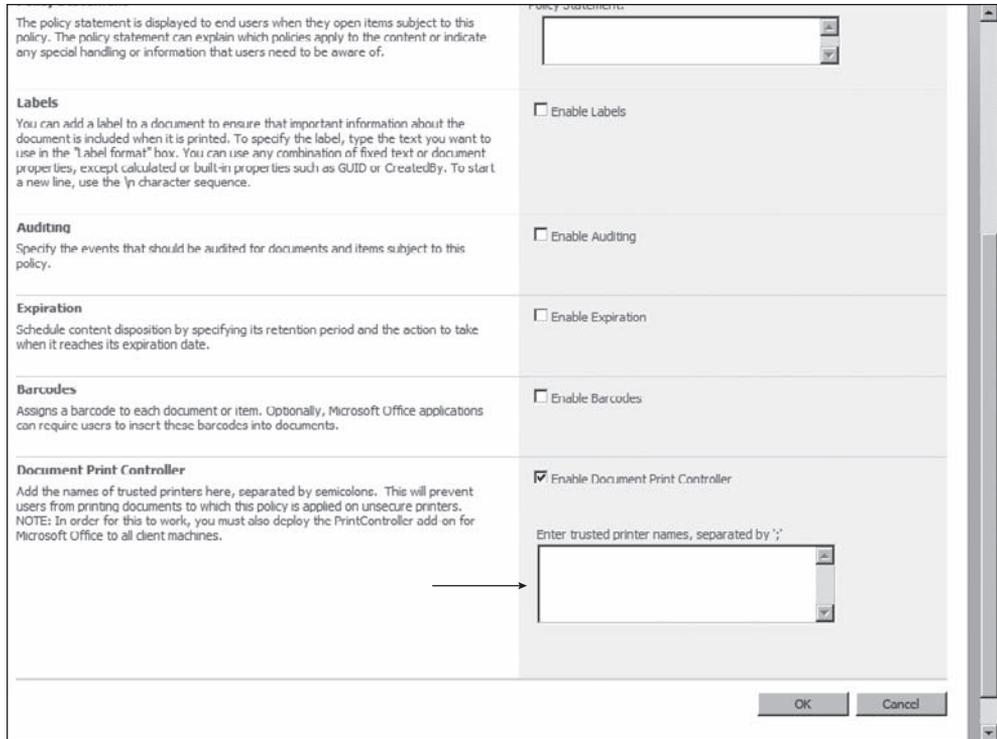


Figure 7-7: Printer Control policy feature settings.

Creating a Print Monitor Add-In for Word 2007

To prevent users from violating the printer control policy, we need some additional code that will run on every client machine. This code must be installed on each client and configured as an application-level add-in for the Office client applications. In this step, we will use Visual Studio Tools for Office to create a print monitor add-in for Word 2007 that will read the trusted printer list from the document and compare it to the currently active printer. If the currently active printer name is not in the list, then any attempt to print the document is aborted with an error message.

When an information policy is attached to a content type, a copy of the policy is embedded within any document that is associated with the type. This means that the entire policy definition travels with the document and is accessible from our add-in code. We can take advantage of this by creating a new VSTO project called *ECM2007.PrintMonitorAddin*. From the New Project dialog as shown in Figure 7-8, we select “Word 2007 Add-in” from the Office/2007 section.

Open the *ThisAddIn.cs* file for editing, and add the following namespaces to the `using` statements at the top of the file:

```
using System.Xml;
using System.Windows.Forms;
```

Chapter 7: Information Management Policy

The first bit of code will handle the `Startup` event to install a handler for the `BeforePrint` event. This handler will be called whenever the user attempts to print the active document. Add the following code inside the `ThisAddIn_Startup` event handler that was auto-generated by the New Project Wizard.

```
// install a handler for the print event
this.Application.DocumentBeforePrint
+= new Word.ApplicationEvent4_DocumentBeforePrintEventHandler (
    Application_DocumentBeforePrint);
```

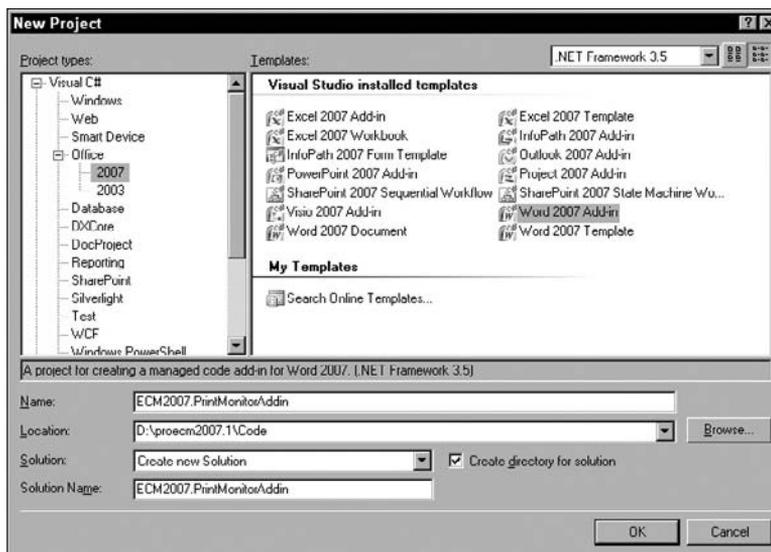


Figure 7-8: New Project dialog for PrintMonitor add-in.

Replace the generated event-handler stub with the following code:

```
/// <summary>
/// This event handler gets the name of the printer the user is
/// attempting to use. It then obtains the list of "trusted" printers
/// as defined by the PrintControl policy associated with the document.
/// If the current printer is not trusted, then a message is displayed
/// and the operation is cancelled.
/// </summary>
/// <param name="Doc"></param>
/// <param name="Cancel"></param>
void Application_DocumentBeforePrint(Word.Document Doc, ref bool Cancel)
{
    if (!PrinterIsTrusted(Application.ActivePrinter))
    {
        MessageBox.Show(String.Format(
            "Sorry. Big Brother does not want you to print this document " +
            "on printer '{0}'! Please select a different printer.",
            Application.ActivePrinter),
            "Printer Is Not Trusted");
    }
}
```

```
        Cancel = true;
    }
}
```

The next routine will return true if the specified printer is a trusted printer as defined by the policy:

```
/// <summary>
/// Determines whether a specified printer is trusted.
/// </summary>
/// <param name="printerName"></param>
/// <returns>true if the specified printer is in the list of trusted
/// printers</returns>
bool PrinterIsTrusted(string printerName)
{
    // Retrieve the list of trusted printers from the attached policy.
    List<string> trustedPrinters = GetTrustedPrintersList();
    // Check whether the specified printer is in the list.
    if (trustedPrinters.Count == 0)
        return true;
    return trustedPrinters.Contains(printerName);
}
```

Finally, we need a routine to extract the list of trusted printers from the information policy attached to the document. For Word 2007 documents the information policy is embedded as a `CustomXMLPart` object within the active document. The collection of custom XML parts is exposed as a property of the `Microsoft.Office.Interop.Word.Document` object. The following code performs the lookup operation:

```
/// <summary>
/// Retrieves the list of trusted printers from the PrintControl policy
/// associated with the document.
/// </summary>
/// <remarks>
/// This method searches through the list of custom XML parts in
/// the document until it finds one matching the PrintControl policy URN.
/// </remarks>
/// <returns>the list of trusted printers from the attached policy</returns>
List<string> GetTrustedPrintersList()
{
    List<string> result = new List<string>();
    const string policyNamespace = "urn:ecm401:policy.printcontrol";
    Office.CustomXMLParts parts = Application.ActiveDocument.CustomXMLParts;
    foreach (Office.CustomXMLPart part in parts)
    {
        if (part.XML.Contains(policyNamespace))
        {
            // extract the list of trusted printers.
            XmlDocument xmlDoc = new XmlDocument();
            xmlDoc.LoadXml(part.XML);
            XmlNamespaceManager nsmgr = new XmlNamespaceManager(xmlDoc.NameTable);
            nsmgr.AddNamespace("p", policyNamespace);
            XmlNode node = xmlDoc.SelectSingleNode("p:data/p:printers", nsmgr);
            if (node != null)
            {

```

Chapter 7: Information Management Policy

```
string[] printers = node.InnerText.Split(";".ToCharArray());
foreach (string printer in printers)
    result.Add(printer);
break;
    }
}
}
return result;
}
```

To test our work, we can simply run the project in the debugger and open a document from the document library we used to test the information policy. When we try to print the document, we should see a dialog like the one shown in Figure 7-9.

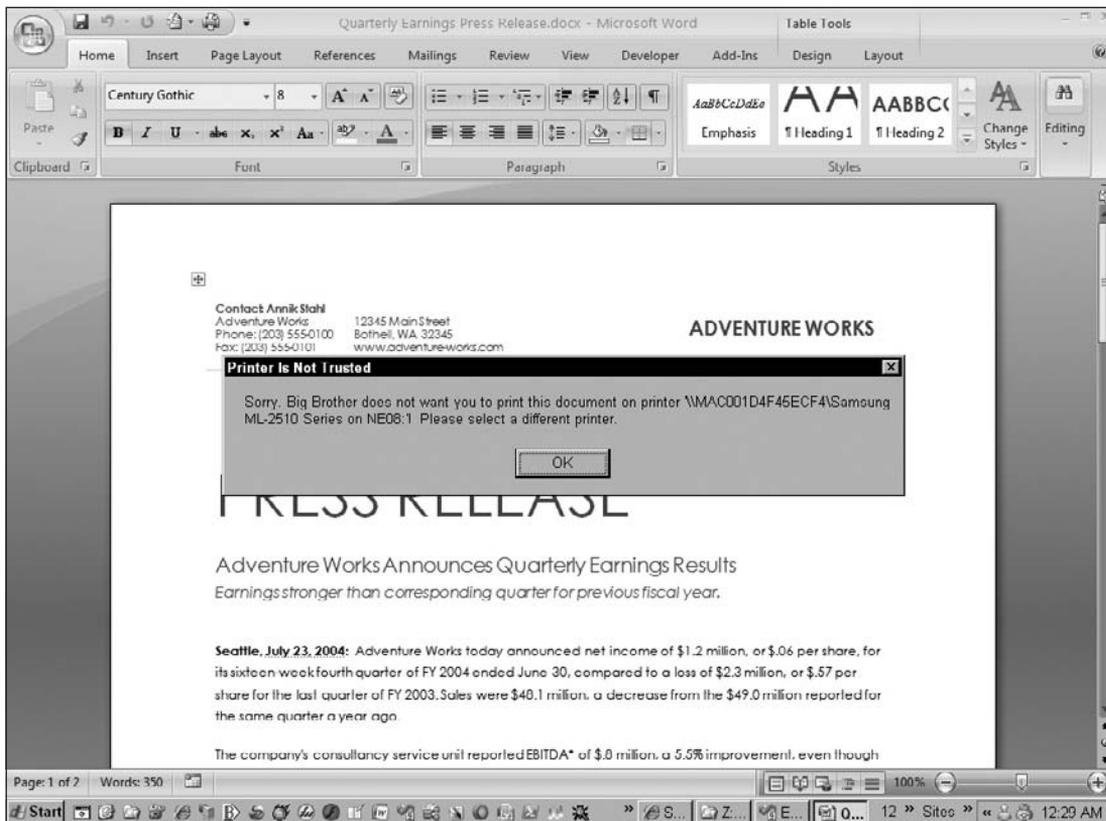


Figure 7-9: Printer policy error dialog.

In actual practice, when creating a VSTO add-in, we would need to also create a setup project to install the add-in on client machines.

Summary

Microsoft Office SharePoint Server 2007 provides an extensible framework for defining and applying custom information policies to individual list items. The Information Management Policy framework enables administrators to attach custom policy statements and custom code to lists or to content types, eliminating the need to modify and redeploy the code if the policy statement changes or if the policy must be removed or applied to a different set of objects.

This chapter described the Information Policy framework architecture and introduced the policy feature and policy resource components, showing how they work together to provide a complete package of functionality for a given policy. The chapter also described the information policy life cycle to help you understand how the code that is implemented in a policy feature is invoked when changes are made to policy settings or list items.

Since the declaration and deployment of information policy components require a precise coordination between the XML documents used to describe the components and the code used to implement them, this chapter showed how to create a library of reusable components and interfaces to make it easier to create custom policy features and policy resources.

In order to build effective information policy–based solutions, it is often necessary to build cooperating code that must run on the client. Several strategies exist for passing information between the server and the client to synchronize the behavior of client and server components. The Information Policy framework supports this by embedding the policy statement directly within Office 2007 documents as custom XML objects that can be accessed using client-side code.

8

Information Policy and Record Retention

Since the Information Policy framework was developed precisely to deal with records management issues, you can learn a lot by studying the default implementations of the built-in policy features. In this chapter, we'll look at the Expiration Policy Feature, which uses a straightforward design pattern that addresses the competing goals of enabling administrators to define complex expiration policies while allowing developers to extend the policy feature with custom expiration formulas and actions. We'll also look beyond the core implementation to gain an understanding of the special Expiration Timer Job and its role in ensuring that the retention period for individual list items is calculated correctly and that any associated actions are performed in a timely manner. Finally, we'll look for ways to extend our evolving File Plan schema to include support for automatically applying an expiration policy to documents and list items.

The Expiration Policy Feature

In Chapter 7, we explored the Information Policy architecture and the structure of custom policy features and policy resources. In this chapter, let's take a deeper look at the built-in Expiration Policy Feature to gain a better understanding of how it works. I'm calling attention to this particular policy feature because it implements an interesting design pattern that we can apply to our own policy features and policy resources.

The Expiration Policy Feature is used to control how long documents are held in the content database. This is such a core requirement that much of the Information Policy architecture was built specifically to support this scenario. The primary goal is to provide a way to allow content managers to define both the formula used to determine the expiration date of an item as well as the action to take when that date arrives.

Chapter 8: Information Policy and Record Retention

As you can imagine, there are several challenges to making this work efficiently. The obvious problem is determining when to calculate the expiration date. The next problem is figuring out when to check the date of each item without affecting the performance of the system as a whole. It turns out that the Information Policy architecture provides a convenient way for SharePoint to address these issues. The first part has to do with optimization.

Given the fact that there may be hundreds of thousands of list items in the content database at any given time, we obviously need a timer job that runs periodically to determine whether a particular item has *expired*. But since document retention is managed by a specific information policy feature, we can safely ignore any items to which that policy feature has not yet been applied.

So far, so good.

The second thing to note is that we can go ahead and compute the expiration date at the moment the policy feature is activated. This allows the policy feature to place a pre-computed date into a custom field associated with the list item. That way, our timer job can easily compare the system date to the stored date to determine if the item has expired.

While this may seem like a good idea at first, it actually limits the kinds of expiration formulas we can build. We could not, for example, build dynamic expiration formulas that determine the expiration date based on changing external data because the expiration formula is called only once — when the policy feature is activated or its settings are changed.

Creating Custom Expiration Formulas and Actions

You will recall from Chapter 7 that an information policy consists of one or more policy features, which are classes that implement the `IPolicyFeature` interface. Each policy feature may rely on one or more policy resources, which are typically used to provide low-level support for the policy feature implementation. Consequently, policy features and policy resources are generally implemented together, and there is no restriction on how they are structured. The idea is to place as few constraints as possible on how the policy features and policy resources interact, leaving it to the discretion of the designer to declare the appropriate interfaces needed to fulfill a given set of requirements.

This open architecture is exploited very effectively by the built-in Expiration Policy Feature as illustrated in Figure 8-1.

To take advantage of this design, you create and install your own custom expiration formulas as policy resources that are linked to the Expiration Policy Feature. Policy resources must implement the `IPolicyResource` interface. Policy resources are linked to a specific policy feature by specifying the feature identifier in the policy resource definition, which is used to tell SharePoint which policy feature the resource is associated with. Because of this mechanism, each resource can be linked to only one policy feature at a time, although a policy feature can be associated with more than one policy resource.

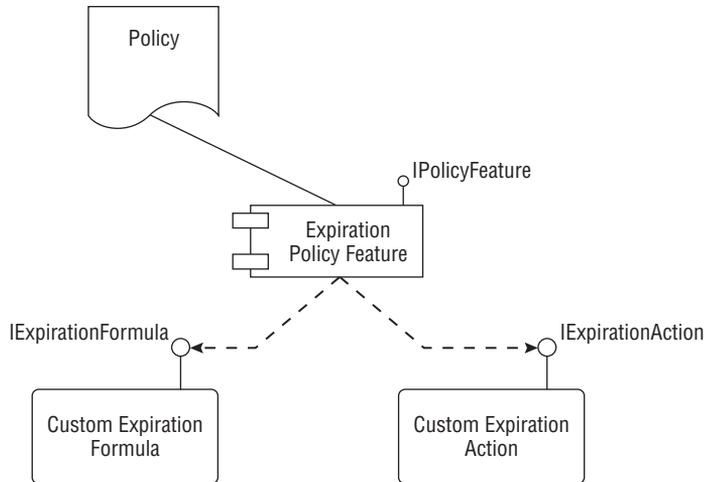


Figure 8-1: Expiration Policy Feature architecture.

The Expiration Policy Feature communicates with your custom expiration formula policy resource through a special interface provided for that purpose. The `IExpirationFormula` interface is declared in the `Microsoft.Office.RecordsManagement.PolicyFeatures` namespace. This interface declares a single method that is called to compute the expiration date for a given list item. The interface definition is shown in Listing 8-1.

Listing 8-1: `IExpirationFormula` interface

```
using Microsoft.SharePoint;
using System;
using System.Xml;

namespace Microsoft.Office.RecordsManagement.PolicyFeatures
{
    public interface IExpirationFormula
    {
        DateTime? ComputeExpireDate(SPListItem item, XmlNode parametersData);
    }
}
```

The `DateTime?` syntax was introduced as part of the .NET Framework 2.0. It means that the method returns a nullable type, which solves many problems like storing null values in a database.

All you have to do to define a custom expiration formula is create a class that implements this interface and then load it into the SharePoint environment as a valid policy resource component. Writing the component is easy. The tricky part is creating the appropriate CAML code needed to register the component correctly so that both SharePoint and the Expiration Policy Feature will recognize it.

Expiration Formulas

Using the helper classes we developed in Chapter 7, we can easily build a custom expiration formula. Our custom formula will simply calculate the expiration date for a list item based on the current date and time. First, we derive a new class from our `SharePointPolicyResource` component and then implement the `IExpirationFormula` interface as shown in Listing 8-2.

Listing 8-2: A custom expiration formula component

```
using System;
using System.ComponentModel;
using ECM2007.InformationPolicy;
using Microsoft.Office.RecordsManagement.PolicyFeatures;
using Microsoft.SharePoint;

namespace ECM2007.DocumentExpiration
{
    [Name("Custom Expiration Formula")]
    [Description("Calculates document expiration based on current time.")]
    [RunInstaller(true)]
    public class CustomExpirationFormula : SharePointPolicyResource,
        IExpirationFormula
    {
        /// <summary>
        /// Override to associate this custom formula component with the
        /// correct policy feature for document expiration.
        /// </summary>
        protected override string FeatureId
        {
            get
            {
                return "Microsoft.Office.RecordsManagement.PolicyFeatures.
                    Expiration";
            }
        }

        /// <summary>
        /// Override to specify the policy resource type.
        /// </summary>
        protected override string ResourceType
        {
            get
            {
                return "DateCalculator";
            }
        }

        #region IExpirationFormula Implementation

        /// <summary>
```

```
/// Simple implementation that expires the item immediately.
/// </summary>
public Nullable<DateTime> ComputeExpireDate(SPListItem item,
    System.Xml.XmlNode parametersData)
{
    return DateTime.Now;
}

#endregion
}
}
```

Notice the two overridden properties `FeatureId` and `ResourceType`. These are provided by our base class to enable us to specify the identifier of the policy feature to which our policy resource belongs, and any additional information needed by that policy feature. For the `FeatureId` property, we return the value `Microsoft.Office.RecordsManagement.PolicyFeatures.Expiration`, which uniquely identifies the built-in Expiration Policy Feature. For the `ResourceType` we must return the value `DateCalculator`, which is a special value that the Expiration Policy Feature recognizes as an expiration formula.

For this simple example, we'll implement the `ComputeExpireDate` method to just return the current date as the expiration date. This will cause the associated list item to expire immediately. In actual practice, we could have retrieved some data from the list item itself or pulled configuration data from a separate list and then used it to compute the expiration date. A typical scenario might require that we correlate the expiration date of one item with the value of some field in another item. The important point to remember is that the expiration date is computed when the policy is applied, and not when the expiration date arrives. This means that the `ComputeExpireDate` method will be called only once for each list item unless something happens that causes the policy to be applied to the item again, such as changing the policy settings as explained earlier.

If we wanted to implement a more sophisticated solution that modifies the expiration date when some other event occurs, then we would need to programmatically modify the policy so that SharePoint recomputes the expiration date for all affected items. We could do this, for example, from an event receiver attached to the master item. This would require careful coordination between the various components of the solution, however, to ensure that a specific policy is active and available for modification.

You will recall from previous chapters that we took advantage of the Installer tool provided by the .NET Framework (`Installutil.exe`) to install our server resources during development by executing custom installer components within an assembly from the command line. We can do it again here simply by adding the appropriate command to the post-build events in Visual Studio, as shown in Figure 8-2.

Now when we build the project, our custom expiration formula is automatically installed into the local SharePoint farm. Figure 8-3 shows our new formula as it appears in the dropdown list of the Policy configuration page.

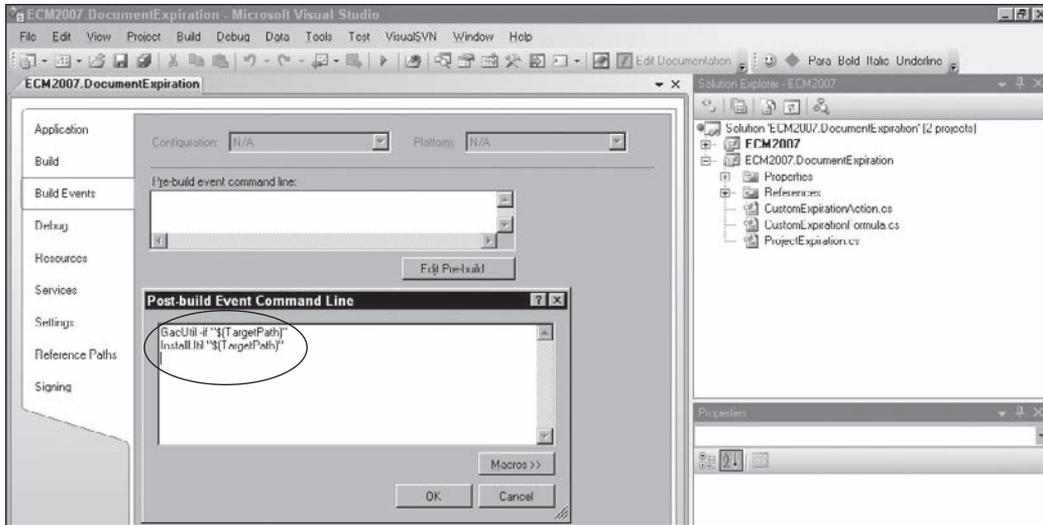


Figure 8-2: InstallUtil called from post-build events.

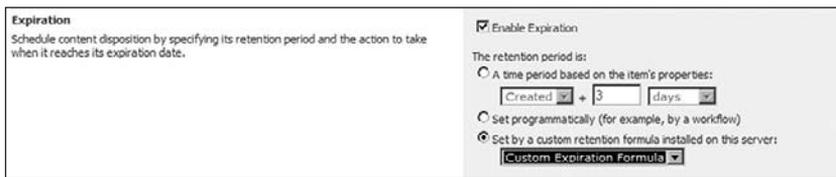


Figure 8-3: Custom expiration formula selection.

Expiration Actions

Custom expiration formulas allow us to control the computed expiration date, but it's also useful to develop custom expiration actions to control what happens when a document expires. This is done in a very similar fashion to the expiration formulas, except we instead implement the `IExpirationAction` interface and provide a different value for the `ResourceType` property included in the policy resource CAML definition. Instead of returning the value `DateCalculator`, we return the value `Action` to tell the Expiration Policy Feature that this policy resource implements an expiration action. Here, we implement a custom expiration action that adds an item to a custom expiration history list whenever an item expires. Listing 8-3 shows the completed component.

Listing 8-3: CustomExpirationAction class

```
using System;
using System.ComponentModel;
using System.Xml;
using ECM2007.InformationPolicy;
```

Chapter 8: Information Policy and Record Retention

```
using Microsoft.Office.RecordsManagement.PolicyFeatures;
using Microsoft.SharePoint;

namespace ECM2007.DocumentExpiration
{
    [Publisher("John F. Holliday")]
    [Name("Expiration Event Logger")]
    [Description("Writes to the expiration history list when an item expires.")]
    [RunInstaller(true)]
    public class CustomExpirationAction : SharePointPolicyResource,
        IExpirationAction
    {
        protected override string FeatureId
        {
            get
            {
                return "Microsoft.Office.RecordsManagement.PolicyFeatures.
                    Expiration";
            }
        }

        protected override string ResourceType
        {
            get
            {
                return "Action";
            }
        }

        #region IExpirationAction Members

        /// <summary>
        /// Handles the action to be performed when the item expires.
        /// </summary>
        /// <remarks>
        /// Writes an entry to the expiration history list.
        /// </remarks>
        /// <param name="item"></param>
        /// <param name="parametersData"></param>
        /// <param name="expiredDate"></param>
        void IExpirationAction.OnExpiration(SPListItem item,
            XmlNode parametersData, DateTime expiredDate)
        {
            SPSecurity.RunWithElevatedPrivileges(delegate()
            {
                using (SPSite site = new SPSite(item.Web.Url))
                using (SPWeb web = site.OpenWeb())
                {
                    new ExpirationHistoryList(web).WriteEntry(
                        string.Format("The item '{0}' in list '{1}' of web '{2}'
                            has expired",
                                item.Title, item.ParentList.Title, web.Title));
                }
            });
        }
    }
}
```

Continued

Listing 8-3: CustomExpirationAction class (continued)

```
    }  
  
    #endregion  
}  
}
```

To see the expiration action in operation, we'll create a generic list that displays a message for each expired item. We can use a simple helper class that creates the list if it does not already exist and then appends an item containing a specified string.

```
public class ExpirationHistoryList  
{  
    private SPList m_list = null;  
    public ExpirationHistoryList(SPWeb web)  
    {  
        // Create or open a custom list named "Expiration History".  
        const string name = "Expiration History";  
        const string description = "Records expiration events.";  
        try { m_list = web.Lists[name]; } catch { }  
        if (m_list == null)  
        {  
            try  
            {  
                m_list = web.Lists[  
                    web.Lists.Add(name, description, SPListTemplateType.  
                        GenericList)  
                ];  
                m_list.OnQuickLaunch = true;  
                m_list.Update();  
            }  
            catch (Exception x)  
            {  
                Trace.WriteLine("Failed to create expiration history list - "  
                    + x.Message, "CustomExpirationAction");  
            }  
        }  
    }  
  
    public void WriteEntry(string description)  
    {  
        try  
        {  
            SPListItem newItem = m_list.Items.Add();  
            newItem["Title"] = description;  
            newItem.Update();  
        }  
        catch (Exception x)  
        {  
            Trace.WriteLine("Failed to write to expiration history list - "  
                + x.Message, "CustomExpirationAction");  
        }  
    }  
}
```

Now when we re-build the project, our custom expiration action is installed, and we can navigate back to the Policy configuration page to select the expiration action in the dropdown list, as shown in Figure 8-4.

Expiration
Schedule content disposition by specifying its retention period and the action to take when it reaches its expiration date.

Enable Expiration

The retention period is:

A time period based on the item's properties:
Created + 3 days

Set programmatically (for example, by a workflow)

Set by a custom retention formula installed on this server:
Custom Expiration Formula

When the item expires:

Perform this action:
Expiration Event Logger

Start this workflow:
Collect Feedback

Figure 8-4: Custom expiration action selection.

The Expiration Timer Job

The final piece of the expiration puzzle is understanding how the expiration timer job works. I already mentioned that in order for expiration to work efficiently without affecting the performance of the farm, there must be a timer job that checks periodically for list items that have expired. Although it doesn't have to check every list item in the content database, it must still run as a timer job. There is a specific timer job, called the *Expiration* timer job, that does just that.

Information Policy features are configured from the Central Administration web site on the Operations tab under the "Security Configuration" section. Clicking on the "Information management policy configuration" link takes us to the Policyfeatures.aspx page, shown in Figure 8-5.

Central Administration > Operations > Policy Features and Resources

Information Management Policy Configuration

This list displays all of the available information management policy feature for use within lists, libraries, and content types.

Name	Description	Publisher	Availability
Labels	Generates labels that can be inserted in Microsoft Office documents to ensure that document properties or other important information are included when documents are printed. Labels can also be used to search for documents.	Microsoft	Available
Auditing	Audits user actions on documents and list items to the Audit Log.	Microsoft	Available
Expiration	Automatic scheduling of content for processing, and expiry of content that has reached its due date.	Microsoft	Available
Barcodes	Generates unique identifiers that can be inserted in Microsoft Office documents. Barcodes can also be used to search for documents.	Microsoft	Available
Document Print Controller	Maintains a list of 'trusted' printers so that administrators can control where documents are printed.	John F. Holiday	Available

Figure 8-5: Policy Configuration page in Central Administration.

Chapter 8: Information Policy and Record Retention

From here, we'll click on the "Expiration" link to get to the Configure Expiration page shown in Figure 8-6.

Configure Expiration

Settings
Configure the system-wide settings for this policy feature.

Automatically find and process expired items
Schedule this process to run:
 Daily
 Weekly on Sunday
Between 11 PM 00
and 11 PM 30
Process Expired Items Now

Availability
Specify whether this feature is available for use in lists, libraries and content types.

Resources
Configure settings for each of the resources available to this policy feature.

Status
 Available for use in new site and list policies.
 Decommissioned: Unavailable to new site and list policies, but still available in existing policies that use it.

Records Center Expiration Action
Expires abandoned files submitted to the Records Center server.

Custom Expiration Formula
Calculates document expiration based on current time.

Expiration Event Logger
Writes to the event log when an item expires.

Save **Cancel**

Figure 8-6: Configure Expiration page.

This page provides controls for scheduling the expiration timer job as well as a button for running the job immediately. At the bottom of the page is the current set of installed policy resources that return a FeatureId property value of `Microsoft.Office.RecordsManagement.PolicyFeatures.Expiration`.

Returning to the main site, we can now create a sample document and have it expire immediately. Although the document expires immediately by virtue of our custom policy, the expiration history list won't show up until we run the expiration timer job. To do that, we can press the "Process Expired Items Now" button from Central Administration and wait for a minute or so until the timer job actually runs. Figure 8-7 shows the resulting entry when the expired document is processed and our custom action is called.

Professional Records Management

Welcome System Account | My Site | My Links

Professional Records Management | Records Center

This List: Expiration History

Site Actions

Professional Records Management > Expiration History

Expiration History

Records expiration events.

New - Actions - Settings - View: **All Items**

Title
The item 'Sample Document' in list 'Shared Documents' of web 'Professional Records Management' has expired

Figure 8-7: Custom expiration history list showing expired documents.

Planning for Retention

Now that we understand how the Expiration Policy Feature works, we can refactor the file planning exercise so that we can include document retention into the file plan. The advantage of doing this is that we can use it to more fully encapsulate the life cycle of a particular record type. By including more detailed document retention instructions into the file plan, it becomes a kind of *living document specification* that can enhance and simplify the processing of a document throughout its life cycle.

Extending the File Plan to Include Expiration Policies

We would like to end up with information in the file plan that includes document retention instructions that can be applied directly to the Expiration Policy Feature. The default file planning worksheet already includes general information about document retention. It describes the retention period in terms of a fixed number of days, weeks, months, or years and may include a general statement about what to do when that period expires, such as “archive,” “destroy,” and so on.

Now, we’d like to capture more detailed information that describes how to calculate the expiration date and what steps to take when the retention period ends. We can drive this specification from the fields that are included in the built-in Expiration Policy Feature. The following table shows the additional columns needed to capture this information:

Element	Description
Expiration Policy Type	Specifies how the expiration date should be calculated. Valid values include: <code>time-span</code> , <code>programmatic</code> , or <code>custom</code> .
Expiration Period Start	Specifies how to determine the starting date for calculating the retention period. This can be specified as the value of a document property or a static value.
Expiration Period Units	Specifies the unit of measurement for the expiration period. Valid values include <code>days</code> , <code>weeks</code> , <code>months</code> , or <code>years</code> .
Expiration Formula	Describes the method used to calculate the expiration period. This can be an actual formula or a reference to a custom formula implemented elsewhere.
Expiration Action	Specifies what to do when the retention period ends. Valid values are <code>Archive</code> , <code>Delete</code> , <code>StartWorkflow</code> , or <code>Custom</code> .
Expiration Action Data	Specifies additional data needed to perform the expiration action.

Once we’ve added the columns to file plan schema, the next step is to add support to the file plan.

Adding Expiration Policy Support to the Dynamic File Plan

We can now extend the dynamic file plan schema and the associated file-planning components we developed in Chapter 5 to support the automatic construction of document retention policies whenever a file plan is executed. Once we’ve added the required elements to the schema and regenerated the

Chapter 8: Information Policy and Record Retention

components, we can then write some additional code that translates the supplied field values into actual policies when the content types and document libraries are created during file plan execution.

You will recall that our file plan schema includes an element type called `RecordSpecification` that is used to describe each official record type. We will now extend that element definition to include an `Expiration` subelement based on a new complex type called `ExpirationPolicyFeature`.

```
<xs:complexType name="ExpirationPolicyFeature">
  <xs:annotation>
    <xs:documentation>
      Specifies how the record expiration policy feature should
      be configured for a record type.
    </xs:documentation>
  </xs:annotation>
  <xs:all>
    <xs:element name="Policy" type="ExpirationPolicy" nillable="true"/>
    <xs:element name="Action" type="ExpirationAction" nillable="true"/>
  </xs:all>
</xs:complexType>
```

This element consists of two subelements: one to describe the expiration policy and another to describe the expiration action. The `ExpirationPolicy` element is shown below:

```
<xs:complexType name="ExpirationPolicy" mixed="true">
  <xs:annotation>
    <xs:documentation>
      Specifies the expiration policy details for a record series.
    </xs:documentation>
  </xs:annotation>
  <xs:choice>
    <xs:element name="TimePeriod" type="ExpirationPeriod"/>
    <xs:element name="Function" type="ExpirationFunction"/>
    <xs:element name="Custom" type="ExpirationFormula"/>
  </xs:choice>
  <xs:attribute name="Type" type="ExpirationPolicyType"/>
</xs:complexType>
```

Keeping in mind that we will ultimately have to translate this information into an actual policy attached to the content type for each record, we'll factor the individual pieces into separate elements so that when the `XSD.exe` utility converts them into classes, there will be a separate class for each subcomponent. Thus, we'll have separate classes for the `ExpirationPeriod`, `ExpirationFunction`, and `ExpirationFormula` elements referenced in the element. Note that these are declared within an `xs:choice` element so there can be only one specified at a time.

We'll also use an enumerated type to specify the type of policy being applied so that we can extend it easily.

```
<xs:simpleType name="ExpirationPolicyType">
  <xs:annotation>
    <xs:documentation>
      Specifies the type of expiration policy to apply to the record.
    </xs:documentation>
  </xs:annotation>
</xs:simpleType>
```

```
</xs:annotation>
<xs:restriction base="xs:string">
  <xs:enumeration value="TimeSpan"/>
  <xs:enumeration value="Programmatic"/>
  <xs:enumeration value="Custom"/>
</xs:restriction>
</xs:simpleType>
```

The `ExpirationPeriod` simply declares a starting date and a span of time, but each of these can be customized to specify the document property used as the starting date and the units used to calculate the span.

```
<xs:complexType name="ExpirationPeriod" mixed="true">
  <xs:annotation>
    <xs:documentation>
      Specifies a time period for expiration.
    </xs:documentation>
  </xs:annotation>
  <xs:attribute name="FromDate" type="ExpirationPeriodType"/>
  <xs:attribute name="Span" type="ExpirationPeriodSpanType"/>
</xs:complexType>

<xs:simpleType name="ExpirationPeriodType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Created"/>
    <xs:enumeration value="Modified"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="ExpirationPeriodSpanType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Days"/>
    <xs:enumeration value="Months"/>
    <xs:enumeration value="Years"/>
  </xs:restriction>
</xs:simpleType>
```

Next, we add the `ExpirationFunction` and `ExpirationFormula` elements. These are essentially placeholders to capture the name of a custom expiration function or workflow and the name of any custom expiration formula that might be installed in the system. Depending on how you choose to implement the associated components, additional information can be included here, such as the name of an assembly and class that implements the custom formula, or an identifier for looking up additional information when the file plan is executed.

These are specified as mixed types so that the name can be written directly within the element tags instead of having to use an attribute.

```
<xs:complexType name="ExpirationFunction" mixed="true">
  <xs:annotation>
    <xs:documentation>
      Specifies the use of an unspecified programmatic function
      (such as a workflow) that will calculate the expiration date.
    </xs:documentation>
  </xs:annotation>
```

```
</xs:annotation>
</xs:complexType>

<xs:complexType name="ExpirationFormula" mixed="true">
  <xs:annotation>
    <xs:documentation>
      Specifies the name of a custom expiration formula to use.
    </xs:documentation>
  </xs:annotation>
</xs:complexType>
```

With the `ExpirationFormula` element defined, we can add the specification for the `ExpirationAction` element. This element can be declared as one of three types: `BuiltIn`, `StartWorkflow`, or `Custom`.

```
<xs:complexType name="ExpirationAction" mixed="true">
  <xs:annotation>
    <xs:documentation>
      Specifies the expiration action details for a record series.
    </xs:documentation>
  </xs:annotation>
  <xs:choice>
    <xs:element name="BuiltIn" type="BuiltInExpirationAction"/>
    <xs:element name="StartWorkflow" type="WorkflowExpirationAction"/>
    <xs:element name="Custom" type="CustomExpirationAction"/>
  </xs:choice>
</xs:complexType>
```

The `BuiltInExpirationAction` is simply a string that specifies the name of one of the built-in expiration actions, such as `Delete`.

Since the list of built-in expiration actions is known, we could declare them as an enumerated type. I've chosen to leave it as a string to avoid having to revisit the schema in case additional actions are added to the platform in the future.

Similarly, the `WorkflowExpirationAction` simply captures the name of the workflow to start when the retention period ends, and the `CustomExpirationAction` captures the name of the custom action to search for and attach to the policy.

```
<xs:complexType name="WorkflowExpirationAction" mixed="true">
  <xs:annotation>
    <xs:documentation>
      Specifies the name of a workflow to start when the
      record expires.
    </xs:documentation>
  </xs:annotation>
</xs:complexType>

<xs:complexType name="CustomExpirationAction" mixed="true">
  <xs:annotation>
    <xs:documentation>
      Specifies the name of a custom expiration action
      to execute when the record expires.
    </xs:documentation>
  </xs:annotation>
</xs:complexType>
```

```
</xs:annotation>  
</xs:complexType>
```

With these elements in place, we can now write the code needed to create an expiration policy for a record type when the file plan is executed. To do this, we need to extend the `CreateContentTypes` method of the `RecordSpecification` class so that it calls a method of the embedded `ExpirationPolicyFeature` class that then creates and attaches the policy. The `ExpirationPolicyFeature` class is generated as a partial class along with the other classes from the schema. We can extend this class by writing the code shown in Listing 8-4.

Listing 8-4: ExpirationPolicyFeature.cs

```
using System;  
using System.Text;  
using Microsoft.Office.RecordsManagement.InformationPolicy;  
using Microsoft.SharePoint;  
  
namespace ECM2007.RecordsManagement  
{  
    public partial class ExpirationPolicyFeature  
    {  
        /// <summary>  
        /// Creates and attaches any expiration policy which may have been  
        /// specified along with the record.  
        /// </summary>  
        /// <param name="type">Identifies the content type to which the expiration  
        /// policy will be attached.</param>  
        public void CreateExpirationPolicy(SPContentType type)  
        {  
            try  
            {  
                // check if there is a policy attached to the content type  
                // if not, then create one so we can configure it  
                Microsoft.Office.RecordsManagement.InformationPolicy.Policy  
                    targetPolicy =  
                    Microsoft.Office.RecordsManagement.InformationPolicy.Policy.  
                        GetPolicy(type);  
                if (targetPolicy == null)  
                {  
                    Microsoft.Office.RecordsManagement.InformationPolicy.Policy.  
                        CreatePolicy(type, null);  
                    targetPolicy = Microsoft.Office.RecordsManagement.  
                        InformationPolicy.Policy.GetPolicy(type);  
                }  
  
                // add the expiration policy to the content type  
                const string expirationPolicyFeatureId =  
                    "Microsoft.Office.RecordsManagement.PolicyFeatures.Expiration";  
                if (targetPolicy.Items[expirationPolicyFeatureId] == null)  
                    targetPolicy.Items.Add(expirationPolicyFeatureId, "");  
  
                PolicyItem expirationPolicyItem =
```

Continued

Listing 8-4: ExpirationPolicyFeature.cs (continued)

```
        targetPolicy.Items[expirationPolicyFeatureId];
        expirationPolicyItem.CustomData = GetCustomDataString();
        expirationPolicyItem.Update();

    }
    catch (Exception x)
    {
        Helpers.HandleException(this, x);
    }
}

/// <summary>
/// Builds the custom data string used to specify the expiration policy.
/// </summary>
private string GetCustomDataString()
{
    StringBuilder sb = new StringBuilder("<data>");
    sb.Append(this.Policy.ToString());
    sb.Append(this.Action.ToString());
    sb.Append("</data>");
    return sb.ToString();
}
}
```

This code creates the expiration policy by building a custom data string, which it attaches to a new policy object associated with the content type. The custom data string is constructed in two parts. The first part describes the formula, and the second part describes the action.

The format of the custom data string is determined by the implementer of each policy feature. In this case, the custom data is formatted so that it will be recognized and parsed by the Expiration Policy Feature.

Listings 8-5 and 8-6 show the overrides of the `ToString()` method for the generated `ExpirationPolicy` and `ExpirationAction` classes, respectively. These methods simply retrieve the properties from the data file and then generate the appropriate CAML that identifies the formula and the action to perform.

Listing 8-5: ExpirationPolicy.cs

```
using System.Text;

namespace ECM2007.RecordsManagement
{
    public partial class ExpirationPolicy
    {
        /// <summary>
        /// Calculates the custom data string needed to configure the
        /// expiration policy in the SharePoint environment.
        /// </summary>
        public override string ToString()
        {

```

Chapter 8: Information Policy and Record Retention

```
StringBuilder sb = new StringBuilder("<formula ");

switch (this.Type)
{
    case ExpirationPolicyType.TimeSpan:
    {
        ExpirationPeriod period = this.Item as ExpirationPeriod;
        sb.AppendFormat("id=\"{0}\">",
            "Microsoft.Office.RecordsManagement.PolicyFeatures.
            Expiration.Formula.Builtin");
        sb.AppendFormat("<number>{0}</number>", period.Text);
        sb.AppendFormat("<property>{0}</property>",
            GetPeriodProperty(period));
        sb.AppendFormat("<period>{0}</period>",
            period.Span.ToString().ToLower());
        break;
    }
    case ExpirationPolicyType.Programmatic:
    {
        ExpirationFunction function = this.Item as
            ExpirationFunction;
        sb.AppendFormat("id=\"{0}\">", function.Text);
        break;
    }
    case ExpirationPolicyType.Custom:
    {
        ExpirationFormula formula = this.Item as ExpirationFormula;
        sb.AppendFormat("id=\"{0}\">", formula.Text);
        break;
    }
}

sb.Append("</formula>");
return sb.ToString();
}

/// <summary>
/// Retrieves the name of the property to use when calculating the
/// start of the retention period.
/// </summary>
/// <returns></returns>
private string GetPeriodProperty(ExpirationPeriod period)
{
    switch (period.FromDate)
    {
        case ExpirationPeriodType.Created:
            return "Created";
        case ExpirationPeriodType.Modified:
            return "Modified";
        default:
            return "Expiration";
    }
}
}
}
```

Listing 8-6: ExpirationAction.cs

```
using System.Text;

namespace ECM2007.RecordsManagement
{
    public partial class ExpirationAction
    {
        /// <summary>
        /// Retrieves the custom data string needed to configure the
        /// expiration action in the SharePoint environment.
        /// </summary>
        public override string ToString()
        {
            StringBuilder sb = new StringBuilder("<action ");

            BuiltInExpirationAction builtinAction = this.Item as
                BuiltInExpirationAction;
            if (builtinAction != null)
            {
                sb.AppendFormat("type=\"action\" id=\"{0}\",",
                    GetActionId(builtinAction));
            }
            else
            {
                WorkflowExpirationAction workflowAction = this.Item as
                    WorkflowExpirationAction;
                if (workflowAction != null)
                {
                    // assumes that the workflow tag contains the workflow GUID
                    // this code could be extended to retrieve the workflow id
                    // from its name
                    sb.AppendFormat("type=\"workflow\" id=\"{0}\",",
                        workflowAction.Text);
                }
                else
                {
                    CustomExpirationAction customAction = this.Item as
                        CustomExpirationAction;
                    if (customAction != null)
                    {
                        // assumes that the custom action tag contains the name of
                        // the custom action class that implements the action
                        sb.AppendFormat("id=\"{0}\",", customAction.Text);
                    }
                }
            }

            sb.Append(">");
            sb.Append("</action>");
            return sb.ToString();
        }
    }
}
```

```
    }

    /// <summary>
    /// Retrieves the id of the builtin action.
    /// </summary>
    private string GetActionId(BuiltInExpirationAction action)
    {
        switch (action.Type)
        {
            case BuiltInExpirationActionType.Delete:
                return "Microsoft.Office.RecordsManagement.PolicyFeatures.
                    Expiration.Action.MoveToRecycleBin";
            default:
                return string.Empty;
        }
    }
}
}
```

Using these examples as a starting point, you can easily apply the same techniques to quickly build a set of custom components that can address unique document retention requirements in a variety of scenarios. We started by capturing the information and rules needed to determine the retention period for a given record type, and then we added more rules to specify the appropriate actions to take when the retention period ends. All of this information is captured in a single file plan schema and drives the implementation of an extensible set of components we can work with easily. Over time, as more and more use cases are analyzed, we can continually revisit the file plan specification and look for additional ways to add value to the component library.

Summary

The MOSS document retention strategy builds on the core Information Policy framework by providing additional extensibility points for implementing custom expiration formulas and actions. In this chapter, we explored the built-in Expiration Policy Feature to gain a deeper understanding of the development techniques used by the MOSS product team to implement the default policy features and policy resources that are supplied as part of the MOSS Records Management platform architecture. We can apply the same techniques when developing our own custom policy features and resources.

Leveraging the custom Information Policy components we have been developing throughout the book as part of our ECM2007 component library, we saw how to create and deploy custom expiration formulas and expiration actions so that they can be added to information policies by a content administrator and attached to documents and list items.

We also extended the file-planning worksheet so that it can be used to capture enough information for setting up expiration policies, and we extended the dynamic file plan schema and associated classes that we developed in Chapter 2 to process the additional information needed to create expiration policies automatically whenever the file plan is executed.

9

Information Policy and Record Auditing

Another important policy feature that is included out-of-the-box with MOSS is the Auditing Policy feature, which we'll look at in this chapter. Before we can explore the Auditing Policy feature, we have to first take a look at the auditing framework that is provided by MOSS. Auditing is an important component of any content management system and is particularly important for records management, where any change to a document or list item that is being tracked as an official record is stored in the content database along with the content itself. So, we'll first look at the auditing architecture within MOSS and examine ways to create audit records, control how much information is kept in the content database, and also explore different ways to extract and view that information for a given record or set of records. Then we'll see how to associate auditing components with information policy through the default implementation of the Auditing Policy feature. Finally, we'll revisit the file planning worksheet and schema so that they include instructions for enabling auditing on a given record type.

Understanding Auditing in SharePoint

Content auditing is an essential part of records management, and there are some basic observations we can make about auditing in general. The fundamental requirement is to keep track of changes to content throughout its life cycle. Such changes may include modifications to the metadata associated with an item as well as to the item content itself. Depending on the type of business process involved, audit records themselves may be the subject of an auditing event. For example, if you are providing tools for an administrator to review and delete audit records, then additional audit events should be generated so that the act of deleting the audit records is also recorded. It is therefore part of our general auditing requirement that we track changes to the audit records as well as to any configuration settings that may influence how auditing events are generated.

One thing to keep in mind is that any auditing mechanism that tracks audit records would have to account for recursion and avoid modifying audit records during the auditing process.

Chapter 9: Information Policy and Record Auditing

SharePoint addresses these requirements by using a generalized auditing infrastructure whereby audit records are created in the content database whenever operations are performed on the content elements that are stored within it. This auditing mechanism is disabled by default and must be enabled through the user interface or programmatically either through utility code that runs on the server or by writing and activating a feature. This means that when we create a SharePoint site or site collection, auditing is turned off and no audit events are generated. In a Windows SharePoint Services environment, there is no user interface provided for viewing audit records or for enabling the audit events. The reason for this is obvious when you think about the many ways that direct access to the auditing layer could be abused. We don't want to allow any user to casually manipulate the auditing mechanism. On the other hand, we do need the ability to provide some level of configuration for administrative users. It should also be easy for those users to view the current set of audit events in the form of a report or list. MOSS gives us both capabilities out-of-the-box.

Auditing events can be generated for many kinds of content elements in a SharePoint environment, not only for documents and list items. The following objects can be the subject of an auditing event:

- Site collections
- Sites
- Lists
- Folders
- Document and list items

For each type of object, the auditing infrastructure defines a set of monitored operations, which are controlled using a bitmask. The `SPAuditMaskType` enumeration specifies the following monitorable events that can be enabled for all elements, such as the ones listed above, that expose an `SPAudit` object:

Audit Mask	Description
All	All event types and actions are monitored.
CheckIn	An entry is created when the object is checked in.
CheckOut	An entry is created when the object is checked out.
ChildDelete	An entry is created when one of the object's child objects is deleted (such as when deleting a file from a folder or a site from a site collection).
Copy	An entry is created when the object is copied.
Delete	An entry is created when the object is deleted.
Move	An entry is created when the object is moved.
None	No event types or actions are monitored.
ProfileChange	An entry is created when a profile associated with an object is changed.
SchemaChange	An entry is created when the object's schema is modified.

Audit Mask	Description
Search	An entry is created when the object is searched.
SecurityChange	An entry is created when a change is made to the object's security configuration.
Undelete	An entry is created when the object is restored from the recycle bin.
Update	An entry is created when the object's properties are updated.
View	An entry is created whenever the object is viewed by a user.
Workflow	An entry is created when the object is used in a workflow activity.

Once a mask has been associated with an *SPAudit* object, SharePoint begins monitoring what happens to those objects in the content database.

Enabling and Disabling Auditing

To enable auditing in a Windows SharePoint Services environment, we can create a simple feature that modifies the audit mask of the SharePoint site in which it is activated or deactivated. The following code shows what has to be done:

```
public class AuditingFeature : SPFeatureReceiver
{
    public override void FeatureActivated(SPFeatureReceiverProperties properties)
    {
        SPSite site = properties.Feature.Parent as SPSite;
        if (site != null)
        {
            // Store the current audit flags.
            site.RootWeb.Properties["AuditFlags"] =
                site.Audit.AuditFlags.ToString();
            site.RootWeb.Properties.Update();

            // Enable auditing for all items in the site collection.
            site.Audit.AuditFlags = SPAuditMaskType.All;
            site.Audit.Update();
        }
    }

    public override void FeatureDeactivating(SPFeatureReceiverProperties properties)
    {
        SPSite site = properties.Feature.Parent as SPSite;
        if (site != null)
        {
            // Restore the original audit flags.
            site.Audit.AuditFlags = (SPAuditMaskType)
                Enum.Parse(typeof(SPAuditMaskType),
                    site.RootWeb.Properties["AuditFlags"]);
        }
    }
}
```

```
        site.Audit.Update();
    }

    public override void FeatureInstalled(SPFeatureReceiverProperties properties) {}
    public override void FeatureUninstalling(SPFeatureReceiverProperties properties)
    {}
}
```

The feature activation code first retrieves the current audit mask for the site collection and then stores it in the property bag associated with the site so that the deactivation code can restore the site to its previous settings. In this example, we are turning on auditing for *all* event types. This means that SharePoint will start storing auditing events for nearly everything that happens within the site collection. This has obvious implications for performance and storage. With auditing enabled, there is a performance cost, but probably not a noticeable one since most interactions on a site are confined to a single page request. The real cost comes with the additional storage required to maintain the audit records. Thus, turning on all audit events can dramatically increase the size of the content database. One way to deal with this might be to periodically purge the audit entry collection after saving it to external storage.

Purging the audit history is not provided as an out-of-the-box feature of SharePoint and would have to be custom-coded.

Managing Audit Entries

Audit records are stored in the content database along with the content being audited. This has several advantages. It allows the audit history to be backed up along with the content, and it reduces the risk that the audit history and the referenced records might get out-of-sync or, worse, that the audit history will be lost. Since the audit configuration may change over time, the history of those changes and the resulting audit records can be more easily analyzed and correlated if everything is kept together in one place.

The built-in auditing support covers all content-related operations that can occur within the SharePoint environment, but there is also support for creating our own custom audit entries. This covers cases in which the out-of-the-box auditing events are inadequate, and it allows us to define high-level auditing events that may depend on external factors or that represent the aggregation of primitive events.

This ability to define custom audit records is an important part of the auditing framework. As an example, we might have auditing enabled for a site and want to monitor changes that are made to a particular list definition. In addition to tracking the modifications themselves, we might also want to keep a record of any changes that are made by a particular person or group — or perhaps an attempt to modify the schema by a person or group other than those to whom the list has been assigned. In cases such as this, we could build an event receiver that is called whenever the list schema changes and then writes a custom audit entry for the list item, but only if the current user is not a member of the list administrator's group. The generation of the custom audit record could be governed by a policy, allowing the text and payload of the custom audit entry to change, depending on how the policy is defined. The following code snippet shows how these kinds of custom audit entries might be written if we had, say, a custom `AuthorizedToUpdate` method that determined whether the user had the appropriate permissions:

```
public class ListItemMonitor : SPItemEventReceiver
{
    public override void ItemUpdated(SPItemEventProperties properties)
```

```
{
    base.ItemUpdated(properties);
    SPListItem item = properties.ListItem;

    if (!AuthorizedToUpdate(item, item.Web.CurrentUser))
    {
        StringBuilder sb = new StringBuilder("<PolicyEnforcement>");
        sb.AppendFormat("<Site>{0}</Site>", item.Web.Url);
        sb.AppendFormat("<Item>{0}</Item>", item.ID);
        sb.AppendFormat("<User>{0}</User>", item.Web.CurrentUser.LoginName);
        sb.Append("</PolicyEnforcement>");

        item.Audit.WriteAuditEvent(SPAuditEventType.Custom,
            "UnauthorizedUpdate", sb.ToString());
        item.Audit.Update();
    }
}
```

The interesting thing about custom audit entries is that they are not associated with a bitmask. Instead, they are associated with an arbitrary string that indicates the *source* of the event. Additionally, audit entries carry a payload in the *EventData* field of the *SPAuditEntry* object, so it is a relatively simple matter to read and interpret them in whatever way is necessary.

Reading audit entries requires elevation of privileges to prevent casual access to the audit tables. Writing audit entries does not require elevated privileges so that the current user is always recorded along with the event. This is important to remember when writing code like that shown above, because when the audit entry is written to the content database, it is associated with the user under whose account the code is executing.

In the code shown, the current user's login name is also being written explicitly into the payload of the audit event.

Audit Reporting

Office SharePoint Server provides built-in audit reporting. To view these built-in audit log reports, the Reporting feature must first be activated. To do this, follow these steps:

1. Navigate to the Site Collection Features page.
2. Scroll down until you see the Reporting feature.
3. Click on the Activate button and then return to the Site Settings page.
4. In the Site Collection Administration section, there is now a link for "Audit log reports." Click on this link to open the View Auditing Reports page shown in Figure 9-1.
5. Click on any of the links to generate and view a report in Microsoft Excel.

The default audit reports are created by an application page called *Reporting.aspx*, which writes ExcelML to a file and then returns the file to the user.

Chapter 9: Information Policy and Record Auditing

The term ExcelML refers to “Excel Markup Language,” which is an XML representation of a Microsoft Excel workbook that is based on the XML Spreadsheet Schema that was introduced by Microsoft in 2001. This should not be confused with OpenXML, which is a more formal open standard for representing any kind of document content using the XML language. Microsoft Excel 2002 and later versions can read raw ExcelML files, enabling the creation of spreadsheets on platforms such as web servers that do not have the Excel product installed.

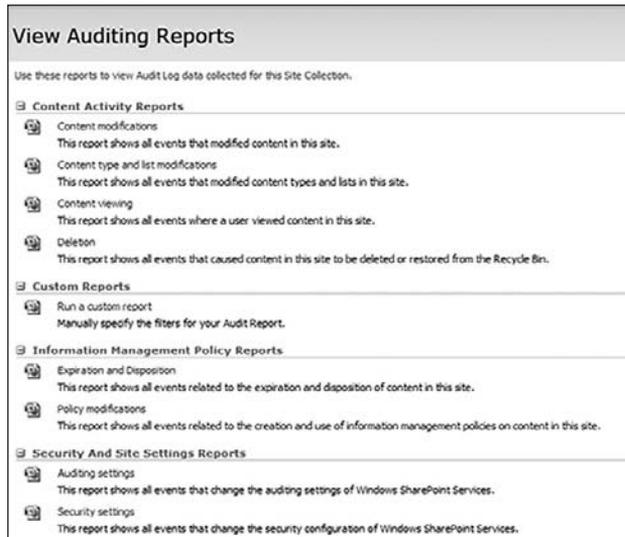


Figure 9-1: Default audit reports in MOSS.

Since they are based on ExcelML, the out-of-the-box reports can be somewhat difficult to read, especially if they contain many entries. One alternative would be to save the report to disk and then apply an XSLT stylesheet to transform the report into something more readable. Other alternatives include writing a custom application page for displaying audit records or building an audit reporting web part. We'll take the second approach later in the next section.

The following table lists the values of the `SPAuditEventType` enumeration, which are written into each audit record:

Audit Event Type	Description
<code>AuditMaskChange</code>	A change occurred in the types of events being audited for the object.
<code>CheckIn</code>	Check-in of the object
<code>CheckOut</code>	Check-out of the object
<code>ChildDelete</code>	A child object has been deleted from the object.

Audit Event Type	Description
ChildMove	A child of the object has been moved.
Copy	The object has been copied.
Custom	A custom event has occurred.
Delete	The object was deleted.
EventsDeleted	Some audited events connected with the object were deleted.
Move	The object was moved.
ProfileChange	A profile associated with the object was changed.
SchemaChange	The object schema was changed.
Search	A search query was performed on the object.
SecGroupCreate	A group was created for a site collection.
SecGroupDelete	A group was deleted from a site collection.
SecGroupMemberAdd	A new member was added to a group in a site collection.
SecGroupMemberDel	A member was removed from a group in a site collection.
SecRoleBindBreakInherit	Security permissions are no longer inherited from the object's parent.
SecRoleBindInherit	Security permissions are now inherited from the object's parent.
SecRoleBindUpdate	Security permissions for the object were changed.
SecRoleDefBreakInherit	A permission level (role) inheritance was removed from the object.
SecRoleDefCreate	A new permission level (role) was created for the object.
SecRoleDefDelete	A permission level (role) was removed from the object.
SecRoleDefModify	A permission level (role) associated with the object was modified.
Undelete	The object was restored from the recycle bin.
Update	A new object was created or an existing object's properties were modified.
View	The object was viewed by a user.
Workflow	The object was accessed by a workflow.

We can use these values along with the `SPAuditItemType` and `SPAuditLocationType` enumerations to generate custom reports or to build custom web parts that focus on a particular set of auditable events.

Using the Auditing Policy Feature

Now that we understand how auditing works in the MOSS environment, we can see how auditing instructions are attached to an information policy so they can be associated with a content type or document library. The Auditing Policy feature is implemented similarly to other policy features, but it is structured so that whenever an item is processed, the audit flags are set according to instructions that have been configured by the administrator of the policy. Those instructions are simply the list of audit flags that should be enabled for each item. Figure 9-2 shows the user interface that is provided for setting the audit flags.

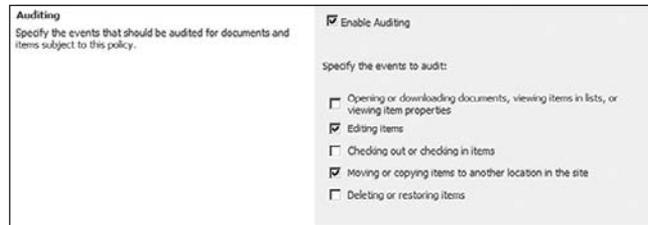


Figure 9-2: Auditing Policy user interface.

These are the same flags that appear on the Configure Audit Settings page, except for the Lists, Libraries, and Sites event masks, which do not apply to list items.

Auditing in a Records Center Site

Unlike a more traditional Records Repository that is used only for storing official records, a Records Center site in SharePoint supports collaboration on the records that it contains. This is one of the things that sets SharePoint apart from other systems. Since records managers routinely need to examine the *state* of each record in addition to its content and standard metadata, it is useful to provide them with tools that present the information in a way they can work with more easily. In this section, we'll create a simple web part for viewing audit records and also create a custom user interface for viewing the audit history of submitted records just to see what would be involved in creating similar tools for records managers.

Creating an Audit Viewer Web Part

In this section, we'll create a simple web part for viewing audit records in a site collection. This web part will enable *any* user to view the audit log, although in actual practice, we would most likely include security trimming code to prevent casual viewing of the audit log by unauthorized users.

For the purposes of this book, we will ignore security trimming and simply focus on the auditing functionality. However, if you want to extend the example to filter the displayed audit records based on the profile of the current user, then you have several options. You could determine the permissions for the current user as they relate to the individual objects being audited, or you could map the roles or group affiliations of the current user to a more general set of restrictions that apply only to audit records.

We'll start by creating a feature that we'll activate to install the web part, and we'll scope the feature to `Site` since it will install web parts into the web part gallery. Figure 9-3 shows the project structure.

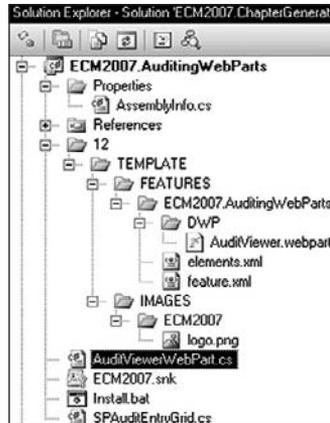


Figure 9-3: Auditing Web Parts feature project.

Next, we'll create the web part itself by deriving a new class from `System.Web.UI.WebControls.WebParts.WebPart`. Our web part will contain an `SPGridView` control to display the audit table entries. To make it easier to reuse the grid in other web parts, controls, and pages, we'll implement the audit entry management code inside the grid by deriving a new class from the `SPGridView` class and then hosting the specialized grid as a child control of the web part.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Drawing;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using Microsoft.SharePoint;
using Microsoft.SharePoint.WebControls;

namespace ECM2007.AuditingWebParts
{
    /// <summary>
    /// This web part displays a grid containing audit records
    /// associated with the current web, allowing the user to
    /// filter the grid based on a specified audit mask.
    /// </summary>
    public class AuditViewerWebPart : WebPart
    {
        public SPAuditEntryGrid Grid { get; private set; }

        /// <summary>
        /// Create the child controls that implement the
        /// web part functionality.
    }
}
```

Chapter 9: Information Policy and Record Auditing

```
    /// </summary>
    protected override void CreateChildControls()
    {
        this.Controls.Clear();
        this.Controls.Add(this.Grid =
            new SPAuditEntryGrid(SPCContext.Current.Site));
    }
}
```

We want to generalize the grid's behavior, so we'll simply pass the current `SPSite` object to its constructor. Later, we can extend the grid to accept different objects as determined by the contexts in which it is used.

Every grid consists of rows and columns, so before declaring the grid itself, let's create a class we can use to configure the columns. Our `ColumnInfo` class will hold the column header and sorting expression along with the type and format of the information the column will display.

```
/// <summary>
/// Represents a column in the grid.
/// </summary>
public class ColumnInfo
{
    public string header;
    public string sortExpression;
    public Unit width;
    public AuditEntryField field;
    public Type fieldType;

    public enum AuditEntryField {
        User, DocumentUrl, ItemType, ItemId, EventType, EventTime
    };

    public void Update(SPSite site, DataRow row, SPAuditEntry entry)
    {
        switch (field)
        {
            case AuditEntryField.User:
                {
                    string userName = string.Empty;
                    try
                    {
                        userName = site.RootWeb.SiteUsers.GetByID(
                            entry.UserId).Name;
                    }
                    catch
                    {
                        userName = entry.UserId.ToString();
                    }
                    row[field.ToString()] = userName;
                    break;
                }
            case AuditEntryField.DocumentUrl:
                row[field.ToString()] = entry.DocLocation;
                break;
        }
    }
}
```

```
        case AuditEntryField.ItemType:
            row[field.ToString()] = entry.ItemType.ToString();
            break;
        case AuditEntryField.ItemId:
            row[field.ToString()] = entry.ItemId.ToString();
            break;
        case AuditEntryField.EventType:
            row[field.ToString()] = entry.Event;
            break;
        case AuditEntryField.EventTime:
            row[field.ToString()] = entry.Occurred.ToLocalTime();
            break;
    }
}
}
```

Our `AuditEntryField` enumeration identifies the type of information that will be displayed in the column. The `fieldType` member specifies the underlying .NET type of the data stored in each cell. We'll use this to configure the data table that will be bound to the grid. The `Update` method will be called for each column in a row to extract the relevant field values from the `SPAuditEntry` object and place it into the appropriate row cell.

Now we can declare the `SPAuditEntryGrid` class, which inherits from `SPGridView`. The constructor creates a new `DataTable` and an `SPAuditQuery` object that we will use to retrieve the audit records. We'll also override the `OnLoad` method to refresh the grid using a helper method we will implement along with a couple of diagnostic logging and exception handling routines. To set up the grid, we'll need a collection of `ColumnInfo` objects, which we can create easily using a static collection initializer as follows:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using Microsoft.SharePoint;
using Microsoft.SharePoint.WebControls;

namespace ECM2007.AuditingWebParts
{
    /// <summary>
    /// This class extends the SPGridView control to display
    /// a collection of audit entries based on an audit query.
    /// </summary>
    public class SPAuditEntryGrid : SPGridView
    {
        public SPAuditQuery Query { get; private set; }
        public SPSite SiteCollection { get; private set; }
        public DataTable DataTable { get; private set; }
    }
}
```

Chapter 9: Information Policy and Record Auditing

```
/// <summary>
/// Static initializer for column configuration.
/// </summary>
protected List<ColumnInfo> m_columns = new List<ColumnInfo>() {

    new ColumnInfo(){ field=ColumnInfo.AuditEntryField.User,
        fieldType=typeof(string), header="User" },

    new ColumnInfo(){ field=ColumnInfo.AuditEntryField.DocumentUrl,
        fieldType=typeof(string), header="Url" },

    new ColumnInfo(){ field=ColumnInfo.AuditEntryField.ItemType,
        fieldType=typeof(string), header="Type" },

    new ColumnInfo(){ field=ColumnInfo.AuditEntryField.ItemId,
        fieldType=typeof(string), header="Id" },

    new ColumnInfo(){ field=ColumnInfo.AuditEntryField.EventType,
        fieldType=typeof(string), header="Event" },

    new ColumnInfo(){ field=ColumnInfo.AuditEntryField.EventTime,
        fieldType=typeof(DateTime), header="Time",
        width=new Unit(120) },

};

/// <summary>
/// Constructor.
/// </summary>
/// <param name="query"></param>
public SPAuditEntryGrid(SPSite siteCollection)
{
    Log("ctor");
    this.DataTable = new DataTable();
    this.Query = new SPAuditQuery(
        this.SiteCollection = siteCollection);
}

/// <summary>
/// Override to refresh the grid.
/// </summary>
/// <param name="args"></param>
protected override void OnLoad(EventArgs args)
{
    Log("OnLoad");
    Refresh();
}

/// <summary>
/// Diagnostic output routine.
/// </summary>
/// <param name="format"></param>
/// <param name="args"></param>
```

```
protected void Log(string format, params object[] args)
{
    string category = GetType().Name;
    Trace.WriteLine(string.Format(format, args)+"...", category);
}

/// <summary>
/// Generic exception handler.
/// </summary>
/// <param name="x"></param>
protected void HandleException(Exception x)
{
    Log("Exception occurred: {0}", x.ToString());
}
}
```

The overridden `OnInit` routine loops through the `ColumnInfo` table adding columns to both the `DataTable` and the grid. Using the `ColumnInfo` table avoids having duplicate column names and will make it easier to extend later. While we're at it, we'll also enable sorting, filtering, and paging.

The line "this.PagerTemplate = null" in the `OnInit` code is required because of a bug in the `SPGridView` control that fails to display the paging tabs. To work around it, you have to set the `PagerTemplate` property to null after the grid has been added to the controls collection but before the `BindData` method is called. For more information, see Paul Robinson's blog post entitled, "SPGridView: Adding Paging to SharePoint When Using Custom Data Sources," at <http://blogs.msdn.com/powlo/archive/2007/03/23/Adding-paging-to-SPGridView-when-using-custom-data-sources.aspx>.

```
/// <summary>
/// Called when the control is to be initialized to setup
/// the grid columns.
/// </summary>
/// <param name="args"></param>
protected override void OnInit(EventArgs args)
{
    try
    {
        // setup the columns
        Log("Initializing columns");
        foreach (ColumnInfo column in m_columns)
        {
            this.DataTable.Columns.Add(
                column.field.ToString(), column.fieldType);

            SPBoundField field = new SPBoundField();
            field.HeaderText = column.header;
            field.DataField = column.field.ToString();
            if (column.width != null)
                field.ControlStyle.Width = column.width;
            if (!string.IsNullOrEmpty(column.sortExpression))
                field.SortExpression = column.sortExpression;
            else

```

Chapter 9: Information Policy and Record Auditing

```
        field.SortExpression = column.field.ToString();
        this.Columns.Add(field);
    }

    // configure the grid
    Log("Setting grid properties");
    this.AutoGenerateColumns = false;
    this.AllowFiltering = true;
    this.HeaderStyle.Font.Bold = true;
    this.AlternatingRowStyle.CssClass = "ms-alternating";

    this.PageSize = 12;
    this.AllowPaging = true;
    this.PageIndexChanging += new GridViewPageEventHandler(
        SPAuditEntryGrid_PageIndexChanging);
    this.PagerTemplate = null;

    this.AllowSorting = true;
    this.Sorting += new GridViewSortEventHandler(
        SPAuditEntryGrid_Sorting);
}
catch (Exception x)
{
    HandleException(x);
}
}
```

Refreshing the grid involves clearing the table and then *executing* the audit query against the site collection. Our design makes it possible to adjust the query before loading the grid. As an example, we could restrict the query so that it operates only on particular event types, users, or lists. We could also restrict it so that it only returns events that occur within a particular range of dates.

Once we have the collection of audit entries, we add them to new table rows by calling the `ColumnInfo Update` method, passing the site collection, row, and audit entry objects.

```
/// <summary>
/// Helper method to refresh the grid based on the current
/// query definition.
/// </summary>
private void Refresh()
{
    try
    {
        this.DataTable.Clear();

        // Execute the query to get the audit entry collection.
        // This requires elevation of privileges.
        Log("Executing audit query");
        SPSecurity.RunWithElevatedPrivileges(delegate()
        {
            SPAuditEntryCollection entries =
                this.SiteCollection.Audit.GetEntries(this.Query);

            // Fill the data table with the entry data.
```

```
Log("Initializing data table with {0} entries", entries.Count);
foreach (SPAuditEntry entry in entries)
{
    DataRow row = this.DataTable.Rows.Add();
    foreach (ColumnInfo col in m_columns)
        col.Update(this.SiteCollection, row, entry);
}
});

// Bind the data table to the grid.
Log("Binding the data table to the grid");
// Recreate sort if needed.
if (ViewState["SortDirection"] != null &&
    ViewState["SortExpression"] != null)
{
    this.DataTable.DefaultView.Sort =
        ViewState["SortExpression"].ToString()
        + " " + ViewState["SortDirection"].ToString();
}
this.DataSource = this.DataTable.DefaultView;
this.DataBind();
}
catch (Exception x)
{
    HandleException(x);
}
}
```

Note that we are running this code within a delegate passed to the `SPSecurity.RunWithElevatedPrivileges` method. This ensures that our web part will work in any user context because SharePoint requires that code that accesses audit records must run as a site administrator. However, this may limit the usefulness of the web part for use in production. We could address this by either checking whether the current user is an administrator or performing some other test before instantiating the grid. Of course, we could also handle the exception and simply inform the user of his or her woefully inadequate permissions.

Sorting in an `SPGridView` requires a little finesse in the callback to manage the `ViewState` so that the sort expression and direction are coordinated properly between the grid and the underlying data table view. We do this by first checking the `ViewState` for any previously defined sort expression or direction so we can use them if they are available. Otherwise, we use the default values.

Thanks to Paul Robinson (blogs.msdn.com/powlo) for explaining the `ViewState` trick.

```
void SPAuditEntryGrid_Sorting(object sender, GridViewSortEventArgs e)
{
    string lastExpression = "";
    if (ViewState["SortExpression"] != null)
        lastExpression = ViewState["SortExpression"].ToString();

    string lastDirection = "asc";
    if (ViewState["SortDirection"] != null)
        lastDirection = ViewState["SortDirection"].ToString();

    string newDirection = "asc";
```

Chapter 9: Information Policy and Record Auditing

```
if (e.SortExpression == lastExpression)
    newDirection = (lastDirection == "asc") ? "desc" : "asc";

ViewState["SortExpression"] = e.SortExpression;
ViewState["SortDirection"] = newDirection;

this.DataTable.DefaultView.Sort = e.SortExpression + " " + newDirection;
}
```

Since we have enabled paging, we also need to handle the `PageIndexChanging` event as follows:

```
void SPAuditEntryGrid_PageIndexChanging(
    object sender, GridViewPageEventArgs e)
{
    this.PageIndex = e.NewPageIndex;
    this.DataBind();
}
```

Now we just need to package it up and send it to SharePoint. For that, we'll need a `.webpart` file to configure the web part in the gallery:

```
<webParts>
  <webPart xmlns="http://schemas.microsoft.com/WebPart/v3">
    <metaData>
      <type name="ECM2007.AuditingWebParts.AuditViewerWebPart, ECM2007.
        AuditingWebParts, Version=1.0.0.0, Culture=neutral,
        PublicKeyToken=eb8a6a1622425a15" />
      <importErrorMessage>Cannot import this Web Part.</importErrorMessage>
    </metaData>
    <data>
      <properties>
        <!-- Standard Web Part properties -->
        <property name="ChromeType" type="chrometype">Default</property>
        <property name="Title" type="string">Audit Viewer Web Part</property>
        <property name="Description" type="string">Displays audit records for
        items in the current site.</property>
        <property name="CatalogIconImageUrl" type="string">/_layouts/images/
        ecm2007/logo.png</property>
        <property name="TitleIconImageUrl" type="string">/_layouts/images/
        ecm2007/logo.png</property>
        <property name="ExportMode" type="exportmode">All</property>
      </properties>
    </data>
  </webPart>
</webParts>
```

and we'll need a `SafeControl` entry in the `web.config` file:

```
<SafeControl Assembly="ECM2007.AuditingWebParts, Version=1.0.0.0, Culture=neutral,
  PublicKeyToken=eb8a6a1622425a15" Namespace="ECM2007.AuditingWebParts"
  TypeName="*" Safe="True" />
```

Figure 9-4 shows the web part displayed on a page. Clicking on the column headers sorts and filters the rows accordingly.

User	URI	Type	Id	Event	Time
System Account	Legal Documents/SharePoint Class Template.docx	Document	c5aa3ac3-814a-4927-87b7-6f03173de25	Custom	9/18/2008 7:43:47 AM
System Account	Legal Documents/FR1.bit	Document	c326d8f8-4835-4c47-a214-765a42c9eb12	Custom	9/18/2008 6:56:01 AM
System Account	Legal Documents/ECM401 Cover.docx	Document	f7432187-d121-4189-a8b7-7feb2bc05cd	Custom	9/18/2008 7:28:02 AM
System Account	Shared Documents/Sample Document.docx	Document	450d8984-d194-4703-a2c2-815b56689721	Custom	11/30/2008 8:41:32 PM
System Account	Legal Documents/SharePoint ECM Analysis.xlsx	Document	b6cb852b-0e12-4c0f-9122-98a1b62c8ba3	Custom	9/18/2008 7:47:46 AM
System Account	Legal Documents/ECM401 Cover.docx	Document	6a3a8d20-6854-4b9a-b6c1-a6ddc1146359	Custom	9/17/2008 11:46:49 PM
System Account	Shared Documents/ExpenseReport1.xlsx	Document	f94b447b-b70b-47ed-8c7a-d306c32ed149	AudMaskChange	10/21/2008 10:29:51 PM
System Account	Shared Documents/ExpenseReport1.xlsx	Document	f94b447b-b70b-47ed-8c7a-d306c32ed149	Delete	11/30/2008 7:09:30 AM
System Account	Shared Documents/Sample Document.docx	Document	b7a8f8e4c1-450d-4703-a2c2-815b56689721	Custom	11/30/2008 8:10:51 PM

Figure 9-4: Audit Viewer Web Part.

The Audit Viewer Web Part we've created retrieves audit records from the content database and displays them in a grid for the user to review. It demonstrates how to obtain the audit records based on an `SPAuditQuery` and how to decode the audit entry fields for display. It does not support filtering or security trimming, which are features you will most likely want to add before putting the web part into production. In the next section, we will deal with a different kind of audit review mechanism that is unique to Records Center sites.

Viewing Historical Audit Records

In the context of a Records Center, where each incoming record may be submitted with its own audit history (from interactions with the document prior to submission), you may wish to provide an additional layer of support for those users who are tasked with examining or otherwise working directly with that audit history. Although the recorded information is similar to that stored within the content database for *active* documents and list items, the audit history records are static, and they are stored in a separate location. In this section, we will create a custom application page and an associated Edit Control Block (ECB) command that will allow a records manager to view the audit history (as opposed to the active audit entries) for a given record in a more user-friendly display than that provided by simply viewing the audit history file.

Figure 9-5 shows how audit history files are stored for an incoming record. A special folder is created, and within this folder, the individual XML data files are placed that contain the actual audit entries.



Type	Name	Modified
Folder	Audit History	3/21/2009 11:54 AM
Folder	Properties	3/21/2009 11:39 AM
Document	Lorem Ipsum New_KQWEP9	3/21/2009 11:54 AM
Document	Lorem Ipsum New_OASW7G	4/17/2009 12:21 PM
Document	Lorem Ipsum_MJ50CV	3/21/2009 11:39 AM

Figure 9-5: Audit history for a submitted record.

The file that the Records Center generates is an XML file that contains an `AuditEntry` element for each audit record, along with additional information about the source web site, the item and its type, the user ID, the document location, and the type of event that occurred. The audit history is stored in a file having the same name as the record (including the date stamp), but with an XML extension inside the Audit History folder. This file only exists if audit records were submitted with the file. Listing 9-1 shows an example of the information that is contained within this file.

Listing 9-1: Sample audit history file

```
<AuditData>
  <AuditEntry>
    <SiteId>4cfb9fd1-c3a8-4215-bea4-a7d58e835d2c</SiteId>
    <ItemId>b9a0e871-37e7-4cc1-ba10-1cde82ff1f53</ItemId>
    <ItemType>Document</ItemType>
    <UserId>1073741823</UserId>
    <DocLocation>Legal Documents/Lorem Ipsum.doc</DocLocation>
    <LocationType>Url</LocationType>
    <Occurred>3/21/2009 3:53:42 PM</Occurred>
    <Event>Update</Event>
    <EventSource>SharePoint</EventSource>
    <EventData>
      <Version>
        <Major>1</Major>
        <Minor>0</Minor>
      </Version>
    </EventData>
  </AuditEntry>
  <AuditEntry>
    <SiteId>4cfb9fd1-c3a8-4215-bea4-a7d58e835d2c</SiteId>
    <ItemId>b9a0e871-37e7-4cc1-ba10-1cde82ff1f53</ItemId>
    <ItemType>Document</ItemType>
    <UserId>1073741823</UserId>
    <DocLocation>Legal Documents/Lorem Ipsum New.doc</DocLocation>
    <LocationType>Url</LocationType>
    <Occurred>3/21/2009 3:54:24 PM</Occurred>
    <Event>View</Event>
    <EventSource>SharePoint</EventSource>
    <EventData>
      <Version>
        <Major>0</Major>
        <Minor>6</Minor>
```

```
</Version>
</EventData>
</AuditEntry>
</AuditData>
```

Figure 9-6 shows the structure of the ECM2007.AuditHistoryViewer feature. The main elements are the elements.xml file shown in Listing 9-2, which contains the custom ECB command, and the AuditHistory.aspx page shown in Listing 9-3, which contains a placeholder for a customized SPGridView that we will use to display the audit records.

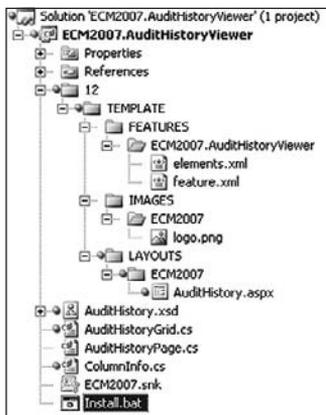


Figure 9-6: ECM2007 .AuditHistoryViewer feature.

Listing 9-2: elements.xml

```
<Elements xmlns="http://schemas.microsoft.com/sharepoint/">
  <CustomAction
    Id="ECM2007.AuditHistoryViewer"
    Location="EditControlBlock"
    Title="View Audit History"
    RegistrationType="List"
    RegistrationId="101">

    <UrlAction Url="~site/_layouts/ECM2007/AuditHistory.aspx?
      List={ListId}&amp;ID={ItemId}" />
  </CustomAction>
</Elements>
```

Listing 9-3: AuditHistory.aspx

```
<%@ Assembly Name="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
  PublicKeyToken=71e9bce111e9429c" %>
<%@ Assembly Name="ECM2007.AuditHistoryViewer, Version=1.0.0.0, Culture=neutral,
```

Continued

Listing 9-3: AuditHistory.aspx (continued)

```
    PublicKeyToken=eb8a6a1622425a15" %>

<%@ Page Language="C#" MasterPageFile="~/_layouts/application.master"
    Inherits="ECM2007.AuditHistoryViewer.AuditHistoryPage"
    EnableViewState="true" EnableViewStateMac="true" %>

<%@ Register TagPrefix="SharePoint"
    Namespace="Microsoft.SharePoint.WebControls"
    Assembly="Microsoft.SharePoint, Version=12.0.0.0, Culture=neutral,
    PublicKeyToken=71e9bce111e9429c" %>

<asp:Content ID="PageTitle" runat="server"
    ContentPlaceHolderID="PlaceHolderPageTitle">
    View Record Audit History
</asp:Content>

<asp:Content ID="PageTitleInTitleArea" runat="server"
    ContentPlaceHolderID="PlaceHolderPageTitleInTitleArea">
    Audit History For: '<asp:Label ID="AuditedRecordName" runat="server" />'
</asp:Content>

<asp:Content ID="Main" runat="server"
    ContentPlaceHolderID="PlaceHolderMain">
    <p>
        This page displays the audit records, if any, that were submitted to the
        records center along with the file. These audit entries do not reflect
        subsequent interactions with the record after it was submitted.
    </p>
    <!-- Placeholder for the Audit Entry Grid -->
    <asp:Placeholder ID="phGrid" runat="server" />
</asp:Content>
```

To simplify the binding of audit records to the grid, we first convert the raw XML data into a `DataTable` and then bind the table to the grid. To further simplify the parsing of the audit history file, we can use a schema and then generate wrapper classes. The easiest way to generate a schema from an existing XML data file is to load the file into Visual Studio and then select the “Create Schema” command from the XML menu item. Once the schema is created, we can simply generate the deserialization classes we need using the XSD.EXE command line tool. Using the resulting wrapper classes, we can then write the code we need to add the audit entries to the `DataTable` for binding to the grid. Listing 9-4 shows the `Refresh` method for an `AuditHistoryGrid` component that follows the same structure as the `SPAuditEntryGrid` we created for the `AuditViewerWebPart`.

Listing 9-4: AuditHistoryGrid.Refresh

```
/// <summary>
/// Helper method to refresh the grid from historical audit entries.
/// </summary>
private void Refresh()
{
    try
```

```
{
    this.DataTable.Clear();

    // Retrieve the collection of audit entries from
    // the attached list item. These are the static entries
    // that were submitted along with the record.

    Log("Loading audit records from the list item.");

    // Locate and open the containing folder.
    SPFolder thisFolder = FindContainingFolder(
        Item.ParentList.RootFolder.SubFolders, Item);
    if (thisFolder != null && thisFolder.Exists)
    {
        // Locate and open the "Audit History" folder.
        SPFolder auditHistoryFolder = thisFolder.SubFolders["Audit History"];
        if (auditHistoryFolder != null && auditHistoryFolder.Exists)
        {
            // Locate an XML file that matches the name of
            // the selected item.
            string auditFileName =
                Path.GetFileNameWithoutExtension(Item.Name) + ".xml";
            SPFile auditFile = auditHistoryFolder.Files[auditFileName];

            if (auditFile.Exists)
            {
                SPSite site = Item.Web.Site;

                // Deserialize the file into an "AuditData" component.
                XmlSerializer ser = new XmlSerializer(typeof(AuditData), "");
                using (Stream auditStream = auditFile.OpenBinaryStream())
                {
                    AuditData auditData =
                        (AuditData)ser.Deserialize(auditStream);
                    foreach (AuditDataAuditEntry entry in auditData.AuditEntry)
                    {
                        DataRow row = this.DataTable.Rows.Add();
                        foreach (ColumnInfo col in m_columns)
                            col.Update(site, row, entry);
                    }
                }
            }
        }
    }

    // Bind the data table to the grid.
    Log("Binding the data table to the grid");

    // Recreate sort if needed.
    if (ViewState["SortDirection"] != null &&
        ViewState["SortExpression"] != null)
    {
```

Continued

Listing 9-4: AuditHistoryGrid.Refresh (continued)

```
        this.DataTable.DefaultView.Sort =
            ViewState["SortExpression"].ToString()
            + " " + ViewState["SortDirection"].ToString();
    }
    this.DataSource = this.DataTable.DefaultView;
    this.DataBind();
}
catch (Exception x)
{
    HandleException(x);
}
}
```

When we're done, we can activate the feature and select the new "View Audit History" command from the ECB dropdown for any given record. Figure 9-7 shows the Audit History page for a typical record.

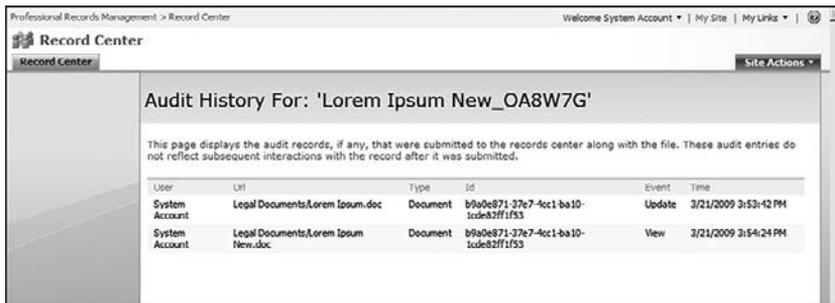


Figure 9-7: Audit History page.

Additional Considerations for Auditing

There are many ways to leverage the Auditing API for records management. The ability to write custom audit entries opens the door for all kinds of audit-related processing and post-processing. They essentially allow us to add layers of *meaning* on top of the predefined audit events without adding to the core set of event types. As an example, consider a `Delete` event for a financial document in a Records Center. Using an item event receiver, we could monitor the deletion of any such document on, say, a business holiday and then write a custom audit entry that basically says, "A questionable action was detected that appears to violate our policy XYZ." Furthermore, the conditions governing how the custom audit entries are created could be defined in an actual policy attached to the content type or to the list containing the item.

It's always a good idea to keep track of when the audit log is viewed and by whom, and as mentioned earlier, it's equally important to consider adding security-trimming logic for any controls and/or web parts that allow users to view audit records. It might also be advisable to provide a separate user interface that allows only administrative users to view and modify audit settings.

Extending the File Plan Schema to Support Auditing Policy

Extending the file plan schema to support auditing is a simple matter of adding the available audit masks to the schema and then writing some code to create the appropriate policy feature for the content type associated with the record type. To do this, we first copy the enumeration values of the audit mask and add them to the schema:

```
<xs:complexType name="AuditingPolicyFeature">
  <xs:annotation>
    <xs:documentation>
      Specifies how the auditing policy feature should be configured
      for a record type.
    </xs:documentation>
  </xs:annotation>
  <xs:sequence>
    <xs:element name="Audit" type="AuditMask" minOccurs="0"
      maxOccurs="unbounded" nillable="true"/>
  </xs:sequence>
</xs:complexType>

<xs:simpleType name="AuditMask">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Viewing"/>
    <xs:enumeration value="Editing"/>
    <xs:enumeration value="CheckingOut"/>
    <xs:enumeration value="Copying"/>
    <xs:enumeration value="Deleting"/>
  </xs:restriction>
</xs:simpleType>
```

Next, we regenerate the wrapper classes so that we can access the new data structure, and then we extend the `CreateContentTypes` method of the `RecordSpecification` class so that it adds the auditing policy to the content type when the file plan is executed:

```
// the auditing policy is specified as an array of audit masks - if any were
// specified, then enable the policy feature using a static method of the
// extended AuditingPolicyFeature class
if (this.Auditing.Length > 0)
{
    AuditingPolicyFeature.CreatePolicy(type, this.Auditing);
}
```

The additional code needed to create the new audit policy for a content type is shown in Listing 9-5.

Listing 9-5: Adding an auditing policy to a content type

```
public partial class AuditingPolicyFeature
{
    /// <summary>
```

Continued

Listing 9-5: Adding an auditing policy to a content type *(continued)*

```
/// Creates and attaches any auditing policy which may have been
/// specified along with the record.
/// </summary>
/// <param name="type">Identifies the content type to which the auditing
/// policy will be attached.</param>
/// <param name="auditFlags">specifies the audit flags that should be
/// set</param>
public static void CreateAuditingPolicy(SPContentType type, AuditMask[]
    auditFlags)
{
    try
    {
        // check if there is a policy attached to the content type
        // if not, then create one so we can configure it
        Microsoft.Office.RecordsManagement.InformationPolicy.Policy
            targetPolicy =
            Microsoft.Office.RecordsManagement.InformationPolicy.Policy.
                GetPolicy(type);
        if (targetPolicy == null)
        {
            Microsoft.Office.RecordsManagement.InformationPolicy.Policy.
                CreatePolicy(type, null);
            targetPolicy = Microsoft.Office.RecordsManagement.
                InformationPolicy.Policy.GetPolicy(type);
        }

        // add the auditing policy to the content type
        const string auditingPolicyFeatureId =
            "Microsoft.Office.RecordsManagement.PolicyFeatures.PolicyAudit";
        if (targetPolicy.Items[auditingPolicyFeatureId] == null)
            targetPolicy.Items.Add(auditingPolicyFeatureId, "");

        PolicyItem auditingPolicyItem =
            targetPolicy.Items[auditingPolicyFeatureId];
        auditingPolicyItem.CustomData = GetCustomDataString(auditFlags);
        auditingPolicyItem.Update();
    }
    catch (Exception x)
    {
        Helpers.HandleException(typeof(AuditingPolicyFeature), x);
    }
}

/// <summary>
/// Builds the custom data string used to specify the auditing policy.
/// </summary>
private static string GetCustomDataString(AuditMask[] auditFlags)
{
    StringBuilder sb = new StringBuilder("<Audit>");
    foreach (AuditMask mask in auditFlags)
    {
```

```
switch (mask)
{
    case AuditMask.CheckingOut:
        sb.Append("<CheckInOut/>");
        break;
    case AuditMask.Copying:
        sb.Append("<MoveCopy/>");
        break;
    case AuditMask.Deleting:
        sb.Append("<DeleteRestore/>");
        break;
    case AuditMask.Editing:
        sb.Append("<Update/>");
        break;
    case AuditMask.Viewing:
        sb.Append("<View/>");
        break;
}
sb.Append("</Audit>");
return sb.ToString();
}
```

Summary

In this chapter, we examined the auditing support that is built into the SharePoint platform to understand how to control the information that is added to the content database for tracking what happens to documents and list items during the content life cycle. We then explored ways to enhance the user experience related to auditing by creating a web part that displays audit records for any active document or list item in the current site.

Since records managers may need to review historical audit entries that are submitted along with incoming records in addition to active audit entries that are written by SharePoint to the content database, we created a SharePoint feature that extended the Edit Control Block (ECB) with a custom command for retrieving and displaying historical audit records.

Finally, we examined how the built-in Audit Policy feature enables a records manager to attach audit masks to information policies. We then extended the dynamic file plan schema introduced in Chapter 5 to include auditing instructions and extended the dynamic file plan components to automatically generate auditing policies during content type creation when the file plan is executed.

10

Managing Physical Records

This chapter is about using the records management functionality provided by SharePoint to manage records that exist primarily on paper and must be stored physically. When you think about it, dealing with physical records is so fundamentally different from managing electronic documents that the challenges should be obvious. First, there is the problem of keeping the physical documents and any associated metadata synchronized with the corresponding database record or list item being used to manage them. Next, there are the problems of securing the physical repository and tracking what happens to the documents it contains. Then there are the purely logistic issues involved in finding and retrieving documents that have already been stored in the repository and ensuring that they find their way back to the right spot.

Clearly, dealing with physical records properly involves additional costs. There are the real costs associated with maintaining a secure storage facility and the paperwork and personnel costs required to manage both the transient and final disposition of documents. Then there are the soft costs that flow from the additional time needed to coordinate the flow of electronic and non-electronic data throughout the organization. These problems are, of course, not new, but they do present a unique set of challenges for any Rights Management Service (RMS), and especially for one designed to support collaboration and workflow.

Physical Records and List Items

Office SharePoint Server 2007 is already designed to handle physical records. The features needed for tracking electronic records, such as content types, workflows, forms, and information policies, work equally well for lists and for document libraries. The difference is that the storage of physical records is handled outside of the Records Center. The Record Routing Table is purely a storage mechanism and is designed to work in conjunction with document libraries, which provides the storage medium. Therefore, the Record Routing mechanism is not used for managing physical records. Instead, you simply create a custom list to capture the metadata needed for tracking the physical record and then enhance the list with workflows, forms, and information policies to coordinate the management part that is handled by SharePoint with the storage part that is handled externally. Figure 10-1 depicts the overall processing sequence. The sections that follow describe the steps needed to manage physical records using list items.

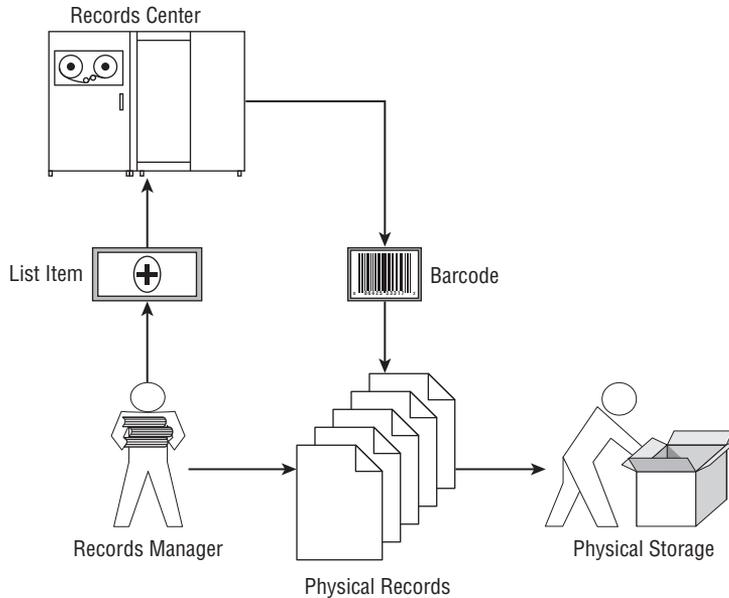


Figure 10-1: Physical records processing.

Create a Physical Record Content Type

The first step is to create a content type to describe each physical record type. Content types greatly simplify the management of physical records because they can capture metadata, workflow, and information policy requirements. For example, we can declare a generic content type called *Physical Record* to handle all physical records and to define the key elements needed to manage them. Later, we can extend the concept if necessary by inheriting from this content type.

The basic information needed in the Physical Record content type is not that different from the information we are already gathering in our file plan. The following table shows the minimal set of columns that might be used for tracking a typical physical record. The actual columns you choose will depend on your specific requirements.

Column	Type	Description
Title	Single line of text	Specifies the record type.
Description	Multiple lines of text	Describes the purpose of the record.
Category	Single line of text	Describes the record category.
Media	Single line of text	Specifies the media, such as “photograph” or “brochure.”

Column	Type	Description
Location	Single line of text	Specifies the location in which the record is stored.
Manager's Name	Single line of text	Identifies the person who is responsible for managing the record.

Create a Physical Records List

After creating a content type for each physical record, create a new generic list to represent their actual location. Give it a representative name such as *Physical Record* and enable management of content types from the Advanced Settings page. Add the new Physical Record content type to the list of content types on the Customize Physical Records page. Now as items are placed into the actual repository, you can create corresponding list items of type Physical Record in the list. The challenge, of course, will be keeping the list synchronized with what is actually happening in the real world. Later in the chapter, we'll look at ways to approach this using workflow. For now, there is one additional feature that can help to keep things in order, and that is information management policy.

Configure Information Policy Features

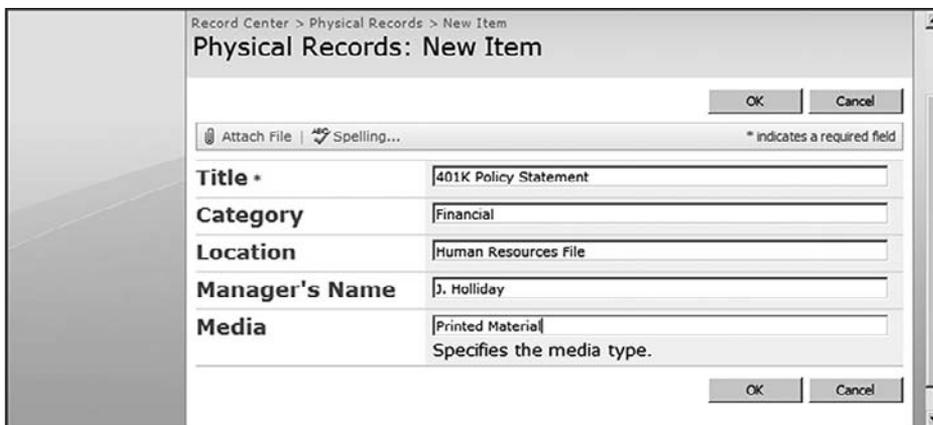
The Barcode information policy feature provides a convenient mechanism for identifying physical records because the barcode is generated from within the Records Center and is then affixed to the physical record. Later, when the record is checked out or moved from one physical location to another, the barcode can be scanned to quickly locate the corresponding list item so that it can be updated accordingly.

One advantage of enabling the Barcode policy feature at the content type level is that you don't have to repeat the configuration operation for every list you create to track physical records. The information policy follows the content type instead of the list. On the other hand, enabling the policy feature at the list level ensures that generic items also have a barcode assigned.

To enable the Barcode policy feature for the Physical Record content type, navigate to the Content Type Settings page, and choose the "Information management policy settings" link. From the Information Management Policy Settings: Physical Record page, click on the "Define a policy" radio button and press OK. Place a checkmark in the "Enable Barcodes" feature and press OK. Now whenever a new instance of Physical Record is created, a barcode will be generated automatically.

To prove it, navigate to the custom Physical Records list and choose New ⇨ Physical Record from the toolbar. Fill in the properties on the Physical Records: New Item page as shown in Figure 10-2, and press OK.

To see the generated barcode, you have to view the item properties by selecting the View Item command from the Edit Control Block of the new item. Figure 10-3 shows the item with the barcode as it will appear when affixed to the record.



Record Center > Physical Records > New Item

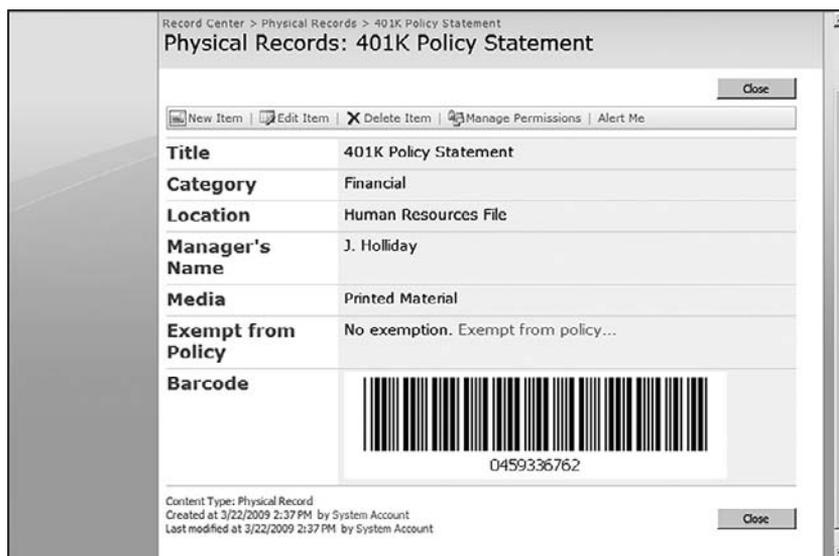
Physical Records: New Item

Attach File | Spelling... * indicates a required field

Title *	401K Policy Statement
Category	Financial
Location	Human Resources File
Manager's Name	J. Holliday
Media	Printed Material Specifies the media type.

OK Cancel

Figure 10-2: Physical Record Item properties.



Record Center > Physical Records > 401K Policy Statement

Physical Records: 401K Policy Statement

New Item | Edit Item | X Delete Item | Manage Permissions | Alert Me

Title	401K Policy Statement
Category	Financial
Location	Human Resources File
Manager's Name	J. Holliday
Media	Printed Material
Exempt from Policy	No exemption. Exempt from policy...
Barcode	 0459336762

Content Type: Physical Record
Created at 3/22/2009 2:37 PM by System Account
Last modified at 3/22/2009 2:37 PM by System Account

Close

Figure 10-3: Physical Record Item with barcode.

Physical Records and Folders

Another approach to dealing with physical records is to create folders within the Records Center that represent the physical locations in which the records are stored. This technique is convenient because it maps directly to what is happening in the real world. To facilitate this approach, you can create folder content types for each physical location being modeled.

Folder content types are special content types that apply only to folders, but they have the same characteristics as other content types for specifying columns, workflows, and information policies. To create a folder content type, navigate to the Site Content Type Gallery of the Records Center and select Create to open the New Site Content Type page shown in Figure 10-4. Enter a name for the folder, such as *DVD Rack*. Select “Folder Content Types” from the dropdown list under “Select parent content type from” and then select “Folder” as the parent content type and click OK to create the new type.

Figure 10-4: Creating a folder content type.

From the Site Content Type settings page, add some additional columns similar to the columns created for the Physical Record content type, such as *Category* and *Location*. Select the “Information management policy settings” link to enable the Barcode policy feature.

Next, create a new content type called *DVD* that inherits from Physical Record. Add additional columns as needed for the presentation, department, subject, serial number, and so on. You would use this kind of content type to represent individual DVDs in a given rack within the physical storage location.

Finally, enable all three content types (Physical Record, DVD, and DVD Rack) on the Physical Records list as shown in Figure 10-5. Now you can create a new folder that represents a physical DVD rack, and it will be assigned a barcode automatically, which can be printed and affixed to the physical rack. Through a periodic review of the DVDs contained in the physical rack, you can now easily add new items to the folder to match the DVDs contained in the rack.

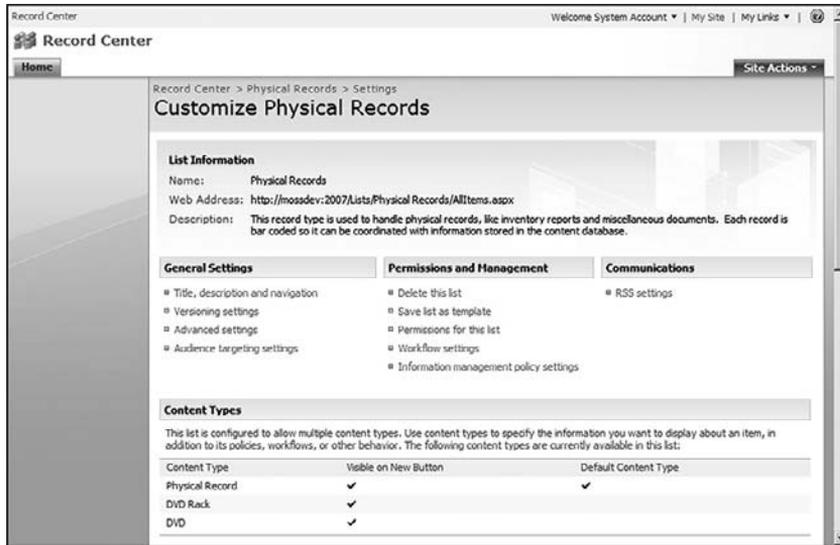


Figure 10-5: Physical records list settings.

Automating the Process

To simplify the steps required for setting up the Records Center to handle physical records, we'll create a SharePoint feature called `PhysicalRecordsFeature` to do the heavy lifting for us. We'll start by creating a new solution in Visual Studio that will hold the SharePoint feature project and adding a reference to the ECM2007 support library we've been using throughout the book. We'll use the Information Policy components in the support library to create the Content Type, List, and Barcode policies that will be installed by the feature. We'll also need to reference the `Microsoft.Office.Policy`, `Microsoft.SharePoint`, and `System.Configuration.Install` assemblies. The last one is needed because of code in the ECM2007 support library. Figure 10-6 shows the solution layout.

The project layout shown in this example includes components that are created automatically by the custom SharePoint feature project template we constructed in Chapter 2.

The feature.xml file specifies this as a Web scoped feature with a `ReceiverAssembly` and `ReceiverClass` that are called when the feature is activated.

```
<Feature xmlns="http://schemas.microsoft.com/sharepoint/"
  Id="1e1b1079-81a5-4da1-9f9c-9890b07f7821"
  Title="ECM2007 Physical Records"
  Description="Creates a custom list and associated content type for managing
  physical records. The content type is automatically bound to the barcode
  policy feature to facilitate tracking physical records linked to list items."
  ImageUrl="ECM2007\logo.png"
  Version="1.0.0.0"
  Scope="Web"
  Hidden="FALSE"
  ReceiverAssembly="ECM2007.PhysicalRecordsFeature, Version=1.0.0.0,
  Culture=neutral, PublicKeyToken=eb8a6a1622425a15"
```

```

ReceiverClass="ECM2007.PhysicalRecordsFeature.FeatureReceiver"
>
<ElementManifests>
  <ElementManifest Location="elements.xml" />
</ElementManifests>
</Feature>

```

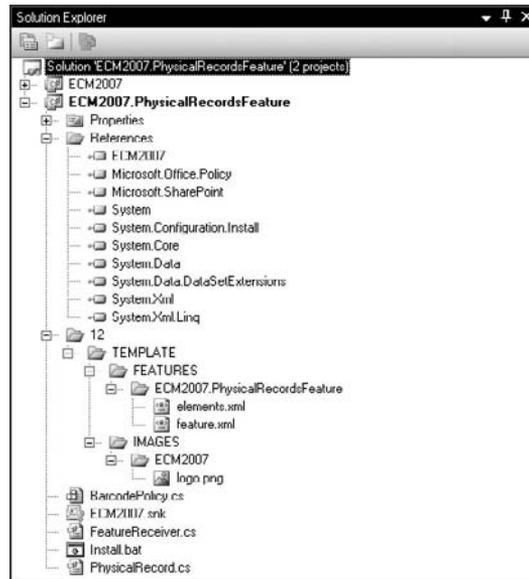


Figure 10-6: Physical Records feature project.

The `FeatureReceiver` class inherits from `SPFeatureReceiver` and implements the `FeatureActivated` method to set up the Records Center site. The code we will write requires the following namespaces:

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Diagnostics;
using Microsoft.SharePoint;
using ECM2007.ContentTypes;
using ECM2007.Utilities;
using ECM2007.InformationPolicy;
using Microsoft.Office.RecordsManagement.PolicyFeatures;
using Microsoft.Office.RecordsManagement.RecordsRepository;
using Microsoft.Office.RecordsManagement.InformationPolicy;

```

The first component we need is a Physical Record content type. For this, we will use the `SharePointContentType` attribute class from the ECM2007 support library to create a new content type based on attributes that we declare in a separate `PhysicalRecord` helper class as follows:

```

using System;
using System.Collections.Generic;

```

Chapter 10: Managing Physical Records

```
using System.Linq;
using System.Text;
using ECM2007.ContentTypes;

namespace ECM2007.PhysicalRecordsFeature
{
    [
        SharePointContentType(BaseType="Item",
            Description="This record type is used to handle physical records, like
            inventory reports and miscellaneous documents. Each record is bar
            coded so it can be coordinated with information stored in the content
            database.",
            Group="Professional Records Management",
            Name="Physical Record",
            Sealed=false)
    ]
    public class PhysicalRecord
    {
        [FieldRef("Description",
            Description="Describes the record",
            DisplayName="Description")]
        public string Description { get; set; }

        [FieldRef("Category",
            Description="The record category",
            DisplayName="Category")]
        public string Category { get; set; }

        [FieldRef("Location",
            Description="The physical location where the record is stored",
            DisplayName="Location")]
        public string Location { get; set; }

        [FieldRef("Media",
            Description="The media type of the record, such as 'Print' or 'Tape'",
            DisplayName="Media")]
        public string Media { get; set; }
    }
}
```

Next, we need to enable the Barcode policy feature for the content type so that whenever a physical record item is created, a barcode is automatically generated for it. To do this, we first create a `BarcodePolicy` object and then use it to enable the Barcode policy feature for the content type. Add a new class called `BarcodePolicy` to the project that inherits from the `SharePointPolicy` component defined in the `ECM2007.InformationPolicy` namespace of the ECM2007 support library. The `SharePointPolicy` component exposes a `Create` method that reads the required properties from a separate class, which declares the policy name and policy statement in the constructor as follows:

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

```
using ECM2007.InformationPolicy;

namespace ECM2007.PhysicalRecordsFeature
{
    [Guid("69A35EA8-1BCC-40bd-896C-D93CE617EBBF")]
    public class BarcodePolicy : SharePointPolicy
    {
        public BarcodePolicy()
            : base(
                "PhysicalRecordBarcodePolicy",
                "This policy enables the barcode feature for managing physical
                records."
            )
        {
        }
    }
}
```

Listing 10-1 shows the full implementation of the feature receiver class. The first thing we do is check to ensure that the feature is being activated on a Records Center web site. This will keep users from activating it on other kinds of sites where the functionality makes no sense. To do that, we compare the `WebTemplateId` of the web site to that of the Records Repository as published by the Records Management API.

We then proceed to create the Physical Records list using the `SharePointList` component from our custom ECM2007 support library. This is a simple generic list that we will continue to configure as we build up the other required components. The `SharePointPolicy.Create` method returns the new `Policy` object from the `Microsoft.Office.RecordsManagement.InformationPolicy` namespace and associates it with the content type. All that remains is to enable the Barcode policy feature, which is accomplished by adding the correct policy feature identifier to the internal list of policy items that SharePoint associates with the policy object. The Records Management API makes this easy by exposing the correct policy feature identifier as a public static string property of the Barcode object (`Microsoft.Office.RecordsManagement.PolicyFeatures.Barcode.PolicyId`).

Listing 10-1: PhysicalRecordsFeature feature receiver

```
namespace ECM2007.PhysicalRecordsFeature
{
    /// <summary>
    /// Handles events during feature installation and activation.
    /// </summary>
    public class FeatureReceiver : SPFeatureReceiver
    {
        public override void FeatureActivated(SPFeatureReceiverProperties properties)
        {
            SPWeb web = properties.Feature.Parent as SPWeb;
            if (web != null)
            {
                // Perform an initial check to ensure that the feature can only be
                // activated on a record center site.
                if (web.WebTemplateId != RecordsRepositoryCommon.
                    RecordsRepositoryWebTemplateID)
            }
        }
    }
}
```

Continued

Listing 10-1: PhysicalRecordsFeature feature receiver (continued)

```
{
    throw new SPEException("This feature must be activated in a Records
        Center site.");
}

// Create the "Physical Records" list.
SPList physicalRecords = SharePointList.Create(web,
    SPListTemplateType.GenericList, "Physical Records",
    "Use this list to manage physical records by creating list items that
        include the required metadata.");

if (physicalRecords == null)
{
    throw new SPEException("Failed to create the physical records list.");
}

// Create the PhysicalRecord content type.
SPContentType spPhysicalRecord = SharePointContentType.Create(
    web, typeof(PhysicalRecord));
if (spPhysicalRecord == null)
{
    throw new SPEException("Failed to create the physical record content
        type.");
}

// Turn on the barcode policy feature for the content type.
Policy barcodePolicy = SharePointPolicy.Create(
    spPhysicalRecord, typeof(BarcodePolicy));
if (barcodePolicy == null)
{
    Helpers.Log(this,
        "Failed to create barcode policy for physical record content
            type.");
}
else
{
    barcodePolicy.Items.Add(Barcode.PolicyId, string.Empty);
    barcodePolicy.Update();
}

// Enable content type management on the list
// and add the physical record content type.
try
{
    physicalRecords.ContentTypesEnabled = true;
    physicalRecords.ContentTypes.Add(spPhysicalRecord);
    physicalRecords.OnQuickLaunch = true;
    physicalRecords.Update();
}
catch (SPEException spex)
{
    // Get here if the content type is already associated with the list.
```

```

        Helpers.HandleException(this, spex);
    }
    else
    {
        Helpers.Log(this, "Invalid feature scope - expected SPWeb");
    }
}

public override void FeatureDeactivating(SPFeatureReceiverProperties properties)
{
}

public override void FeatureInstalled(SPFeatureReceiverProperties properties)
{
}

public override void FeatureUninstalling(SPFeatureReceiverProperties properties)
{
}
}
}

```

Finally, we enable content types on the list, add the Physical Record content type to the collection of content types that are attached to the list, add the list to the QuickLaunch bar, and update the list in the content database. When the feature is activated, the custom list is created with the new content type attached and the Barcode policy automatically enabled for physical records management. Figure 10-7 shows the resulting list in the SharePoint UI.

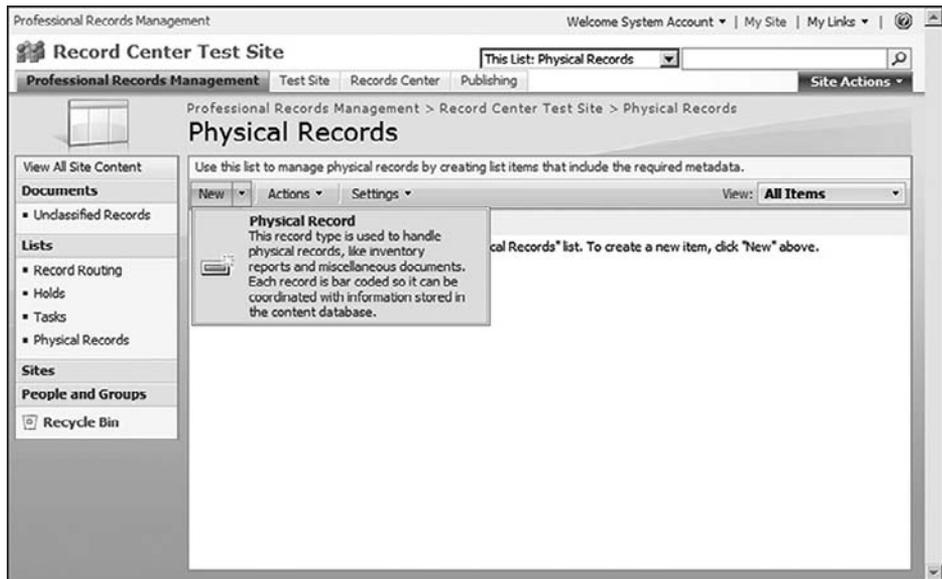


Figure 10-7: Physical Records feature activated.

Physical Records and Workflow

Although the preceding approach described using list items and barcodes works well to keep things synchronized, there is still a substantial gap between the electronic list item being managed by SharePoint and the physical document being stored separately. One problem is that there are a lot of “moving parts” to the overall scheme, which increases the complexity of the job facing the records manager using the system. As with anything else, increased complexity brings higher cost.

The complexity is increased because there is more for the records manager to do, and this procedure must be applied to every record being managed. Even the simple process of looking up the barcode in order to retrieve the right box containing the needed records can end up requiring a lot of extra time and effort that could be spent more productively elsewhere.

Although it is possible to set up business rules that control record routing and policies that specify required metadata, and so on, real-world scenarios typically involve *active* documents that are either moving into or out of a business process. Consequently, it is often insufficient to handle only the transfer of records from the collaborative environment into the repository. What is really required is some sort of workflow process to oversee the coordination of related tasks that generally need to occur at the same time. This is particularly true for highly regulated industries.

In early 2007, Microsoft published a case study entitled “Streamlining Records Management Using SharePoint Server 2007 Workflow,” in which they described an approach taken by the Microsoft Legal and Corporate Affairs (LCA) Records Management team. The power of this approach really shows how workflow can be used to fill in the gaps between electronic and physical information stores.

The example scenario presented by Microsoft involved a legal department and the problems they faced attempting to manage all of the different touch points that flowed out of their need to archive managed electronic documents into a physical inventory. We can extrapolate from that scenario to develop an overall strategy for dealing with physical records in general. The key elements are as follows:

1. A team is responsible through their collaborative environment to mark a document as a record.
2. The record is placed into a container, possibly along with other documents.
3. The container and the documents within it are labeled or barcoded to allow them to be uniquely identified.
4. The labels are bound to a list item in the Records Center.
5. The container is placed into inventory.

Over time, this day-to-day process will create a lot of containers, and at various times, personnel will have to retrieve a particular document from a particular container — either because of a routine audit or through the execution of some periodic review process. No matter how good the labeling system is, there will be some cost associated with going into the records management system, performing a search to locate the records of interest, and then figuring out which container the document is in. Additional costs will flow from physically locating the appropriate container, checking it out of storage, and retrieving the document, while making sure that all of the procedures required to keep that document associated with the corresponding list item are followed correctly. Once the document is out-of-the-box, it may also be necessary to lock the electronic record to ensure that no inconsistent metadata creeps into the system. The entire process may also be governed by continually evolving laws and regulations designed to ensure that the records management procedures are effective in achieving their goals.

Workflow is one way to tie all of this together so that the entire process can be easily understood by all of the stakeholders involved. It is not the only approach available. For example, it is certainly possible to create specialized SharePoint timer jobs that process requests for physical records and perform the necessary updates to the content database, and so on. However, the tight integration between SharePoint and Windows Workflow Foundation provides a convenient integration point for developing an effective overall strategy for coordinating activities related to the management of physical records.

Using workflow, there are programmatic elements (Workflow Activities) that produce tasks created within SharePoint to assist in the management process. The Microsoft solution involved allocating those tasks among the different roles involved in the business process they were already using to manage physical record inventories for their legal department. The final solution included a specially constructed workflow that was initiated by any request to retrieve a physical document. Figure 10-8 shows the overall workflow.

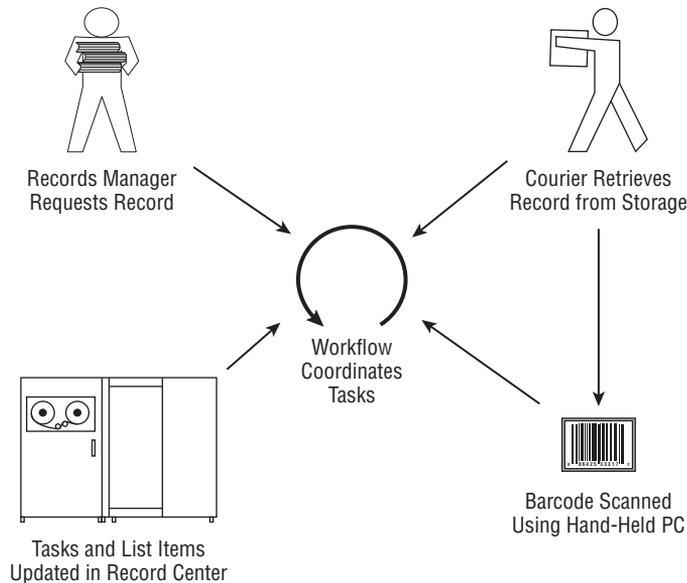


Figure 10-8: Managing physical records using workflow.

When the electronic record was created originally, a barcode was generated at the same time and was then placed physically on the document associated with the list item. The physical document was then moved into the offsite repository. Later, when a request was made to retrieve the document for review, the specially constructed workflow created a task that was assigned to a specific group of people responsible for retrieving documents from physical inventory. In this case, it was a team of couriers armed with hand-held barcode readers who understood the business processes involved. With the appropriate instructions in hand, these couriers would then fetch the documents and scan the barcode label. A separate Windows Mobile application would then inform the workflow to complete the retrieval task and create more tasks assigned to the team responsible for taking possession of the physical documents, reviewing them, and so on. At the same time, the workflow locked the electronic record to prevent further updates until the physical document was returned to storage.

Chapter 10: Managing Physical Records

The same strategy can be applied to any scenario involving external record storage. Whether or not you are using an actual courier, the real benefit comes from delegating the responsibility for task coordination to the workflow process and from maintaining the association between the Records Center list item and the physical document through a unique identifier. Figure 10-9 shows the generalized workflow.

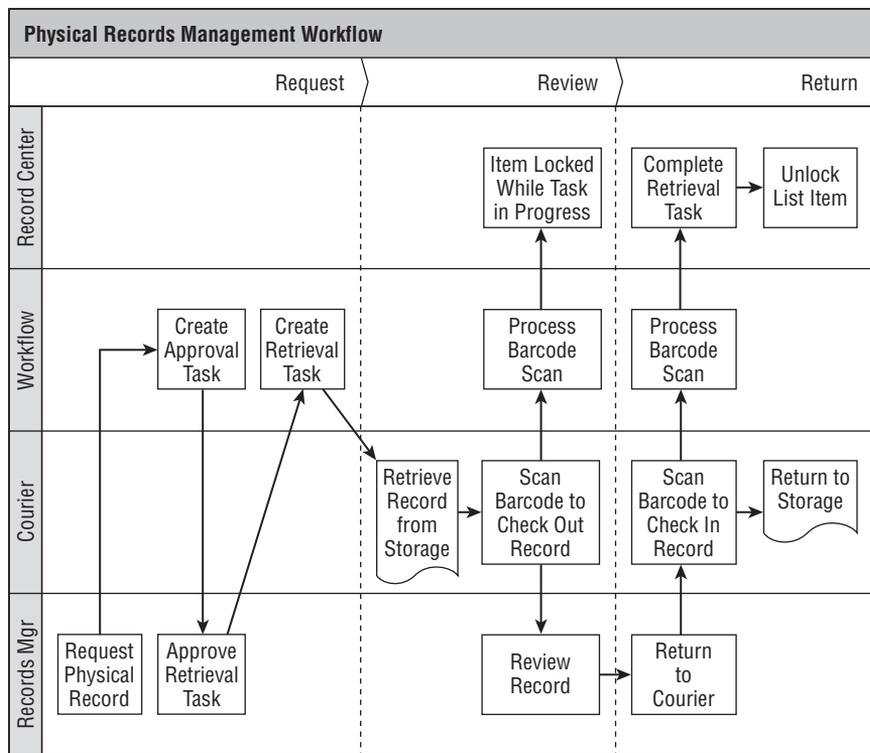


Figure 10-9: Physical records management workflow.

Summary

This chapter showed how Office SharePoint Server 2007 is set up to handle physical records by managing them as list items from within the Records Center site rather than as documents submitted to the Records Repository. Whereas electronic records require both *storage* and *management* support, physical records only need to be managed because they are stored outside of the Records Center.

The management support provided by SharePoint for handling physical records is greatly enhanced by the availability of information management policy. Specifically, the Barcode policy feature makes it possible to track physical records easily by automatically generating a barcode whenever a corresponding list item is created. Information policy used in the conjunction with content types provides a complete solution, but presents greater complexity to the records manager who is responsible for ensuring that all of the steps are performed correctly for each record. This increased complexity can be reduced through the use of custom workflows, which can help to coordinate the various activities involved.

11

Suspending Record Processing Using Holds

The Hold Feature is provided as part of the Records Management infrastructure to deal with situations in which the default processing for an individual record needs to be suspended for some reason. A typical example of this would be a litigation support scenario where a court order is issued to *freeze* the normal processing for a specific set of documents that are related to the case. The primary purpose of a hold is to prevent a group of records from being destroyed until after the associated event has been reconciled. Since holds are intended primarily to prevent the deletion of records, the Holds architecture is tied indirectly to the Expiration Policy Feature described in Chapter 8, but only if the designated expiration action is to delete the document. Whenever a hold is applied to an item, it is not possible to remove it from a list. Figure 11-1 shows the relationship between the items in a document library or list and the Holds list. Each item is linked to one or more hold items, which are stored as list items in the Holds list.

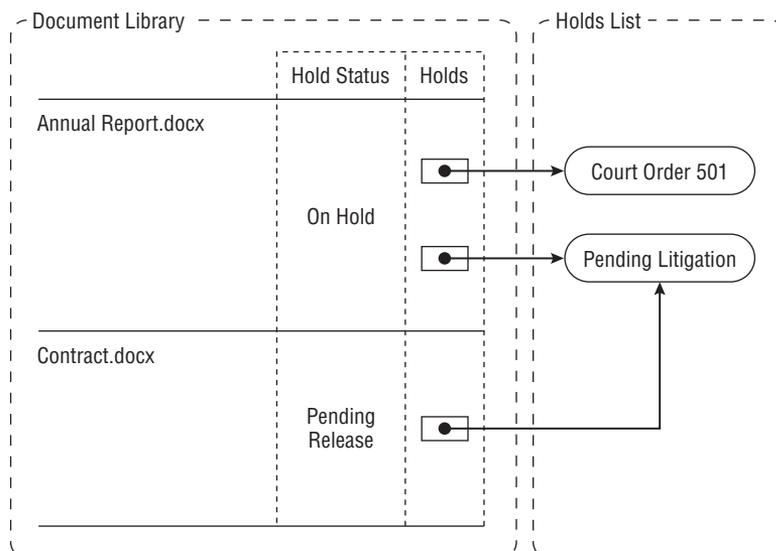


Figure 11-1: Holds and list items.

The Holds Architecture

When a Records Center web site is provisioned, it is automatically configured for holds processing. A Boolean property called `IsHoldEnabled` is added to the property collection associated with the web site to indicate that it has been set up to manage holds. At the same time, another property called `HasRecordCenter` is added to the root web of the site collection to indicate that the site collection is now associated with a Records Management site that is now active within the site collection.

Next, the built-in document parser is disabled for the entire web site. This turns off automatic property promotion and demotion for document libraries, which could inadvertently cause the properties of a document to change whenever document library column values are modified, thus voiding the purpose of having a Records Repository in the first place.

Because the document parser configuration setting applies to all document libraries in the web site, it carries implications for any document library that depends on property promotion and demotion. It essentially means that any other document libraries you create in the web site that are not involved in managing records will no longer promote or demote properties the way that standard document libraries do. This may require some additional training for records managers who may be accustomed to editing document properties through the standard document library user interface.

The master Holds list is then created, and the fields listed in the following table are added to it as well as to any content types associated with the list:

Field	Type	Description
<i>Description</i>	<i>Note</i>	Provides a brief description of the purpose of the hold.
<i>ManagedBy</i>	<i>User</i>	Identifies the user who is responsible for managing the hold.
<i>HoldStatus</i>	<i>HoldStatusField</i>	Contains the current hold status.
<i>HoldCount</i>	<i>Text</i>	The current number of items to which this hold applies.
<i>ReportDate</i>	<i>DateTime</i>	The most recent date on which a holds report was generated.

Minor versioning is also enabled for the list, and a default view is created that displays the title with a link to the item, the item description, and the *ManagedBy* and *HoldStatus* fields. Finally, a special event receiver is attached to the list for the `ItemDeleting`, `ItemAdding`, `ItemAdded`, and `ItemUpdating` events.

When a hold is created, SharePoint adds an additional column to the list to display the current status of the hold, as shown in Figure 11-2. This column is implemented using the internal `HoldsField` and `HoldsFieldControl` classes described below.

Because Edit Control Block (ECB) menu items are generated using JavaScript embedded in the web page, it is still possible for end-users to attempt to delete an item to which a hold has been applied, simply by entering the appropriate URL into the address bar of the web browser. However, any such attempt causes the Holds API to throw an exception, which appears to the end-user in a standard error page, as shown in Figure 11-3.

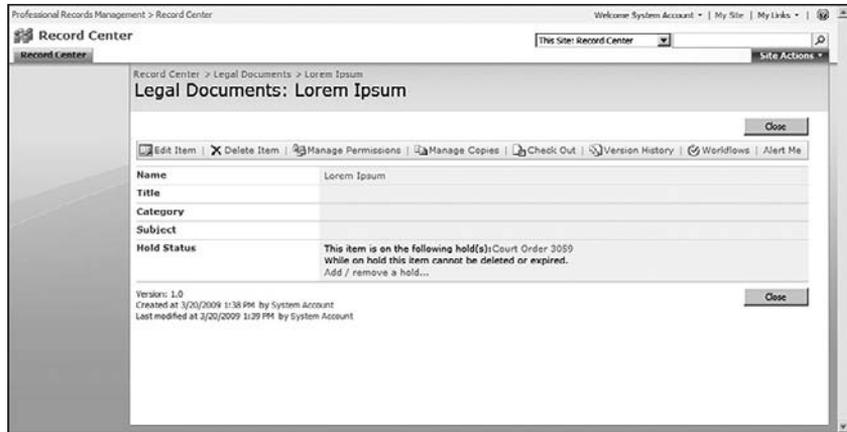


Figure 11-2: List item with hold applied.



Figure 11-3: Holds Error page.

Following is a list of the key components involved in holds processing. The Holds list and the Hold Reports list are specialized SharePoint lists. The other components are part of the object model and are declared within the `Microsoft.Office.RecordsManagement.Holds` namespace that is implemented in the `Microsoft.Office.Policy.dll`.

- ❑ **The Holds List** — The processing of holds requires a location in which to store the description of each hold that will be placed on list items. This location is a special list called a *Holds list* whose items describe each hold. Interestingly, unlike other specialized lists that are used throughout the Records Center, the Holds list is not associated with any content type. Instead, several columns are added directly to the list to keep track of the title and description of the hold. This information is then used programmatically to create `Hold` objects within the content database.

Chapter 11: Suspending Record Processing Using Holds

- ❑ **The Hold Class** — The `Hold` class implements the core functionality used to apply and remove holds from list items. It also exposes several static methods that support the Holds architecture as shown in Listing 11-1.
- ❑ **The HoldsField Class** — The `HoldsField` class is attached by the framework to a document library or list in the Records Center site to manage the collection of holds that have been applied to items in the list. The value that is stored in the field is managed by the `HoldsFieldControl` class, which is the designated base field control for the class. This class and all its members are reserved for internal use, and there are no public methods that can be called for this class.
- ❑ **The HoldsFieldControl Class** — The `HoldsFieldControl` displays the list of holds that are currently applied to the list item. It compiles the list in the `Render` method by decoding the list of hold identifiers stored in the `SPFieldMultiChoiceValue` object used to format the field value. Each item is displayed as a link to the list item describing the hold.
- ❑ **The HoldStatusField Class** — The `HoldStatusField` is attached to a document library or list to display the current hold processing status for the item. This class and all its members are reserved for internal use, and there are no public methods that can be called for this class.
- ❑ **The HoldStatusFieldControl Class** — The `HoldStatusFieldControl` works in conjunction with the `HoldStatusField` to display the current hold status.
- ❑ **The Hold Reports List** — A special timer job called the *Hold Processing and Reporting job* is scheduled automatically by the Records Management framework to run daily. This job is responsible for compiling a report of the holds that have been applied to various list items.

Listing 11-1: The Hold class

```
namespace Microsoft.Office.RecordsManagement.Holds
{
    /// <summary>
    /// Represents specific properties of a hold.
    /// Holds can be placed on documents to exempt them
    /// from the enforcement of the expiration policy
    /// that is applied to them.
    /// </summary>
    public class Hold
    {
        /// <summary>
        /// Constructs a new Hold object as a list item in
        /// the specified list in the Records Center.
        /// </summary>
        public Hold();

        /// <summary>
        /// Gets the most current holds report for the specified document.
        /// </summary>
        /// <returns>
        /// A string that represents the most current holds report for
        /// the specified document.
        /// </returns>
        /// <param name="hold">
        /// The document for which you want to get a holds report.
    }
}
```

Chapter 11: Suspending Record Processing Using Holds

```
/// </param>
public static string GetHoldReportName(SPListItem hold);

/// <summary>
/// Gets a list of items currently on hold for the specified
/// SharePoint site.
/// </summary>
/// <returns>
/// Returns an SPList object that represents a list of the
/// items on hold for the specified site.
/// </returns>
/// <param name="web">
/// The site for which you want the holds report.
/// </param>
public static SPList GetHoldReportsList(SPWeb web);

/// <summary>
/// Returns the list where the holds are stored for the
/// specified Records Center.
/// </summary>
/// <returns>
/// Returns an SPList object that represents the list where the
/// holds are stored for this Records Center.
/// </returns>
/// <param name="web">
/// The Records Center for which you want the list where the
/// holds are stored.
/// </param>
public static SPList GetHoldsList(SPWeb web);

/// <summary>
/// true if the specified document currently has a hold
/// applied to it; otherwise, false.
/// </summary>
/// <returns>
/// A Boolean value that specifies whether the document
/// currently has a hold applied to it.
/// </returns>
/// <param name="item">
/// The document for which you want to know if
/// a hold has been applied.
/// </param>
public static bool IsItemOnHold(SPListItem item);

/// <summary>
/// true if the specified document currently has the specified
/// hold applied to it; otherwise, false.
/// </summary>
/// <returns>
/// A Boolean value that specifies whether the document
/// currently has the specified hold applied to it.
/// </returns>
/// <param name="item">
/// The document for which you want to know if the specified
```

Continued

Listing 11-1: The Hold class (continued)

```
    /// hold has been applied.
    /// </param>
    /// <param name="holdID">
    /// The integer that identifies the list item that represents the hold.
    /// </param>
    public static bool IsItemOnHold(SPLListItem item, int holdID);

    /// <summary>
    /// Releases any documents on the specified SharePoint site
    /// from the specified hold.
    /// </summary>
    /// <param name="web">
    /// The site on which the documents you want released from
    /// this hold are located.
    /// </param>
    /// <param name="holdID">
    /// The integer that identifies the list item that
    /// represents the hold.
    /// </param>
    /// <param name="comments">
    /// Any comment you want to write to the audit log when
    /// the hold is released.
    /// </param>
    public static void ReleaseHold(int holdID, SPWeb web, string comments);

    /// <summary>
    /// Releases the specified document from the specified hold.
    /// </summary>
    /// <param name="item">
    /// The document for which you want the specified hold released.
    /// </param>
    /// <param name="holdID">
    /// The integer that identifies the list item that represents the hold.
    /// </param>
    /// <param name="comments">
    /// Any comment you want to write to the audit log
    /// when the hold is released for the specified document.
    /// </param>
    public static void RemoveHold(int holdID, SPLListItem item, string
        comments);

    /// <summary>
    /// Releases all the specified items from the specified hold.
    /// </summary>
    /// <returns>
    /// A Boolean value that represents whether the hold was
    /// successfully removed from the specified items.
    /// </returns>
    /// <param name="items">
    /// A collection of the list items for which you want
    /// to release the hold.
    /// </param>
    /// <param name="holdID">
```

Chapter 11: Suspending Record Processing Using Holds

```
    /// The integer that identifies the list item that
    /// represents the hold.
    /// </param>
    /// <param name="comments">
    /// Any comment you want to write to the audit log when
    /// the hold is released for the specified documents.
    /// </param>
    public static bool RemoveHold(int holdID, SPLListItemCollection items,
        string comments);

    /// <summary>
    /// Applies the specified hold to the specified document.
    /// </summary>
    /// <param name="item">
    /// The document to which you want the specified hold applied.
    /// </param>
    /// <param name="holdID">
    /// The integer that identifies the list item that represents the hold.
    /// </param>
    /// <param name="comments">
    /// Any comment you want to write to the audit log
    /// when the hold is applied to the specified document.
    /// </param>
    public static void SetHold(int holdID, SPLListItem item, string comments);

    /// <summary>
    /// Applies the specified hold to all the specified items.
    /// </summary>
    /// <returns>
    /// A Boolean value that represents whether the hold
    /// was successfully applied to the specified items.
    /// </returns>
    /// <param name="items">
    /// A collection of the list items to which you
    /// want the specified hold applied.
    /// </param>
    /// <param name="holdID">
    /// The integer that identifies the list item
    /// that represents the hold.
    /// </param>
    /// <param name="comments">
    /// Any comment you want to write to the audit log
    /// when the hold is applied to the specified documents.
    /// </param>
    public static bool SetHold(int holdID, SPLListItemCollection items, string
        comments);

    public static string HoldECBItemFeatureGuid { get; }
    public static string HoldsFieldInternalName { get; }
    public static string HoldsListColumn_Description { get; }
    public static string HoldsListColumn_HoldStatusInternal { get; }
    public static string HoldsListColumn_ManagedBy { get; }
    public static string HoldsListColumn_ReportDateInternal { get; }
}
}
```

Chapter 11: Suspending Record Processing Using Holds

Let's take a closer look at the Hold Reports list. The purpose of the Hold Processing and Reporting timer job is to generate an XML file that lists the holds that have been created in the Records Center site and the items to which they have been applied. The report is generated in a format that enables it to open automatically in Microsoft Excel. Whenever a report is generated, it is stored in the Hold Reports list, which resides in the Records Center web site, as shown in Figure 11-4.

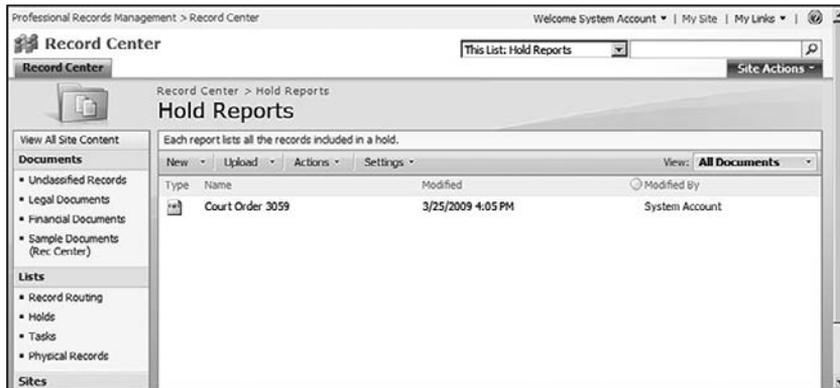


Figure 11-4: Hold Reports list.

The Hold Report template is copied from the Hold feature folder located on the server machine (in the 12 hive) to the Forms subfolder within the root folder of the Hold Reports list (in the content database) when the web site is first provisioned. This template is an XML file (ExcelML) that defines the layout and content of the holds report so that it can be opened directly in Microsoft Excel. When the timer job runs, a copy of the template is loaded into memory, and the data from each hold is injected to produce the report.

Although the code that generates the holds report is designed to be called only from the timer job, it is still possible to generate a hold report programmatically by calling the static overloaded `ProcessAndReport` method on the `Hold` class. Two versions of this method are available, one to generate a report for all holds in the Web, and a second to generate a report for an individual hold item. Listing 11-2 shows an example of the code needed to generate the report. A separate report file is created for each hold item. If the report already exists, then it is replaced with the latest version of the report.

Listing 11-2: Generating a holds report programmatically

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.SharePoint;
using Microsoft.Office.RecordsManagement.Holds;

namespace RunHoldsReport
{
    class Program
    {
```

```
static void Main(string[] args)
{
    const string url = "http://mossdev:9995/records";
    using (SPSite site = new SPSite(url))
    using (SPWeb web = site.OpenWeb())
    {
        Hold.ProcessAndReport(web);
    }
}
```

Creating and Removing Holds

To apply a hold to a list item, use the `Hold.SetHold` method. This method checks to ensure that the target item is not a folder or a workflow task. If it is, then an exception is thrown because holds are not supported for these types. Then a check is made to ensure that the current user has permission to edit list items.

If holds are not enabled for the parent list, then an attempt is made by the `SetHold` method to re-provision the list. Re-provisioning the list ensures that the correct event receivers are attached and that the list contains the required fields for managing the hold. Another check is made to ensure that the root folder is properly marked to indicate that the list has items on hold. This is a secondary consistency check to make sure that everything is properly configured. Finally, a new `SPFieldMultiChoiceValue` object is created for the `HoldsField` associated with the list. The hold identifier is appended to the field, and the `HoldCount` is incremented.

If the target file is currently checked-out, it is forcibly checked-in. An audit event record is then written to the content database that includes the hold identifier, the hold name, and any user comments supplied to the method.

Placing a Hold

To place a hold on a collection of list items, use code such as the following, which executes a query to obtain the `SPListItemCollection` object, retrieves the hold to be applied from the Holds list, and then calls the `Hold.SetHold` method to apply the hold:

```
SPList holdsList = Hold.GetHoldsList(web);
SPListItem holdItem = holdsList.Items[GetHoldItemIndex()];

SPQuery itemQuery = CreateItemQuery();
SPList legalDocuments = web.Lists["Legal Documents"];
SPListItemCollection itemsToPlaceOnHold = legalDocuments.GetItems(itemQuery);

Hold.SetHold(holdItem.ID, itemsToPlaceOnHold,
    "Processing has been suspended for legal documents.");
```

The `GetHoldItemIndex()` and `CreateItemQuery()` methods are placeholders for custom helper routines that are implemented in this example.

Chapter 11: Suspending Record Processing Using Holds

The overloaded `SetHold` method accepts an `SPListItemCollection` to make it easy to apply a hold to a collection of items. Later, we'll see how to take advantage of the Search & Process API to efficiently place holds on items that may reside in separate lists.

Removing a Hold

To remove a hold from a particular item, invoke the `Hold.RemoveHold` method. This method searches the `SPFieldMultiChoiceValue` field value from the `HoldsField` associated with the item. It then removes the specified value from the field and adjusts the `HoldCount` field. It also ensures that the file is checked-in and then writes an audit entry containing the hold ID, the hold name, and any user comments supplied to the method. The code to remove a hold looks like the following, which removes a specific hold from all items in a document library:

```
Hold.RemoveHold(holdItem.ID, legalDocuments.Items,
    "Hold removed from legal documents.");
```

You can also remove a hold from a collection of list items using the `Hold.RemoveHold(int holdId, SPListItemCollection items, string comments)` method. This method simply iterates over the collection calling the individual `RemoveHold` method.

To release *all* documents in the web site from a given hold, invoke the `Hold.ReleaseHold` method. When this method is called, the hold status column for the document is set to `HoldIsPendingRelease`, and an audit event is written that includes the hold identifier, the hold name, and any user comments supplied to the method.

The naming of these routines is a little odd. There are actually three overloads for the `Hold.RemoveHold` routine. The first overload accepts an `SPWeb` object, and the other two accept `SPListItem` and `SPListItemCollection` objects as the second parameter, respectively. However, the `Hold.RemoveHold(int, SPWeb)` routine does not accept the third `comments` string parameter, and it is indicated as being "reserved for internal use" in the SDK documentation, so you should not call it directly to release a hold from all documents in the web site. Instead, you should call the `Hold.ReleaseHold(int holdId, SPWeb web, string comments)` method, which accepts the `comments` string parameter.

The Search & Process API

The Records Management API includes support for Search & Process operations. This feature allows for the execution of a keyword query and then performing a custom operation on every list item that is returned from the query. I've included the description of this subsystem here in the Holds chapter, because applying and removing holds is one of the most common scenarios in which it is used.

The Search & Process feature is exposed through the `Microsoft.Office.RecordsManagement.SearchAndProcess` namespace and can only be accessed from code. The purpose of this API is to provide a way to process list items efficiently without having first to retrieve them from the content database and then iterate over the collection in memory. It also allows SharePoint to schedule each call into the assembly that implements the operation in a way that does not affect the overall performance of the system.

Chapter 11: Suspending Record Processing Using Holds

To use the Search & Process API, you first create an assembly that contains one or more classes that implement the `IProcess` interface. This interface is declared in the `Microsoft.Office.RecordsManagement.SearchAndProcess` namespace as follows:

```
public interface IProcess
{
    bool ProcessItem(SPLListItem item, string[] args, out string msg);
}
```

The return code from the `ProcessItem` method indicates whether the item was processed successfully, and the output string receives a description of the processing results. The `args` parameter contains an array of additional arguments that may be specified by the user.

The Search & Process API can be confusing at first because of the way it is structured. In order to ensure that the processing of items returned by the query is handled efficiently, everything — including the query itself — must be scheduled and coordinated with the SharePoint system. This is done using the `SearchAndProcessItem` class. After creating an instance of this class and setting the required property values, you then call the `Add` method to schedule the process with SharePoint. Listing 11-3 shows an example.

Listing 11-3: Search & Process example

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.SharePoint;
using Microsoft.Office.RecordsManagement.SearchAndProcess;
using Microsoft.Office.RecordsManagement.Internal;

namespace SearchAndPlaceOnHold
{
    public class JobScheduler
    {
        public void ScheduleSearchAndProcessJob(SPWeb web)
        {
            SearchAndProcessItem job = new SearchAndProcessItem();
            Type processor = typeof(Microsoft.Office.RecordsManagement.Internal.
                AddToHold);
            job.AssemblyName = processor.Assembly.GetName().FullName;
            job.ClassName = processor.FullName;
            job.KeywordQuery = "Enron";
            job.Add(web, DateTime.Now, Guid.Empty, 0, false, Guid.Empty,
                Guid.Empty, web.CurrentUser.ID);
        }
    }
}
```

In this example, a keyword query is defined to locate documents that contain the keyword `Enron` so that the resulting documents can be placed under a hold. It turns out that placing documents on hold is so common that the `Microsoft.Office.Policy` assembly already includes an implementation

Chapter 11: Suspending Record Processing Using Holds

of the `IProcess` interface needed to perform this operation. This implementation is provided by the `AddToHold` class from the `Microsoft.Office.RecordsManagement.Internal` namespace. So all we have to do is set up the `SearchAndProcessItem` instance so that it specifies the appropriate `AssemblyName` and `ClassName` and then call the `Add` method to schedule it for execution the next time the SharePoint Master Timer job runs.

A keyword query is different from a normal CAML query. Keyword queries use the Windows SharePoint Services Search Keyword syntax to execute a query against MOSS Enterprise Search, whereas CAML queries operate directly against the content database.

Summary

This chapter provided an overview of the Hold feature in MOSS that prevents records from being deleted or expired, allowing auditors to effectively suspend the normal operation of policies and other processes related to the disposition of records. MOSS provides an integrated set of components that enable the creation of `Hold` objects that are linked to individual documents or list items through a `HoldField` custom field that is bound to each record through a column on the list. This mechanism enables the Hold processing layer to leverage standard MOSS mechanisms such as event receivers to protect individual records from being destroyed.

We examined the changes that are made to a Records Center web site in order to support holds processing, and we saw how to add and remove holds programmatically. We also explored the Search & Process API, which offers an efficient alternative to iterating manually through a collection of list items to add or remove holds by providing a way to schedule the processing of each item using a keyword query and the SharePoint Master Timer job.

12

Building and Deploying Custom Routers

The SharePoint records management components are built around an extensible framework for processing records that are submitted to the records center. That framework can be extended to include custom record routing logic by building a router component and then associating it with a particular record series type. The custom router is called by the framework after a record of that type is received by the records center but before the record is placed into the target document library. This gives us an opportunity to add post-processing logic that participates in the routing mechanism. As an example, this architecture could be used to check the integrity of incoming records and reject them if they fail to conform to specific validation requirements. In this chapter, we will explore the custom routing framework in detail by building several different types of custom routers.

Building Custom Routers

In Chapter 4, when we looked into the process of transferring files into the records center, you saw that the default routing architecture took care of placing the document into the appropriate document library. Now if we go back to the record vault and take another look at the routing type for legal documents, you see that when we edit the item, we don't have an option to add any kind of custom router. That's because we haven't built and installed any custom routers yet. They haven't been registered in the system. Custom routers are registered in a manner similar to information policy features. Once we create a custom router and register it, then it will show up in the list of available routers and we can select it for a given record series type.

A significant limitation of the MOSS routing architecture is that it only supports attaching a single router to a given record type at a time. This presents a challenge if we want to leverage the routing framework to apply more than one type of rule. Thus, using a custom router for validation would preclude using a second router for other purposes, such as de-duplication. One approach might be to place special rules in the target document library itself or to use information policy features. This would not be ideal because the record would already be in the repository, thus defeating the purpose of having a custom router in the first place.

Chapter 12: Building and Deploying Custom Routers

A better approach would be to create a *master router* that loads a secondary set of *routing action* components to perform the actual routing. Such a system might include configuration pages for an administrator that displayed the list of installed routing actions and then allowed the administrator to map selected routing actions to the current list of record routing types. The master router would then examine the record type associated with an incoming record, load the specified routing actions, and then call them in sequence to achieve the desired result. This would effectively create a *routing pipeline* that supports as many actions as needed.

Constructing such a framework would involve many technical challenges and is beyond the scope of this book. First of all, it would have to be transactional so that the entire pipeline of routing actions is performed as an atomic operation. Otherwise, a record could be partially processed, which would undermine the entire purpose of having a records center at all. Second, the ordering of actions might be significant, so there would need to be some way to establish a prescribed sequence for executing the custom actions, and so on. Third, there would have to be a central location for storing the collection of installed routing actions and their configuration information, and you would also have to hook into the standard routing configuration mechanisms (CAML views, etc.) provided by the Records Center site definition in order to enable a records administrator to select and configure the individual routing actions.

Building a custom router involves creating a .NET class that implements the `IRouter` interface. This interface is defined in the `Microsoft.Office.RecordsManagement.RecordsRepository` namespace. This interface declares a single method called `SubmitFile`, as shown in Listing 12-1.

Listing 12-1: `IRouter` interface

```
using Microsoft.SharePoint;
using System;

namespace Microsoft.Office.RecordsManagement.RecordsRepository
{
    public interface IRouter
    {
        RouterResult OnSubmitFile(
            string recordSeries,
            string sourceUrl,
            string userName,
            ref byte[] fileToSubmit,
            ref RecordsRepositoryProperty[] properties,
            ref SPList destination,
            ref string resultDetails);
    }
}
```

When the `SubmitFile` method is called, we get the record series name, the source URL, the user, the bytes associated with the file, the property bag, the destination list, and a string in which to store the result details. Most of this information maps to the information provided by the submitter using the Official File Web Service to submit a record. Some of it, however, comes from the initial processing performed by SharePoint to determine where the record will be stored.

Although the destination `SPList` parameter is declared using the `ref` keyword, it is not intended to be changed. SharePoint will still send the record to the designated destination unless an error occurs during the process. We'll look at this more closely when we build a redirecting router later in the chapter.

The `RouterResult` return type is used to tell SharePoint what to do with the file. There are three different router results that can be returned: `SuccessContinueProcessing`, `SuccessCancelFurtherProcessing`, and `RejectFile`. We need two success values to handle the situation in which you determine that a given file should not be stored in the repository but don't want to abandon the entire operation. Although it's up to the calling application to determine what to do when an error condition arises, you need the `SuccessCancelFurtherProcessing` value to indicate that the application should consider the operation a success, but that the document was not fully processed for some reason. A description of the reason is returned in the `resultDetails` parameter. The `resultDetails` are also returned to the caller along with the `RejectFile` result code.

Building the router is one thing, but deploying the router is another. This is another area where we can leverage the `InstallUtil` utility and build an abstract base class for routers that inherits from the `Installer` component class. We could then apply the `RunInstaller` attribute to your derived router class so that administrators can install our custom router from the command line. It would also be useful to have the option of installing the router using a `FeatureActivated` feature receiver.

In the following sections, we'll extend the record routing capability of the records center to redirect, track, and filter incoming records based on document metadata as well as custom business rules. This will require that we create several custom routers that will be installed by a feature that can be activated on a given records center site.

Let's start by creating a new SharePoint Feature project in Visual Studio and give it the name **ECM2007.CustomRouting**. We'll set the feature scope to *Web* so that it can be activated on any SharePoint web site, as shown in Figure 12-1.

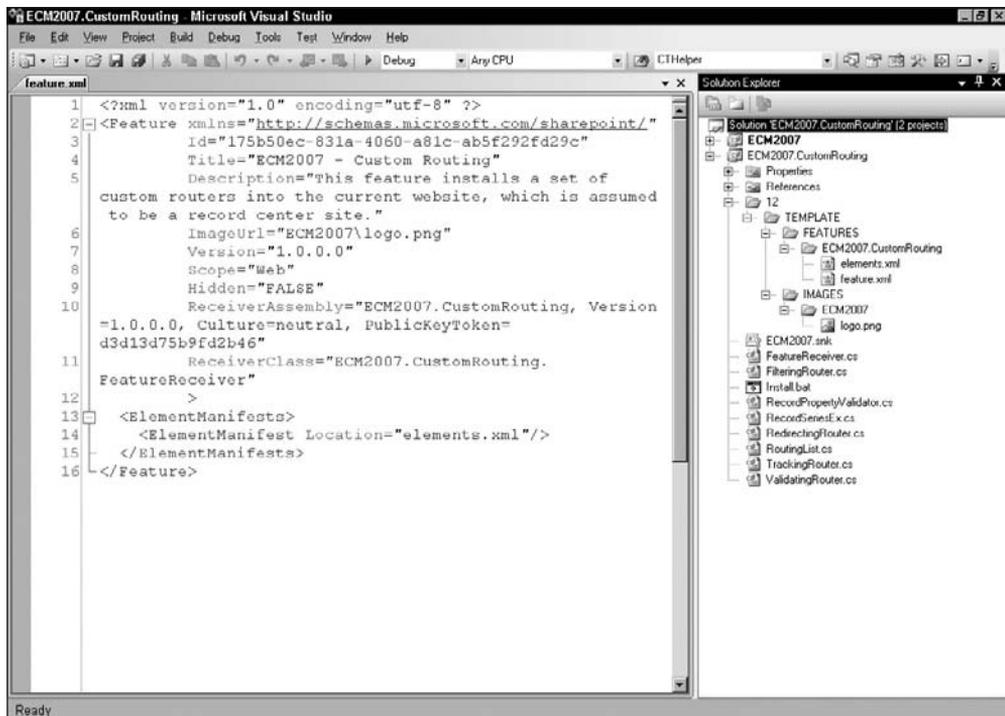


Figure 12-1: Custom routing feature.

Chapter 12: Building and Deploying Custom Routers

Our feature will take advantage of utility code from the ECM2007 class library that we've been using throughout the book. In this case, we'll be using classes from the `ECM2007.RecordsManagement` namespace. Again, we'll simply add the project to our solution and then add a reference to it and to the `System.Configuration.Install` assembly that the ECM2007 project depends on. Since we will also be relying on classes in the `Microsoft.Office.RecordsManagement` namespace, we will need to add a reference to the `Microsoft.Office.Policy.dll` file. The initial project should resemble Figure 12-2.

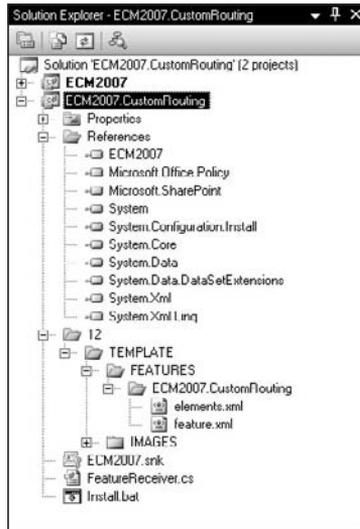


Figure 12-2: Custom routing solution setup.

The first router we create will simply filter incoming records based on incoming metadata. This router will examine selected properties associated with each incoming record and then decide whether to accept or reject the file based on the property values. In this example, the filtering rules will be hard-coded into the logic of the `SubmitFile` method. In actual practice, we would read the rules in from an external file or from a database system.

Creating a Simple Filtering Router

Add a new class to the project called `FilteringRouter` and open the new code file for editing. Add the following lines to the set of using statements at the top of the file:

```
using System.Diagnostics;
using Microsoft.Office.RecordsManagement.RecordsRepository;
using WSS=Microsoft.SharePoint;
using ECM2007.RecordsManagement;
```

The `WSS=` prefix is required because there is a conflict between the `RecordsRepositoryProperty` object declared in the `Microsoft.SharePoint` namespace and an object of the same name declared in the `Microsoft.Office.RecordsManagement.RecordsRepository` namespace.

The `ECM2007.RecordsManagement` namespace contains utility classes that simplify the construction of various SharePoint components. One of these is the `SharePointRouter` base class that understands how to install and uninstall custom routers and also provides a default implementation of the `IRouter` interface that all custom routers must implement. We will derive our custom router from this abstract base class and then override the `OnSubmitFile` method.

Next, we'll modify the generated class declaration so that it matches the following code:

```
[Name("ECM2007 Filtering Router")]
public class FilteringRouter : SharePointRouter
{
}
```

The `NameAttribute` class is a custom attribute class provided in the `ECM2007` utility library for attaching a name to any type. The `SharePointRouter` base class looks for this attribute and then uses it to determine the router name when the custom router is installed.

The `IRouter` interface declares a single method called `OnSubmitFile` that will contain our custom routing logic. We handle this by adding the code shown in Listing 12-2 to the class definition.

Listing 12-2: `FilteringRouter.OnSubmitFile`

```
/// <summary>
/// Custom implementation that validates the submitted files contents.
/// </summary>
protected override RouterResult OnSubmitFile(
    string recordSeries,
    string sourceUrl,
    string userName,
    ref byte[] fileToSubmit,
    ref RecordsRepositoryProperty[] properties,
    ref WSS.SPList destination,
    ref string resultDetails)
{
    // setup the default result...
    RouterResult result = RouterResult.SuccessContinueProcessing;
    try
    {
        if (!(ValidateContent(ref resultDetails, ref fileToSubmit)
            && ValidateMetadata(ref resultDetails, ref properties)))
        {
            result = RouterResult.RejectFile;
        }
    }
    catch (Exception x)
    {
        EventLog.WriteEntry("FilteringRouter", String.Format(
            "Exception occurred: {0}", x.Message));
        // Cancel if we encounter problems.
        result = RouterResult.SuccessCancelFurtherProcessing;
    }
    return result;
}
```

Chapter 12: Building and Deploying Custom Routers

Next, we apply custom business rules to validate the content and metadata of each incoming record using two utility methods called `ValidateContent` and `ValidateMetadata`:

```
/// <summary>
/// Checks the file content for validity. This can be any algorithm we like.
/// </summary>
bool ValidateContent(ref string resultDetails, ref byte[] fileToSubmit)
{
    return true;
}
```

We complete the router implementation by adding the following code to the class definition:

```
/// <summary>
/// Checks the metadata properties for validity and consistency.
/// </summary>
/// <param name="resultDetails"></param>
/// <param name="properties"></param>
/// <returns></returns>
bool ValidateMetadata(ref string resultDetails, ref RecordsRepositoryProperty[]
properties)
{
    foreach (RecordsRepositoryProperty property in properties)
    {
        if (property.Name.Equals("ContentType"))
        {
            // Only accept certain content types?
            if (property.Value.Equals("Document"))
            {
                Log("Rejecting generic document.");
                resultDetails = "Generic documents are not allowed.";
                return false;
            }
        }
        else if (property.Name.Equals("File_x0020_Type"))
        {
            // Only accept certain file extensions?
            if (property.Value.Equals("xls"))
            {
                Log("Rejecting Excel file.");
                resultDetails = "Excel Files are not allowed.";
                return false;
            }
        }
        else if (property.Name.Equals("_IsCurrentVersion"))
        {
            // Only accept current versions of documents.
            if (!property.Value.Equals("True"))
            {
                Log("Rejecting older version.");
                resultDetails = "Only current versions of documents can be stored
                    in the repository.";
                return false;
            }
        }
    }
}
```

```
    }  
  }  
  return true;  
}
```

Now that the router is created, we need to install it.

Installing the Router

In order for our router to be recognized within the SharePoint environment, it must be added to the global collection of routers for the Web on which our feature is activated. To accomplish this, we will use a factory method of the `SharePointRouter` utility class. In the `FeatureReceiver.cs` file, implement the `FeatureActivated` and `FeatureDeactivating` methods using the following code:

In this example, I am using my custom SharePointFeature Visual Studio project template, which is described in Chapter 2. You can also use your own favorite tool instead, making whatever minor modifications are necessary in order to deploy the feature into SharePoint each time the project is built.

```
/// <summary>  
/// Override the feature activation event to declare custom routers.  
/// </summary>  
/// <param name="properties"></param>  
public override void FeatureActivated(SPFeatureReceiverProperties properties)  
{  
    using (SPWeb web = properties.Feature.Parent as SPWeb)  
        SharePointRouter.AddRouter(web, typeof(FilteringRouter));  
}  
  
/// <summary>  
/// Override the feature deactivating event to remove custom routers.  
/// </summary>  
/// <param name="properties"></param>  
public override void FeatureDeactivating(SPFeatureReceiverProperties properties) {  
    using (SPWeb web = properties.Feature.Parent as SPWeb)  
        SharePointRouter.RemoveRouter(web, typeof(FilteringRouter));  
}
```

The next step is to close the file, build the project, and install the feature into the local SharePoint farm.

Activating the Router

There are two steps that must be completed before our custom router is available in the records center site. First, we must activate the feature. Second, we must associate the router with one or more record series types.

To activate the feature, navigate to the Site Settings page of the records center site and select the Site Features link. The `ECM2007.CustomRouter` feature should appear in the list of features. We can then activate the feature and navigate to the home page of the records center site and select a record routing type from the Record Routing list, for example, Financial Document. Then, from the Record Routing: Financial Document page, scroll to the bottom of the page and select your custom router from the list, as shown in Figure 12-3. Press OK to update the routing table.

Chapter 12: Building and Deploying Custom Routers

You can select any routing type you like. “Financial Document” was chosen here just to illustrate the process.

The screenshot shows a dialog box titled "Record Routing - Financial Document". It contains several fields and a dropdown menu. The "Title" field is "Financial Document". The "Description" field contains "Financial records such as annual reports, financial statements and related materials." The "Location" field is "Financial Documents" with a sub-description: "The title of the library where records matching this record routing item should be stored. Libraries used to store submitted records cannot be deleted." The "Aliases" field is "Annual Report/Financial Statement" with a sub-description: "A '/' delimited list of alternative names that represent this record routing entry." The "Default" field has an unchecked checkbox with the text: "If checked, this routing item will be used for submitted records that do not match the title or aliases of any other record routing item." The "Router" field has a dropdown menu open, showing the following options: "Do not perform additional record processing", "Do not perform additional record processing", "ECM2007 Filtering Router", "ECM2007 Tracking Router", and "ECM2007 Redirecting Router". The "ECM2007 Tracking Router" option is highlighted. To the right of the dropdown is the text "matching this". At the bottom of the dialog are "OK" and "Cancel" buttons. A footer at the very bottom reads: "Created at 12/2/2008 7:33 AM by System Account" and "Last modified at 12/2/2008 7:53 AM by System Account".

Figure 12-3: Filtering router selection.

Now we can submit a variety of file types to the records center from any document library. But when we try to submit an Excel spreadsheet, it is rejected. The same will be true for generic documents and older versions of existing documents.

The next router we create will be used to track incoming records by writing an entry to a custom list in the records center site. The same logic could be applied to write custom auditing records into the content database or to an external SQL database.

Creating a Tracking Router

Start by adding a new class to the project named `TrackingRouter` and then open the file for editing. As when creating the filtering router, add the following `using` statements at the top of the file:

```
using System.Diagnostics;
using Microsoft.Office.RecordsManagement.RecordsRepository;
using WSS=Microsoft.SharePoint;
using ECM2007.RecordsManagement;
```

Next, modify the class declaration so it matches the following code:

```
[Name("ECM2007 Tracking Router")]
public class TrackingRouter : SharePointRouter
{
}
```

Chapter 12: Building and Deploying Custom Routers

To simplify creating and updating the history list, we will use a nested wrapper class. Insert the following code inside the `TrackingRouter` class definition.

```
/// <summary>
/// A custom wrapper class that facilitates creating a list
/// and writing an entry to it.
/// </summary>
internal class RecordRouterHistoryList
{
    const string listTitle = "Record Routing History";
    const string listDescription = "This list is used by the TrackingRouter to
        record incoming parameter values.";

    const string colRecordSeries = "Record Series";
    const string colSourceUrl = "Source Url";
    const string colUserName = "User Name";
    const string colFileSize = "File Size";
    const string colDestination = "Destination";

    private WSS.SPList m_list = null;

    public RecordRouterHistoryList(WSS.SPList sourceList)
    {
        // Create the list in the same web as the source list.
        WSS.SPWeb web = sourceList.ParentWeb;

        // Create or open a custom list
        try { m_list = web.Lists[listTitle]; }
        catch { /* list does not exist - eat the exception */ }
        if (m_list == null)
        {
            try
            {
                m_list = web.Lists[
                    web.Lists.Add(listTitle, listDescription,
                        WSS.SPListTemplateType.GenericList)
                ];

                m_list.Fields.Add(colRecordSeries, WSS.SPFieldType.Text, false);
                m_list.Fields.Add(colSourceUrl, WSS.SPFieldType.URL, false);
                m_list.Fields.Add(colUserName, WSS.SPFieldType.Text, false);
                m_list.Fields.Add(colFileSize, WSS.SPFieldType.Number, false);
                m_list.Fields.Add(colDestination, WSS.SPFieldType.Text, false);
                m_list.OnQuickLaunch = true;
                m_list.Update();
            }
            catch
            {
                // if we get here, there is a catastrophic failure within
                // the SharePoint API - calling code will throw another
                // exception and log the error
            }
        }
    }
}

/// <summary>
```

```
/// Writes an entry to the list.
/// </summary>
public void WriteEntry(
    string recordSeries,
    string sourceUrl,
    string userName,
    int fileSize,
    string destination)
{
    WSS.SPLListItem item = m_list.Items.Add();
    item[colRecordSeries] = recordSeries;
    item[colSourceUrl] = sourceUrl;
    item[colUserName] = userName;
    item[colFileSize] = fileSize;
    item[colDestination] = destination;
    item.Update();
}
}
```

With our history list defined, we can override and implement the `OnSubmitFile` method. Add the following code to the `TrackingRouter` class definition:

```
/// <summary>
/// Custom implementation that writes incoming parameters
/// to a custom list. Creates the list if it does not exist.
/// </summary>
protected override RouterResult OnSubmitFile(
    string recordSeries,
    string sourceUrl,
    string userName,
    ref byte[] fileToSubmit,
    ref RecordsRepositoryProperty[] properties,
    ref WSS.SPLList destination,
    ref string resultDetails)
{
    // setup the default result...
    RouterResult result = RouterResult.SuccessContinueProcessing;
    try
    {
        // Write an entry to the history list.
        new RecordRouterHistoryList(destination).WriteEntry(
            recordSeries, sourceUrl, userName,
            fileToSubmit.Length, destination.Title);

        // Write an event log entry to capture the properties.
        EventLog.WriteEntry("TrackingRouter Properties",
            string.Format("Source Url = '{0}'\nProperties:\n{1}",
                sourceUrl, ExtractProperties(properties)));
    }
    catch (Exception x)
    {
        // Cancel if we encounter problems writing to the list.
        // (could reject with resultDetails)
        EventLog.WriteEntry("TrackingRouter",
            String.Format("Exception occurred: {0}", x.Message));
    }
}
```

```
        result = RouterResult.SuccessCancelFurtherProcessing;
    }
    return result;
}
```

This method performs the dual function of writing to the history list and also to the system event log, which it also uses to record any problems encountered while writing to the history list. To make it easier to deal with the custom properties that have been provided along with the incoming file, a separate helper method is used. Add the following code to the `TrackingRouter` class definition:

```
/// <summary>
/// Returns a string containing all of the properties in the array.
/// </summary>
/// <param name="properties"></param>
/// <returns></returns>
string ExtractProperties(RecordsRepositoryProperty[] properties)
{
    StringBuilder sb = new StringBuilder();
    foreach (RecordsRepositoryProperty property in properties)
        sb.AppendFormat("{2}{0}={1}", property.Name, property.Value, sb.Length > 0
            ? " ;" : "");
    return sb.ToString();
}
```

This completes the tracking router implementation. As a final step, we must add code to the `FeatureActivated` and `FeatureDeactivating` feature receiver methods to register and unregister the router in the SharePoint environment.

Open the `FeatureReceiver.cs` file and add the following line to the `FeatureActivated` method:

```
SharePointRouter.AddRouter(web, typeof(TrackingRouter));
```

Add the following line to the `FeatureDeactivating` method:

```
SharePointRouter.RemoveRouter(web, typeof(TrackingRouter));
```

Save and re-build the project, recycle the application pool or perform an `IISRESET`, and then deactivate and reactivate the feature. Select a record series type and enable the ECM2007 Tracking Router custom router. Now when you send a document to the repository that matches the selected record series type, you should see a new list named *Record Routing History* with an entry for each incoming file.

The final router we create will redirect incoming records to different document libraries based on metadata associated with each record. This is a standard requirement for record routing and the code we write can easily be extended for use in our own solutions.

Creating a Redirecting Router

As with the other routers in this set, start by adding a new class to the project named `RedirectingRouter`. Open the code file for editing and add the appropriate `using` statements at the top of the file:

```
using System.Diagnostics;
using Microsoft.Office.RecordsManagement.RecordsRepository;
```

Chapter 12: Building and Deploying Custom Routers

```
using WSS=Microsoft.SharePoint;
using ECM2007.RecordsManagement;
```

Modify the class declaration so it matches the following code:

```
[Name("ECM2007 Redirecting Router")]
public class RedirectingRouter : SharePointRouter
{
    // Write exceptions to the trace log.
    void HandleException(Exception x)
    {
        Log(string.Format("Exception occurred: {0}",x.ToString()));
    }
}
```

This implementation of `OnSubmitFile` will redirect incoming records based on the collection of metadata properties attached to the file. The metadata properties are provided in the `RecordsRepositoryProperties` array, which is passed to the method by SharePoint.

Add the following code to the `RedirectingRouter` class definition:

```
/// <summary>
/// Custom implementation that redirects incoming records
/// based on the metadata properties attached to the file.
/// </summary>
protected override RM.RouterResult OnSubmitFile(
    string recordSeries,
    string sourceUrl,
    string userName,
    ref byte[] fileToSubmit,
    ref RM.RecordsRepositoryProperty[] properties,
    ref SPList destination,
    ref string resultDetails)
{
    // setup the default result...
    RM.RouterResult result = RM.RouterResult.SuccessContinueProcessing;
    try
    {
        // get the content type from the property array
        string contentTypeName = string.Empty;
        foreach (RM.RecordsRepositoryProperty property in properties)
            if (property.Name.Equals("ContentType")) contentTypeName =
                property.Value;

        // use the content type name and properties to determine the correct
        // destination
        SPList newDestination = destination;
        if (AdjustDestination(contentTypeName, sourceUrl, userName,
            ref properties, ref newDestination))
        {
            string sourceFileName = Path.GetFileNameWithoutExtension(sourceUrl);

            // store the file into the new destination
            if (SaveDocument(contentTypeName, sourceUrl, userName, ref
```

```
        fileToSubmit, ref properties, ref newDestination, ref resultDetails))
    {
        // succeeded in saving the document...
        Log(string.Format("Saved '{0}' to '{1}'", sourceFileName,
            destination.Title));
    }
    else
    {
        // failed to save the document...
        Log(string.Format("Document save failed for '{0}'",
            sourceFileName));
    }

    // return success but cancel further processing, since we
    // are taking responsibility for storing the file...
    result = RM.RouterResult.SuccessCancelFurtherProcessing;
}
else
{
    // redirection is not required
    // continue processing normally
    Log(string.Format("Redirection not required for content type '{0}'",
        contentTypeName));
    result = RM.RouterResult.SuccessContinueProcessing;
}
}
catch (Exception x)
{
    // cancel processing if we encounter an error...
    HandleException(x);
    result = RM.RouterResult.SuccessCancelFurtherProcessing;
}
return result;
}
```

The goal of a redirecting router is to change the target location of each incoming file based on a custom algorithm.

The SharePoint API implies (by use of the `ref` keyword on the destination `SPList` parameter) that you can change the destination target simply by changing this value to reference another list. In actual practice, this does not work unless there is a problem storing the file into the original location. In order to change the destination, you have to do it explicitly in your custom router code.

Instead of relying on SharePoint to copy the file, we will use a helper method to copy the file ourselves depending on whether an adjustment to the destination is required. We will use another helper method to make that determination based on the incoming content type and metadata properties. Add the following helper method to the `RedirectingRouter` class definition:

```
// Attempts to adjust the destination list based on incoming metadata.
bool AdjustDestination(string contentTypeName, string sourceUrl, string userName,
    ref RM.RecordsRepositoryProperty[] properties,
    ref SPList destination)
{
```

Chapter 12: Building and Deploying Custom Routers

```
// if no content type name was provided, then no special processing is
// possible...
if (string.IsNullOrEmpty(contentTypeName))
    return false;

// if the content type name matches an existing document library, then use that
// as the new destination library...
SPList targetList = SharePointList.Find(destination.ParentWeb,
    contentTypeName);
if (targetList != null)
{
    Log(string.Format("Reusing existing list '{0}'", targetList.Title));
    destination = targetList;
}
else
{
    // target list does not exist, so create a new document library...
    destination = SharePointList.Create(destination.ParentWeb,
        SPListTemplateType.DocumentLibrary, contentTypeName,
        "Created by the RedirectingRouter");
    // no special fields added to the default "Document" content type
    destination.ContentTypesEnabled = true;
    destination.OnQuickLaunch = true;
    destination.Update();
}

return true;
}
```

To perform the actual copy operation, add the `SaveDocument` helper method to the class definition:

```
// Stores the document using metadata to determine where to place it within the
// target list.
bool SaveDocument(string contentTypeName, string sourceUrl, string userName,
    ref byte[] fileToSubmit, ref Microsoft.Office.RecordsManagement.
    RecordsRepository.RecordsRepositoryProperty[] properties,
    ref SPList destination, ref string resultDetails)
{
    SPFolder targetFolder = null;
    SPListItem item = null;
    SPFile file = null;
    string fileName = Path.GetFileNameWithoutExtension(sourceUrl);

    try
    {
        // get a folder in the destination list using the user name
        int pos = userName.LastIndexOf('\\');
        string actualUserName = userName.Substring(pos >= 0 ? pos : 0);
        targetFolder = SharePointList.CreateFolder(destination, actualUserName);
    }
    catch (Exception x1)
    {
        HandleException(new Exception("Failed to get target folder", x1));
    }

    try
```

```
{
    // add the document to the folder
    file = targetFolder.Files.Add(fileName, fileToSubmit);
    item = file.Item;
}
catch (Exception x2)
{
    HandleException(new Exception("Failed to add document to folder", x2));
}

// set the content type if recognized...
try
{
    SPContentType ct = destination.ContentTypes[contentTypeName];
    item["ContentTypeId"] = ct.Id;
    item.Update();
}
catch (Exception x3)
{
    HandleException(new Exception("Failed to update content type id", x3));
}

// copy any matching properties into the new item...
foreach (RM.RecordsRepositoryProperty property in properties)
{
    if (item.Fields.ContainsField(property.Name))
    {
        try
        {
            // check the validity of the target field...
            SPField field = item.Fields.GetField(property.Name);
            if (field != null && !field.ReadOnlyField &&
                (field.Type != SPFieldType.Invalid) &&
                (field.Type != SPFieldType.WorkflowStatus) &&
                (field.Type != SPFieldType.File) &&
                (field.Type != SPFieldType.Computed) &&
                (field.Type != SPFieldType.User) &&
                (field.Type != SPFieldType.Lookup) &&
                (!field.InternalName.Equals("ContentType")))
            {
                item[property.Name] = property.Value;
            }
        }
        catch (Exception x)
        {
            resultDetails = string.Format("Exception occurred while
            saving '{0}': {1}", fileName, x.Message);
            return false;
        }
    }
}

item.Update();
return true;
}
```

Chapter 12: Building and Deploying Custom Routers

Open the `FeatureReceiver.cs` file again and add the registration code to the feature receiver methods. Add this line to the `FeatureActivated` event receiver method:

```
SharePointRouter.AddRouter(web, typeof(RedirectingRouter));
```

Add this line to the `FeatureDeactivating` event receiver method:

```
SharePointRouter.RemoveRouter(web, typeof(RedirectingRouter));
```

Re-build the project, and then deactivate and activate the feature and associate the redirecting router for one or more record series types.

Extending the File Plan Schema to Support Custom Routing

In Chapter 5, we developed a file plan schema that allows us to create dynamic file plan components based on an XML data file and then *execute* the file plan to populate the record center with the content types, document libraries, and routing types needed for a given set of record definitions. Since custom routing can play a critical role in the disposition chain for a record, it makes sense to extend the file plan schema so that we can capture this additional information and then use it to enhance the file plan execution so that the custom router is automatically associated with the routing type.

To do this, we simply add a field to the file plan schema that allows the *end-user* (the person filling out the form) to specify which router he or she wants to use. The schema already includes a `RecordSpecification` element that describes each record type. Extend this element with a `Router` element using the following code:

```
<xs:element name="Router" type="xs:string" minOccurs="0">
  <xs:annotation>
    <xs:documentation>
      Specifies the name of the custom router to associate with
      this record type. The specified router must be installed separately
      into the record center by an administrator.
    </xs:documentation>
  </xs:annotation>
</xs:element>
```

Once the field has been added to the schema, we then regenerate the wrapper class using the `XSD.EXE` utility so that the field is available to our `FilePlan` component implementation:

```
[System.CodeDom.Compiler.GeneratedCodeAttribute("xsd", "2.0.50727.42")]
[System.SerializableAttribute()]
[System.Diagnostics.DebuggerStepThroughAttribute()]
[System.ComponentModel.DesignerCategoryAttribute("code")]
[System.Xml.Serialization.XmlTypeAttribute(
  Namespace="http://schemas.johnholliday.net/FilePlan.xsd")]
public partial class RecordSpecification {
  /* code omitted */
  private string routerField;
```

```
public string Router {
    get {
        return this.routerField;
    }
    set {
        this.routerField = value;
    }
}
```

Finally, we add some code to the `CreateRecordSeries` method that picks up the router name and then programmatically associates it with the new record series. The `RecordSeriesCollection.Add` method already accepts the router name as a parameter. Whereas before we passed an empty string, now we can pass the actual router name. If the specified router does not exist, the record series will still be created, but the router will not be attached.

```
/// <summary>
/// Creates the record series table entry for this record type.
/// </summary>
/// <param name="filePlan">a FilePlan object</param>
/// <param name="recordCenter">the records center site</param>
public void CreateRecordSeries(FilePlan filePlan, SPWeb recordCenter)
{
    if (this.Type == RecordType.Electronic)
    {
        // access the routing table for the record center
        RecordSeriesCollection routingTable =
            new RecordSeriesCollection(recordCenter);

        routingTable.Add(this.Name, this.SafeLocation, this.Description,
            this.Aliases, this.Router, false);
    }
}
```

Summary

This chapter examined the extensible routing architecture provided by the Office SharePoint Server records management API. The MOSS record routing framework can be used to address a variety of requirements related to the final disposition of incoming records. Typically, custom routers are used for filtering, tracking, and redirecting records to different locations based on the record metadata. This chapter showed how to build these types of routers and described some of the limitations of the current MOSS routing architecture, suggesting strategies for working around them. This chapter also showed how to extend the dynamic File Plan Schema introduced in Chapter 5 to include additional information about custom routing and showed the code needed to automatically associate the router with a record type when the file plan is executed.

13

Maintaining Record Integrity

In Chapter 1, we identified information integrity as a core records management requirement because any degradation in record integrity can affect many business processes and can also lead to legal liability. Often, companies must not only enforce a Records Management Policy that ensures records are not tampered with, but they must also prove that no tampering could have occurred. In this chapter, we will expand the notion of record integrity to include more than just creating tamper-proof records. We would also like to ensure that incoming records have *valid* metadata, which requires having the appropriate range of values in specific columns that are suitable for a given purpose. In this way we can ensure that a particular piece of content can be used to drive other business processes.

Building a Content Validation Framework

We would like to de-couple content validation from the core custom routing mechanism. Content validation is such a key requirement, it will often be necessary to incorporate some form of content validation logic into many different routing solutions. One way to do that is to build a separate validation framework that we can call from anywhere.

Although we are building the validation framework in the context of records management, it is important to note that this framework is suitable for inclusion in any SharePoint solution where you want to disallow certain content based on a precise set of validation rules.

In this section, we will attack the problem of validating list items by constructing a simple XML-based metadata parsing utility that examines the values of list item fields to determine if they match a given set of values. The matching values are expressed in XML so that we can attach them to a content type or to a router, store them in a separate list, or read them from an external database. Later, we will build a Validating Router that uses the validation framework to accept or reject incoming records based on a custom set of validation rules.

We want to end up with an XML file that describes all of the validation rules to be applied to an item, as well as the actions to be taken if the validation fails. For example, suppose our `SalesProposal` content type includes a field that specifies the proposal type as either `FixedBid`

Chapter 13: Maintaining Record Integrity

or `TimeAndMaterials`. We might then have a validation rule that requires all fixed bid proposals to have a bid amount over, say, \$5,000, in order to be accepted as an official record. Anything less than that should be rejected from the Records Center. Although somewhat contrived, we would like to express this rule using an XML fragment similar to Listing 13-1.

Listing 13-1: ProposalValidationRules.xml

```
<?xml version="1.0" encoding="utf-8" ?>
<ListItemValidator xmlns="http://tempuri.org/ListItemValidator.xsd">
  <Rule Name="MinimumBidAmount">
    <Match>
      <Eq>
        <Field>ProposalType</Field>
        <Value Type="String">FixedBid</Value>
      </Eq>
    </Match>
    <Try>
      <Geq>
        <Field>Bid Amount</Field>
        <Value Type="Decimal">5000</Value>
      </Geq>
    </Try>
    <Catch>
      <Throw>The bid amount must be at least 5000</Throw>
    </Catch>
  </Rule>
  <Rule Name="MinimumHours">
    <Match>
      <Eq>
        <Field>Proposal Type</Field>
        <Value Type="String">TimeAndMaterials</Value>
      </Eq>
    </Match>
    <Try>
      <Geq>
        <Field>Estimated Hours</Field>
        <Value Type="Number">300</Value>
      </Geq>
    </Try>
    <Catch>
      <Throw>The estimated hours must be at least 300.</Throw>
    </Catch>
  </Rule>
</ListItemValidator>
```

For any given list item, we can define a set of rules that are evaluated in the context of the item. Each rule is evaluated in sequence using a *match-try-catch* algorithm. If the logical expression contained in the `Match` element evaluates to `true`, then the logical expression in the `Try` element is evaluated. If the result is `true`, then the rule succeeds; otherwise, the rule fails and the `Catch` element is evaluated. If all of the rules in the set succeed, then the list item is considered valid; otherwise, it is invalid.

This set of rules states that for a given list item, two validation rules must be applied. The first declares that list items with a field named *Proposal Type* that equals the string `FixedBid` must have a decimal

field named *Bid Amount*, and the value contained in that field must be greater than or equal to \$5,000. The second declares that list items with a proposal type of *TimeAndMaterials* must have a number field named *Estimated Hours*, and the value contained in that field must be greater than or equal to 300.

Using this approach, we can easily create validation rules that can be applied to list items in many different contexts. The rules can be stored anywhere and can be evaluated independently of the items themselves. We can attach rules to content types, document libraries, timer jobs, site collections, or any persistable SharePoint object. We start by defining a validation schema.

Defining a Validation Schema

It is important that our validation schema be flexible enough to handle simple logical expressions. It would be nice to have a more elaborate language for building logical expressions, but we don't want to spend a lot of energy developing an expression parser. We could use regular expressions, but we also want the syntax to be readable since we would ultimately like to provide a tool for administrators to create and maintain validation rules — perhaps in a SharePoint list. Also, by using XML, we can provide administrators with the actual schema to help them construct syntactically correct rules.

It turns out that we can build a simple expression evaluator using XML elements that describe standard expressions with operators like *GEQ*, *LEQ*, and so on. Begin by creating a new XML schema file and creating a top-level element called *ListItemValidator*:

```
<?xml version="1.0" encoding="utf-8" ?>

<xs:schema id="ListItemValidator"
  targetNamespace="http://tempuri.org/ListItemValidator.xsd"
  elementFormDefault="qualified"
  xmlns="http://tempuri.org/ListItemValidator.xsd"
  xmlns:mstns="http://tempuri.org/ListItemValidator.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <!-- This is the top-level element that declares the validation rules -->
  <xs:element name="ListItemValidator">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Rule" type="Rule"
          minOccurs="1" maxOccurs="unbounded" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The top-level element contains a sequence of *Rule* elements. We set the *minOccurs* attribute to 1 to require at least one rule. The *maxOccurs* attribute is set to *unbounded* to allow for any number of rules. Since we specified a *type* other than *xs:string*, we need to add a definition for the *Rule* element:

```
<!-- This element declares an individual rule using the Match/Try/Catch pattern -->
<xs:complexType name="Rule">
  <xs:sequence>
    <xs:element name="Match" type="Expression" minOccurs="1" maxOccurs="1"/>
    <xs:element name="Try" type="Expression" minOccurs="1" maxOccurs="1"/>
    <xs:element name="Catch" type="Exception" minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
```

Chapter 13: Maintaining Record Integrity

```
<xs:attribute name="Name" type="xs:string" use="required"/>
</xs:complexType>
```

The `Rule` element declares an ordered sequence of subelements representing the `Match`, `Try`, and `Catch` elements, which are of type `Expression`. We set the `minOccurs` attribute of the `Catch` element to 0 so that it is optional. The `Match` and `Try` elements are required. There can be only one instance of each of these elements. The `Name` attribute allows the user to provide a name for the rule, which can be used in error messages.

Next, we declare the `Expression` element. This element can take many forms, each representing a different kind of expression. To set this up, we need an `xs:choice` element within the `xs:sequence` that is required, but limited to a single instance. This allows the XML validation to ensure that at least one of the subelements is present. Using this approach, we can map different keywords such as `And`, `Or`, and `Not` to the appropriate expression type. We need four types of expressions to describe validation rules — `UnaryLogicalExpression`, `LogicalExpression`, `FieldExpression`, and `FieldRangeExpression`:

```
<xs:complexType name="Expression">
  <xs:sequence>
    <xs:choice minOccurs="1" maxOccurs="1">
      <xs:element name="Any" type="xs:string"/>
      <xs:element name="Not" type="UnaryLogicalExpression"/>
      <xs:element name="And" type="LogicalExpression"/>
      <xs:element name="Or" type="LogicalExpression"/>
      <xs:element name="Eq" type="FieldExpression"/>
      <xs:element name="Neq" type="FieldExpression"/>
      <xs:element name="Gtr" type="FieldExpression"/>
      <xs:element name="Geq" type="FieldExpression"/>
      <xs:element name="Less" type="FieldExpression"/>
      <xs:element name="Leq" type="FieldExpression"/>
      <xs:element name="Contains" type="FieldExpression"/>
      <xs:element name="InRange" type="FieldRangeExpression"/>
    </xs:choice>
  </xs:sequence>
</xs:complexType>
```

The `UnaryLogicalExpression` element allows us to express the negation of some other expression. It is declared not as a sequence of subelements, but as a single choice between the subelement types.

```
<xs:complexType name="UnaryLogicalExpression">
  <xs:choice minOccurs="1" maxOccurs="1">
    <xs:element name="And" type="LogicalExpression"/>
    <xs:element name="Or" type="LogicalExpression"/>
    <xs:element name="Eq" type="FieldExpression"/>
    <xs:element name="Neq" type="FieldExpression"/>
    <xs:element name="Gtr" type="FieldExpression"/>
    <xs:element name="Geq" type="FieldExpression"/>
    <xs:element name="Less" type="FieldExpression"/>
    <xs:element name="Leq" type="FieldExpression"/>
    <xs:element name="Contains" type="FieldExpression"/>
    <xs:element name="InRange" type="FieldRangeExpression"/>
  </xs:choice>
</xs:complexType>
```

The `LogicalExpression` element allows us to describe a logical relationship between two expressions. It requires exactly two subelements.

```
<xs:complexType name="LogicalExpression">
  <xs:choice minOccurs="2" maxOccurs="2">
    <xs:element name="Not" type="UnaryLogicalExpression"/>
    <xs:element name="Eq" type="FieldExpression"/>
    <xs:element name="Neq" type="FieldExpression"/>
    <xs:element name="Gtr" type="FieldExpression"/>
    <xs:element name="Geq" type="FieldExpression"/>
    <xs:element name="Less" type="FieldExpression"/>
    <xs:element name="Leq" type="FieldExpression"/>
    <xs:element name="Contains" type="FieldExpression"/>
    <xs:element name="InRange" type="FieldRangeExpression"/>
  </xs:choice>
</xs:complexType>
```

The `FieldExpression` element allows us to identify a list item field by name and specify its value. This is used as part of a larger expression in which we need to compare the value of a field to some other value.

```
<xs:complexType name="FieldExpression">
  <xs:sequence>
    <xs:element name="Field" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="Value" type="ValueType" minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>
```

The `FieldRangeExpression` element is useful for testing whether the value of a list item field falls within a given range. This is particularly useful for date value types.

```
<xs:complexType name="FieldRangeExpression">
  <xs:sequence>
    <xs:element name="Field" type="xs:string" minOccurs="1" maxOccurs="1"/>
    <xs:element name="From" type="ValueType" minOccurs="1" maxOccurs="1"/>
    <xs:element name="To" type="ValueType" minOccurs="1" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>
```

The `ValueType` element is an atomic element used in the preceding expression elements to declare a scalar value of a given type. The `Type` attribute is used to coerce the supplied string into the designated underlying type at run time. It is declared as a `FieldDataType` so we can control what types the user is allowed to enter.

```
<xs:complexType name="ValueType">
  <xs:simpleContent>
    <xs:extension base="xs:string">
      <xs:attribute name="Type" type="FieldDataType" use="optional"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

The `Exception` element accepts a string that is used to communicate validation faults to the calling process. This message could be written to a log file or displayed to the user through the user interface.

```
<xs:complexType name="Exception">
  <xs:sequence>
    <xs:element name="Throw" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

The `FieldDataType` element specifies the recognized set of data types for scalar values. This could be extended to enable the user to specify any .NET type by adding an `xs:enumeration` element called, for example, `TypeSpecifier`. The specified string could then be interpreted as the name of a type that would then have to be resolved using .NET Reflection.

```
<xs:simpleType name="FieldDataType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Number"/>
    <xs:enumeration value="Date"/>
    <xs:enumeration value="String"/>
    <xs:enumeration value="Decimal"/>
    <xs:enumeration value="Double"/>
    <xs:enumeration value="Boolean"/>
  </xs:restriction>
</xs:simpleType>
```

Building Validation Components

Now that we have a validation schema, we can generate wrapper classes and extend them to build validation components. Our goal is to have a set of components that we can call from a variety of scenarios to validate list items, document properties, and any other object that might be the subject of validation rules. As an example, we might call a validator component from an event receiver attached to a list as illustrated in Figure 13-1.

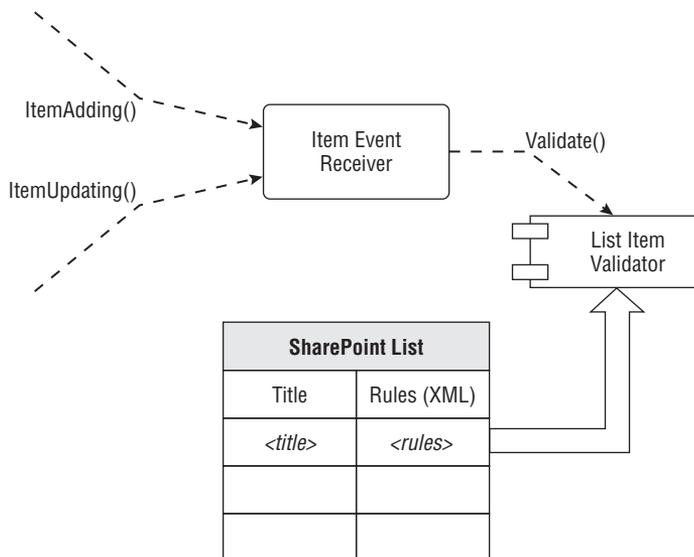


Figure 13-1: Validator called from an event receiver.

In this case, the validation rules might be attached directly to the list item, perhaps in a special column based on a custom field type. Or, we might want to associate the validation rules to a content type, as shown in Figure 13-2.

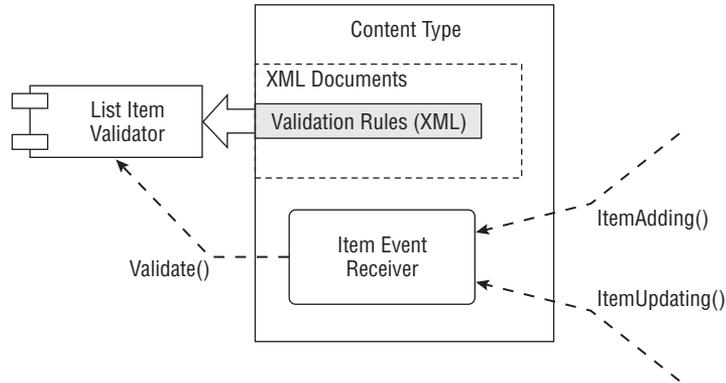


Figure 13-2: Validation rules in a content type.

In this case, the validation rules could be stored in an XmlDocument attached to the content type. This would facilitate building a custom policy feature wherein the policy is attached to a content type. Yet another scenario might involve a custom router, as shown in Figure 13-3.

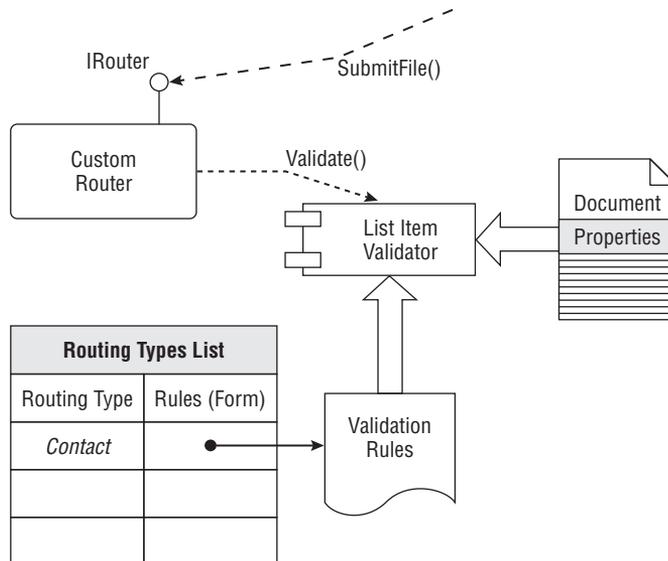


Figure 13-3: Validator called from a custom router.

Here, the validation rules could be attached directly to the routing table, so that for each routing type an administrator could specify the validation rules to be applied to incoming records. The custom router would then send the document properties to the validator instead of a list item.

Chapter 13: Maintaining Record Integrity

To build our validator component from the schema, we'll use the `XSD.EXE` custom tool introduced earlier to produce partial classes for each of the schema elements, and then we'll extend them so that callers can de-serialize a set of rules and pass objects in to be validated.

The first thing we need is a `ListItemValidator` constructor and some static factory methods we can use to create a new instance from data loaded from some other object. We at least need to have factory methods that create a `ListItemValidator` instance from a file or from an XML string. We'll include some public methods to allow the caller to easily determine the current validation state and to retrieve any error message that might have been generated. We'll also add an event that the caller can subscribe to so that whenever the validator detects a validation failure, a callback function can be invoked automatically.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Text;
using System.Xml;
using System.Xml.Serialization;
using Microsoft.SharePoint;

namespace ECM2007.Validation
{
    /// <summary>
    /// Extends the generated class to include static factory methods and to fire
    /// events
    /// when validation fails for a given list item.
    /// </summary>
    public partial class ListItemValidator
    {
        public bool IsValid { get; set; }
        public string ErrorMessage { get; set; }
        public event ValidationHandler ValidationFailed;
        public delegate void ValidationHandler(SPLListItem item, string message);

        #region Deserialization Methods

        public static ListItemValidator FromXml(string xmlString)
        {
            XmlSerializer ser = new XmlSerializer(typeof(ListItemValidator), "");
            ListItemValidator validator = null;
            try
            {
                validator = ser.Deserialize(new StringReader(xmlString)) as
                    ListItemValidator;
            }
            catch { }
            return validator;
        }

        public static ListItemValidator FromFile(string filename)
        {

```

```
        XmlSerializer ser = new XmlSerializer(typeof(ListItemValidator), "");
        ListItemValidator validator = null;
        try
        {
            validator = ser.Deserialize(new FileStream(filename,
                FileMode.Open)) as ListItemValidator;
        }
        catch { }
        return validator;
    }
#endregion
}
}
```

Before we get into the validator itself, let's take a look at how the expressions themselves are processed. This is the heart of the validator component and contains the code that does the actual work of comparing field values and taking the appropriate actions.

To start out, there are two scenarios we want to handle with regard to validating list items. First, we need to validate individual list items directly. This will allow us to mix in validation calls from wherever we have access to a list item instance, such as after executing a CAML query or when responding to an event receiver. Second, we'd like to validate list items indirectly, by passing in another structure from which we extract the actual list item. For example, it might be useful to have a method that understands the `SPItemEventDataCollection` object that is passed to list item event-receiver methods. Listing 13-2 shows the two public static methods that handle these scenarios.

Listing 13-2: Static expression evaluation methods

```
/// <summary>
/// Applies an operation to a list item for different kinds of expressions.
/// </summary>
public static bool Evaluate(SPListItem item, ItemChoiceType1 op, object
    expression)
{
    if (expression is FieldExpression)
        return ((FieldExpression)expression).Eval(item, op);
    else if (expression is FieldRangeExpression)
        return ((FieldRangeExpression)expression).Eval(item, op);
    else if (expression is LogicalExpression)
        return ((LogicalExpression)expression).Eval(item, op);
    else if (expression is UnaryLogicalExpression)
        return ((UnaryLogicalExpression)expression).Eval(item, op);
    return false;
}

/// <summary>
/// Applies an operation to a property set for different kinds of
    expressions.
```

Continued

Listing 13-2: Static expression evaluation methods *(continued)*

```
/// </summary>
public static bool Evaluate(SPItemEventDataCollection properties,
    ItemChoiceType1 op, object expression)
{
    if (expression is FieldExpression)
        return ((FieldExpression)expression).Eval(properties, op);
    else if (expression is FieldRangeExpression)
        return ((FieldRangeExpression)expression).Eval(properties, op);
    else if (expression is LogicalExpression)
        return ((LogicalExpression)expression).Eval(properties, op);
    else if (expression is UnaryLogicalExpression)
        return ((UnaryLogicalExpression)expression).Eval(properties, op);
    return false;
}
```

The static methods delegate to the separate `Eval` methods shown in Listings 13-3 and 13-4. These methods simply decode the operation and then delegate to the appropriate subexpression, each of which exposes its own `Eval` method.

Listing 13-3: `ListItem` expression evaluation method

```
/// <summary>
/// Determines if the values of the specified list item fields matches
/// this expression.
/// </summary>
/// <param name="item">the list item to test</param>
/// <returns>true if the item matches the rule, otherwise false</returns>
public bool Eval(SPListItem item)
{
    try
    {
        ItemChoiceType1 op = this.ItemElementName;
        if (Item is FieldExpression)
            return ((FieldExpression)Item).Eval(item, op);
        else if (Item is FieldRangeExpression)
            return ((FieldRangeExpression)Item).Eval(item, op);
        else if (Item is LogicalExpression)
            return ((LogicalExpression)Item).Eval(item, op);
        else if (Item is UnaryLogicalExpression)
            return ((UnaryLogicalExpression)Item).Eval(item, op);
        else if (ItemElementName == ItemChoiceType1.Any)
            return true;
    }
    catch
    {
    }
    return false;
}
```

Listing 13-4: `SPItemEventDataCollection` expression evaluation method

```
/// <summary>
    /// Processes a collection of item data properties.
    /// </summary>
    /// <param name="properties"></param>
    /// <returns></returns>
    public bool Eval(SPItemEventDataCollection properties)
    {
        try
        {
            ItemChoiceType1 op = this.ItemElementName;
            if (Item is FieldExpression)
                return ((FieldExpression)Item).Eval(properties, op);
            else if (Item is FieldRangeExpression)
                return ((FieldRangeExpression)Item).Eval(properties, op);
            else if (Item is LogicalExpression)
                return ((LogicalExpression)Item).Eval(properties, op);
            else if (Item is UnaryLogicalExpression)
                return ((UnaryLogicalExpression)Item).Eval(properties, op);
            else if (ItemElementName == ItemChoiceType1.Any)
                return true;
        }
        catch
        {
        }
        return false;
    }
}
```

Our validation model is based on a simple pattern that applies the validation rules to list items that match a set of conditions. To express those conditions, we use `FieldExpression` and `FieldRangeExpression` elements. The code shown in Listing 13-5 compares a list item (or a `SPItemEventDataCollection` object) to a value and returns true if it matches.

The return value is false if the specified field value is null.

Listing 13-5: `FieldExpression.cs`

```
using System;
using System.Collections.Generic;
using System.Text;
using Microsoft.SharePoint;

namespace ECM2007.Validation
{
    /// <summary>
    /// Applies an operator to a given field value to determine a match.
    /// </summary>
    public partial class FieldExpression
```

Continued

Listing 13-5: FieldExpression.cs (continued)

```
{
    public bool Eval(SPListItem item, ItemChoiceType1 op)
    {
        object fieldValue = Expression.GetFieldValue(item, this.Field);
        if (fieldValue == null)
            return false;
        return this.Value.Eval(op, fieldValue);
    }

    public bool Eval(SPItemEventDataCollection properties, ItemChoiceType1 op)
    {
        object fieldValue = Expression.GetFieldValue(properties, this.Field);
        if (fieldValue == null)
            return false;
        return this.Value.Eval(op, fieldValue);
    }
}
```

Similarly, we can also test whether a given field falls within a range of values. This is useful for validating list items that fall within a range of dates or numeric values.

```
public partial class FieldRangeExpression
{
    /// <summary>
    /// Determines if a field value is within a given range.
    /// </summary>
    /// <param name="item"></param>
    /// <param name="op"></param>
    /// <returns></returns>
    public bool Eval(SPListItem item, ItemChoiceType1 op)
    {
        object fieldValue = Expression.GetFieldValue(item, this.Field);
        if (fieldValue == null) return false;
        return
            this.From.Eval(ItemChoiceType1.Geq, fieldValue) &&
            this.To.Eval(ItemChoiceType1.Leq, fieldValue);
    }

    public bool Eval(SPItemEventDataCollection properties, ItemChoiceType1 op)
    {
        object fieldValue = Expression.GetFieldValue(properties, this.Field);
        if (fieldValue == null) return false;
        return
            this.From.Eval(ItemChoiceType1.Geq, fieldValue) &&
            this.To.Eval(ItemChoiceType1.Leq, fieldValue);
    }
}
```

The rest of the implementation falls out of the schema as declared. For example, Listing 13-6 shows the code for handling logical expressions.

Listing 13-6: LogicalExpression

```

public partial class LogicalExpression
{
    public bool Eval(SPLListItem item, ItemChoiceType1 op)
    {
        switch (op)
        {
            case ItemChoiceType1.And:
                return Expression.Evaluate(item, op, Items[0]) &&
                    Expression.Evaluate(item, op, Items[1]);
            case ItemChoiceType1.Or:
                return Expression.Evaluate(item, op, Items[0]) ||
                    Expression.Evaluate(item, op, Items[1]);
        }
        return false;
    }

    public bool Eval(SPIItemEventDataCollection properties,
        ItemChoiceType1 op)
    {
        switch (op)
        {
            case ItemChoiceType1.And:
                return Expression.Evaluate(properties, op, Items[0]) &&
                    Expression.Evaluate(properties, op, Items[1]);
            case ItemChoiceType1.Or:
                return Expression.Evaluate(properties, op, Items[0]) ||
                    Expression.Evaluate(properties, op, Items[1]);
        }
        return false;
    }
}

```

The `ValueType` code shown in Listing 13-7 handles the low-level value comparisons needed for evaluating expressions. The value of any given object is determined by the operator passed in. The set of operators is defined within the schema, and the comparison is driven by the data type of the field as declared within the validation rule.

Listing 13-7: ValueType.cs

```

/// <summary>
/// Handles low-level value comparisons.
/// </summary>
public partial class ValueType
{
    public bool Eval(ItemChoiceType1 op, object fieldValue)
    {
        FieldDataType dataType =
            this.TypeSpecified ? this.Type : FieldDataType.String;
        switch (dataType)
        {

```

Continued

Listing 13-7: ValueType.cs (continued)

```
case FieldDataType.Date:
{
    DateTime dateTest = DateTime.Parse(this.Value);
    DateTime dateValue = DateTime.Parse(fieldValue.ToString());
    switch (op)
    {
        case ItemChoiceType1.Eq:
            return dateValue.Equals(dateTest);
        case ItemChoiceType1.Geq:
            return dateValue.CompareTo(dateTest) >= 0;
        case ItemChoiceType1.Gtr:
            return dateValue.CompareTo(dateTest) > 0;
        case ItemChoiceType1.Leq:
            return dateValue.CompareTo(dateTest) <= 0;
        case ItemChoiceType1.Less:
            return dateValue.CompareTo(dateTest) < 0;
        case ItemChoiceType1.Neq:
            return !dateValue.Equals(dateTest);
        case ItemChoiceType1.Any:
            return true;
    } break;
}
case FieldDataType.Boolean:
{
    break;
}
case FieldDataType.Decimal:
{
    decimal decTest = decimal.Parse(this.Value);
    decimal decValue = decimal.Parse(fieldValue.ToString());
    switch (op)
    {
        case ItemChoiceType1.Eq:
            return decValue.Equals(decTest);
        case ItemChoiceType1.Geq:
            return decValue.CompareTo(decTest) >= 0;
        case ItemChoiceType1.Gtr:
            return decValue.CompareTo(decTest) > 0;
        case ItemChoiceType1.Leq:
            return decValue.CompareTo(decTest) <= 0;
        case ItemChoiceType1.Less:
            return decValue.CompareTo(decTest) < 0;
        case ItemChoiceType1.Neq:
            return !decValue.Equals(decTest);
    }
    break;
}
case FieldDataType.Double:
{
    double doubleTest = double.Parse(this.Value);
    double doubleValue = double.Parse(fieldValue.ToString());
    switch (op)
```

```

    {
        case ItemChoiceType1.Eq:
            return doubleValue.Equals(doubleTest);
        case ItemChoiceType1.Geq:
            return doubleValue.CompareTo(doubleTest) >= 0;
        case ItemChoiceType1.Gtr:
            return doubleValue.CompareTo(doubleTest) > 0;
        case ItemChoiceType1.Leq:
            return doubleValue.CompareTo(doubleTest) <= 0;
        case ItemChoiceType1.Less:
            return doubleValue.CompareTo(doubleTest) < 0;
        case ItemChoiceType1.Neq:
            return !doubleValue.Equals(doubleTest);
    }
    break;
}
case FieldDataType.Number:
{
    int intTest = int.Parse(this.Value);
    int intValue = int.Parse(fieldValue.ToString());
    switch (op)
    {
        case ItemChoiceType1.Eq:
            return intValue.Equals(intTest);
        case ItemChoiceType1.Geq:
            return intValue.CompareTo(intTest) >= 0;
        case ItemChoiceType1.Gtr:
            return intValue.CompareTo(intTest) > 0;
        case ItemChoiceType1.Leq:
            return intValue.CompareTo(intTest) <= 0;
        case ItemChoiceType1.Less:
            return intValue.CompareTo(intTest) < 0;
        case ItemChoiceType1.Neq:
            return !intValue.Equals(intTest);
    }
    break;
}
case FieldDataType.String:
{
    string stringTest = this.Value;
    string stringValue = fieldValue.ToString();
    switch (op)
    {
        case ItemChoiceType1.Eq:
            return stringValue.Equals(stringTest);
        case ItemChoiceType1.Geq:
            return stringValue.CompareTo(stringTest) >= 0;
        case ItemChoiceType1.Gtr:
            return stringValue.CompareTo(stringTest) > 0;
        case ItemChoiceType1.Leq:
            return stringValue.CompareTo(stringTest) <= 0;
        case ItemChoiceType1.Less:
            return stringValue.CompareTo(stringTest) < 0;
        case ItemChoiceType1.Neq:

```

Continued

Listing 13-7: `ValueType.cs` (continued)

```
                return !stringValue.Equals(stringTest);
            }
            break;
        }
    }
    return false;
}
}
```

Since our validator will potentially be used in many different scenarios, we want to have a flexible mechanism for detecting when the validation fails. One approach is to create a validator object, call the validation method, and then check if any errors have occurred. Another approach is to attach an event handler that is called whenever validation fails. Using event handlers will make it easier for developers to trap validation failure events without having to write code for each validation scenario. Listing 13-8 shows the code needed to enable both techniques.

Listing 13-8: Validation helper methods

```
#region Validation Methods

/// <summary>
/// Helper to fire validation events.
/// </summary>
protected void OnValidationFailed(SPListItem item, string message)
{
    this.IsValid = false;
    this.ErrorMessage = message;
    if (ValidationFailed != null)
        ValidationFailed(item, message);
}

/// <summary>
/// Helper to set flags for later interrogation by caller.
/// </summary>
/// <param name="properties"></param>
/// <param name="message"></param>
protected void OnValidationFailed(SPItemEventDataCollection properties, string
message)
{
    this.IsValid = false;
    this.ErrorMessage = message;
    if (ValidationFailed != null)
        ValidationFailed(null, message);
}

#endregion
```

Now we can write the last bit of code that evaluates the validation rules in sequence to determine which rules match a given list item. The matching rules are then fired, raising the `ValidationFailed` event if the rule evaluation fails.

```
/// <summary>
/// Validates all items in a collection.
/// </summary>
/// <param name="items"></param>
/// <returns></returns>
public bool Validate(SPListItemCollection items)
{
    foreach (SPListItem item in items)
        if (!Validate(item))
            return false;
    return true;
}

/// <summary>
/// Evaluates the validation rules in sequence to determine which rules match
/// this list item.
/// Matching rules are then fired, raising the ValidationFailed event on failure.
/// </summary>
/// <param name="item">the list item to be validated</param>
/// <returns>true if the validation succeeded</returns>
public bool Validate(SPListItem item)
{
    try
    {
        this.IsValid = true;
        this.ErrorMessage = string.Empty;
        EventLog.WriteEntry("ListItemValidator: Validating Item", item.Title);

        foreach (Rule rule in this.Rule)
        {
            Log("ListItemValidator: Validating Rule", "Rule = " + rule.Name);
            if (rule.Match.Eval(item))
            {
                Log("ListItemValidator: Matched Rule", "Rule = " + rule.Name);
                if (!rule.Try.Eval(item))
                {
                    Log("ListItemValidator: rule.Try.Eval(item)", "Failed -
                        Firing Event...");
                    OnValidationFailed(item, rule.Catch.Throw);
                }
                else
                {
                    Log("ListItemValidator: rule.Try.Eval(item)", "Succeeded");
                }
            }
        }
    }
    catch
    {
        return false;
    }
    return true;
}
```

Using the Validation Framework with a Self-Validating Proposal Content Type

The typical validation scenario for a content type would involve creating a `ListItemValidator` component by loading the validation rules from an XML string stored in a SharePoint list or attached to the content type. The validation code would be called from the `ItemAdding` and `ItemUpdating` methods of an item event receiver to determine if the item is valid. If it were not, then the operation would be canceled with an appropriate error message. The following section illustrates this approach.

Using our ECM2007 content type components, we can easily declare a Self-Validating Proposal type that uses the validation framework to ensure that any proposal added or uploaded into a document library conforms to a given set of validation rules.

```
using System;
using System.Diagnostics;
using System.IO;
using System.Reflection;
using Microsoft.SharePoint;
using ECM2007.ContentTypes;
using ECM2007.Validation;

namespace ECM2007.ValidatingProposal
{
    [
        SharePointContentType(
            Name = "Validating Proposal",
            Description = "A Sales Proposal Content Type with Validation",
            BaseType = "Document",
            Group = "ECM2007",
            Hidden = false,
            XmlDocuments =
                "res://ProposalRules.xml[http://tempuri.org/ListItemValidator.xsd]")
    ]
    public class ProjectProposal : SPItemEventReceiver
    {
        const string VALIDATION_NAMESPACE =
            "http://tempuri.org/ListItemValidator.xsd";

        public enum ProposalTypeEnum
        {
            FixedBid,
            TimeAndMaterials
        }

        void Log(string message)
        {
            Trace.WriteLine(message, "ProjectProposal");
        }

        #region Field References

        [FieldRef("ProposalType",
            DisplayName = "Proposal Type",
            Required = true)]
```

```

public ProposalTypeEnum ProposalType { get; set; }

[FieldRef("BidAmount",
    DisplayName = "Bid Amount")]
public decimal BidAmount { get; set; }

[FieldRef("EstHours",
    DisplayName = "Estimated Hours")]
public int EstimatedHours { get; set; }

#endregion

#region Event Receiver Methods

/// <summary>
/// Prevent the creation of invalid proposals.
/// </summary>
/// <param name="properties"></param>
public override void ItemAdding(SPIItemEventProperties properties)
{
    base.ItemAdding(properties);
    Log("ItemAdding");
    ValidateFromResourceFile(ref properties, "ProposalRules.xml");
}

/// <summary>
/// Prevent the updating of invalid proposals.
/// </summary>
/// <param name="properties"></param>
public override void ItemUpdating(SPIItemEventProperties properties)
{
    base.ItemUpdating(properties);
    Log("ItemUpdating");
    ValidateFromResourceFile(ref properties, "ProposalRules.xml");
}

#endregion
}
}

```

In this example, since we are building the content type as a .NET component, we can store the validation rules as an embedded resource directly in the assembly. The following routine shows how to load the embedded resource and convert it to an XML string. The string is then passed to one of the static factory methods to obtain an initialized instance of the `ListItemValidator`.

```

void ValidateFromResourceFile(ref SPIItemEventProperties properties, string
    rulesFile)
{
    try
    {
        Assembly asm = Assembly.GetExecutingAssembly();
        string resourcePath = asm.GetName().Name + "." + rulesFile;

        Log("Resource path = " + resourcePath);
        Stream stream = asm.GetManifestResourceStream(resourcePath);
    }
}

```

```
    if (stream == null)
    {
        Log("Failed to load rules file: " + resourcePath);
        return;
    }

    Log("Calling validator");
    StreamReader reader = new StreamReader(stream);
    ListItemValidator validator =
        ListItemValidator.FromXml(reader.ReadToEnd());
    bool isValid = validator.Validate(properties.AfterProperties);
    if (!isValid)
    {
        properties.Cancel = true;
        properties.ErrorMessage = validator.ErrorMessage;
        Log(string.Format("Validation failed: {0}", validator.ErrorMessage));
    }
}
catch (System.Exception x)
{
    Log(String.Format("Exception during validation: {0}", x.Message));
    EventLog.WriteEntry("Validating Proposal", "Exception during validation: "
        + x.ToString());
}
}
```

Other scenarios might require that the validation rules can be changed after the content type code is deployed. In such scenarios, the validation rules would need to be stored in the SharePoint content database. A convenient location is to store them as an `XmlDocument` attached to the content type itself. The following routine shows how to locate the content type object from the `SPIItemEventProperties` and then extract the validation rules from the `XmlDocuments` collection:

```
void ValidateFromXmlDocument(ref SPIItemEventProperties properties, string nsRules)
{
    try
    {
        Log("ValidateFromXmlDocument");
        // get the web associated with the object
        using (SPWeb web = properties.OpenWeb())
        {
            // get the content type
            string ctName = properties.AfterProperties["ContentType"].ToString();
            Log("Retrieving content type: " + ctName);

            SPContentType ctProposal = web.ContentTypes[ctName];
            if (ctProposal == null)
                Log("Failed to retrieve content type " + ctName);

            if (ctProposal != null)
            {
                // Create a list item validator by loading the rules from the
                // content type.
            }
        }
    }
}
```


Listing 13-9: FeatureReceiver.cs (continued)

```
    /// <summary>
    /// Override to create the project proposal content type.
    /// </summary>
    /// <param name="properties"></param>
    public override void FeatureActivated(SPFeatureReceiverProperties
        properties)
    {
        SPSite siteCollection = properties.Feature.Parent as SPSite;
        if (siteCollection != null)
        {
            // Create the Project Proposal content type
            SPContentType ctProposal =
                SharePointContentType.Create(siteCollection.RootWeb,
                    typeof(ProjectProposal));
        }
    }

    /// <summary>
    /// Override to remove the project proposal content type.
    /// </summary>
    /// <param name="properties"></param>
    public override void FeatureDeactivating(SPFeatureReceiverProperties
        properties)
    {
        SPSite siteCollection = properties.Feature.Parent as SPSite;
        if (siteCollection != null)
        {
            SharePointContentType.Delete(siteCollection.RootWeb,
                typeof(ProjectProposal));
        }
    }
    public override void FeatureInstalled(SPFeatureReceiverProperties
        properties) { }
    public override void FeatureUninstalling(SPFeatureReceiverProperties
        properties) { }
}
}
```

Now we can activate the feature and create an instance of the proposal. Entering invalid values as shown in Figure 13-4 generates the error page shown in Figure 13-5.

Other ways to use the validation framework include building a Validating Router for a Records Center or creating a validation policy feature using information management policy. To create a validation policy using the Information Policy framework, we would build a custom policy feature with a user interface that allows an administrator to enter the rules as an XML string. The policy feature would then store the rules for use later, either by adding them to the property bag of the site or attaching them to the list or to the content type associated with the policy. At the same time, the policy feature would install event receivers for the `ItemAdding` and `ItemUpdating` events similar to what we have done here. When the event receivers are called, the validation rules are retrieved, and the item is validated.

Professional ECM 2007 > Validated Proposals > Consulting proposal > Edit Item

Validated Proposals: Consulting proposal

OK Cancel

X Delete Item | Spelling... * indicates a required field

Name * Consulting proposal .docx

Title Consulting Proposal

Proposal Type * FixedBid

Bid Amount 450

Estimated Hours

Created at 12/3/2008 8:13 AM by System Account
Last modified at 12/3/2008 8:14 AM by System Account

OK Cancel

Figure 13-4: Entering invalid proposal values.



Figure 13-5: Proposal validation error page.

Using the Validation Framework to Build a Validating Router

A Validating Router would work slightly differently. Instead of installing event receivers and validating a list item, the router would validate the properties associated with the incoming record. In order to validate an incoming record, we need a place to store the validation rules. Since the incoming record series name is uniquely associated with an `SPLISTITEM` in the Record Routing Table, we can attach the validation rules to the matching record series item in the Records Center site. This allows the records manager to copy-and-paste the rules' XML into a column of the routing type to control how the records associated with that type are validated. The router will then locate the item and extract the rules to evaluate them against the incoming document properties.

Chapter 13: Maintaining Record Integrity

To set this up, we'll need a helper class that exposes a method we can call to ensure that the custom field has been added to the routing table before we attempt to access it.

```
/// <summary>
/// Wrapper class used to manipulate the record routing table
/// in a records center site.
/// </summary>
public static class RoutingList
{
    // The name of a custom field used to store validation rules.
    public const string RULES_FIELDNAME = "ValidationRules";

    // Default validation rules if none are specified for a series.
    public const string DEFAULT_VALIDATIONRULES =
        "<ListItemValidator><Rule Name=\"Default\"></Rule>
        </ListItemValidator>";

    /// <summary>
    /// Ensures that the record routing table is properly
    /// configured to hold validation rules.
    /// </summary>
    /// <param name="web"></param>
    /// <returns></returns>
    public static SPList EnsureSetupValidation(SPWeb web)
    {
        SPList list = null;

        try
        {
            // Get the routing table as an SPList.
            SPList routingTable = web.GetList(RecordSeries.ListUrl(web));

            // Check if the routing table list contains the rules field.
            if (!routingTable.Fields.ContainsField(RULES_FIELDNAME))
            {
                // Add a column to the list to hold validation rules
                string result = routingTable.Fields.Add(RULES_FIELDNAME,
                    SPFieldType.Note, false);
                SPField rulesField = routingTable.Fields[RULES_FIELDNAME];
            }
        }
        catch (Exception x)
        {
            Helpers.HandleException(typeof(RoutingList), x);
        }
        return list;
    }
}
```

We also need a routine to convert a `RecordSeries` object into an `SPListItem` so we can retrieve the validation rules that have been added by the records manager:

```
/// <summary>
/// Retrieves the SPListItem associated with a given RecordSeries object.
```

```
/// </summary>
public static SPLListItem GetRecordSeriesItem(SPWeb web, RecordSeries
    series)
{
    try
    {
        SPLList routingTable = web.GetList(RecordSeries.ListUrl(web));
        SPLListItemCollection items = routingTable.Items;
        foreach (SPLListItem seriesItem in items)
            if (seriesItem.Title.Equals(series.Name))
                return seriesItem;
    }
    catch (Exception x)
    {
        Helpers.HandleException(typeof(RoutingList), x);
    }
    return null;
}
```

Finally, we can tie all this together with another static method that retrieves the validation rules from a given Web and record series:

```
/// <summary>
/// Extracts any validation rules that may be defined for a given record
/// series, while ensuring that the routing table is properly configured
/// for validation.
/// </summary>
public static string GetValidationRules(SPWeb recordCenter, RecordSeries
    series)
{
    string result = RoutingList.DEFAULT_VALIDATIONRULES;
    try
    {
        RoutingList.EnsureSetupValidation(recordCenter);
        SPLListItem seriesItem =
            RoutingList.GetRecordSeriesItem(recordCenter, series);
        if (seriesItem != null)
        {
            string test = seriesItem[RoutingList.RULES_FIELDNAME]
                .ToString();
            if (!string.IsNullOrEmpty(test))
                result = test;
        }
    }
    catch (Exception x)
    {
        Helpers.HandleException(typeof(RecordSeries), x);
    }
    return result;
}
```

Chapter 13: Maintaining Record Integrity

Next, we need to extend the `ListItemValidator` so that it supports validation against an array of `RecordsRepositoryProperty` objects instead of just `SPLListItem` and `SPIItemEventDataCollection` objects. To do this, we simply add an additional `Eval` method to each of the following classes:

- `Expression`
- `FieldExpression`
- `FieldRangeExpression`
- `LogicalExpression`
- `UnaryLogicalExpression`

The new `Eval` method can be created by simply copying the code from the existing `Eval(SPIItemEventDataCollectionProperties)` method and changing the `properties` parameter type to `RecordsRepositoryProperty[]`. The following code shows the added `Eval` method for the `Expression` class:

```
/// <summary>
/// Processes a collection of records repository properties.
/// </summary>
/// <param name="properties"></param>
/// <returns></returns>
public bool Eval(RecordsRepositoryProperty[] properties)
{
    try
    {
        ItemChoiceType1 op = this.ItemElementName;
        if (Item is FieldExpression)
            return ((FieldExpression)Item).Eval(properties, op);
        else if (Item is FieldRangeExpression)
            return ((FieldRangeExpression)Item).Eval(properties, op);
        else if (Item is LogicalExpression)
            return ((LogicalExpression)Item).Eval(properties, op);
        else if (Item is UnaryLogicalExpression)
            return ((UnaryLogicalExpression)Item).Eval(properties, op);
        else if (ItemElementName == ItemChoiceType1.Any)
            return true;
    }
    catch (Exception x)
    {
        Helpers.HandleException(this, x);
    }
    return false;
}
```

Finally, we can subclass the `ListItemValidator` class to create a specialized class for handling record properties. Our `RecordPropertyValidator` class simply defines a custom `Validate` method that accepts an array of `RecordsRepositoryProperty` objects and then calls the appropriate `Eval` methods shown above.

```
public class RecordPropertyValidator : ListItemValidator
{
    public bool Validate(RecordsRepositoryProperty[] properties)
```

```

{
    try
    {
        this.IsValid=true;
        this.ErrorMessage = string.Empty;
        foreach (Rule rule in this.Rule)
            if (rule.Match.Eval(properties))
                if (!rule.Try.Eval(properties))
                {
                    OnValidationFailed((SPLListItem)null, rule.Catch.Throw);
                    return false;
                }
    }
    catch (System.Exception x)
    {
        Helpers.HandleException(typeof(RecordPropertyValidator), x);
        return false;
    }
    return true;
}
}

```

Now we can use the `RoutingList` wrapper class to write the code for a custom router that validates incoming documents. The essential elements are that the `OnSubmitFile` method must retrieve the `RecordSeries` object specified for the record and then access the routing table to extract any validation rules that may have been added by the administrator for that routing type. Figure 13-6 shows what this might look like in a Records Center site.

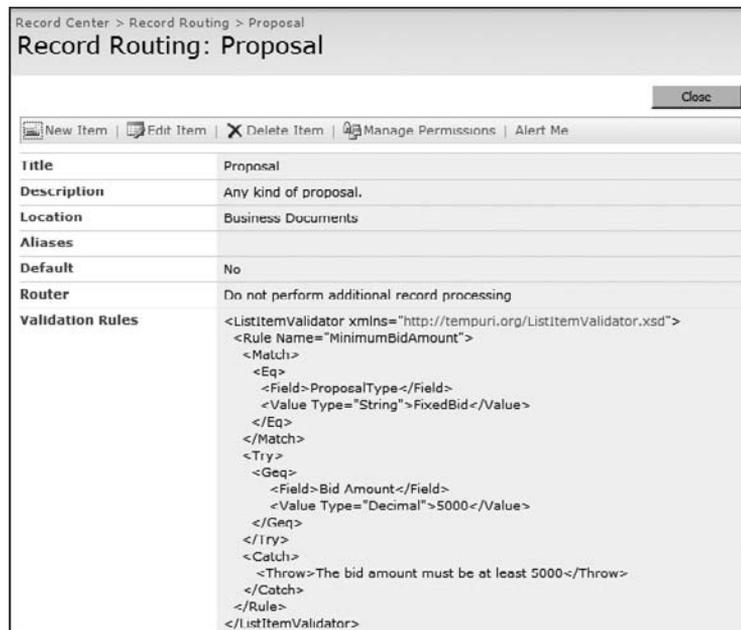


Figure 13-6: Routing type extended with validation rules.

Chapter 13: Maintaining Record Integrity

Listing 13-10 shows the code for the *ValidatingRouter* class.

Listing 13-10: ValidatingRouter.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using Microsoft.Office.RecordsManagement.RecordsRepository;
using WSS=Microsoft.SharePoint;
using ECM2007.RecordsManagement;
using ECM2007.Validation;

namespace ECM2007.CustomRouting
{
    [Name("ECM2007 Validating Router")]
    public class ValidatingRouter : SharePointRouter
    {
        /// <summary>
        /// Calls the RecordsRepositoryValidator to check the validity
        /// of document properties that were submitted with the file.
        /// </summary>
        protected override RouterResult OnSubmitFile(
            string recordSeries,
            string sourceUrl,
            string userName,
            ref byte[] fileToSubmit,
            ref RecordsRepositoryProperty[] properties,
            ref WSS.SPList destination,
            ref string resultDetails)
        {
            // setup the default result...
            RouterResult result = RouterResult.SuccessContinueProcessing;
            try
            {
                // Get the web associated with the records center
                // from the destination list.
                WSS.SPWeb web = destination.ParentWeb;

                // Get the named record series object.
                RecordSeries series = new RecordSeries(web, recordSeries, true);

                // Make sure the routing list is properly configured.
                RoutingList.EnsureSetupValidation(web);

                // Get the validation rules from the record series.
                string validationRules = RoutingList.GetValidationRules(web,
                    series);

                if (!string.IsNullOrEmpty(validationRules))
                {
                    // Create a validator from the rules.
                    RecordPropertyValidator validator =
```

```
        ListItemValidator.FromXml(validationRules)
            as RecordPropertyValidator;

        // Validate the document properties.
        if (!validator.Validate(properties))
        {
            // Store the result details from the error message in the
            // validator
            resultDetails = validator.ErrorMessage;

            // Set the return value to reject the file.
            result = RouterResult.RejectFile;
        }
    }
}
catch (System.Exception x)
{
    // Cancel if we encounter problems validating the properties.
    EventLog.WriteEntry("ValidatingRouter",
        String.Format("Exception occurred: {0}", x.Message));
    result = RouterResult.SuccessCancelFurtherProcessing;
}
return result;
}
}
```

Summary

Record integrity is a core requirement for any records management system. This chapter explored ways to fulfill that requirement by showing how to build a flexible content validation framework that can be applied to any list item to validate its metadata against a given set of rules. Leveraging the expressive power of XML, this chapter developed a validation schema that was then used to generate wrapper classes for the key elements of the schema. These wrapper classes were then extended to support the validation of `SPLListItem` and `SPIItemEventDataCollection` objects so they could be called easily from event receivers on list items or content types.

The chapter showed how to integrate the validation framework by building a Self-Validating Proposal content type that illustrated the steps needed to invoke the validator from an event receiver. The chapter also showed how to extend the framework to include support for validating incoming records by building a Validating Router component that applies validation rules attached to record routing types in the Records Repository.

14

Managing Electronic Mail Records

The purpose of this chapter is to provide an overview of how MOSS is set up to handle e-mail records. If we're talking about Office SharePoint Server 2007 and e-mail, then we have to include Exchange 2007 in the discussion because of the tight integration it provides. It turns out that Exchange 2007 is designed to work in tandem with MOSS to provide end-to-end "content life-cycle" support that includes Messaging Records Management (MRM). MRM is an additional layer within Exchange 2007 that seeks to simplify the process of staying in compliance with company policy, government regulations, or other requirements.

E-mail comprises the majority of enterprise content, much of which is directly related to official communications that become part of the official record and history related to particular business activities. It's not difficult to imagine scenarios in which e-mails must be treated as something more than simply transient messages. For example, an e-mail message can have attachments. If the message is to be treated as an official record, then its attachments would also need to be included in the Records Management strategy.

Failure to develop an effective system for managing e-mail records can have serious financial and personal consequences, as shown by often cited court cases such as *Murphy Oil v. Fluor Daniel* (2002), *United States v. Philip Morris* (2004), and *CIBC World Markets v. Genuity Capital Markets* (Canada, 2005).

In the Murphy Oil case, the plaintiff requested discovery of e-mail messages that were stored on backup tapes over the course of 14 months. The defendant estimated it would take 6 months and more than \$6 million to produce them. In the Philip Morris case, the court fined the defendant company \$2.75 million because it continued to delete e-mail messages in spite of a litigation hold.

In the Canadian CIBC case, Genuity was being sued for an alleged trade secret infringement. It was learned during discovery that Genuity did not maintain a centrally managed electronic mail archive, so the court allowed forensic experts to examine *all* PCs and smart phones of *all* of its employees, including those belonging to their spouses and children.

Chapter 14: Managing Electronic Mail Records

Planning for e-mail records management can be viewed as an extension of the File Plan strategy we used in Chapter 1 to design document-based records management solutions. Whether implementing the plan using the out-of-the-box features of Exchange 2007 or integrating it with a SharePoint Records Center site, the same information needs to be gathered. To accommodate the special requirements for handling e-mail, we would need to extend the semantic definition of *File Plan* to include descriptions of the components that must also be created and/or configured to support the submission and processing of e-mail messages and attachments. To that end, it would be an added bonus if we can find ways to extend our Dynamic File Plan concept from Chapter 5 so that the processing rules may be used to configure Outlook 2007 and Exchange 2007, as well as the SharePoint Records Center.

How would this work? First, we would have to extend our File Plan Schema to include additional metadata that describes the retention policies so they can be used to configure the e-mail system. Such metadata might include rules that specify permissions for certain groups and users, or that specify retention periods and actions for *e-mail messages* as distinct from *documents* for the same file type, while another set of metadata elements might control the processing of attachments, and so on. Once we've extended the File Plan Schema, we can then look at ways to *realize* or *render* a given file plan within Exchange 2007 and/or Outlook 2007 in the same way that we "executed" the File Plan InfoPath form within SharePoint. In that way, we can ensure that the same file plan description applies equally to any custom configuration that may be applied to the SharePoint Records Center.

Figure 14-1 shows a high-level overview of the overall processing strategy. We have managed folders in Exchange that allow us to set up rules that are attached to each folder. There is the client-side story provided by Outlook to move messages into those managed folders. And then you have a link between the centrally managed folders in Exchange and the Records Center site in SharePoint. That link is maintained not through the typical route that uses the Web Services (WS) layer to send documents to the Repository, but actually by simply forwarding e-mails from the managed folders to the Repository.

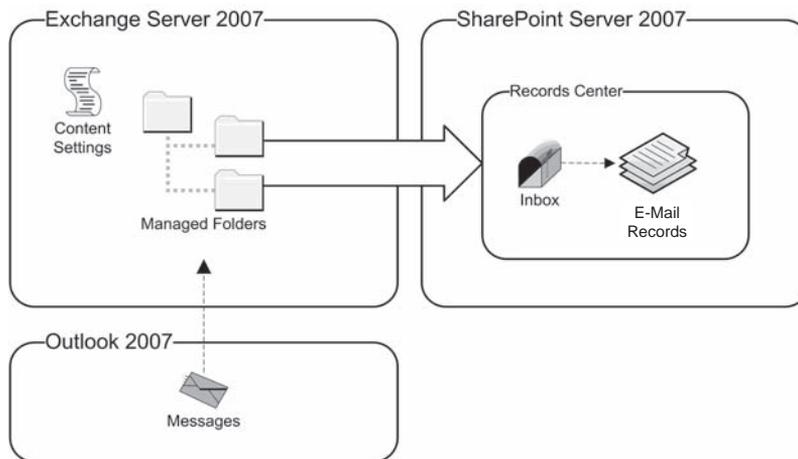


Figure 14-1: E-mail Records Management architecture.

There are three parts to the overall solution. First, how do we set up the managed folders in Exchange and what tools are available for configuring them, both manually and programmatically? We'll go through the manual steps of creating a managed folder, and then we'll look at how this might be done programmatically from code implemented in the dynamic file plan. Second, how will users interact

with the centrally managed folders to designate certain messages as *official* messages? This typically requires only that users see the managed folders in Outlook 2007 so they can drag official messages into the appropriate folders. It might also include setting up rules in the Outlook client to move or copy messages to those folders automatically. The third component requires that the SharePoint Farm is properly configured to receive incoming e-mails and that the Records Center site exposes an e-mail address that can be used for submitting e-mail messages. We'll look at the steps needed to set this up and also examine what happens within the Records Center as those e-mail messages are processed.

It is not technically necessary to use Exchange 2007 managed folders to support MRM, since Outlook can be configured to simply forward e-mail messages directly to the Records Center. However, using Exchange has the added advantage of enabling administrators to control the flow of official e-mail messages centrally without having to reconfigure each e-mail client individually whenever the location or e-mail address of the Records Center changes, or when additional processing rules and policies must be applied to e-mail messages.

Configuring Exchange 2007

E-mail records management is called *Messaging Records Management* (MRM) in Exchange. It is designed around the concept of *managed folders*. A number of managed folders are created by default. For example, the Inbox is a managed folder that automatically appears in each user's mailbox. Managed folders are designed to support MRM through the use of policies called *mailbox policies*, and each managed folder has a `FolderType` associated with it. The Inbox is of type `InboxFolder`. Each user's mailbox can have only one mailbox policy assigned to it, and each policy can be associated with only one managed folder of a given type.

To configure Exchange 2007 for records management, we must perform the following steps:

1. Create a managed folder for each e-mail record type you want to manage.
2. Configure the managed content settings for the folder to control what happens to messages added to the folder.
3. Create a managed folder mailbox policy. The policy may contain a policy statement that is displayed to users so they understand the purpose of the folder and how it should be used.
4. Add managed folders to the mailbox policy to make the folders available to end-users. When the policy is associated with a mailbox, the managed folders appear in Outlook, allowing users to route e-mail messages to the managed folders in Exchange.
5. Run the `Managed Folder Assistant` utility to complete the configuration within Exchange. This utility is like a timer job that can be scheduled or run immediately. Until it is executed, the managed folders will not appear in the user interface (UI).

The following sections describe these tasks in greater detail. The first thing to do to configure Exchange 2007 is to create managed folders.

Creating Managed Folders

This section shows how to create a managed folder using the user interface of the `Exchange Management Console` and programmatically using the `Exchange Management Shell`. The `Exchange Management Console` is a tool that is included with Exchange Server to perform

Chapter 14: Managing Electronic Mail Records

administrative tasks. The Exchange Management Shell provides a command-line interface to the same functionality based on Windows PowerShell cmdlets.

To create a managed folder using the user interface provided by the Exchange Management Console, perform the following steps:

1. Select the Mailbox node under the Organization Configuration node.
2. Choose the `New Managed Custom Folder` command. The dialog shown in Figure 14-2 is shown. This dialog includes several options that control what the user sees in Outlook. These options are useful for letting the user know the purpose of the folder and how it should be used. This can be particularly important for records management where policy statements are often used to keep users informed and to avoid the claim that they did not know about the policy.
3. Fill out the form, click on the `New` button, and then click on the `Finish` button to close the dialog. The new managed folder now appears in the console window.

New Managed Custom Folder

New Managed Custom Folder
 Completion

New Managed Custom Folder
Managed custom folders are mailbox folders with settings that control the content within the folder.

Name:
OfficialMessages

Display the following name when the folder is viewed in Office Outlook:
Official Messages

Storage limit (KB) for this folder and its subfolders:

Display the following comment when the folder is viewed in Outlook:
This folder is provided for the purpose of saving official communications that will be copied to the records center for long-term storage based on a centrally managed retention period.

Do not allow users to minimize this comment in Outlook.

Managed custom folders are a premium feature of messaging records management. Each mailbox that has managed custom folders requires an Exchange enterprise client access License (CAL).

Help < Back New Cancel

Figure 14-2: Creating a managed folder using the Exchange Management Console.

Another way to create a managed folder is to use the Exchange Management Shell shown in Figure 14-3, which provides a command-line interface to many of the functions that are available through the console.

```

Machine: john-9dcb5dfb1e | Scope: johnholliday.local
Welcome to the Exchange Management Shell!

Full list of cmdlets:           get-command
Only Exchange cmdlets:        get-excommand
Cmdlets for a specific role:   get-help -role *UM* or *Mailbox*
Get general help:              help
Get help for a cmdlet:         help <cmdlet-name> or <cmdlet-name> -?
Show quick reference guide:    quickref
Exchange team blog:           get-exblog
Show full output for a cmd:    <cmd> | format-list

Tip of the day #45:
Forgot what the available parameters are on a cmdlet? Just use tab completion!
Type:

Set-Mailbox -<tab>

When you type a hyphen (-) and then press the Tab key, you will cycle through all
the available parameters on the cmdlet. Want to narrow your search? Type part
of the parameter's name and then press the Tab key. Type:

Set-Mailbox -Prohibit<tab>

[PS] C:\Documents and Settings\Administrator>_

```

Figure 14-3: The Exchange Management Shell.

This is an important capability for our purposes, because our ultimate goal is to extend the dynamic file plan component so we can automate the Exchange 2007 configuration by calling methods implemented by our `FilePlan` component. To do that, we can call the interfaces that are exposed by the Exchange Management Shell, and therefore employ many of the same techniques that are typically used by Exchange administrators who are familiar with the shell. As an example, the following command creates a managed folder using the `New-ManagedFolder` cmdlet within the Exchange Management Shell.

```
New-ManagedFolder -Name OfficialMessages -FolderName "Official Messages"
```

In order to extend our `FilePlan` component so that we can create managed folders as part of the File Plan execution, we need a way to invoke commands in the Exchange Management Shell from .NET. To do that, we'll add a utility class to the ECM2007 foundation classes that will facilitate communication with Exchange. We'll add this class to the ECM2007 project in Visual Studio.

By adding methods to the FilePlan component, we are essentially adding a layer of support for building tools that can be used by administrators to help them correctly configure the SharePoint Farm as well as other servers for a given set of record types.

To handle the configuration of an Exchange 2007 server, one approach would be to customize the InfoPath 2007 form we created in Chapter 5 to capture the File Plan data so that it includes a Configure Exchange command. Using such a form, an administrator running the InfoPath client directly on the Exchange Server machine could then open any File Plan document stored in a SharePoint form library and then configure the server directly from within InfoPath.

Another option might be to create a Windows PowerShell cmdlet that could be deployed to the Exchange Server and loaded into the Exchange Management Shell. This cmdlet would accept the URL of the File Plan data file as a parameter and then load it to automatically configure the Exchange server.

Exchange Server 2007 commands are implemented as Windows PowerShell cmdlets that are called from within the Exchange Management Shell. Our utility class will do the same. Therefore, we must reference the PowerShell namespaces, which are located in the `System.Management.Automation` assembly.

Chapter 14: Managing Electronic Mail Records

Although this assembly is installed into the Global Assembly Cache when PowerShell is installed, it does not show up in the list of .NET assemblies when you try to add the reference to your project. To get around this, we'll add the reference manually to the .csproj file using the following procedure:

1. Right-click on the project node in the Visual Studio Solution Explorer and choose "Unload Project."
2. Right-click on the project node again and choose "Edit <projectname>."
3. Search for the ItemGroup containing Reference nodes, and add the line **<Reference Include="System.Management.Automation"/>**.
4. Save the file and reload it into Visual Studio.

We can now create the `ExchangeManagementShell` utility class as shown in Listing 14-1.

For more examples of how to call Exchange Management Shell commands from managed code, see the following MSDN article: <http://msdn.microsoft.com/en-us/library/bb332449.aspx>.

Listing 14-1: Exchange Management Shell utility class

```
using System;
using System.IO;
using System.Collections;
using System.Collections.Generic;
using System.Reflection;
using System.Text;
using System.Management.Automation;
using System.Management.Automation.Host;
using System.Management.Automation.Runspaces;

namespace ECM2007
{
    /// <summary>
    /// Provides a wrapper for calling methods in the
    /// Exchange Management Shell.
    /// </summary>
    public class ExchangeManagementShell
    {
        public static Runspace Runspace;
        public static RunspaceConfiguration Configuration;
        public static RunspaceInvoke RunspaceInvoke;
        public const string SnapInName =
            "Microsoft.Exchange.Management.PowerShell.Admin";

        /// <summary>
        /// Initializes the runspace that will be used to execute
        /// PowerShell commands. Happens only once.
        /// </summary>
        private static void Init()
        {
            if (Configuration == null)
            {
```

```
Configuration = RunspaceConfiguration.Create();
AppDomain.CurrentDomain.AssemblyResolve
    += new ResolveEventHandler(AssemblyResolver);

PSSnapInException ex;
PSSnapInInfo info=Configuration.AddPSSnapIn(SnapInName,out ex);
if (ex != null)
{
    Helpers.HandleException(typeof(ExchangeManagementShell), ex);
}
Runspace = RunspaceFactory.CreateRunspace(Configuration);
Runspace.Open();
}
}

/// <summary>
/// Adds the path to the Exchange 2007 assemblies so that
/// PowerShell can find them.
/// </summary>
private static Assembly AssemblyResolver(object obj,
    ResolveEventArgs args)
{
    if (args.Name.Contains("Microsoft.Exchange"))
    {
        string programFiles = Environment.GetFolderPath(
            Environment.SpecialFolder.ProgramFiles);
        string exchangePath = Path.Combine(programFiles,
            @"Microsoft\Exchange Server\bin\");
        return Assembly.LoadFrom(
            Path.Combine(exchangePath,
                args.Name.Split(', ')[0] + ".dll"));
    }
    return null;
}

/// <summary>
/// Retrieves the shared static RunspaceInvoke object.
/// </summary>
public static RunspaceInvoke Invoke
{
    get
    {
        Init();
        if (RunspaceInvoke == null)
            RunspaceInvoke = new RunspaceInvoke(Runspace);
        return RunspaceInvoke;
    }
}

/// <summary>
/// Invokes a simple command.
/// </summary>
/// <param name="command">a PowerShell command</param>
/// <returns>the PowerShell result set</returns>
```

Continued

Listing 14-1: Exchange Management Shell utility class (continued)

```
public static ICollection<PSObject> Run(string command)
{
    return Invoke.Invoke(command);
}

/// <summary>
/// Invokes a command with parameters.
/// </summary>
/// <param name="command">a PowerShell command</param>
/// <param name="input">input parameters to the command</param>
/// <returns>the PowerShell result set</returns>
public static ICollection<PSObject> Run(string command,
    IEnumerable input)
{
    return Invoke.Invoke(command, input);
}

/// <summary>
/// Invokes a command and retrieves any errors.
/// </summary>
/// <param name="command">a PowerShell command</param>
/// <param name="errorList">receives any error output</param>
/// <returns>the PowerShell result set</returns>
public static ICollection<PSObject> Run(string command,
    out IList errorList)
{
    return Run(command, null, out errorList);
}

/// <summary>
/// Invokes a command with parameters and retrieves any errors.
/// </summary>
/// <param name="command">A PowerShell command</param>
/// <param name="input">input parameters</param>
/// <param name="errorList">receives any error output</param>
/// <returns>the PowerShell result set</returns>
public static ICollection<PSObject> Run(string command,
    IEnumerable input, out IList errorList)
{
    return Invoke.Invoke(command, input, out errorList);
}
}
```

With this utility class now available in the library, we can use it to create a managed folder in Exchange using code like the following:

```
ICollection<PSObject> results
= ECM2007.ExchangeManagementShell.Run(
    "New-ManagedFolder -Name OfficialMessages -FolderName \"Official Messages\"");
```

We will use code similar to this when we extend the `FilePlan` component to support setting up managed folders automatically. First, we need to see how managed content settings are configured in Exchange.

Setting Up the Records Center E-Mail Address

When you create a Records Center site using the Records Center site definition, it checks to see if incoming e-mail has been enabled in Central Administration for site collections in the farm. If it has, then an e-mail address is automatically provisioned using the incoming e-mail settings that were specified in the Central Administration web site.

This e-mail address then appears in the Records Center announcements list as shown in Figure 14-4 and is automatically associated with special processing code that is responsible for parsing incoming e-mail messages and associating them with the correct routing type. This is the e-mail address you will use when setting up Managed Content Settings in Exchange.



Figure 14-4: Records Center e-mail address.

The important thing to remember here is that SharePoint does not configure the mailbox for you. It only allocates the e-mail address that it uses internally to determine which incoming messages are intended for the Records Center. In order to direct e-mail messages from the Exchange Server to the SharePoint Server, you have to create a contact object in the Exchange 2007 management console that contains the correct SMTP mail alias needed to route messages to the SharePoint Server. In the case of the Records Center shown in Figure 14-4, you would add the e-mail address `recordscenter_CI6TLK@MOSSDEV` to the contact object in Exchange and create a send connector pointing `moss.development` to the SharePoint Server.

Configuring Managed Content Settings

Once a managed folder is available in Exchange, it can be configured to control how long items remain in the folder and what to do when they expire. You can either delete the messages when their retention period ends or *journal* the messages to a separate storage location such as a SharePoint Records Center. This is done using Managed Content Settings and is the way that Exchange 2007 enables e-mail retention policies.

The retention policies that you set up in Exchange are in addition to any retention policies that might also be configured within the Records Center site. In this example, we will not specify a retention policy in Exchange, but instead delegate the retention policy to the Records Center.

Chapter 14: Managing Electronic Mail Records

To create Managed Content Settings using the Exchange Management Console, perform the following steps:

1. Select the Mailbox node under the Organization Configuration node in the Management Console window, and then navigate to the Managed Custom Folders tab and select the folder to be configured.
2. Right-click on the folder, select “New Managed Content Settings,” and then enter a name and a notification message for the user. For the Message type, leave the default set to “All Mailbox Content.”
3. Leave the “Length of retention period” checkbox unchecked and click “Next” to move to the Journal page. Since the management of messages affected by this policy will be delegated to the Records Center, we will not specify retention settings here. Instead, we will route them directly to the Records Center in the next step.
4. On the Journal page, select the “Forward copies to” checkbox and then click “Browse” to select the contact you created earlier for the SharePoint Records Center.
5. Enter a label that will be applied to messages that are forwarded to the Records Center. This is very important, because the label you enter is what will be used to select the Routing Type that controls how the message will be processed upon arrival at the Records Center. Think of this as the *Content Type* for all messages in the folder to which these settings will apply.
6. Finally, select the message format to be used when the message is forwarded. Normally, this will be the “Outlook Message Format (*.msg).” Using this format ensures that attachments are sent in their native format. The default “Exchange MAPI Message Format (TNEF)” embeds the attachments as a binary stream within the body of the message, making them more difficult to work with.
7. Click “Next” and then “New” and “Finish” to complete the wizard and create the settings.

In order to extend the `FilePlan` component, we again need to invoke the appropriate Exchange Management Shell commands. Listing 14-2 shows the necessary code added to the `FilePlan` component that calls the `ExchangeManagementShell` utility class from a new `SetupExchangeFolder` static method. This method accepts four parameters: the folder identifier and name, the target record routing type, and the Records Center e-mail address. When called, it creates a new managed folder and then completes the configuration by using the supplied parameters to configure the managed content settings for the new folder.

Listing 14-2: FilePlan extended to set up exchange

```
namespace ECM2007.RecordsManagement
{
    public partial class FilePlan
    {
        public static void SetupExchangeFolder(string folderId, string folderName,
            string recordRoutingType, string recordsCenterEmailAddress)
        {
            // create the managed folder in Exchange
            ICollection<PSObject> results = ExchangeManagementShell.Run(
```

```
        string.Format("New-ManagedFolder -Name {0} -FolderName \"{1}\"",
            folderId, folderName));

    // configure the content settings
    const string messageClass = "All Mailbox Content";
    const string messageFormat = "MSG";
    StringBuilder command = new
    StringBuilder("New-ManagedContentSettings");
    command.AppendFormat(" -FolderName {0}", folderName);
    command.AppendFormat(" -MessageClass {0}", messageClass);
    command.AppendFormat(" -Name {0}ContentSettings", folderId);
    command.AppendFormat(" -RetentionEnabled $false");
    command.AppendFormat(" -JournalingEnabled $true");
    command.AppendFormat(" -AddressForJournaling {0}",
        recordsCenterEmailAddress);
    command.AppendFormat(" -LabelForJournaling {0}", recordRoutingType);
    command.AppendFormat(" -MessageFormatForJournaling {0}",
        messageFormat);
    results = ExchangeManagementShell.Run(command.ToString());
    }
}
```

Using this method, we can simply extend the File Plan Schema to include a new record type, called `Official Message`, and then add a method to our `FilePlan` object to configure Exchange. This method might be invoked from a command-line utility or a custom PowerShell command to process an XML file containing a file plan on an existing Exchange 2007 Server. Listing 14-3 shows the modified `RecordType` element in the schema, and Listing 14-4 shows the new `FilePlan.ConfigureExchange` method.

Listing 14-3: File Plan Schema support for official messages

```
<xs:simpleType name="RecordType">
  <xs:annotation>
    <xs:documentation>
      Describes the incoming record type, which is used to determine how
      the file
      should be processed in the record center.  Electronic documents are
      placed
      into document libraries.  Physical documents are placed into custom lists
      with special columns used to store the document barcode or label.
    </xs:documentation>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:enumeration value="Electronic"/>
    <xs:enumeration value="Barcoded"/>
    <xs:enumeration value="Labeled"/>
    <xs:enumeration value="OfficialMessage"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 14-4: FilePlan.ConfigureExchange method

```
/// <summary>
/// Configures an existing Exchange 2007 server with the
/// managed folders and content settings needed to implement
/// this file plan for a given records center.
/// </summary>
public bool ConfigureExchange(string recordsCenterEmailAddress)
{
    try
    {
        foreach (RecordSpecification record in this.Records)
            if (record.Type == RecordType.OfficialMessage)
            {
                FilePlan.SetupExchangeFolder(record.SafeName, record.Name,
                    record.SafeName, recordsCenterEmailAddress);
            }
        return true;
    }
    catch (Exception x)
    {
        Helpers.HandleException(this, x,
            "Failed to configure Exchange Server");
    }
    return false;
}
```

Creating a Mailbox Policy

Managed policies provide a way to create a logical grouping of managed folders that can be attached to a user's mailbox all at once. In fact, only one mailbox policy can be associated with any given mailbox, but there can be any number of managed folders associated with the policy. This gives administrators the flexibility they need to assign groups of folders to different users. In the context of MRM, it means that the policy is the primary tool that administrators have for controlling the flow of official e-mail records into the Repository.

To create a mailbox policy using the Exchange Management Console, perform the following steps:

1. Select "New Managed Folder Mailbox Policy" from the Mailbox node under the Organization Configuration node in the console tree, and enter the policy name into the "Managed folder mailbox policy name" field of the form. This will be the name you use to reference the policy in subsequent steps.
2. Add the managed folders you want to associate with the policy by clicking on the "Add" button in the "Specify the managed folders to link with this policy" section. This opens a dialog that lists the available managed folders to select from.
3. Click "New" and then "Finish" to close the form and create the policy.

To create a policy using the `ExchangeManagementShell` component, we can use code similar to the following:

```
string name="MRMPolicy";
string folders="OfficialMessages";
```

```

string command="New-ManagedFolderMailboxPolicy";
ExchangeManagementShell.Run(
    string.Format("{0} -Name {1} -ManagedFolderLinks \"{2}\"",
        command, name, folders)
);

```

Running the Managed Folder Assistant

After setting up the managed folders, content settings, and policies on the Exchange Server, we then have to schedule a job that will run at predetermined intervals to propagate the objects out to user mailboxes. This process is similar to scheduling SharePoint timer jobs. In Exchange 2007 parlance, it is known as the *Managed Folder Assistant*. We basically need to tell Exchange when to run the Managed Folder Assistant by specifying a schedule that will apply to the entire server.

To schedule the Managed Folder Assistant using the Exchange Management Console, perform the following steps:

1. Select “Mailbox” from the Server Configuration node of the console, and then open the “properties” dialog and select the Messaging Records Management tab.
2. Select “Use Custom Schedule” in the “Schedule the Managed Folder Assistant” section, and click on the Customize button to display the schedule dialog. Use the controls in the “Schedule” section to specify when you want the program to run.

Figure 14-5 shows an example of the completed dialog.

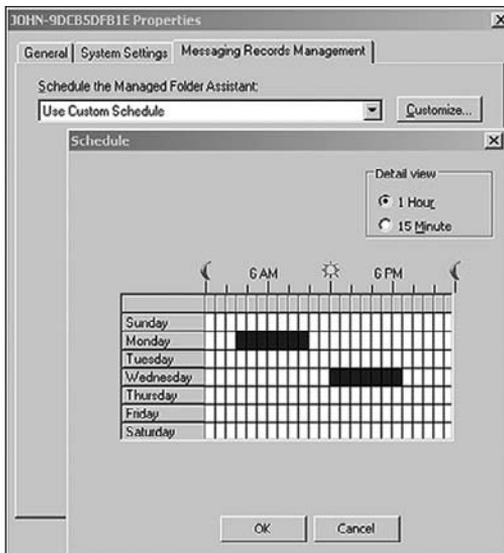


Figure 14-5: Scheduling the Managed Folder Assistant.

We don't really need to start the Managed Folder Assistant from code related to the `FilePlan` because it's more of an administrative procedure that should be left to the Exchange Administrator. In a typical

Chapter 14: Managing Electronic Mail Records

scenario, we will load some number of FilePlan definitions from disk and then process them to set up the appropriate Managed Folders, Managed Content Settings, and MRM Policies in Exchange. Then the Exchange Administrator will schedule or run the Managed Folder Assistant at some later time to propagate the objects out to users. However, during solution development, you can use the following command to start the Managed Folder Assistant immediately:

```
ExchangeManagementShell.Run("Start-ManagedFolderAssistant");
```

Handling the Folders in Outlook 2007

Once the Managed Folder Assistant runs, users will see the new managed folders in the Outlook client as shown in Figure 14-6. At this point, they can either drag-and-drop e-mail messages into the appropriate folder or set up Outlook rules to route incoming e-mails to the managed folders automatically. E-mails that are routed into managed folders are then processed according to the associated content settings, ultimately being forwarded to the Records Center within SharePoint for their final disposition.

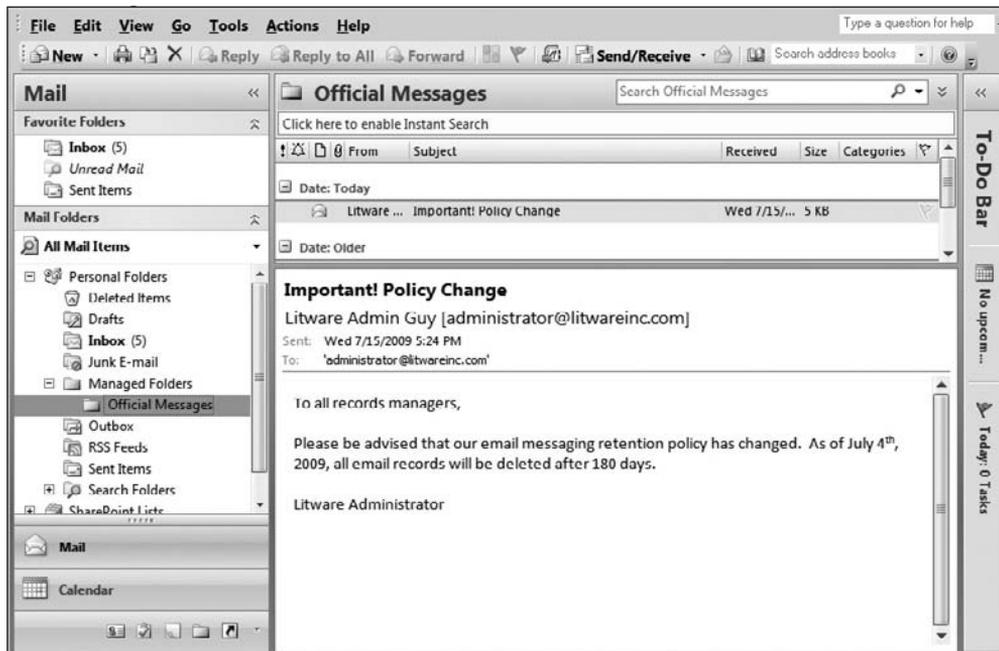


Figure 14-6: Managed folders in Outlook 2007.

Handling Missing Properties

The standard Records Center processing sequence ensures that all of the required properties are present for each incoming record by comparing the items in the incoming property array with the columns of the target document library as indicated by the routing type associated with the record. If any of the columns are marked as *required* and the matching property values are not present in the array, then the record is placed into the Holding Zone and the result details (a text string) are formatted to indicate that additional information is needed before the record can be processed.

For incoming e-mail records, the result details are routed back to the person who submitted the message, along with a link to a custom view that is generated by the Records Center for the purpose of collecting the required metadata. The record is not processed until the missing metadata is supplied.

The generated message is not sent to the submitter immediately, but is postponed until the next scheduled time slot for processing missing properties. This time slot is scheduled by the administrator in Central Admin. When the message is sent, it links back to an aggregated view that displays a grid that enables the submitter to enter metadata for all pending records at one time. Figure 14-7 shows an example of the generated form.

The screenshot shows a web browser window with the address bar displaying "Litware Records > Missing Properties > (no title) > Edit Item". The page title is "Missing Properties: (no title)". Below the title, there are "OK" and "Cancel" buttons. A toolbar contains a "Delete Item" checkbox and a note "* indicates a required field". The form has three main sections: "Required Data *" with a text input field, "Starting Date *" with a date picker, and "Ending Date *" with a date picker. At the bottom, it shows "Created at 7/15/2009 11:04 PM by System Account" and "Last modified at 7/15/2009 11:04 PM by System Account", along with "OK" and "Cancel" buttons.

Figure 14-7: Generated form for missing properties.

Summary

This chapter provided an overview of the available tools for managing official electronic mail records using Office SharePoint Server 2007, Outlook 2007, and Exchange 2007, including setting up a Records Center to receive electronic mail and using Exchange Managed Folders to facilitate centralized routing of e-mail messages to the Records Center site.

We explored the Exchange 2007 components and tools for creating the necessary components as well as options for configuring them programmatically. We also showed how to extend the Dynamic File Plan component from Chapter 5 so that it can be used to control the automatic configuration of Exchange using the same data files we used earlier to configure the Records Center site.

15

Using Workflow to Manage Records

One thing that is striking about enterprise content management in general is the dynamic interplay that exists between content and business processes. This should not be surprising, since all business processes depend on information in some form. However, understanding the converse relationship — how content depends on business processes — requires a different level of examination and ultimately may lead to the construction of different kinds of tools.

Business processes certainly produce as well as consume information. Indeed, the very purpose of many business processes, particularly those focused on document creation, is to do precisely that. But there are other processes that may produce or modify content simply as by-products of their primary function, and those by-products feed into yet other processes in a semi-coordinated dance that hopefully brings us closer to the holy grail of “increased productivity.”

Consider the lowly expense report — clearly one of the most underappreciated mainstays of the business report pantheon. Whether filling it out directly or indirectly through a form, each row in the report might spawn any number of separate and distinct business processes. There could be a line-item approval workflow that is conditionally invoked for any expense item with an amount greater than a specified threshold value. Even the determination of that value might be the result of a separate business process run periodically to assess the overall financial health of the organization. There could be an aggregate approval workflow that determines whether an expense report qualifies for special examination based on factors having nothing to do with the report itself, but instead may be driven by things like the frequency of unapproved items submitted by a particular user. As shown in Figure 15-1, this might, in turn, result in the generation of an employee evaluation document and the creation of a related set of tasks to be performed by a manager or HR personnel.

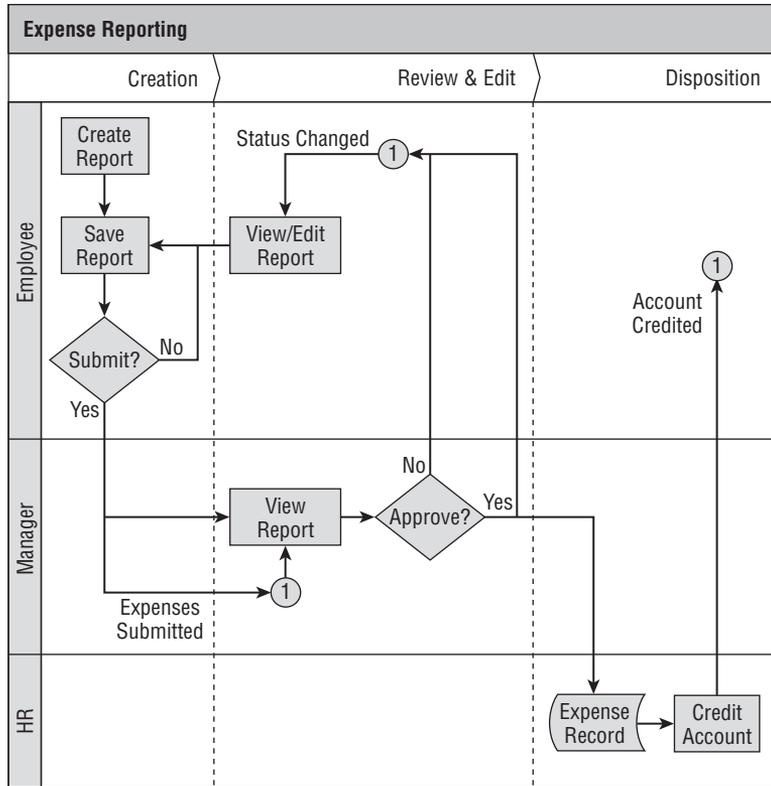


Figure 15-1: Expense reporting workflow.

True enterprise content management must account for the way in which a particular content element will be used. If workflow and Enterprise Content Management (ECM) are to deliver on the promise of increasing day-to-day productivity, then business analysts must be able to describe business processes in the language of their business and in terms of the flow of content into and out of the system as a whole. In the context of workflow development, this means having a richer set of workflow activities to work with. It is much easier to describe a business process that involves the generation of an audit report using a `GenerateAuditReport` activity than to spell out the individual steps needed to produce one.

The ECM team at Microsoft released an interesting set of tools at the beginning of the product cycle called the "ECM Starter Kit." It included a variety of code samples intended to highlight the ECM features of the MOSS 2007 product. The samples included code for document management and content processing, records management, information policy, and workflow. The workflow samples were particularly interesting because they included a special library called *ECMActivities*. This library contained several custom workflow activities that purported to interact with the enterprise content management features of SharePoint in such a way that they could be used in other workflows. It is not clear whether the term *other workflows* was intended to include *non-SharePoint workflows*, but the overall concept resonated on an intuitive level. If the idea was to promote the development of high-level ECM workflow activities so they could be plugged into larger business processes without having to deal with the lower-level activities provided by the built-in SharePoint workflow activities, then it seemed like a good

idea. Unfortunately, the ECM Starter Kit does not seem to have gained much traction during the year or so that followed its introduction. Perhaps it came too early in the product life cycle. Nevertheless, it still seems like a good idea, and luckily it is still included in the SharePoint Server 2007 SDK.

The purpose of this chapter is to explore the possibility of extending the set of workflow activities that are available out-of-the-box for building records management solutions. Part of this involves identifying general-purpose ECM workflow activities that can be applied to any SharePoint site, list, or list item. But it also includes identifying more specialized workflow activities that may be tied to specific business processes, and then developing a methodology for identifying and building them.

As part of this exploration, we will start by examining the ECM workflow activities that were included in the original ECM Starter Kit and then look for ways to improve and possibly generalize them even further. Following the lead of the ECM team at Microsoft, we will then attempt to identify and develop some additional activities that were not part of the original set, but that add value to any SharePoint ECM solution. Finally, we will focus on a few activities that are specific to some of the records management code already developed in previous chapters, such as the validation framework for content validation and the file plan schema.

A SharePoint Workflow Primer

What is a workflow? Think of it as a long-running process that has built-in support for persistence. *Long-running* means that it functions like any other program but can be suspended or resumed automatically by the Workflow Foundation run time. The way that the Workflow Foundation architecture allows us to build on that notion is by breaking the overall process down into discrete activities. These are the atomic operations that exist for any given application domain. What's unique about the Workflow Foundation approach is that activities may contain other activities. There is the ability to compose new activities from existing ones that is an essential ingredient for creating a set of tools that we can build on and continue to add value to over time.

There are two basic types of workflow programs. One is the sequential workflow, like the one shown in Figure 15-2, which is modeled like a flowchart where you have a beginning and an end state with branches in between.

There is also the state machine workflow, where you have many state transitions that can occur at any time. With sequential workflows, the flow of control moves in one direction through the logic, whereas state machine workflows follow more of an event-driven model. As it turns out, the state machine model is a more natural way to describe human interactions than the sequential workflow. Humans tend to operate in interrupt-driven fashion when processing information in response to stimuli that come from all directions all the time. It's no surprise, then, that the state machine model works well for describing human-based workflows. Figure 15-3 shows a state machine for managing task reminders.

Workflow activities are .NET classes that are written in managed code (VB .NET or C#) and then compiled into assemblies called *workflow activity libraries*. Within those classes, certain properties and methods are implemented, which are exposed in the same way they would be for normal classes except that additional wrappers are added so they can participate in the workflow process. When those properties and methods are exposed, the workflow run time can use *.NET Reflection* to find those properties and methods and then bind them to other activities dynamically as the workflow runs.

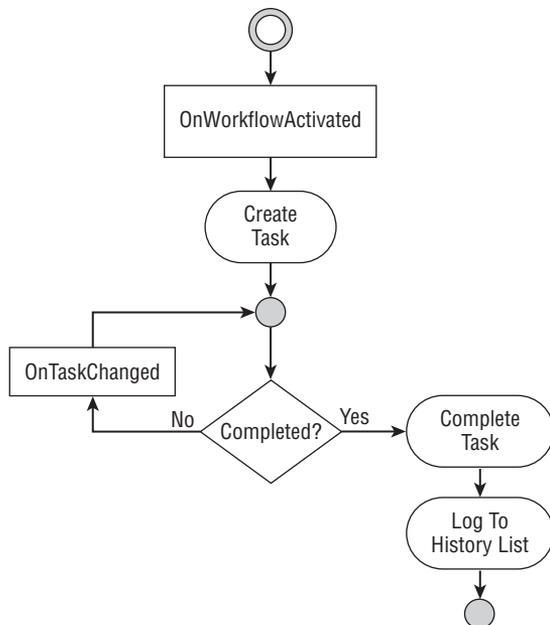


Figure 15-2: A sequential workflow.

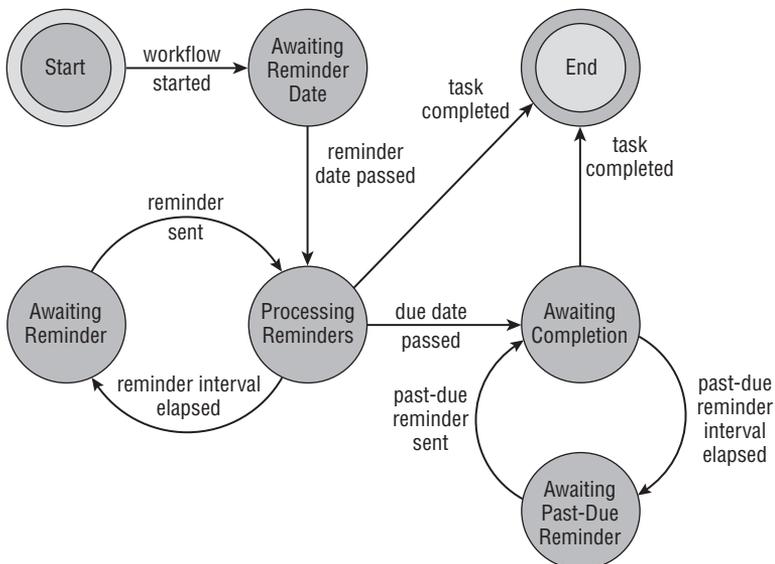


Figure 15-3: State machine workflow for managing tasks.

.NET Reflection is an essential ingredient for the success of the workflow run time. Without the ability to reflect dynamically over the assembly to figure out how to connect the published workflow properties and methods to other activities, it would not be possible to write workflow programs as a collection of activities wired together. Instead, it would be necessary to write each workflow program as a monolithic implementation of a particular set of requirements. This was a problem that many developers faced when trying to build workflows prior to the emergence of the Workflow Foundation architecture. Part of the problem was that it was very hard to model the workflow generically and yet stay focused on the problem at hand. There was always the need to work on the entire workflow program at one time. With the emergence of the activity library, it is now possible to break off a piece of the functionality and focus on it independently. This makes it possible to design and build workflow components. Our goal here is to extend this notion to include the construction of ECM workflow components.

Workflow activities also expose interfaces that can be invoked by other activities. This allows the workflow run time to establish and enforce contracts between activities. Just like other .NET classes, workflow activities can include an event-firing mechanism. So it is possible to write workflow activities that handle events that are generated by the workflow run time and that fire events that enable other activities to respond to them.

This workflow architecture and the tools provided to construct workflows are easy to understand at the conceptual level. In practice, there is a lot of detail to master. One of the many tasks that face new workflow developers is becoming familiar with the tools that are available in the platform. There are many activities that are provided with the Workflow Foundation, and these are the activities that all other workflows are built on. Starting with the simple code activity, there are simple and complex activities that every developer must learn. There is also the fundamental notion of a *composite* activity that is defined in terms of other activities contained within it. For example, you could think of a sequential workflow as being a composite activity that imposes an order of execution for the child activities contained within it.

In addition to the fundamental activities, there are SharePoint-specific activities, which are distributed in a custom activity library called the *Microsoft.SharePoint.WorkflowActions.dll*, which has been written in conjunction with a specialized workflow layer built into the SharePoint product itself.

It was quite fortuitous that the SharePoint Server 2007 product and the Workflow Foundation were being built at the same time. This allowed them to take advantage of each other's expertise. While the Workflow Foundation team was trying to design an architecture that would be strong enough, yet simple and elegant enough to support sophisticated applications like SharePoint, the SharePoint team was focused on extending the core concepts of the Workflow Foundation to ensure that they could meet all of their requirements. The result is a truly elegant and extensible architecture.

As MOSS was being built, the Workflow Foundation team and the MOSS team collaborated to come up with a core set of activities, but decided to restrict what SharePoint developers could do in several areas. One such restriction limits the ability to define custom interfaces and then load them into the workflow run time as custom *services* that can be discovered by running workflow activities and executing them. This is one of the most powerful features of the Workflow Foundation architecture because it allows you to create custom libraries of shared services that can be called from many different workflow activities. The SharePoint implementation of the workflow run time disallows this capability. We could have used this in lots of ways. For example, we could have used it to construct a custom report generation framework that exposes a standard set of interfaces that could be consumed by any workflow activity. Unfortunately, it is not possible with the current product. It may be possible in the future, but it introduces many complexities that the Workflow Foundation team and SharePoint group decided not to support at this time.

Official Records, Workflow, and Complexity

Imagine a long document of, say, 250 pages. Within that document there are different sections, and within those sections there are paragraphs, sentences, and words. For documents that drive business processes, the same content may be viewed by people who must fulfill different responsibilities depending on the roles they must play in relation to the business process. For any given role, the document might be presented in different ways, perhaps hiding irrelevant sections, highlighting important sentences or words, or organizing the content in a different order.

Sitting *behind* the document, then, is a set of rules that collectively determine how the document should be viewed or routed. These rules make up the intelligence that moves along with the document and provides a particular implementation of the business process.

For single documents, the implementation can be provided by a set of macros or by code in a .NET assembly attached to the document. Adding even a second document to the process can dramatically increase the complexity involved. As more and more documents are added to the business process, then the aggregated rules become increasingly complex, more metadata is needed to capture the rules, and the entire process becomes more difficult to manage. The ability of a given system to deal with this kind of content-driven complexity will depend on its flexibility and scalability, but also on the ease with which business analysts can extend the process model to deal with the new abstractions being introduced.

The complexity of a simple business process involving a single content type could be modeled in terms of the kind of content it is. For example, a text file could be defined as a primitive content component with a complexity of 1. Additional metrics could be defined that measure the “usefulness” of the content, based on the number of roles or responsibilities it supports either as an input or as an output. This could be derived from the role/activity modeling exercise introduced in Chapter 1. The kinds of questions to ask in coming up with a measure of content complexity might include:

- How many ways can the content element be produced?
- Does it require creative input?
- Is it produced as an artifact of some other process?
- Does it require approval?

Consider the typical approval process, wherein there is a decision point associated with a content element. If that decision point is embedded within a flowchart, then the *value* of that content is derived in part from where it appears in the flowchart, and its complexity might therefore be calculated based on the number of subsequent decisions that depend on it. Similarly, we might measure the impact of a wrong decision on the enterprise as a whole and then use that measurement to assign a value or cost to the content element.

Using these kinds of analytical methods, we might develop a *content complexity matrix* based on standard types of documents that are currently in use throughout an organization. For example, we might measure the “cost” of a particular type of report not reaching all of its intended recipients in a timely fashion. Time sheets, sales reports, expense reports, and the like could all be analyzed from this perspective to come up with a complexity matrix for a given workflow based on the kinds of content elements it contains.

The problem with this approach is that the analysis is often too expensive. Although everyone may recognize the value of content reengineering, few would be willing to take the entire system off-line in order to execute it. Since most of the value would be derived from the ability to match a given content element to a particular business process, there is little opportunity to identify (and too little time to implement) specialized tools that might help simplify the analysis. In other words, the hard part of the analysis would already be completed by the time the content is available for further analysis. The content is seen therefore as a by-product of the business process rather than an enabler for achieving greater process efficiency.

However, an interim approach is available using workflow activities. They provide an additional layer within which to maneuver, so that instead of focusing on the complexity of the content itself, we can focus on the workflow patterns that govern its use. Using this approach, we can identify three layers of complexity associated with workflow activities in the records management arena:

1. **Category 1** — Primitive activities
2. **Category 2** — Generic records management activities
3. **Category 3** — Domain-specific activities

Primitive Activities

Primitive activities include the basic set of workflow activities provided by the Workflow Foundation, as well as the SharePoint-specific workflow activities provided by MOSS. The following table lists the primitive activities and their descriptions:

Activity	Description
<code>CallExternalMethod</code>	Works in conjunction with the <code>HandleExternalEvent</code> activity to invoke a local method on a local service.
<code>CancellationHandler</code>	Cleans up a composite activity if part of the activity cancels before its children are done executing.
<code>Code</code>	Allows you to write custom code as part of a workflow.
<code>Compensate</code>	Allows you to compensate for failed transactions that have already committed.
<code>CompensationHandler</code>	Works in conjunction with the <code>Compensate</code> activity to execute a group of activities when a transaction fails after having been committed.
<code>CompensatableSequence</code>	Implements the <code>ICompensatableActivity</code> interface to provide a compensatable version of the <code>Sequence</code> activity.
<code>CompensatableTransactionScope</code>	Implements the <code>ICompensatableActivity</code> interface to provide a compensatable version of the <code>TransactionScope</code> activity.

Continued

Chapter 15: Using Workflow to Manage Records

Activity	Description
ConditionedActivityGroup	Executes child activities based on a set of conditions or code such as a <code>WhenCondition</code> or <code>UntilCondition</code> .
Delay	Works from within a <code>Listen</code> activity to specify a period of time to delay.
EventDriven	Executes child activities in response to a specific event.
EventHandlers	Enables the connection of events to activities.
EventHandlingScope	Allows you to run the main child activity concurrently with <code>EventHandlers</code> activities while they are listening for events.
FaultHandler	Works like a <i>catch</i> statement to handle a specified fault type.
HandleExternalEvent	Waits for an external event to occur, blocking execution of the workflow.
IfElse	Works like an <code>If Else</code> to specify a condition to determine which branch to follow.
IfElseBranch	Contains the activities for <code>IfElse</code> branches.
InvokeWebService	Invokes a Web Service.
InvokeWorkflow	Invokes another workflow.
Listen	Contains multiple branches of event-driven activities. Can only be used in sequential workflows.
Parallel	Contains two or more <code>Sequence</code> activities that execute at the same time.
Policy	Implements a forward-chaining rules engine for specifying business logic separate from the workflow.
Replicator	Creates multiple instances of an activity at run time, allowing you to specify an indeterminate number of times to execute a given activity.
Sequence	Allows you to execute a collection of activities in a pre-defined sequence.
SetState	Allows you to transition between states in a state machine workflow.
State	Represents a state in a state machine workflow.
StateInitialization	Contains child activities that are executed when a state is transitioned to.

Activity	Description
StateFinalization	Contains child activities that are executed when a state is transitioned out of.
Suspend	Suspends the workflow.
SynchronizationScope	Allows you to serialize access to critical code segments similar to thread semantics in programming languages.
Terminate	Immediately terminates the workflow with no recourse.
Throw	Throws an exception that can be handled by a <code>Fault</code> activity.
TransactionScope	Used to wrap transactions, allowing you to roll back in case of failure. Cannot be nested in another <code>TransactionScope</code> activity.
WebServiceFault	Allows you to raise a SOAP exception.
WebServiceInput	Allows you to receive data from a Web Service.
WebServiceOutput	Used to respond to a Web Service request
While	Similar to a <code>While</code> loop that continues until a given condition is met.

The next table shows the set of workflow activities that are specific to SharePoint and are implemented in the `Microsoft.SharePoint.WorkflowActions` assembly.

Activity	Description
CompleteTask	Marks a task as completed.
CreateTask	Creates a task with the specified property values.
CreateTaskWithContentType	Creates a task item using a specified content type.
DeleteTask	Deletes a task item.
EnableWorkflowModification	Enables a workflow modification form so that the modification can be listed as an available workflow modification in the user interface (UI).
InitializeWorkflow	Initializes a new instance of the workflow.
LogToHistoryList	Logs information about the execution of a workflow to the workflow history list.
OnTaskChanged	Responds to changes to a task associated with the workflow.

Continued

Chapter 15: Using Workflow to Manage Records

Activity	Description
OnTaskCreated	Responds to the creation of a task associated with the workflow.
OnTaskDeleted	Responds to the deletion of a task associated with the workflow.
OnWorkflowActivated	Responds to the initiation of a new workflow for a list item.
OnWorkflowItemChanged	Responds to changes in the item associated with the workflow.
OnWorkflowItemDeleted	Responds to the deletion of a workflow item.
OnWorkflowModified	Responds to the submission of a workflow modification form.
RollbackTask	Rolls a workflow task back to its last accepted state.
SendEmail	Creates and sends an e-mail message to a specified list of users.
SetState	Sets the status text for the workflow that is displayed in the UI.
SharePointSequentialWorkflowActivity	Executes a sequence of SharePoint workflow activities.

Useful Activities for Records Management

The availability of generic enterprise content management activities makes it easier to incorporate records management functionality into workflow programs. The following table lists the generic ECM activities that are included in the Microsoft SharePoint ECM Starter Kit:

Activity	Description
WssTask	Creates a task, waits for it to be changed, then completes the task and handles task deletion.
ApplyHold	Puts an item in a Records Center on hold.
SendToRecordsRepository	Sends an item to the Records Center associated with the site.

In addition to these, the general-purpose ECM activities listed in the following table can be built using some of the code developed in the preceding chapters of this book.

Activity	Description
RemoveFromHold	Removes an item in a Records Center from a hold.
ExecuteFilePlan	Executes a dynamic file plan for a Records Center site. The file plan can be supplied by the workflow or loaded from the File Plan gallery.
SetExpirationPolicy	Configures the expiration settings for a new or existing information policy.

Activity	Description
EnableBarcodePolicy	Configures the barcode settings for a new or existing information policy.
EnableLabelPolicy	Configures the labeling settings for a new or existing information policy.
GenerateAuditReport	Generates a custom audit report for a list item and places the report into a specified document library.
ValidateListItem	Uses the <code>ListItemValidator</code> to apply custom validation rules to a list item. The validation rules can be supplied by the workflow or can be loaded from a separate list.

Domain-Specific Records Management Activities

To further simplify the creation of workflows that support records management, domain-specific activities can be derived from a detailed analysis of the workflow associated with a particular regulation or business rule that is driving the need for records management. The overriding goal here is to identify high-value targets for automating a complex business process that is required in order to manage records effectively for a given regulatory compliance scenario. Examples might include the generation of a report from data provided during the workflow or the conditional execution of a workflow based on data gathered in a document that has now been published or approved. The following table lists a few examples:

Regulation	Activity	Description
HIPAA	ValidatePHIContent	Checks whether an item contains <i>Protected Health Information</i> (PHI) that might require special processing.
SOX	UpdateCEOCertification, UpdateCFOCertification	SOX compliance requires a periodic certification by the CEO and CFO that disclosure controls and procedures are in place. These activities could be used to enforce this requirement by creating the appropriate tasks or generating notifications and reminders for the responsible users.

Workflow Modeling

The easiest way to model a workflow is to start with a Visio diagram. While this works great for trivial workflows, once you get to a certain level of complexity, a lot more effort is involved. Another way to think about this is that sequential modeling is good for “primitive” or atomic operations, but is not as good for describing more complex interactions between activities. As the number of cross-connections increases, so does the overall complexity of the model. Consequently, its value — both in terms of its ability to convey the logical flow and its ability to control that flow — is reduced.

Chapter 15: Using Workflow to Manage Records

As mentioned earlier, state machine workflows are a more natural way to model how humans interact. SharePoint workflows, and ECM workflows in particular, are basically describing human interactions, so state machine workflow modeling offers a lot of promise. Still, there are well-known procedures such as approval or task modification that are good candidates for sequential workflows. Ultimately, ECM solutions will include a mixture of both styles. The choice of a modeling template can also be influenced by the complexity metrics introduced previously, and as shown in the following table:

Complexity	Dominant Characteristic	Modeling Template
Category 1 — Primitive	Sequential	Flowchart
Category 2 — Generic RM	Event-driven	Dataflow diagram
Category 3 — Domain-specific	Role-driven	Cross-functional flowchart/dataflow

Although Visual Studio provides a great workflow development environment, it may not be the best choice for initially modeling the workflow because developers tend to get sucked into “development mode” and lose focus once they are back in the Integrated Development Environment (IDE). It is so easy to start prototyping different ideas and dropping in pieces of code, and it is easy to get carried away with the details of a particular property or data structure. I prefer to start with Visio because I can’t write code in Visio. So the focus stays on design mode. Only after I’ve found the right level of granularity and I’ve shared it with the client and the client agrees do I then go into Visual Studio to implement the model.

Visio is also a good tool for converting role/activity modeling artifacts into a workflow design because the same terminology that the stakeholders have used during the discovery process can also be used to describe the content elements being consumed and produced by the workflow. Since it is a role-driven methodology, it is easy to create a cross-functional flowchart or a cross-functional dataflow diagram and present that to the stakeholders as an additional tool to make sure the workflow design accurately addresses their requirements. Microsoft Visio plus Microsoft Excel provides all the tools needed to determine what content will be created as by-products of workflow activities and how they will feed into other workflow state transitions. This is also where the high-value targets for building domain-specific ECM activities will likely surface.

After the high-value ECM activity targets have been identified, we can focus on the related content elements and either extend them using strongly typed XML schemas or simply use them as a reference to build domain-specific components and workflow activities. As part of this component-building exercise, we can also include additional support for workflow activities by including events that the workflow activities can handle or by exposing interfaces that they can more easily consume.

Finally, we want to expose the records management workflow activities to SharePoint designer so they are available to declarative workflows and ultimately reduce the overall cost of building domain-specific workflows. This process will yield assets that can be reused throughout the organization for any workflow scenario that involves ECM.

Designing workflow activities that work well as declarative “no-code” workflows requires some planning and optimization. Many of the out-of-the-box activities included with SharePoint were designed especially for use in SharePoint Designer and are not supported in the Visual Studio IDE.

Building a Workflow Activity Library

So what does it take to build a custom workflow activity library? The first thing to do is create a separate assembly for the workflow activities using the Workflow Activity Library project template in Visual Studio. This is important primarily because we don't want to complicate the deployment process by introducing code that might not fit within the security constraints needed for deploying the workflow assembly. It also maintains a clean separation between the workflow and non-workflow-related aspects of the component so that it can also be referenced easily from non-workflow code. The Workflow Activity Library project template is located in the New Project dialog under the Workflow node of the language you are using.

The activity library is automatically set up with the appropriate references for building a standard workflow. For SharePoint workflow activities, we must also add references to the `Microsoft.SharePoint` and `Microsoft.SharePoint.WorkflowActions` assemblies.

The Visual Studio Workflow Activity Library project template also extends the user interface so that we can add activities easily to the project. By right-clicking on the project node, we get a convenient context menu command for creating a new activity, as shown in Figure 15-4.

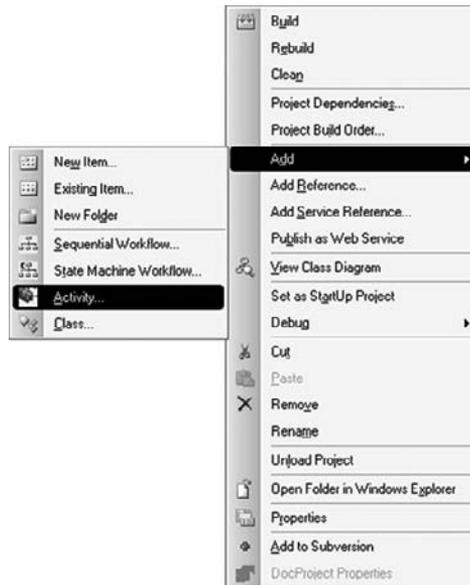


Figure 15-4: Context menu command for creating an activity in Visual Studio.

After creating the activity and giving it a name, the first thing to do is close the workflow design surface and then open the code-behind file by right-clicking on the component file in Solution Explorer and selecting "View Code." Now add using statements for the namespaces needed to code the activity.

```
using Microsoft.SharePoint;
using Microsoft.SharePoint.Workflow;
using Microsoft.SharePoint.WorkflowActions;
```

Chapter 15: Using Workflow to Manage Records

When adding a new activity module to a project, the project template automatically inherits from `SequenceActivity`. In most cases, we should change this to `Activity` unless there is a need to execute child activities in a predefined order. For the ECM workflow activities, a simple `Activity` derivation is all that is needed. The final steps are to add the dependency properties so that the workflow developer can provide the information needed to execute the activity and override the `Execute` method.

With Visual Studio 2008, it's very simple to create a dependency property. Dependency properties are like wrappers around the actual properties or data members that contain the information passed between the workflow activity and the workflow run time. Enabling this requires that each dependency property is marked up with the appropriate attributes so that the workflow run time can interact with them. This can involve a lot of keystrokes that basically just reference the actual property. To make this easier to set up, Microsoft has provided a set of code snippets (which you can access by typing [Ctrl]+K followed by [Ctrl]+X) that insert the appropriate information in a single step, as shown in Figure 15-5.

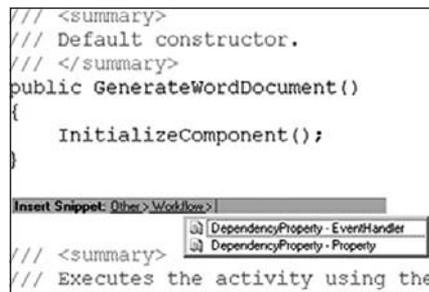


Figure 15-5: Dependency property code snippet.

Finally, after building the activity library, the activities must be added to the toolbox so that workflow developers can use them. This is a simple matter of right-clicking anywhere in the control toolbox, selecting “Choose items,” and then browsing to the assembly and selecting the activities to add. At that point, the activity can be dragged onto the design surface and configured as part of the workflow.

If you are building a workflow as part of the same Visual Studio solution, then the new workflow activity should appear in the toolbox automatically.

Testing Your Activities

To test the new workflow activities, create a new workflow project using the SharePoint 2007 Sequential Workflow project template as shown in Figure 15-6.

Using the SharePoint Workflow Project Wizard, enter the name of the workflow and the address of a SharePoint site to use for testing. This step allows Visual Studio to automatically configure the site by associating the workflow template with a specific list, task list, and workflow history list. It also generates and installs a workflow feature that makes the workflow template available to be executed. Figure 15-7 shows the Wizard dialog as it appears in Visual Studio.

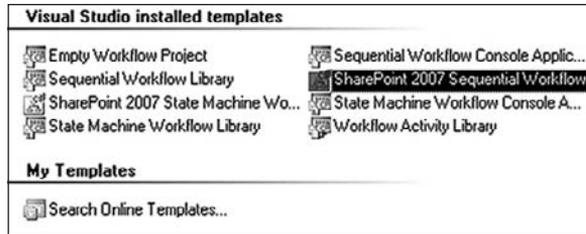


Figure 15-6: Creating a new SharePoint Sequential Workflow project.

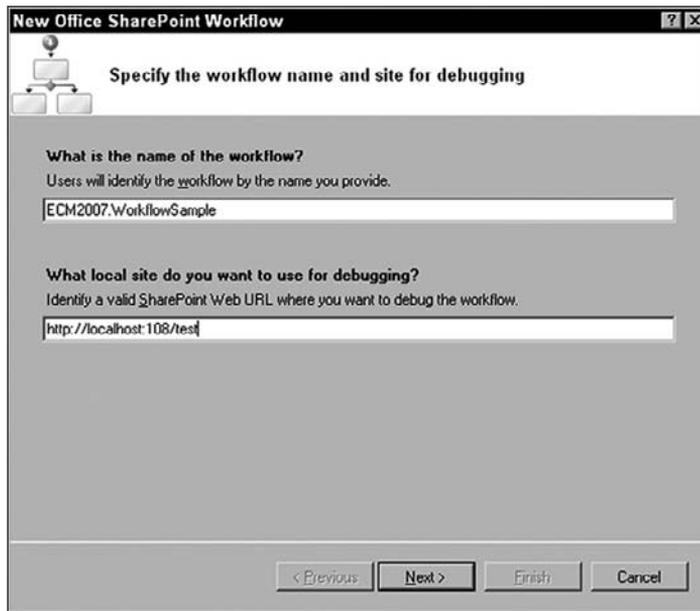


Figure 15-7: Setting up the workflow project for debugging.

Next comes a dialog for selecting which lists the workflow should use, as shown in Figure 15-8, and a final dialog for specifying how the workflow should start, as shown in Figure 15-9.

After everything is done, the project folder will contain a `feature.xml` file and a `workflow.xml` file that are used to install the template into SharePoint when [F5] is pressed. There is no need to add an explicit reference to the custom workflow activity library, since that will be handled when the activity is dragged onto the design canvas from the toolbox.

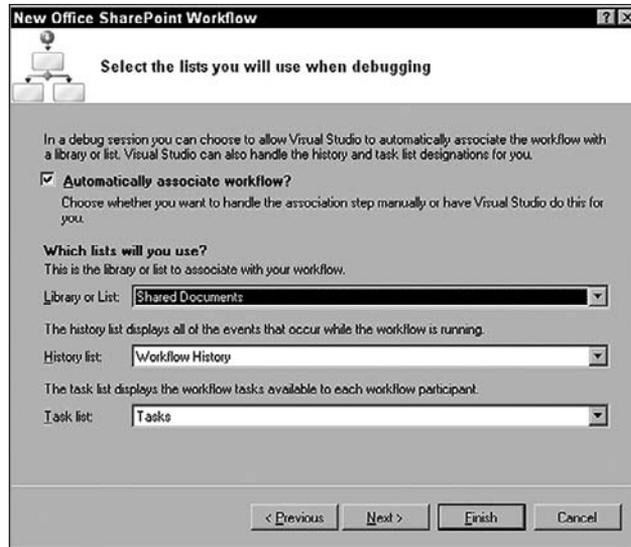


Figure 15-8: Specifying the lists to be used by the workflow.



Figure 15-9: Setting up the workflow instantiation properties.

Building Workflow Activities for Records Management

The following sections step through the creation of general-purpose ECM workflow activities that can be used to build real solutions, as illustrated in Figure 15-10. The first uses the File Plan Schema API developed in Chapter 5 to execute a file plan from data in an XML file attached to a list item in a form library. The second activity uses the list item validator developed in Chapter 13 to validate the metadata contained in a list item. The workflow developer provides the validation rules either directly as an XML string or indirectly as another list item containing an InfoPath form. The validation rules are processed and events are fired, allowing the workflow developer to handle error conditions.

Start by creating a new workflow activity library project in Visual Studio called *ECM2007.WorkflowActivities*. This project will contain each of the activities shown in the sections that follow. After creating the project, delete the starter activity called *Activity1.cs*, and then add references to the `Microsoft.SharePoint` and `Microsoft.SharePoint.WorkflowActions` assemblies. You'll also need to add references to the ECM2007 component library because the activities will call out to components that are declared within that assembly.

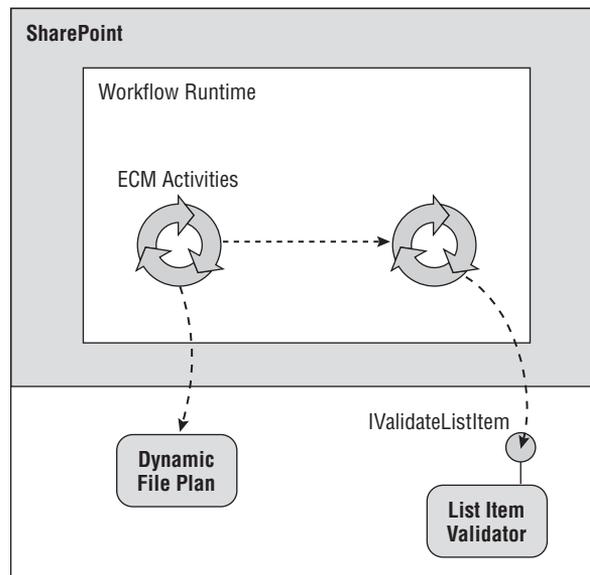


Figure 15-10: Custom records management workflow activities.

To add each of the activities described below, you will right-click on the project node and then select the Add ⇨ Activity command from the context menu. This will bring up the Add New Item dialog with the Workflow node pre-selected. Choose the Activity item template from the “Visual Studio installed templates” section. This will add the new activity with a design surface and a code-beside file.

Do not choose the “Activity (with code separation)” item template, as this would create a separate XAML file for the activity definition.

Executing a File Plan

Listing 15-1 shows a workflow activity that executes a dynamic file plan using the contents of an XML data file stored in the list item associated with the workflow. The workflow developer provides the identifier of a list item that contains the file plan and the URL of the Records Center site to which the file plan should be applied. Optionally, the workflow developer can provide an `Invoke` event handler that is called prior to the file plan execution so that the dependency properties can be initialized.

Listing 15-1: File plan execution workflow activity

```
using System;
using System.ComponentModel;
using System.Drawing;
using System.Workflow.ComponentModel;
using System.Workflow.ComponentModel.Compiler;
using System.Workflow.ComponentModel.Design;
using ECM2007.RecordsManagement;
using Microsoft.SharePoint;
using Microsoft.SharePoint.Workflow;
using Microsoft.SharePoint.WorkflowActions;

namespace ECM2007.WorkflowActivities
{
    [ActivityToolboxDisplayAttribute("ExecuteFilePlan", true)]
    [ToolboxItem(typeof(ActivityToolboxItem))]
    [ToolboxBitmap(typeof(ExecuteFilePlan), "Resources.JFH.ico")]
    public partial class ExecuteFilePlan : Activity
    {
        public ExecuteFilePlan()
        {
            InitializeComponent();
        }

        #region Dependency Properties
        public static DependencyProperty __ContextProperty = DependencyProperty.
            Register("__Context", typeof(WorkflowContext), typeof(ExecuteFilePlan));

        [DescriptionAttribute("__Context")]
        [CategoryAttribute("File Plan")]
        [BrowsableAttribute(true)]
        [ValidationOption(ValidationOption.Required)]
        [DesignerSerializationVisibilityAttribute(DesignerSerializationVisibility.
            Visible)]
        public WorkflowContext __Context
        {
            get
            {
                return ((WorkflowContext)(base.GetValue(ExecuteFilePlan.
                    __ContextProperty)));
            }
            set
            {
            }
        }
    }
}
```

```
        base.SetValue(ExecuteFilePlan.__ContextProperty, value);
    }
}
public static DependencyProperty ListIdProperty = DependencyProperty.
Register("ListId", typeof(string), typeof(ExecuteFilePlan));

[DescriptionAttribute("ListId")]
[CategoryAttribute("File Plan")]
[BrowsableAttribute(true)]
[ValidationOption(ValidationOption.Required)]
[DesignerSerializationVisibilityAttribute(DesignerSerializationVisibility.
Visible)]
public string ListId
{
    get
    {
        return ((string)(base.GetValue(ExecuteFilePlan.ListIdProperty)));
    }
    set
    {
        base.SetValue(ExecuteFilePlan.ListIdProperty, value);
    }
}
public static DependencyProperty ListItemProperty = DependencyProperty.
Register("ListItem", typeof(int), typeof(ExecuteFilePlan));

[DescriptionAttribute("ListItem")]
[CategoryAttribute("File Plan")]
[BrowsableAttribute(true)]
[ValidationOption(ValidationOption.Required)]
[DesignerSerializationVisibilityAttribute(DesignerSerializationVisibility.
Visible)]
public int ListItem
{
    get
    {
        return ((int)(base.GetValue(ExecuteFilePlan.ListItemProperty)));
    }
    set
    {
        base.SetValue(ExecuteFilePlan.ListItemProperty, value);
    }
}

public static DependencyProperty RecordsCenterUrlProperty =
DependencyProperty.Register("RecordsCenterUrl", typeof(string),
typeof(ExecuteFilePlan));

[DescriptionAttribute("RecordsCenterUrl")]
[CategoryAttribute("File Plan")]
[BrowsableAttribute(true)]
[DesignerSerializationVisibilityAttribute(DesignerSerializationVisibility.
Visible)]
public string RecordsCenterUrl
```

Continued

Listing 15-1: File plan execution workflow activity *(continued)*

```
{
    get
    {
        return ((string) (base.GetValue(ExecuteFilePlan.
            RecordsCenterUrlProperty)));
    }
    set
    {
        base.SetValue(ExecuteFilePlan.RecordsCenterUrlProperty, value);
    }
}

public static DependencyProperty InvokeEvent = DependencyProperty.
Register("Invoke", typeof(EventHandler), typeof(ExecuteFilePlan));

[DescriptionAttribute("Invoke")]
[CategoryAttribute("File Plan")]
[BrowsableAttribute(true)]
[DesignerSerializationVisibilityAttribute(DesignerSerializationVisibility.
Visible)]
public event EventHandler Invoke
{
    add
    {
        base.AddHandler(ExecuteFilePlan.InvokeEvent, value);
    }
    remove
    {
        base.RemoveHandler(ExecuteFilePlan.InvokeEvent, value);
    }
}
#endregion

/// <summary>
/// Logs an exception to the SharePoint workflow history list.
/// </summary>
internal static void LogExceptionToWorkflowHistory(Exception e,
    ActivityExecutionContext context, Guid workflowInstanceId)
{
    ISharePointService service = (ISharePointService)context.
    GetService(typeof(ISharePointService));
    if (service == null)
        throw new InvalidOperationException();
    service.LogToHistoryList(workflowInstanceId,
    SPWorkflowHistoryEventType.WorkflowError, 0,
    TimeSpan.MinValue, "Error", e.ToString(), "");
}

/// <summary>
/// Executes the workflow activity.
/// </summary>
```

```
protected override ActivityExecutionStatus
Execute(ActivityExecutionContext executionContext)
{
    try
    {
        // Raise invoke event to execute custom code in the workflow.
        this.RaiseEvent(ExecuteFilePlan.InvokeEvent, this,
            EventArgs.Empty);

        // Elevate privileges to run under the application pool identity.
        SPSecurity.RunWithElevatedPrivileges(delegate
        {
            // open the source website from the context info
            using (SPSite site = new SPSite(__Context.Web.Site.ID))
            using (SPWeb web = site.AllWebs[__Context.Web.ID])
            {
                // locate the list item that contains the file plan data
                SPList sourceList = web.Lists[new Guid(ListId)];
                SPLListItem sourceItem = sourceList.Items.
                    GetItemById(ListItem);

                // extract the file plan data from the item
                FilePlan filePlan = FilePlan.Load(sourceItem);

                // execute the file plan on the target site
                using (SPSite recordsCenterSite = new
                    SPSite(RecordsCenterUrl))
                using (SPWeb recordsCenter = recordsCenterSite.OpenWeb())
                    filePlan.ConfigureRecordCenter(recordsCenter);
            }
        });
    }
    catch (Exception ex)
    {
        LogExceptionToWorkflowHistory(ex, executionContext,
            WorkflowInstanceId);
        throw new Exception("File Plan execution failed", ex);
    }
    return base.Execute(executionContext);
}
}
```

Validating List Item Metadata

We can also create a workflow activity that leverages the content validation components we created as part of the ECM2007 foundation class library. Add another workflow activity called `ItemValidator` using the code shown in Listing 15-2. This workflow activity validates a list item by applying validation rules supplied as an XML string. The workflow developer provides the identifier of the list item to be validated and a string containing the validation rules. Optionally, the workflow developer can provide an `Invoke` event handler that is called before the validation is attempted, and a `Fail` event handler that is called if the validation operation fails.

Listing 15-2: List item validation workflow activity

```
using System;
using System.ComponentModel;
using System.Workflow.ComponentModel;
using System.Workflow.ComponentModel.Compiler;
using ECM2007.Validation;
using Microsoft.SharePoint;
using Microsoft.SharePoint.Workflow;
using Microsoft.SharePoint.WorkflowActions;

namespace ECM2007.WorkflowActivities
{
    [ActivityToolboxDisplayAttribute("ItemValidator", true)]
    [ToolboxItem(typeof(ActivityToolboxItem))]
    public partial class ItemValidator : Activity
    {
        public ItemValidator()
        {
            InitializeComponent();
        }

        #region Dependency Properties
        public static DependencyProperty __ContextProperty = DependencyProperty.
            Register("__Context", typeof(WorkflowContext), typeof(ItemValidator));

        [DescriptionAttribute("__Context")]
        [CategoryAttribute("Item Validator")]
        [BrowsableAttribute(true)]
        [ValidationOption(ValidationOption.Required)]
        [DesignerSerializationVisibilityAttribute(DesignerSerializationVisibility.
            Visible)]
        public WorkflowContext __Context
        {
            get
            {
                return ((WorkflowContext)(base.GetValue(ItemValidator.
                    __ContextProperty)));
            }
            set
            {
                base.SetValue(ItemValidator.__ContextProperty, value);
            }
        }

        public static DependencyProperty ListIdProperty = DependencyProperty.
            Register("ListId", typeof(string), typeof(ItemValidator));

        [DescriptionAttribute("ListId")]
        [CategoryAttribute("Item Validator")]
        [BrowsableAttribute(true)]
        [ValidationOption(ValidationOption.Required)]
        [DesignerSerializationVisibilityAttribute(DesignerSerializationVisibility.
            Visible)]
        public string ListId
```

```
{
    get
    {
        return ((string)(base.GetValue(ItemValidator.ListIdProperty)));
    }
    set
    {
        base.SetValue(ItemValidator.ListIdProperty, value);
    }
}
public static DependencyProperty ListItemProperty = DependencyProperty.
Register("ListItem", typeof(int), typeof(ItemValidator));

[DescriptionAttribute("ListItem")]
[CategoryAttribute("Item Validator")]
[BrowsableAttribute(true)]
[ValidationOption(ValidationOption.Required)]
[DesignerSerializationVisibilityAttribute(DesignerSerializationVisibility.
Visible)]
public int ListItem
{
    get
    {
        return ((int)(base.GetValue(ItemValidator.ListItemProperty)));
    }
    set
    {
        base.SetValue(ItemValidator.ListItemProperty, value);
    }
}
public static DependencyProperty RulesProperty = DependencyProperty.
Register("Rules", typeof(string), typeof(ItemValidator));

[DescriptionAttribute("Rules")]
[CategoryAttribute("Item Validator")]
[BrowsableAttribute(true)]
[DesignerSerializationVisibilityAttribute(DesignerSerializationVisibility.
Visible)]
public string Rules
{
    get
    {
        return ((string)(base.GetValue(ItemValidator.RulesProperty)));
    }
    set
    {
        base.SetValue(ItemValidator.RulesProperty, value);
    }
}
public static DependencyProperty InvokeEvent = DependencyProperty.
Register("Invoke", typeof(EventHandler), typeof(ItemValidator));

[DescriptionAttribute("Invoke")]
[CategoryAttribute("Item Validator")]
```

Continued

Listing 15-2: List item validation workflow activity *(continued)*

```
[BrowsableAttribute(true)]
[DesignerSerializationVisibilityAttribute(DesignerSerializationVisibility.
Visible)]
public event EventHandler Invoke
{
    add
    {
        base.AddHandler(ItemValidator.InvokeEvent, value);
    }
    remove
    {
        base.RemoveHandler(ItemValidator.InvokeEvent, value);
    }
}

public static DependencyProperty FailEvent = DependencyProperty.
Register("Fail", typeof(EventHandler), typeof(ItemValidator));

[DescriptionAttribute("Fail")]
[CategoryAttribute("Fail Category")]
[BrowsableAttribute(true)]
[DesignerSerializationVisibilityAttribute(DesignerSerializationVisibility.
Visible)]
public event EventHandler Fail
{
    add
    {
        base.AddHandler(ItemValidator.FailEvent, value);
    }
    remove
    {
        base.RemoveHandler(ItemValidator.FailEvent, value);
    }
}
}
#endregion

/// <summary>
/// Logs an exception to the SharePoint workflow history list.
/// </summary>
internal static void LogExceptionToWorkflowHistory(System.Exception e,
ActivityExecutionContext context, Guid workflowInstanceId)
{
    ISharePointService service = (ISharePointService)context.
GetService(typeof(ISharePointService));
    if (service == null)
        throw new InvalidOperationException();
    service.LogToHistoryList(workflowInstanceId,
SPWorkflowHistoryEventType.WorkflowError, 0, TimeSpan.MinValue,
"Error", e.ToString(), "");
}
```

```

    }

    /// <summary>
    /// Executes the activity.
    /// </summary>
    protected override ActivityExecutionStatus Execute(
        ActivityExecutionContext executionContext)
    {
        try
        {
            // Raise Invoke event to execute custom code in the workflow.
            this.RaiseEvent(ItemValidator.InvokeEvent, this, EventArgs.Empty);

            // elevate privileges
            SPSecurity.RunWithElevatedPrivileges(delegate
            {
                // open target web from context info
                using (SPSite site = new SPSite(__Context.Web.Site.ID))
                using (SPWeb web = site.AllWebs[__Context.Web.ID])
                {
                    // locate the source list
                    SPList sourceList = web.Lists[new Guid(ListId)];

                    // locate the list item
                    SPListItem sourceItem = sourceList.Items.
                    GetItemById(ListItem);

                    // construct an item validator
                    ListItemValidator validator = ListItemValidator.
                    FromXml(Rules);
                    bool isValid = validator.Validate(sourceItem);

                    if (!isValid)
                    {
                        // notify the workflow that the validation failed
                        this.RaiseEvent(ItemValidator.FailEvent, this,
                        EventArgs.Empty);
                    }
                }
            });
        }
        catch (System.Exception e)
        {
            LogExceptionToWorkflowHistory(e, executionContext,
            WorkflowInstanceId);
            throw new System.Exception("Item validation failed", e);
        }
        finally
        {
        }
    }
    return base.Execute(executionContext);
}
}
}

```

Extending SharePoint Designer to Use Records Management Workflows

The Workflow Design Wizard that is built into SharePoint Designer 2007 can be a powerful tool for creating workflows in the field, provided that the collection of workflow activities is rich enough to describe the business processes needed by a particular organization. Otherwise, the granularity of the tool can be too fine-grained to be useful. Extending the set of available workflow activities to include both general-purpose and domain-specific ECM workflow activities can add significant value by enabling business analysts to harness the power of ECM for use in day-to-day operations.

Extending SharePoint Designer 2007 to use custom workflow involves adding a file to the 12\TEMPLATE\<locale id>\Workflow folder on the SharePoint Server. When SharePoint Designer loads or creates a workflow, any files in that folder with the extension *.ACTIONS* are downloaded and used to configure the workflow designer interface. The default file has the name *WSS.ACTIONS*, and the entries it contains describe the kinds of activities that can be used in a declarative workflow, along with detailed information about how to present the rules used to build conditions, specify actions, and interact with each activity. By providing a customized version of this file, we can completely customize the user interface presented in the workflow designer.

The steps to accomplish this for a given activity library are:

1. Deploy the activity library to the Global Assembly Cache (GAC).
2. Configure SharePoint to recognize the activity library.
3. Create the *.ACTIONS* file to be used by SharePoint Designer.

You can deploy the activity library to the Global Assembly Cache as you normally would, by calling the *GACUTIL.EXE* command-line utility from the *post-build* events in Visual Studio. Next, you must configure SharePoint to recognize the activity library.

Deploying and Registering the Activity Library

After building the activity library and copying it to the GAC, it must be registered as a trusted assembly in the *web.config* file associated with the web application. This is similar to configuring a web part as a safe control, but instead of adding an entry to the *<SafeControls>* section, you add an entry to the *<System.Workflow.ComponentModel.WorkflowCompiler>* section. To do this, edit the *web.config* file for the SharePoint web application and add an *<authorizedType>* element as follows:

```
<authorizedType Assembly="ECM2007.WorkflowActivities, Version=1.0.0.0,
Culture=neutral, PublicKeyToken=0b97b340d4a71524"
Namespace="ECM2007.Workflow" TypeName="*" Authorized="True" />
```

Creating the *.ACTIONS* File

The final step is to create the *.ACTIONS* file that describes the activity to SharePoint Designer. Since this is an XML file, it can be created using Visual Studio or any XML Editor. Unfortunately, the XML schema for this file is not included in the SharePoint SDK. Writing these files manually without some kind of IntelliSense can be quite difficult, so with the help of my colleague Scot Hillier, I created the


```

<xs:restriction base="xs:string">
  <xs:enumeration value="ChooseDocLibItem" />
  <xs:enumeration value="ChooseListItem" />
  <xs:enumeration value="CreateListItem" />
  <xs:enumeration value="Date" />
  <xs:enumeration value="Dropdown" />
  <xs:enumeration value="Email" />
  <xs:enumeration value="Integer" />
  <xs:enumeration value="FieldNames" />
  <xs:enumeration value="ListNames" />
  <xs:enumeration value="Operator" />
  <xs:enumeration value="ParameterNames" />
  <xs:enumeration value="Person" />
  <xs:enumeration value="SinglePerson" />
  <xs:enumeration value="StringBuilder" />
  <xs:enumeration value="Survey" />
  <xs:enumeration value="TextArea" />
  <xs:enumeration value="UpdateListItem" />
  <xs:enumeration value="WritableFieldNames" />
</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name="TypeFrom" type="xs:string" use="optional" />
<xs:attribute name="OperatorTypeFrom" use="optional">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="DropDownMenu" />
      <xs:enumeration value="FieldName" />
      <xs:enumeration value="Left" />
      <xs:enumeration value="Variable" />
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>
<xs:attribute name="Value" type="xs:string" use="optional" />
</xs:complexType>
<!-- ----->
<xs:complexType name="OptionDefinition">
  <xs:attribute name="Name" type="xs:string" use="required" />
  <xs:attribute name="Value" type="xs:string" use="required" />
  <xs:attribute name="TypeFilter" type="xs:string" use="optional" />
  <xs:attribute name="UnaryHides" type="xs:string" use="optional" />
</xs:complexType>
<xs:complexType name="ParametersDefinition">
  <xs:sequence>
    <xs:element name="Parameter" type="ParameterDefinition"
      maxOccurs="unbounded" />
  </xs:sequence>
</xs:complexType>
<xs:complexType name="ParameterDefinition">
  <xs:attribute name="Name" type="xs:string" use="required" />
  <xs:attribute name="Type" type="xs:string" use="required" />
  <xs:attribute name="Direction" use="required" />
  <xs:simpleType>

```

Continued

Listing 15-3: WSSACTIONS.XSD (continued)

```
<xs:restriction base="xs:string">
  <xs:enumeration value="In" />
  <xs:enumeration value="Optional" />
  <xs:enumeration value="Out" />
</xs:restriction>
</xs:simpleType>
</xs:attribute>
<xs:attribute name="InitialValue" type="xs:string" use="optional" />
</xs:complexType>
</xs:schema>
```

The .ACTIONS file describes the public properties exposed by the activity and tells SharePoint Designer how to map those properties into the rules that are displayed to the user. The following code shows a custom .ACTIONS file for the custom activity that validates a list item:

```
<?xml version="1.0" encoding="utf-8" ?>
<WorkflowInfo>
  <Actions Sequential="then" Parallel="and">
    <Action Name="Apply Validation Rules to Item"
      ClassName="ECM2007.WorkflowActivities.ItemValidator"
      Assembly="ECM2007.WorkflowActivities, Version=1.0.0.0,
      Culture=neutral,
      PublicKeyToken=eb8a6a1622425a15"
      AppliesTo="list"
      UsesCurrentItem="true"
      Category="Core Actions">
      <RuleDesigner Sentence="Validate using %1">
        <FieldBind Field="Rules" Text="these rules" Id="1" />
      </RuleDesigner>
      <Parameters>
        <Parameter Name="Rules" Type="System.String, mscorlib"
          Direction="In" />
        <Parameter Name="__Context" Type="Microsoft.SharePoint.
          WorkflowContext, Microsoft.SharePoint.WorkflowActions"
          Direction="In"/>
        <Parameter Name="__ListId" Type="System.String, mscorlib"
          Direction="In" />
        <Parameter Name="__ListItem" Type="System.Int32, mscorlib"
          Direction="In" />
      </Parameters>
    </Action>
  </Actions>
</WorkflowInfo>
```

The `Actions` tag tells SharePoint Designer what to display for each action in the set. Within that, the `Action` tag describes the individual action. The `Name` attribute is what gets displayed in the Designer. The `ClassName` and `Assembly` attributes are used in the generated XAML for the workflow. The interesting part is the way that the `RuleDesigner` and `Parameter` tags work. The `RuleDesigner` tag lets us set up a sentence that gets displayed in the Designer as the user builds the workflow. The `Sentence` attribute then allows us to bind to the activity properties and then substitute their values when the activity is executed.

The `FieldBind` tag includes a `Field` attribute that specifies a given property by name. This is a reference to a specific property exposed by the workflow activity. The `DesignerType` attribute specifies which kind of control to use to gather the data, and the `Id` property specifies the substitution ID to use when building the sentence. Finally, the `Parameter` tag specifies the activity side of the contract and is used when calling the `Execute` method of the activity on the server.

We can declare as many actions as we want in the file. A good rule of thumb is to use a separate `.ACTIONS` file for each logical group of custom activities we want to deploy. Once we've created the `.ACTIONS` file and copied it to the server, we can refresh the site in SharePoint Designer and the custom activity will now appear in the Workflow Designer as shown in Figure 15-11.

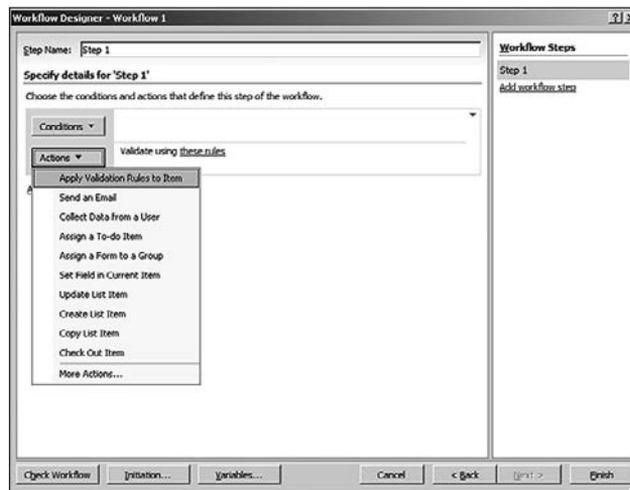


Figure 15-11: Custom activity in the Workflow Designer.

Summary

This chapter introduced SharePoint workflows and stepped through the process of creating custom workflows and workflow activities using the Visual Studio Workflow Designer, with the goal of providing a foundation for identifying and building specialized workflow activities that support the construction of records management and other enterprise content management solutions. We followed the approach taken by the Microsoft ECM team when they created the ECM Starter Kit, which included ECM-specific workflow activities, and then we identified records-management-specific activities based on code developed in previous chapters, such as the list item validator and the dynamic file plan component.

Workflow activities are a lot like web parts. The richer the collection of activities, the more powerful are the workflows that can be built. We looked at ways to leverage the power of Visual Studio to create specialized records management workflow activities and take advantage of the flexibility that is built into SharePoint Designer 2007 to create declarative workflows that require no coding. By extending SharePoint Designer to incorporate our records management workflow actions, we can expose a higher level of functionality to a wider audience of administrators, developers, and business analysts.

16

The DoD 5015.2 Add-On Pack

The Department of Defense (DoD) 5015.2 Add-On Pack was created in response to the fact that the out-of-the-box features of the default MOSS Records Center do not meet the requirements of the DoD 5015 standard. However, many of those requirements can be addressed by applying the typical techniques used to build any solution on the Windows SharePoint Services platform. Viewed in this light, the DoD 5015.2 Add-On Pack provides a great starting point for understanding how to extend the out-of-the-box features of the MOSS platform to address specific regulatory requirements. In fact, the approach taken by the designers of the Add-On Pack truly underscores the power of the MOSS platform and the many extensibility points it offers for handling different kinds of requirements.

Requirements Addressed by the Add-On Pack

The primary goal of the Add-On Pack is to enhance the default Records Center site so that it includes the specific Records Management features required by the DoD 5015 standard. It is not intended to be used in production, but rather as a starting point for developing a compliant solution. It does, however, demonstrate many useful techniques that can be applied to many compliance scenarios.

The specific requirements addressed by the Add-On Pack are:

- Global periods and events
- Disposition and cutoff processing
- Record categories

Chapter 16: The DoD 5015.2 Add-On Pack

- ❑ Vital records
- ❑ Supplemental markings and access controlled columns
- ❑ Record relationships
- ❑ Enhanced search center functionality
- ❑ Workflow-assisted records management

We'll discuss all of these requirements in this section except for workflow-assisted records management, which is explored at the end of this chapter.

New Concepts

In order to address the DoD 5015.2 compliance requirements, the Add-On Pack introduces several concepts that provide the foundation for understanding how its components are intended to work. These concepts are derived directly from the language of the DoD 5015.2 standard.

Global Periods

Global periods are spans of time that are declared centrally so they can be shared throughout the Records Center site. The Add-On Pack creates a special list in which global period entries are defined. Each item specifies the period name, lengths, units, and a starting date. As records are managed, special retention rules can be applied that refer back to that period of time. The main advantage is that the time period can be changed easily without touching the records themselves or the retention policy. The actual expiration date is computed by comparing the record metadata to the information contained in the `GlobalPeriod` object with which it is associated.

Global Events

Global events are events that occur within an organization that may trigger the disposition of one or more records. The standard example given is that of employee termination. Whenever a person is hired in an organization, it is reasonable to assume that at some point the employee relationship will end. At that time, certain processing will have to occur related to the termination of the employer/employee relationship. One way to ensure that the appropriate end-of-employment processing actually happens is to set up an event mechanism that invokes the corresponding process at that time. The Add-On Pack adds `GlobalEvent` objects to help with this kind of scenario.

The problem is that global events are not abstract event definitions. They are actual event *instances* that have to be created for every future occurrence. In the case of the employee relationship, it would mean that a records manager would need to create a separate global employee termination event for each and every employee so that when that employee's employment ends, the appropriate record disposition logic can be invoked. Obviously, this would not work very well for most organizations, even though it technically meets the DoD 5015 standard. A better solution would be to provide a more general eventing mechanism, perhaps based on a set of rules that are evaluated whenever an employee is terminated. It is nevertheless instructive to examine the way the current feature is implemented.

Disposition and Cutoff Processing

According to the DoD specification, to *cut off* a record means to break or end it at regular intervals to permit its disposal or transfer as a completed unit. This concept allows records to be terminated in

batches so that old files can be replaced with new ones while the workflow continues. There are specific rules for when cutoffs should be applied, based on a record's retention period. For example, records with a retention period of less than a year are typically cut off when the retention period ends. On the other hand, records with a retention period of a year or more are typically cut off at the end of each fiscal or calendar year.

The DoD 5015.2 specification allows for the handling of more complex retention rules, which may be driven by many factors that can influence a record throughout its life cycle. However complex the retention rules are, at some point each record must be "disposed of," and, consequently, the meaning of *disposition* can also depend on a specific set of rules. The Add-On Pack provides a way to specify two kinds of disposition rules, called *disposition instructions*. It does this by defining two content types. The *Global Event Disposition Instruction* defines a list item that is triggered by an event in the global events list. The *Record Event Disposition Instruction* defines a list item that is triggered by a date column on an individual record instance. Using these content types and their associated lists, the records manager can set up disposition rules that can then be applied to records by associating them with record categories. The installed timer jobs take care of locating any affected records by finding records that match those categories and then examining and processing any associated disposition instructions.

Global Event Disposition Instruction

The *Global Event Disposition Instruction content type* includes a title, description, and number of units (Days, Months, or Years) used to calculate the disposition period. The actual disposition is specified using a choice field that can have one of the following values:

- Destroy
- Transfer
- Destroy After Transfer

If Transfer is selected, there is also a field for recording the destination path.

The Global Event Disposition Instruction content type includes a lookup field for selecting the global event that will trigger the retention period. When the instruction is processed, the specified event is examined to determine whether it has occurred, and the retention period is calculated based on the disposition offset and units specified in the instruction. If the retention period has elapsed, then the associated records are added to the list of records that are ready to be processed.

Record Event Disposition Instruction

The *Record Event Disposition Instruction content type* operates similarly to the Global Event Disposition Instruction, except that it is triggered by a date column in the record itself. The set of date columns you can use are determined by the presence of special field types that are installed along with the Add-On Pack. When the timer job runs, it looks at the column type specified in the instruction and then searches for matching column types in the affected records. Once a matching column type is located, the date value is used to determine whether the retention period has elapsed for that record.

The Add-On Pack defines three additional content types to support cutoff processing. These are described in the following sections.

Global Period Cutoff Instruction

Global Period Cutoff Instructions are useful for applying disposition rules on calculated dates that depend on a time period. A typical example would be the periodic disposition of financial records at the end of each fiscal year. Instead of calculating the cutoff period for a given record based on the date that the record was created, a *Global Period Cutoff Instruction* enables the disposition rule to be applied to all records that were created anytime during the fiscal year. Placing the rules in the cutoff instruction allows the disposition rules to be modified without having to change the records themselves. The cutoff instruction simply includes a title, description, and a lookup to the `Global Period` object to which it applies.

Global Event Cutoff Instruction

The *Global Event Cutoff Instruction content type* is used to apply disposition rules for event-based retention periods when the event applies to a category.

Folder Event Cutoff Instruction

The Add-On Pack allows global events to be associated with folders. This provides a location-based disposition mechanism whereby the disposition rules are applied to all records contained in the folder when the global event period ends. The *Folder Event Cutoff Instruction content type* allows the records manager to set up a cutoff that is also based on the folder's `Global Event` trigger. Thus, whenever the event is triggered, the cutoff instruction is applied.

Record Categories

The Add-On Pack creates a special content type for defining record categories. This content type is provided as part of the File Plan Builder, which is simply a set of lists that are pre-configured to enable the creation of these special list items. The true power of the Record Category content type comes from its ability to reference multiple disposition instructions through a lookup field that searches for all of the available disposition rules that have been created in the site.

This is where you start to see the “big picture” and everything starts to fall into place with the Add-On Pack. By using record categories, you can create groups of disposition instructions and associate them with other objects, such as records and folders, and the workflows and timer jobs included in the Add-On Pack will apply them consistently to the affected records. It adds that extra layer of indirection that provides the flexibility needed to address real-world scenarios. Since record categories behave as aggregate instruction sets that are applied to groups of records at once, they are also a convenient place to include other information that can be used to control how those records are processed, for example, whether the records associated with the category should be treated as *permanent* records and are therefore prevented from being deleted, or whether they are marked as *vital* records that require periodic review. Consequently, additional fields are included within the content type to accommodate these indicators.

Vital Records

Vital records are special records that are considered critical for the continuity of an organization. If the organization suffers a catastrophic event, these are the records that will be examined by forensic accountants and others involved in the aftermath. These are things like property deeds, financial instruments, and the like.

DoD 5015 requires that a certified records management system must provide a means for designating certain users or groups to routinely review and manage vital records for the organization. Thus, there must be a way to designate a particular folder or category as one that contains vital records and to assign them to particular users or groups. This is done using a special field called a *Vital Records Indicator*, which is added to a Record Folder, along with another field that specifies the amount of time between reviews of vital records and the user ID or the name of the group responsible for reviewing the records in the folder.

Supplemental Markings and Access Constraining Columns

The term *supplemental markings* refers to the idea that certain documents carry special processing instructions in the form of *markings* that are intended only to control their processing and handling, and not to specify how they are classified. Specifically, the DoD 5015 standard requires that a certified software system support the use of supplemental markings to grant access to individual records for designated users and groups. This feature is part of the record-level security requirement. Two forms of record-level security are supported: supplemental markings and access control columns (or access *constraining* columns).

Supplemental markings are created as items in the Supplemental Markings list. The user specifies the name of the marking and the list of users and groups who will be allowed to assign the marking to a record. Once created and assigned to a record, the specified users and groups will be the only ones allowed to view the record after the Advanced Access Control timer job runs and updates the item-level permissions on the corresponding records.

Users can also use the Item Level Access Control page to declare columns as “access constraining.” Once a column is declared as access constraining, its value must be chosen from a predefined selection list specified by the Records Center administrator. A special list is provided to enable the administrator to add values for these columns and to associate users and groups with specific values.

Record Relationships

Record relationships are used to link records together in a hierarchy or in a sequence. The Add-On Pack defines the following three relationships by default:

- Succession** — One record supersedes another.
- Rendition** — One record is a copy of another.
- Attachments** — One record is attached to another.

The user creates and manages record relationships by creating a Record Relationship Template item in the Record Relationship Templates list of the Records Center site. Two types of template are available — hierarchical or peer. A hierarchical relationship is a *directed* relationship between two records, whereas a peer relationship is undirected. The template itself consists of a title, description, and custom text that is used to describe the relationship in views where the record relationship is displayed. For example, if a new relationship template with the title “Cross Reference” is created, then the custom text might be entered as “a cross reference with” to indicate how the records are related to one another.

Once the template has been created, the new relationship will appear in the list of available relationships that is presented when the user clicks on the hyperlink displayed by the `RecordRelationshipsFieldControl`.

Enhanced Search

The enhanced search functionality provided by the Add-On Pack includes the following additional features, which are not found in the standard Records Center:

- The ability to specify the order in which selected properties are displayed for users, and the ability for users to specify which properties they wish to see
- The ability to search within the current result set
- The ability of administrators to configure the default properties available for searching
- The ability to search on null and blank columns and to use wildcard characters

This functionality is implemented in the RecordSearch.aspx application page that is installed into the Layouts folder. Figure 16-1 shows the user interface that end-users see after the administrator has configured record search fields.

Search the Records Center

Use this page to create an organized view of specific records across all categories and folders. This page is similar to the Advanced Search page, but it also enables you to:

- Use wildcard characters (* and ?) when searching a text property
- Search for an empty or missing property
- Display results in a customizable table
- Sort results by any property
- Search within the results of a previous search

Find records with...

All of these words:

The **exact** phrase:

Any of these words:

None of these words:

Add property restrictions...

Where the Property... (Pick Property) Contains

Change how the search results are displayed...

Results per page:

Select the columns to display in the results table:

Figure 16-1: Enhanced record search page.

End-users will see an error page until the record search fields are configured.

To configure the search fields, an administrator must open the “Manage Record Center Search” link from the Site Settings page. This page, shown in Figure 16-2, displays a list of managed search columns that can be selected for inclusion on the search page. Now, when the user navigates to the record search page, the property dropdown will display only the columns that have been provisioned by the administrator.

Manage Record Center Search Columns

Use this page to select what columns appear on the Records Center Search page.

Managed Search Columns
The list contains all indexed columns used by records. Select the columns that people can use when searching the Records Center. You can also configure the default view for the search.
If a column does not appear on the list, the search crawler may not have found it yet.

Show on Search Page	Column Name	Include in Default Table	Default Position from Left
<input checked="" type="checkbox"/>	Format	<input type="checkbox"/>	1
<input checked="" type="checkbox"/>	Originating Organization	<input type="checkbox"/>	2
<input checked="" type="checkbox"/>	Publication Date	<input type="checkbox"/>	3
<input checked="" type="checkbox"/>	Record Relationships	<input type="checkbox"/>	4
<input checked="" type="checkbox"/>	Record Type	<input type="checkbox"/>	5
<input checked="" type="checkbox"/>	Subject	<input type="checkbox"/>	6
<input checked="" type="checkbox"/>	Supplemental Marking Column	<input type="checkbox"/>	7
<input type="checkbox"/>	Content Type	<input type="checkbox"/>	8
<input type="checkbox"/>	Document Created By	<input type="checkbox"/>	9
<input type="checkbox"/>	Addressee(s)	<input type="checkbox"/>	10
<input type="checkbox"/>	Author or Originator	<input type="checkbox"/>	11
<input type="checkbox"/>	Date Filed	<input type="checkbox"/>	12
<input type="checkbox"/>	Date Received	<input type="checkbox"/>	13
<input type="checkbox"/>	Date Superseded	<input type="checkbox"/>	14
<input type="checkbox"/>	Location	<input type="checkbox"/>	15
<input type="checkbox"/>	Media Type	<input type="checkbox"/>	16
<input type="checkbox"/>	NullField	<input type="checkbox"/>	17
<input type="checkbox"/>	Other Addressee(s)	<input type="checkbox"/>	18
<input type="checkbox"/>	Parent Folder	<input type="checkbox"/>	19
<input type="checkbox"/>	Parent Category	<input type="checkbox"/>	20

Figure 16-2: Record search field configuration page.

Installing the Add-On Pack

The Add-On Pack comes as a Windows installer (MSI) file that is run on any web server in the SharePoint farm. To run the installer, you must be logged in as a Farm Administrator. When the installer is executed, it deploys a SharePoint solution package called *DoD_5015_2_Add_On_Pack* to every web application in the farm. The solution package, in turn, installs and activates the features needed to provide the additional functionality required by the solution.

To install the Add-On Pack, perform the following steps:

1. Download the installation package from the Microsoft Download center at www.microsoft.com/downloads/. You can search for "Office SharePoint Server 2007 DoD 5015.2 Resource Kit."
2. Run the MSI file on any machine in the SharePoint farm. You must be logged in as a Farm Administrator in order to run the installer. You can double-click on the installer from Windows Explorer, or run it from a command line to specify either of two command-line parameters. Specifying **/SkipAdd True** on the command line will prevent the installer from adding the solution to the Farm solution store. Specifying **/SkipDeploy True** will add the solution to the Farm, but will not automatically deploy it.

Keep in mind that there are three steps involved in the installation of the Add-On Pack. First, there is the machine-level installation, which copies the DLLs onto the SharePoint Server. The second part is the SharePoint installation, which adds the solution to the SharePoint farm, thereby making it available for deployment. Specifying `!SkipAdd True` performs only the machine-level installation, requiring the solution package to be installed manually. Finally, there is the actual SharePoint deployment step, which occurs after the solution package is added to the farm. Specifying `!SkipDeploy True` performs only the machine-level and the SharePoint installation, so that the solution shows up in the Central Administration web site but is not automatically deployed.

3. Once the solution is deployed, create a new Records Center site based on the default site definition. The site collection feature should automatically activate. If you already have a Records Center site created, you will have to activate the site collection feature manually. This feature must be activated before the web-level features are activated, and you must be logged in as a Farm Administrator in order to activate it.
4. Activate the web-level features on the Records Center site to create the custom lists and pre-populate them with default values. The Add-On Pack only supports activation in the root web of the site collection. There are two features that can be activated. The Web Components feature sets up the lists and data, and the E-mail Router feature installs a custom router that can handle e-mail records submitted from an Exchange 2007 managed folder.

The Web Components feature must be activated before the E-mail Router feature and may only be activated from the command line using STSADM.

When you install the Add-On Pack and create a Records Center site, it looks like any other Records Center site until you activate the web-scoped features. Then it takes on different characteristics because of the specialized lists and other components that are enabled. Although the site collection features are activated automatically when the solution is deployed, the web-scoped features have to be activated manually after creating the site. Figure 16-3 shows the home page of an enhanced Records Center site.

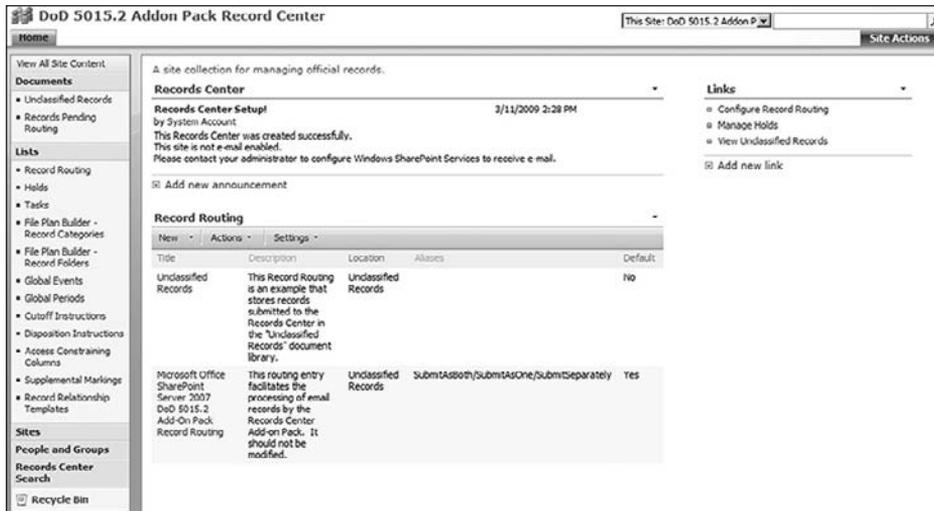


Figure 16-3: Enhanced Records Center home page.

The Components Installed by the Add-On Pack

The Add-On Pack installs the following components that together provide the extended functionality needed for DoD 5015.2 compliance:

- The RecordCenterRouter feature
- The RecordCenterAddonPack web feature
- The RecordCenterAddonPack SiteWorkflows feature
- Custom field types
- Custom field controls
- Content types
- Timer jobs
- Workflows
- STSADM commands

The following sections describe each component and the functionality it provides.

The RecordCenterRouter Feature

The RecordCenterRouter feature is a web-scoped feature that is activated on the Records Center site. It includes an activation dependency on the Records Center Add-On Pack web feature so that it cannot be activated without the rest of the Add-On Pack components installed. This feature installs a custom router that provides the extended record routing logic required by the DoD 5015 standard.

See the section at the end of the chapter entitled, “The Records Center E-mail Router,” for more information about how the router works.

The RecordCenterAddonPack_Web Feature

The RecordCenterAddonPack web feature is responsible for installing most of the other components used by the Add-On Pack. It also double-checks upon activation to make sure that the feature is being activated by a user with Farm Administrator privileges. Also during activation, the feature performs some additional checks to ensure that the required components have all been installed properly and that the custom lists have been populated with default values. For instance, the default groups, roles, and relationships are pre-populated with data, and special web properties are added to the `SPSite` and `SPWeb` objects associated with the Records Center site so that the other components can access them.

Once the basic components have been set up, the feature activation code creates the task and history lists needed by the DoD 5015.2 workflows and then creates the necessary workflow associations and sets up the workflow templates so that they appear on the corresponding lists. Finally, it installs the custom timer job components that are used to periodically check each list for items that require special processing.

The RecordCenterAddonPack_SiteWorkflows Feature

The RecordCenterAddonPack_SiteWorkflows feature installs the custom workflows and InfoPath forms used by the Add-On Pack. There are three primary workflows and one auxiliary workflow installed by the feature to support cutoff approval, disposition, and vital records review. The individual workflows are described in detail in the section below entitled, “Workflows.”

Custom Field Types and Field Controls

In order to enable the enhanced functionality needed by various features, the Add-On Pack creates several custom field types that are used to handle special requirements imposed by the DoD 5015.2 specification. Field types in SharePoint behave like data types in a programming language. They define the type and format of data that can be stored in a column attached to a list item. Certain features of the DoD 5015.2 specification require that default values are provided for fields associated with a record. This applies to choice fields, text fields, and user selection fields. Other features of the Add-On Pack require the pre-selection of lookup values so that records managers are not required to set them manually. Similarly, certain fields must be marked read-only. The following table lists the custom field types that are installed by the Add-On Pack to provide these capabilities:

Field/Control Type	Description
DefaultingChoiceField, DefaultingLookupField, DefaultingTextField, DefaultingUserField	Automatically checks if the field is associated with a record folder or category and sets itself to the appropriate default value.
ExtendedDateTimeField	Adds support for converting null date values to text and for rendering the value in HTML. Also overrides the <code>GetValidatedString</code> routine to provide better date validation logic.
ParentFoldersLookupField	Ensures that the control is only placed on valid pages within the Records Center site, and then populates a <code>DropDownList</code> control with the names of parent record folders relative to the current page within the site.
ReadOnlyDateTimeField, ReadOnlyLookupField, ReadOnlyTextField	Sets the <code>ControlMode</code> to <code>SPControlMode.Display</code> to render the control as read-only.
RecordRelationshipsField	Displays record relationships in a modal dialog, allowing the user to select items from the list.
RecordRoutingField	Performs a query to obtain the list of pending e-mail records and displays them in a <code>DropDownList</code> along with the target folder URL for each item.
RecordVersionsField	Displays record versions in a modal dialog, allowing the user to select items from the list.

Content Types

The Add-On Pack creates several content types that control the functionality of various components. In many cases, the components check whether the item being acted on is based on one of these content types (see following table):

Content Type	Description
Access Constraining Column	Allows the user to define a choice value for an access-constraining column.
Correspondence Record	Use this content type for records that are communications between people. The <code>Email Record</code> content type is a child of this type.
Email Record	All e-mail records submitted to the Records Repository should be of this type.
Folder Event Cutoff Instruction	Creates a cutoff instruction that is triggered by an event trigger on a category's folders.
Global Event Cutoff Instruction	Creates a cutoff instruction that is triggered by an event from the Global Events list.
Global Period Cutoff Instruction	Creates a cutoff instruction that causes cutoff to reoccur every time a global periods ends.
Nonelectronic Record	All non-electronic records submitted to the Records Repository should be of this type. The physical storage of these records is managed outside the electronic repository.
Record Category	Defines a record category for the site.
Record Category On Record Folders List	Items of this type are used to navigate to the correct category so that users can add a folder in that category.
Record Folder Container	Creates a container for record folders.
Record Folder	Provides a way to group records together so that their disposition occurs simultaneously.
Record	All records submitted to the Records Repository should be of this type or a child of this type.

Timer Jobs

Much of the functionality provided by the Add-On Pack is handled using SharePoint Timer Jobs. This allows the specialized processing to be scheduled independently so they don't affect the performance of the Records Center.

- **E-mail Record Processing Timer Job** — This job searches for pending e-mail records and schedules them for review.

Chapter 16: The DoD 5015.2 Add-On Pack

- ❑ **Folder Hold Timer Job** — This job is responsible for ensuring that folder holds are applied to all records contained in the folder.
- ❑ **Cutoff Approval Job** — This job is responsible for initiating the Cutoff Approval workflow for all affected folders and categories.
- ❑ **Record Local Events Job** — The Add-On Pack allows you to extend the content types used to declare records by inheriting from the types it creates in the Records Center. When extending one of these content types, you may add a date column or one of the extended date columns used to specify “Record Local Events.” This timer job searches for any such extensions and updates the hidden list of field names that define Record Local Events.
- ❑ **Advanced Access Control Job** — This job is responsible for updating the declared record permissions that are determined by Access Control columns and Supplemental Markings. When this job runs, it enumerates over all sites in all site collections of the farm in search of any site for which advanced access control has been enabled and then processes them. This determination is made based on the presence of a Boolean property that is added to the property bag of the site when the Add-On Pack web feature is activated.
- ❑ **Vital Records Review Job** — This job is responsible for initiating the Vital Records Review workflow.

Workflows

The following table lists the workflows that are included in the Add-On Pack. The sections that follow describe each workflow in greater detail.

Workflow	Description
Cutoff Approval Workflow	This workflow facilitates the cutoff process. Cutoff allows records managers to batch expiration so that records are processed in groups.
Cutoff Approval Reverse Workflow	This workflow facilitates the cutoff approval reversal process.
Disposition Approval Transfer	This workflow is used to facilitate the transfer or deletion of a record based on its current disposition instruction. It is auto-started by the expiration framework.
Vital Record Review Workflow	This workflow facilitates the periodic review of vital records. If the <i>vital record indicator</i> field is set to Yes, then items in a given folder will undergo a periodic review.

The Cutoff Approval Workflow

The Cutoff Approval workflow is included in the Add-On Pack to allow records with specific cutoff dates to be approved for processing. This workflow facilitates the regular review of records that should

be handled according to cutoff rules by assigning cutoff approval tasks or notifying the appropriate persons if there are no records that meet the cutoff criteria. The flowchart shown in Figure 16-4 illustrates the processing steps involved in the workflow.

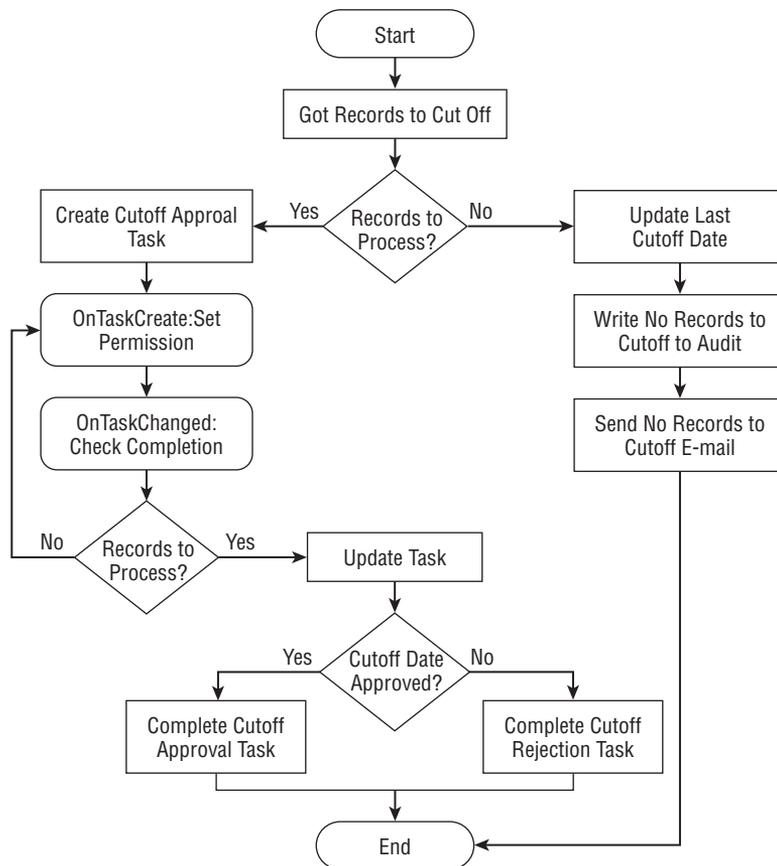


Figure 16-4: Cutoff Approval workflow.

The Cutoff Approval Reverse Workflow

The Cutoff Approval Reverse workflow is used to reverse the cutoff approval process for an item. This workflow simply invokes the `ReverseCutoff` method of the Cutoff Approval workflow, which searches through all of the web sites in the farm that have been enabled for cutoff approval to locate the record folder or record category associated with the item. It then resets the filing status to `Open` and clears the cutoff date.

The Disposition Approval Transfer

The Disposition Approval Transfer workflow facilitates the transfer or deletion of a record based on its current disposition instruction. The disposition of a record may occur independently of any statically defined retention period and may be repeated periodically, depending on the type of disposition and the instructions it contains. The workflow assigns tasks based on the current permissions associated with the record as well as other factors. Figure 16-5 shows the processing steps involved in the workflow.

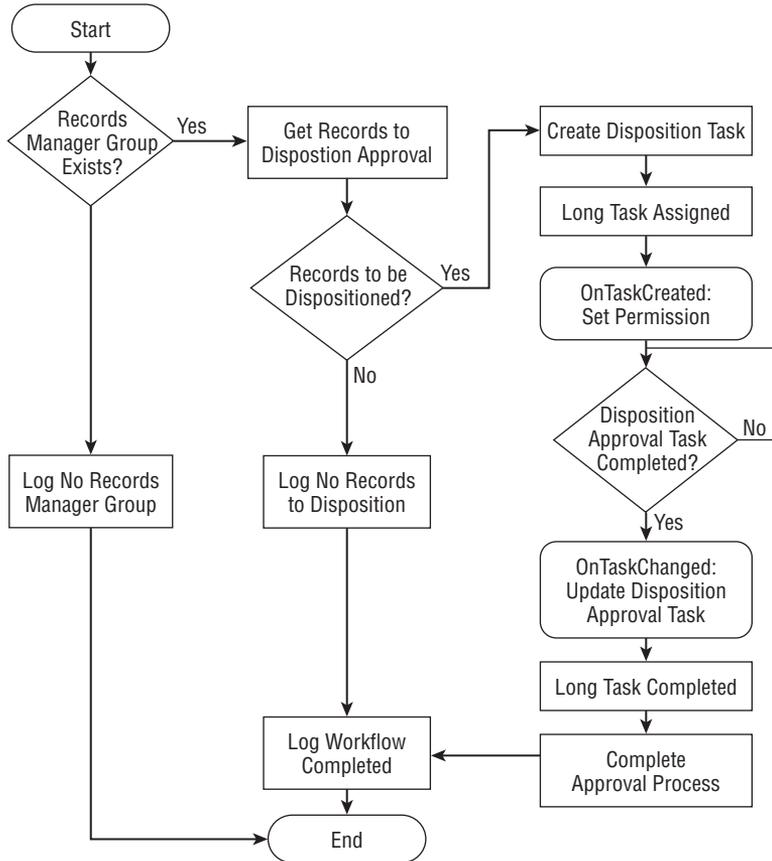


Figure 16-5: Disposition Approval Transfer.

The Vital Record Review Workflow

The Vital Record Review workflow facilitates the periodic review of records contained in folders that have the *vital record indicator* field set to Yes. Figure 16-6 shows the processing logic for this workflow.

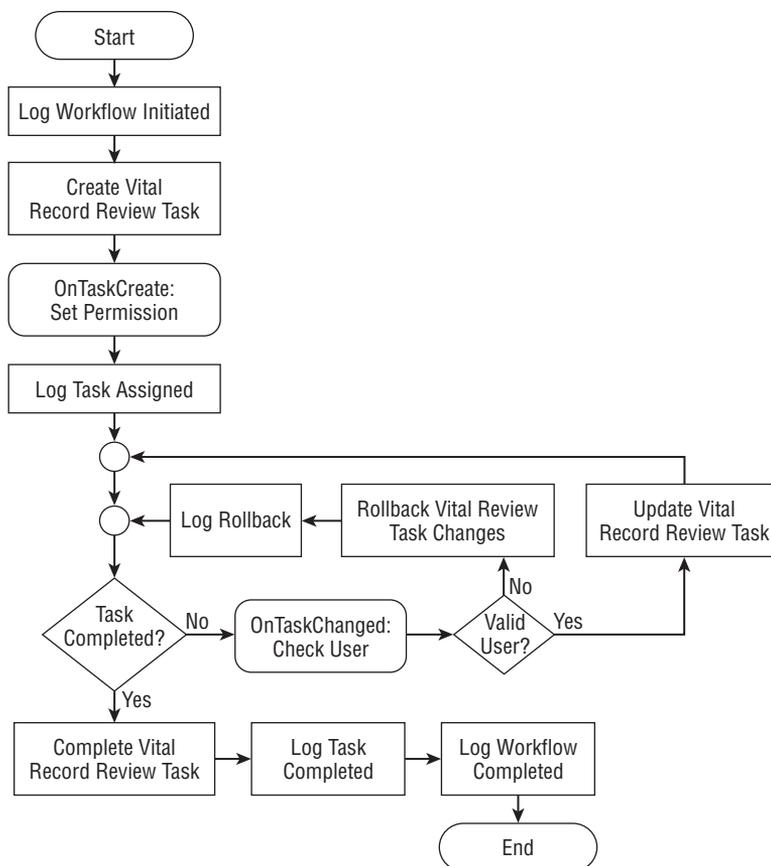


Figure 16-6: Vital Record Review workflow.

STSADM Commands

The Add-On Pack installs a complete set of STSADM commands that enable administrators to work with the enhanced features of the Records Center. All of the commands ending in *Schedule* take a single `-schedule` parameter of the form "Every ## minutes between <start> and <end>", where the start and end values specify a minute between 0 and 59. For example, "every 30 minutes between 0 and 59" would cause the job to run every half-hour. The following table lists the STSADM commands:

Command	Description
<code>SetFolderHoldsProcessingJobsSchedule</code>	Schedules the Folder Hold timer job.
<code>SetVitalRecordsReviewSchedule</code>	Schedules the Vital Records Review job.

Continued

Command	Description
SetRecordLocalEventsListSchedule	Schedules the job that updates the Record Local Events list.
SetAdvancedAccessControlSchedule	Schedules the Advanced Access Control job.
SetCutoffApprovalSchedule	Schedules the Cutoff Approval job.
SetEmailRecordProcessingSchedule	Schedules the Email Record Processing job.
RunJob	This command is provided to enable the execution of any of the Add-On Pack timer jobs. The syntax of the command is "stsadm -o run-job -jobname job", where job is one of Holds, VitalRecordReview, Cutoff, or AdvancedAccessControl.
TruncateAuditLog	This command is provided to enable records management to truncate the audit log, optionally placing the raw audit entries in a file for archival purposes. A date can be provided, which is the date before which audit entries are removed.

The Records Center E-Mail Router

The Add-On Pack installs a special router for handling e-mail records that are submitted from an Exchange 2007 managed folder in Outlook 2007. This router is needed to ensure that the DoD 5015.2 requirements relating to e-mail records are enforced. In particular, the router handles the case where the e-mail message and all its attachments are filed as a single record. Listing 16-1 shows the base implementation of the Records Center router.

Listing 16-1: Records Center router implementation

```
public RouterResult OnSubmitFile(string recordSeries, string sourceUrl,
string userName, ref byte[] fileToSubmit, ref RecordsRepositoryProperty[]
properties, ref SPList destination, ref string resultDetails)
{
    SPWeb parentWeb = destination.ParentWeb;
    parentWeb.Site.AllowUnsafeUpdates = true;
    parentWeb.AllowUnsafeUpdates = true;
    SingleRecordRouter router = null;
    string label = GetLabel(GetDictionary(properties));
    RoutingType type = RoutingType.Default;
    try
    {
        type = (RoutingType) Enum.Parse(typeof(RoutingType), label, true);
```

```
    }
    catch (Exception)
    {
    }
    switch (type)
    {
        case RoutingType.SubmitAsOne:
            router = new SingleRecordRouter();
            break;

        case RoutingType.SubmitSeparately:
            router = new MultipleSeparateRecordRouter();
            break;

        case RoutingType.SubmitAsBoth:
            router = new MultipleRecordRouter();
            break;

        default:
            router = new DefaultRecordRouter();
            break;
    }
    try
    {
        router.Store(parentWeb, fileToSubmit, userName);
    }
    catch
    {
        if (router != null)
        {
            router.Dispose();
        }
        router = new DefaultRecordRouter();
        router.Store(parentWeb, fileToSubmit, userName);
    }
    finally
    {
        router.Dispose();
    }
    parentWeb.AllowUnsafeUpdates = false;
    parentWeb.Site.AllowUnsafeUpdates = false;
    return 1;
}
```

The router first calculates the routing type by examining the submitted routing label, which can indicate that the person submitting the record wants to submit the e-mail message by itself, together with its attachments, or both. There is also the option of processing the record normally using the default routing mechanism. If there is a problem storing the record using one of the specialized routing methods, then the default method is used. If storing the message as a single record, the router first checks to ensure that the submitted file is, in fact, an e-mail message. It then adds the message content as a new file to the Records Pending Routing list.

Summary

The DoD 5015 specification is widely considered the “gold standard” for evaluating records management software systems. In May 2007, the U.S. Department of Defense Joint Interoperability Test Command tested the new functionality added to the MOSS 2007 Records Center by the DoD 5015.2 Add-On Pack and awarded certification, making it the baseline for DoD 5015.2 compliance on the SharePoint platform. This chapter introduced the DoD 5015.2 Add-On Pack as an example of how the default Records Center site definition can be extended to accommodate more exacting records management requirements such as those imposed by DoD 5015.

The Add-On Pack installs several features and components into the MOSS environment. These components work together to provide the extended functionality required by the 5015.2 standard. We examined the necessary steps for installing and configuring the Add-On Pack and explored its features and components to see how they work and also to use them as reference points for implementing similar features in other solutions.

Index

A

access

- constraining, 407
- content, 88–90
- control, 88

ACLs, 89

Action, 240, 400

Actions, 400

.ACTIONS, 396–401

activities. *See also* role/activity modeling

- information, 11
- role/activity modeling, 8–12
- workflow, 377–380

Add, 305, 306

AddToHold, 306

Aliases, 104

All, 256

AllowDeletion, 104, 105

Annual Report, 172, 175, 177

Application Management, 171

Application Pool Recycler, 21

ApplyHold, 380

APPPool, 162

Archive, 245

args, 305

artifacts, 13–14

.ASCX, 224–225

ASP.NET, 225

ASPX, 97

- file plans, 164
- submission, 172

AssemblyName, 55, 306

Asset Tracking, 135–136

attribute classes, 56, 59–63

attributes, 218

AttributeTargets, 56

Audit History, 110–112

Audit Viewer, 262–271

auditability, 3, 11

AuditEntry, 272

AuditEntryField, 265

AuditHistory.aspx, 273–274

AuditHistoryGrid, 274

AuditHistoryGrid.Refresh, 274–276

auditing

- content types, 277–279
- policy, 255–279
- Records Center, 262–276
- web parts, 262–271

AuditMaskChange, 260

AuditViewerWebPart, 274

<authorizedType>, 396

AuthorizeToUpdate, 258–259

AvailableContentTypes, 54

B

Bar Coded, 151

Barcode, 283–284

BarcodePolicy, 288

BeforePrint, 230

behavior, 43, 97

bool, 106

Brief, 125–126, 128, 172

browser utility, 51–55

btnExecute_Click, 168–169

btn_Submit_Click, 189

BuiltInExpirationAction, 248

ButtonSection, 166

byte(), 106

Bytes, 106

C

C#, 33, 72, 142

CAML. *See* Collaborative Application Markup Language

CAML.NET Intellisense, 22–23

Catch, 326–327, 328

Category, 14

Category, 42, 112

Central Administration, 171, 197, 243

CheckBoxTemplate, 186

CheckIn, 256, 260
check-in, 67–68
CheckOut, 256, 260
check-out, 67–68
child, 197
ChildDelete, 256, 260
ChildMove, 261
CHOICE, 61
classes, 142–157, 217–223. *See also* wrapper classes
 attribute, 56, 59–63
 schema, 36–37
ClassName, 55, 306
code-beside model, 33
Collaborative Application Markup Language (CAML)
 content types, 48–51
 FieldRef, 59
 FilePlan.xsn, 157–158
 keyword queries, 306
 schema, 48
 SchemaXml, 209, 214
 XML, 47
ColumnInfo, 264, 265–271
ColumnInfo Update, 268
ComputeExpireDate, 239
Configuration, 103
Configure Exchange, 359
content
 access, 88–90
 managed, 363–366
 modeling, 3–14
 security, 83–91
 validation, 325–353
Content Type Explorer, 53
Content Type gallery, 173–174
content types, 42–63, 150–152
 auditing, 277–279
 child, 197
 document libraries, 45, 153
 DoD 5015.2 Add-On Pack, 413
 ECM, 39, 97
 encapsulation, 42, 46
 folders, 285
 Global Event Disposition
 Instruction, 405
 Global Period Cutoff Instructions, 406
 hierarchy, 45–46
 lists, 65
 physical records, 154, 282–283
 policy, 205
 policy statement, 198

 sample, 64
 SPItemEventReceiver, 64
 traditional file plans, 125–129
 XML, 47, 152
ContentTypes, 53–54
Contract, 172
ControlMode, 412
Copy, 256, 261
Create, 124, 288
CreateContentTypes, 150–152, 249
CreateFromListItem, 35
CreateItemQuery(), 303
Custom, 261
 custom, 245
custom routing, 25, 108, 307–323
 file plans, 322–323
 requirements identification, 16
CustomAction, 162
customData, 200
CustomExpirationAction, 240–243, 248
CustomSettingsControl, 225
CustomXMLPart, 231
cut off processing, 404–406
Cutoff Approval, 414–415
Cutoff Approval Reverse, 415

D

DataTable, 265, 267, 274
DateCalculator, 240
DateTime, 177, 296
Default, 104
default expiration actions, 103, 105
Default Record Series, 105
Delete, 245, 248, 256, 261, 276
Department of Defense. *See* DoD 5015.2 Add-On Pack
dependencies, 86–87, 384
Description, 104, 105
de-serialization, 144–146, 167
 wrapper classes, 142
 XML, 33
Designer, 6
DesignerType, 401
destinationSPList, 319
diagnostic tracing, 21
_dic_RepositoryUsersGroup, 104
differencing engine, 65
disposition, 15, 404–406
Disposition Approval, 105
Disposition Approval Transfer, 416

Document, 150, 153

document(s)

- categories, 16
- check-out, 66, 67–68
- groups, 16
- metadata, 4
- MOSS, 97
- properties, 41–42
- Records Center, 11
- sessions, 67
- sources, 16–17
- uploads, 66
- users, 11
- versioning, 67

document libraries, 40–42

- content types, 45, 153
- holding zone, 104
- I/O, 96
- property storage, 108
- Record Routing Table, 281
- submissions, 92, 175, 178
- traditional file plans, 125–129
- Unclassified Records, 105
- versioning, 82

document parser, 42, 112, 115, 296

document retention policy statement, 46

DocumentGrid.cs, 182–186

DocumentGrid_RowDataBound, 187

DoD 5015.2 Add-On Pack, 19–20, 403–420

- content types, 413
- timer job, 413–414
- workflow, 414–417

.DOTX, 97

DropDownList, 412

DVDs, 285

dynamic file plans, 15, 117, 130–169

- expiration policies, 245–253
- workflow, 388–391

E

ECB. See Edit Control Block

ECM. See Enterprise Content Management

ECM Starter Kit, 372–373

ECM2007, 23–25

ECM2007.ContentTypes, 24

ECM2007.CustomRouting, 309

ECM2007.DocumentGrid, 188

ECM2007.FeatureWizard, 28–31

ECM2007.InformationPolicy, 288

ECM2007.InformationPolicy.

SharePointPolicyFeature, 218

ECM2007.OfficialFileSend, 188

ECM2007.PrinterControlPolicyFeaure, 224

ECM2007.RecordsManagement, 25, 310

ECM2007.Versioning, 25, 79

ECM.InformationPolicy, 24

Edit Control Block (ECB), 157, 171

- auditing, 271
- File Plan Execution, 158
- JavaScript, 296
- Send To, 109

Edit Properties, 42

elements.xml, 180–182, 273

e-mail, 355–369

E-mail Router, 410

EnableBarcodePolicy, 381

EnableLabelPolicy, 381

EnableViewState, 186

encapsulation, 42, 46

- ECM, 39

encryption, 91

End Date, 128

enhanced search, 408–409

Enterprise Content Management (ECM), 21

- access control, 88
- content types, 39, 97
- encapsulation, 39
- workflow, 372, 396

Eval, 334, 350

event handler, 185, 187

event receiver, 52, 154–157

EventData, 259

EventsDeleted, 261

Excel, 6, 13, 382

Excel Markup Language (ExcelML), 260

- Holds Reports, 302

ExcelML. See Excel Markup Language

Exception, 329–330

Exchange, 357–368

Exchange Management Console, 357

ExchangeManagementShell, 360–363

Execute, 401

ExecuteFilePlan, 380

Expiration, 246

expiration actions, 240–243

expiration formulas, 238–239

expiration policies, 245–253

Expiration Policy Feature, 203, 235–236

ExpirationAction.cs, 252–253

ExpirationFormula

ExpirationFormula, 246–248
ExpirationFunction, 246–248
ExpirationPeriod, 246–248
ExpirationPolicy, 246
ExpirationPolicy.cs, 250–251
ExpirationPolicyFeature, 246
ExpirationPolicyFeature.cs, 249–250
Expression, 328, 350
extensibility, 44, 203
 payload, 97
extension methods, 71
 C#, 72
 .NET Framework, 72
 versioning metrics, 79–82

F

feature stapling, 118
Feature Wizard, 179
FeatureActivated, 287, 309, 313, 317
 FeatureReceiver.cs, 222
 SPFeatureReceiver, 100
FeatureDeactivating, 223, 313, 317
FeatureId, 214, 239, 244
FeatureReceiver, 159–162, 287
FeatureReceiver.cs, 222, 322, 345–346
FEATURES, 179
feature.xml, 180, 286
Field, 22, 401
FieldBind, 401
FieldDataType, 330
FieldExpression, 328–329
FieldExpression.cs, 335–336
FieldRangeExpression, 328–329
FieldRef, 59–63
FieldRefAttribute, 24, 61
FieldValidator, 225
File Plan, 137–142
file plan(s), 14–17
 ASPX, 164
 custom routing, 322–323
 execution, 164–169
 expiration policies, 245
 physical records, 154
 Records Center, 146–149
 schema, 132–134
 traditional, 125–130
 Visual Studio, 140
File Plan Execute, 164–166

File Plan Execution, 131, 158, 167
File Plan gallery, 131, 157–163
FileLeafRef, 181
FilePlan, 132, 322
 data source, 140
 de-serialization, 144–146, 167
 Exchange, 359, 364–365
 serialization, 142–144
FilePlan.ConfigureExchange, 365, 366
FilePlan.cs, 142
FilePlanEx.cs, 144
FilePlan.Load, 167
FilePlan.xsd, 142
FilePlan.xsn, 157–158
FileTrackingRouter, 316
FilteringRouter, 310–313
FilteringRouter.OnSubmitFile, 311–314
Financial Document, 125, 129
Financial Statement, 172
folders
 content types, 285
 document libraries, 40
 managed, 357–363
 Outlook 2007, 368
 physical records, 284–286
forced check-out, 66
forms
 File Plan, 137–142
Forms Designer, 36
 InfoPath, 135–137
 Visual Studio, 138

G

GAC. See **Global Assembly Cache**
GenerateAuditReport, 381
GetHoldItemIndex(), 303
GetRecordRoutingCollection, 95
GetValidatedString, 412
GLBA. See **Gramm-Leach-Bliley Act**
Global Assembly Cache (GAC), 25, 32
Global Event, 404, 406
Global Event Disposition
 Instructions, 405
Global Period, 404
Global Period Cutoff Instructions, 406
Gramm-Leach-Bliley Act (GLBA), 20
grid, 267
GUID, 48

H

Health Insurance Portability and Accountability Act (HIPAA), 17, 20

helper methods
validation, 340–341

Helpers, 56

HIPAA. See Health Insurance Portability and Accountability Act

History List, 105

Hold, 298–301

Hold Reports List, 298

holder, 56

holding zone, 104–105
metadata, 103

Hold.RemoveHold, 304

holds
programs, 302–303
Records Management, 295–306

Holds List, 105, 297

Holds Reports, 297, 302

Hold.SetHold, 303

HoldsField, 296, 298, 304

HoldsFieldControl, 296, 298

HoldStatusField, 296, 298

HoldStatusFieldControl, 298

HTML, 49–51

I

ID, 181

ID="1", 103

IDE. See Integrated Development Environment

IExpirationAction, 203, 240

IExpirationFormula, 203, 237, 238

IISRESET, 317

IMAGES, 179

InfoPath, 134–142

Forms Designer, 135–137
SQL, 138
submission, 137
template, 138
WS, 138
XML, 131, 138, 141
XSL, 134

information

activities, 11
integrity, 3

policy, 195–233
spreadsheets, 15
workflow, 11

Information Policy. See policy

Information Rights Management (IRM), 90–91

InputFormControl, 166

InputFormSection, 166

Install, 215, 222

Installer

.NET Framework, 239

Installer, 228, 309

InstallProjectTemplate.bat, 31

InstallUtil, 204, 215, 217, 309

Installutil.exe, 239–40

Integer, 177

Integrated Development Environment (IDE), 382

IntelliSense, 22–23, 48

InvalidOperation, 205

I/O

document libraries, 96

IPolicyFeature, 25, 200–201, 236

IPolicyFeature

.OnGlobalCustomDataChange, 202

IPolicyFeature.ProcessListItem, 202

IPolicyFeature

.ProcessListItemRemove, 202

IPolicyFeature.Register, 202

IPolicyFeature.Unregister, 202

IPolicyResource, 236

IProcess, 306

IRM. See Information Rights Management

IRouter, 25, 308, 311

IRouter.OnSubmitFile, 108

ISAPI, 92, 198

ISharePointObject, 24, 55, 56

ISharePointPolicy, 24, 205

ISharePointPolicyFeature, 24, 209

ISharePointPolicyResource, 24, 212–216

IsHoldEnabled, 296

Item, 150, 153

Item Level Access Control, 407

ItemAdded, 151, 154–157, 218, 296

ItemAdding, 104, 296, 346

ItemDeleting, 104, 296

ItemEventReceiver, 24, 219, 221

ItemUpdated, 218

ItemUpdating, 104, 296, 346

IWizard, 28

J

JavaScript, 296

K

keyword queries, 306

Keywords, 42

L

Label, 225

Labeled, 151

LAYOUTS, 179, 224

LayoutsPageBase, 166

level, 74

life cycle

content, 4–5

content access, 88, 89–90

payload, 46

policy, 200–202

list(s)

content types, 65

permissions, 85

SPPFile, 65

ListInstance, 158–159

ListItem, 334

ListItemValidator, 327, 332, 342, 350

Load, 144

LoadPostData, 227

Location, 104

location, 108

location-based records management systems, 2

LogicalExpression, 328–329

Lookup, 177

M

mailbox policy, 366–367

Managed By, 105

managed content, 363–366

Managed Folder Assistant, 357, 367–368

managed folders, 357–363

Management of Electronic Records (MoReq), 20

Match, 326–327, 328

Matter, 125–126

maxOccurs, 327

m_customData, 225

Messaging Records Management (MRM), 355–369

metadata

content type definition, 47

content types, 42, 43

document libraries, 40

documents, 4

encapsulation, 42

holding zone, 103

official records, 2

RecordsRepositoryProperties, 318

SPListItemVersion, 73

users, 11

validation, 391–395

XML, 176

Microsoft Office SharePoint Server (MOSS)

Auditing Policy, 255

content types, 45

documents, 97

Records Management, 97–115

Records Repository, 14–15

site provisioning, 117

versioning, 65

Microsoft.Office.Interop.Word

.Document, 231

Microsoft.Office.Policy, 92, 198, 217

Microsoft.Office.Policy.dll, 310

Microsoft.Office

.RecordsManagement, 92, 310

Microsoft.Office.RecordsManagement

.Holds, 297

Microsoft.Office.RecordsManagement

.InformationPolicy

IPolicyFeature, 200–201

Microsoft.Office.RecordsManagement

.InformationPolicy.Policy, 198

Microsoft.Office.RecordsManagement

.Internal, 306

Microsoft.Office.RecordsManagement

.PolicyFeatures, 237

Microsoft.Office.RecordsManagement

.PolicyFeatures.Expiration, 239

FeatureId, 244

Microsoft.Office.RecordsManagement

.RecordsRepository, 103, 308, 310

Microsoft.Office.RecordsManagement

.SearchAndProcess, 304–305

Microsoft.SharePoint, 310

Microsoft.SharePoint.OfficialFile, 171

Microsoft.SharePoint

.WorkflowActions, 379–380

Microsoft.SharePoint
 .WorkflowActions.dll, 375
 minOccurs, 327, 328
 MoReq. See Management of Electronic Records
 MOSS. See Microsoft Office SharePoint Server
 moss.development, 363
 Motion, 125–126, 128, 172
 Move, 256, 261
 MRM. See Messaging Records Management
 MSBuild, 21
 multiple records, 179–193

N

Name, 328, 400
 namespace, 44
 nested class, 219
 .NET
 attribute classes, 56
 workflow, 373
 wrapper classes, 33
 XML, 22, 24
 .NET Framework
 extension methods, 72
 Installer, 239
 VSTA, 138
 .NET Reflector, 51–52, 56, 373, 375
 New Managed Custom Folders, 358
 New Project Wizard, 230
 None, 256
 Note, 296

O

Official File Web Service, 92–96, 171
 OfficialFileCore.SubmitFile, 106
 Official Message, 365
 official records, 1–17
 content modeling, 3–14
 file plans, 14–17
 metadata, 2
 workflow, 376–381
 OfficialFileCore, 92
 OfficialFileCore.SubmitFile, 92
 OfficialFileResult.InvalidArgument, 107
 OfficialFileSelect.aspx, 188–189
 OfficialFileSelect.cs, 190–193
 OfficialFileSend.aspx, 187
 OnCustomDataChange, 212, 221

OnInit, 185, 186, 267
 OnSubmit, 316
 OnSubmitFile, 311, 318, 351
 out String, 106
 Outlook 2007, 368

P

PageIndexChanging, 270
 Parameter, 400, 401
 params, 74
 ParserEnabled, 112
 partial wrapper classes, 35
 payload, 43, 44. See also XML
 extensibility, 97
 life cycle, 46
 payload, 219
 PDF, 40
 Period Ending Date, 128
 Period Starting Date, 128
 permission levels, 83–84
 permission manifest, 88
 permissions, 83–87
 Active Directory, 6
 contributor, 105
 dependencies, 86–87
 inheritance, 84
 list, 85
 personal, 85
 records manager, 17
 site, 86
 submission, 172–173
 personal permissions, 85
 physical records, 281–294
 content types, 154, 282–283
 file plans, 154
 folders, 284–286
 ItemAdded, 154–157
 workflow, 292–294
 PhysicalRecord, 287
 PhysicalRecordsFeature, 289–291
 policy
 adherence, 3
 auditing, 255–279
 Barcode, 283–284
 content types, 205
 custom features, 203–232
 identifying, 15–16
 life cycle, 200

policy (continued)

- mailbox, 366–367
- retention, 235–279
- reusable components, 203–216
- utility method, 206
- virtual property, 206
- XML, 16

Policy, 205

- schema, 198–200
- XML, 198

Policy Configuration, 243

policy features, 236

- classes, 217–223
- custom settings, 224–229
- schema, 206–212
- XML, 208

policy resources, 212–216, 236

- schema, 213–214

policy statement, 46, 91

- content types, 198
- Create, 288

PolicyManifest, 205

Press Coordinator, 9

primitive activities, 377–380

PrinerPolicySettings.ascx, 224

print, 216

print monitor add-in, 229–232

PrintControlPolicyNamespace, 218

PrinterPolicyEventReceiver, 219

PrinterPolicyFeature, 218, 220

PrinterPolicyFeature.cs, 228

PrintPolicySettings, 225

ProcessAndReport, 302

processing layer, 103

Processing Results, 181–182, 186

ProcessItem, 305

ProcessList, 79

ProcessListItem, 211, 221

ProfileChange, 256, 261

programmatic, 245

programs

- holds, 302–303
- Records Center, 122–124
- submissions, 177–178
- versioning, 71–82

properties

- demotion, 41–42, 112–115
- documents, 41–42
- promotion, 41–42, 112–115
- standard, 42
- storage, 108–110

Properties, 106

properties, 350

Properties subfolder, 176

ProposalValidationRules.xml, 326–327

Provision, 124

ProvisionAssembly, 103

ProvisionClass, 103

provisioning layer

- Setup, 103
- workflow, 105

R

ReceiverAssembly, 286

ReceiverClass, 286

record(s). See also official records; physical records

- multiple, 179–193
- vital, 406–407

Record Relationship Templates, 407

Record Routing, 105

Record Routing List, 105

Record Routing Table, 104, 129–130, 177

- document libraries, 281
- SPListItem, 347
- storage, 281

Record Type, 14, 125–126

Record Type, 181

RecordCenterAddonPack_SiteWorkflows, 412

RecordCenterAddonPack_Web, 411

RecordCenterRouter, 411

RecordRelationshipsFieldControl, 407

RecordRouting, 177

Records Center

- auditing, 262–276
- building and configuring, 117–170
- components, 103–106
- documents, 11
- e-mail, 363
- E-mail Router, 418–419
- file plans, 146–149
- file processing, 106–115
- holds, 296
- programs, 122–124
- site definition, 102–103
- template, 15
- users, 11

Records Management, 92, 99–100

- custom actions, 100–101
- holds, 295–306
- MOSS, 97–115

Records Pending Submission, 104, 107

- Records Repository, 94, 117–170**
 - MOSS, 14–15
 - populating, 171–194
 - Records Repository Users Group, 103, 104**
 - RecordSeries, 348, 351**
 - RecordSeriesCollection, 103, 105**
 - recordSeriesName, 106**
 - RecordSpecification, 145–146, 167, 246, 322**
 - CreateContentTypes, 249
 - schema, 132–134
 - RecordSpecification**
 - .CreateContentTypes, 150–152
 - RecordSpecification.Execute, 145, 149–150**
 - RecordsRepositoryProperties, 318**
 - RecordsRepositoryProperty, 94, 310, 350**
 - RecordsRepositoryProperty[], 106**
 - RecordsSpecification, 169**
 - RedirectingRouter, 317–322**
 - ref, 108, 308**
 - Register, 210**
 - RejectFile, 309**
 - RemoveFromHold, 380**
 - RemoveHold, 304**
 - Reporting.aspx, 259**
 - reports, 45**
 - auditing, 259–261
 - ResourceType, 214, 239, 240**
 - resultDetails, 106, 309**
 - ResultTemplate, 186–188**
 - retention, 14, 235–279**
 - ReverseCutoff, 415**
 - Rights Management Services (RMS), 12, 90–91**
 - RMS. See Rights Management Services**
 - role/activity modeling, 6–14**
 - activities, 8–12
 - artifacts, 13–14
 - discovery process, 12–13
 - example, 10–12
 - responsibilities, 8–12
 - RMS, 12
 - roles, 7–8**
 - content access, 88–89
 - identifying, 15
 - Router, 322**
 - RouterResult, 309**
 - RouterResult**
 - .SuccessCancelFurtherProcessing, 107
 - RoutingList, 351**
 - RowDataBound, 185**
 - Rule, 327, 328**
 - RuleDesigner, 400**
 - RunInstaller, 215**
- ## S
- SafeControl, 270**
 - <SafeControls>, 396**
 - Sarbanes-Oxley (SOX), 11, 20**
 - SaveDocument, 320**
 - schema**
 - CAML, 48
 - classes, 36–37
 - content types, 48
 - file plans, 132–134
 - FilePlan, 132
 - living specification, 33
 - Policy, 198–200
 - policy features, 206–212
 - policy resources, 213–214
 - RecordSpecification, 132–134
 - validation, 327–330
 - Visual Studio, 33, 36–37
 - XML, 131
 - Schema Editor, 33**
 - SchemaChange, 256, 261**
 - SchemaXml, 55, 209, 213, 214**
 - sealed class, 52**
 - Search, 257, 261**
 - Search & Process, 304–306**
 - Search Keyword, 306**
 - SearchAndProcessItem, 305–306**
 - SecGroupCreate, 261**
 - SecGroupDelete, 261**
 - SecGroupMemberAdd, 261**
 - SecGroupMemberDel, 261**
 - SecRoleBindBreakInherit, 261**
 - SecRoleBindInherit, 261**
 - SecRoleBindUpdate, 261**
 - SecRoleDefBreakInherit, 261**
 - SecRoleDefCreate, 261**
 - SecRoleDefDelete, 261**
 - security, 17. See also permissions**
 - content, 83–91
 - Security Configuration, 197**
 - SecurityChange, 257**
 - Select, 181–182, 186**
 - Self-Validating Proposal, 342–347**
 - Sells, Chris, 36**
 - Send To, 109**
 - SendToOfficialFile.aspx, 171**

- SendToRecordsRepository, 380**
- Sentence, 400**
- sentViaSMTP, 106**
- SequenceActivity, 384**
- serialization**
 - classes, 142–157
 - FilePlan, 142–144
 - wrapper classes, 142–144
 - XML, 202
- sessions, 67**
- SetExpirationPolicy, 380**
- SetHold, 303, 304**
- Setup, 103**
- SetupExchangeFolder, 364**
- Shared Documents, 174**
- Shared Records Repository, 7**
- SharePoint Developer Network, 20**
- SharePoint Farm, 171**
- SharePoint Features Project, 21**
- SharePointContentType, 56, 57–59, 152, 287**
- SharePointList, 24, 289**
- SharePointObject, 24, 56, 204**
- SharePointPolicy, 24, 288**
- SharePointPolicyFeature, 25, 220, 222, 228**
- SharePointPolicyResource, 25, 214–216, 238**
- SharePointRouter, 25, 311, 313**
- SimpleFeature.vstemplate, 26–27**
- SimpleFeature.zip, 31, 32**
- Site, 262**
- Site Collection Policies, 197**
- site definition, 102–103**
- site permissions, 86**
- site provisioning, 117**
- `<site url>/_vti_bin/`
 - `officialfile.asmx, 93`
- Skills Manager, 10**
- Solution Explorer, 140**
- Source Url, 104**
- sourceUrl, 106**
- SOX. See Sarbanes-Oxley**
- SPAudit, 256**
- SPAuditEntry, 259, 265**
- SPAuditEntryGrid, 265, 274**
- SPAuditEventType, 260–261**
- SPAuditItemType, 261**
- SPAuditLocationType, 261**
- SPAuditMaskType, 256**
- SPAuditQuery, 265, 271**
- SPBasePermissions**
 - `.UpdatePersonalWebParts, 85`
- SPContentType, 52, 56**
- SPControlMode.Display, 412**
- SPDocumentLibrary, 41**
- SPFarm, 54**
- SPFeatureReceiver, 100, 287**
- SPFieldMultiChoiceValue, 303, 304**
- SPFile, 65**
 - de-serialization, 167
 - serialization, 144
- SPFileLevel, 74**
- SPFile.SendToOfficialFile, 171**
- SPFileVersion, 73**
- SPGridView, 181, 263**
 - AuditHistory.aspx, 273
 - SPAuditEntryGrid, 265
 - ViewState, 269
- SPItemEventDataCollection, 333, 335**
- SPItemEventProperties, 344**
- SPItemEventReceiver, 24, 64**
- SPList, 24, 40**
 - location, 108
 - ref, 308
 - SPDocumentLibrary, 41
- SPListEx, 25**
- SPListItem, 72, 348**
 - Record Routing Table, 347
 - serialization, 144
 - versioning metrics, 73–74
- SPListItemCollection, 72, 74–79, 303**
 - ECM2007.Versioning, 79
 - SetHold, 304
- SPListItemCollectionEx, 25**
- SPListItemCollectionEx.cs, 74–79**
- SPListItemVersion, 72–73**
- SPLongOperation, 168, 189**
- SPLongOperation.End, 169**
- spreadsheets, 12, 15**
- SPRoleAssignment, 83**
- SPSecurity**
 - `.RunWithElevatedPrivileges, 269`
- SPSite, 54, 222, 264**
- SPTraceView, 21**
- SPWeb, 54, 106, 169**
- SPWebApplicationBuilder, 124**
- SPWebApplicationBuilder.Create, 124**
- SPWeb.CreateDefaultAssociatedGroups, 84**
- SPWebProvisioningProvider, 103**

SQL, 138
standard properties, 42
Start Date, 128
StartWorkflow, 245
statement, 45–46. See also policy statement
Statement, 125–126, 129
static files, 15
storage
 auditing, 258
 properties, 108–110
 Record Routing Table, 281
 RecordSpecification.Execute, 150
 requirements, 17
String, 106
STSADM, 417–418
STSDEV, 21
Subject, 42, 112, 128
submissions
 ASPX, 172
 document libraries, 92, 175, 178
 forms, 137
 InfoPath, 137
 multiple records, 179–193
 Official File Web Service, 95–96
 permissions, 172–173
 programs, 177–178
 Records Repository, 94
 Setup, 103
SubmitFile, 106, 108, 308, 310
SuccessCancelFurtherProcessing, 309
SuccessContinueProcessing, 309
supplemental markings, 407
System.Attribute, 56
System.Configuration.Install, 24, 310
System.Configuration.Installer, 215
System.Management.Automation, 359
System.Web, 217
System.Web.UI.WebControls.WebParts
 .WebPart, 263
 <System.Workflow.ComponentModel
 .WorkflowCompiler>, 396
System.Xml.Serialization.
 .XmlSerializer, 144

T

Task List, 105
Team Builder, 9
TEMPLATE, 224
Template, 182

TemplateField, 186
templates
 Form Designer, 138
 Hold Reports, 302
 InfoPath, 138
 Records Center, 15
 Visual Studio, 25–32
 vstemplate, 26
Text, 177, 296
Text File, 224
TextBox, 225, 227–228
TextBoxPrinters, 225
this, 72
ThisAddIn_Startup, 230
timer job
 DoD 5015.2 Add-On Pack, 413–414
 expiration, 243–244
time-span, 245
Title, 42, 104, 105
ToString(), 250
ToStringEx, 72
TrackingRouter, 313–317
treeview, 55
TrustedPrintersFieldName, 218
Try, 326–327, 328
12 hive, 25, 92, 198
12/TEMPLATE, 179
12/TEMPLATE/FEATURES, 99
12/TEMPLATES/FEATURES/fields, 49
12/TEMPLATE/SiteTemplates/offfile, 102
Type, 329–330
TypeSpecifier, 330

U

ULS tracing, 21
UnaryLogicalExpression, 328
Unclassified Records, 105
Undelete, 257, 261
Uninstall, 215
Unregister, 210
Update, 257, 261, 265
uploads, 66
URL
 Official File Web Service, 171
 virtual paths, 68
U.S. regulations, 19–20
User, 296
user, 177
User Name, 105

userName, 106
using, 72, 79, 218, 314

V

ValidateContent, 312
ValidateListItem, 381
ValidateManifest, 198
ValidateMetadata, 312
Validating Router, 347–353
ValidatingRouter.cs, 352–353

validation

components, 330–341
content, 325–353
metadata, 391–395
schema, 327–330
Self-Validating Proposal, 342–347
Setup, 103
workflow, 391–395
XML, 325–326

ValidationFailed, 340

ValueType, 329

ValueType.cs, 337–340

VB.NET, 33

version history, 65

versioning, 64–82

document libraries, 82
metrics, 72–82
MOSS, 65
programs, 71–82
SPFile, 65

VersioningTestConsole.cs, 79–82

View, 257, 261

ViewState, 269

virtual methods, 209

virtual paths, 68

virtual property, 206

Visio, 6, 12

workflow, 382

Visual Studio, 20

.ASCX, 224–225
custom tools, 36–37
dependencies, 384
file plans, 140
Forms Designer, 138
IntelliSense, 48
schema, 33, 36–37
template, 25–32
12 hive, 25

WSS.XSD, 22
XML, 32–33
XSD.EXE, 33, 142

Visual Studio Extensions for SharePoint, 21
Visual Studio Tools for Office (VSTO), 138, 216

vital record indicator, 416

Vital Record Review, 416–417

vital records, 406–407

VSTA, 138

vstemplate, 26

VSTO. See **Visual Studio Tools for Office**

W

Web, 106

Web Components, 410

web parts

auditing, 262–271
personal permissions, 85
Setup, 103

Web Service Definition Language (WSDL), 171

Web Services (WS), 11, 138, 356

WebTemplateId, 289

WEBTEMPOFFICE.XML, 102

Windows PowerShell, 358, 359

Windows SharePoint Services (WSS), 21, 39–97
site definition, 102

WizardExtension, 28

Word, 14, 40, 67, 112–113, 229–232

workflow, 16, 105, 371–401

Cutoff Approval, 414–415
Designer, 6
Disposition Approval, 105
Disposition Approval Transfer, 416
DoD 5015.2 Add-On Pack, 414–417
ECM, 372
modeling, 381–382
.NET, 373
official records, 376–381
physical records, 292–294
primitive activities, 377–380
Setup, 103
validation, 391–395
Visio, 382
Vital Record Review, 416–417

Workflow Activity Library, 383–386

Workflow Design Wizard, 396–401

WorkflowExpirationAction, 248

wrapper classes, 142

Employee, 33–36
 .NET, 33
 partial, 35
 serialization, 142–144

WS. See Web Services

WSDL. See Web Service Definition Language

WSP Builder, 21

WSS. See Windows SharePoint Services

WSSACTIONS.XSD, 397–401

WSS=prefix, 310

wssTask, 380

WSS.XSD, 22

X

XML, 13–14, 16, 198

AuditEntry, 272
 CAML, 47
 content types, 47, 152
 customData, 200
 DataTable, 274
 de-serialization, 33
 document libraries, 40
 File Plan, 141
 GetRecordRoutingCollection, 95
 InfoPath, 131, 138, 141
 for maximum flexibility, 32–36

metadata, 176
 namespace, 44
 .NET, 22, 24
 OnCustomDataChange, 212
 policy features, 208
 property storage, 108
 Schema Editor, 33
 serialization, 202
 validation, 325–326
 virtual property, 206
 Visual Studio, 32–33

XMLDocument, 33, 219, 331, 344

XPath, 33

XPathNodeIterator, 95

xs:annotation, 22

xs:choice, 246

.xsd, 33

XsdClassGenerator, 36, 142, 167

xsd:dateTime, 176

XSD.EXE, 33, 142, 322, 332

XSD.exe, 246

xs:enumeration, 330

XSL, 134

XSLT, 49–51, 260

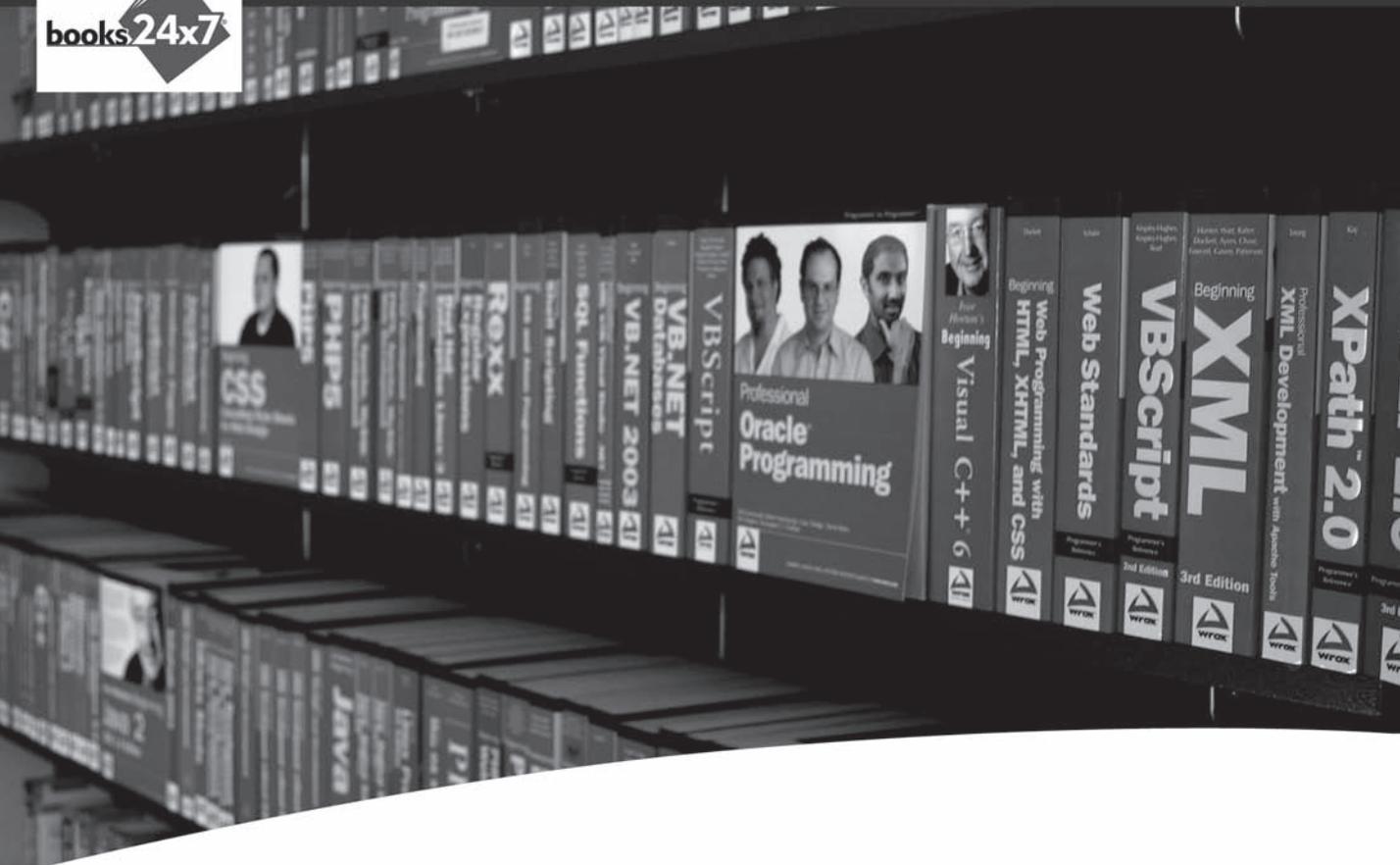
xs:sequence, 328

xs:string, 22–23, 327

powered by

books24x7

Programmer to Programmer™



Take your library wherever you go.

Now you can access more than 200 complete Wrox books online, wherever you happen to be! Every diagram, description, screen capture, and code sample is available with your subscription to the **Wrox Reference Library**. For answers when and where you need them, go to wrox.books24x7.com and subscribe today!

Find books on

- ASRNET
- C#/C++
- Database
- General
- Java
- Mac
- Microsoft Office
- .NET
- Open Source
- PHP/MySQL
- SQL Server
- Visual Basic
- Web
- XML



www.wrox.com

Related Wrox Books

Beginning SharePoint 2007 Administration:

Windows SharePoint Services 3.0 and Microsoft Office SharePoint Server 2007

ISBN: 978-0-470-12529-8

SharePoint MVP Göran Husman walks you through everything from planning and installation to configuration and administration so you can begin developing a production environment.

Beginning SharePoint 2007: Building Team Solutions with MOSS 2007

ISBN: 978-0-470-12449-9

This book provides detailed descriptions and illustrations of the functionality of SharePoint as well as real-world scenarios, offering coverage of the latest changes and improvements to Microsoft Office SharePoint Server 2007.

Building Dashboards for Windows SharePoint Services 3.0 using SharePoint Designer 2007

ISBN: 978-0-470-48566-8

In this e-book only downloadable Wrox Blox, you'll learn how to create powerful Dashboards for Windows SharePoint Services 3.0. It introduces Web Part Pages and out-of-the box Web Parts available in WSS, how to use Web Part Connections to add interactivity in Dashboards, and you'll create advanced Dashboard Views using the Data Form Web Part available with SharePoint Designer 2007.

Professional SharePoint 2007 Design

ISBN: 978-0-470-28580-0

This book outlines all of the steps and considerations a developer should understand in order to design better looking and more successful SharePoint implementations.

Professional SharePoint 2007 Development

ISBN: 978-0-470-11756-9

A thorough guide highlighting the technologies in SharePoint 2007 that are new for developers, with special emphasis on the key areas of SharePoint development: collaboration, portal and composite application frameworks, enterprise search, ECM, business process/workflow/electronic forms, and finally, business intelligence.

Professional Microsoft SharePoint 2007 Reporting with SQL Server 2008 Reporting Services

ISBN: 978-0-470-48189-9

Build customized reports quickly and efficiently with SQL Server 2008 Reporting Services for SharePoint sites and this unique guide. Developers, you'll learn report development and deployment; SharePoint or SQL Server Reporting Services administrators, you'll see how to leverage SharePoint to use SQL Server Reporting Services in SharePoint Integrated Mode.

Professional SharePoint 2007 Web Content Management Development

ISBN: 978-0-470-22475-5

Use this book to learn such things as optimal methods for embarking on web content management projects, ways to implement sites with multiple languages and devices, the importance of authentication and authorization, and how to customize the SharePoint authoring environment.

Real World SharePoint 2007: Indispensable Experiences from 16 MOSS and WSS MVPs

ISBN: 978-0-470-16835-6

This anthology of the best thinking on critical SharePoint 2007 topics is written by SharePoint MVPs—some of the best and most recognized experts in the field. Some of the topics they cover include: Branding, Business Data Connector, Classified Networks, Forms-based Authentication, Information Rights Management, and Zones and Alternate Access Mapping.

Create successful records management solutions from the start

Building effective records management solutions on the SharePoint platform requires a comprehensive development strategy that integrates out-of-the-box components (document libraries, content types, and information policy) with more specialized custom components that can be applied at any stage of the content lifecycle. This helpful book introduces content modeling as an integral part of the development process and guides you through the creation of a reusable set of enterprise content management components that are designed specifically for records management.

- Shows you how to quickly create and manage information policies to control document retention, build custom routers, manage electronic mail records, and more
- Walks you through creating executable file plans that can be applied to multiple records management scenarios
- Introduces invaluable techniques for setting up an effective records management development environment
- Addresses methods for ensuring that incoming records have valid content and metadata
- Reviews building a custom workflow activity library with built-in support for records management
- Offers advice for installing and configuring the DoD 5015.2 Addon Pack and demonstrates how to use it to deploy DoD 5015.2 compliant solutions

John Holliday is an independent consultant and Microsoft MVP for Office SharePoint Server 2007 with more than 25 years of professional software development and consulting experience.

Wrox Professional guides are planned and written by working programmers to meet the real-world needs of programmers, developers, and IT professionals. Focused and relevant, they address the issues technology professionals face every day. They provide examples, practical solutions, and expert education in new technologies, all designed to help programmers do a better job.

Wrox
An Imprint of
 **WILEY**

Business Applications/Microsoft Office

\$49.99 USA
\$59.99 CAN



wrox.com

Programmer Forums

Join our Programmer to Programmer forums to ask and answer programming questions about this book, join discussions on the hottest topics in the industry, and connect with fellow programmers from around the world.

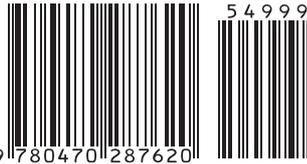
Code Downloads

Take advantage of free code samples from this book, as well as code samples from hundreds of other books, all ready to use.

Read More

Find articles, ebooks, sample chapters and tables of contents for hundreds of books, and more reference resources on programming topics that matter to you.

ISBN 978-0-470-28762-0
5 4 9 9 9



9 780470 287620