Microsoft

Foreword by Andrew Bybee
Principal Program Manager, Microsoft Dynamics CRM

Programming Microsoft Dynamics® CRM 4.0

Jim Steger, Mike Snyder, Brad Bosak, Corey O'Brien, Philip Richardson

sonoma PARTNERS

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Library of Congress Control Number: 2008935422

Printed and bound in the United States of America.

1 2 3 4 5 6 7 8 9   QWT   3 2 1 0 9 8

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Acquisitions Editor: Ben Ryan
Developmental Editor: Devon Musgrave
Project Editor: Lynn Finnel
Editorial Production: ICC Macmillan, Inc.
Technical Reviewer: Elliot Lewis; Technical Review services provided by Content Master, a member of CM Group, Ltd.
Cover: Tom Draper Design

Body Part No. X14-95064

Contents at a Glance

Part I **Overview**

1 Microsoft Dynamics CRM 4.0 SDK Overview 3

2 Development Overview and Environment 19

3 Communicating with Microsoft CRM APIs 49

Part II **Extending Microsoft CRM**

4 Security. 79

5 Plug-ins. 109

6 Programming Workflow 169

7 Form Scripting. 233

8 Developing with the Metadata Service 271

Part III **Advanced Topics**

9 Deployment. 313

10 Developing Offline Solutions 347

11 Multilingual and Multi-Currency Applications. 375

12 Advanced Workflow Programming 407

13 Emulating User Interface with ASP.NET Development 449

14 Developing Custom Microsoft CRM Controls 493

15 Additional Samples and Utilities. 549

Table of Contents

Foreword xv

Acknowledgments xvii

Jim Steger's Acknowledgments xviii

Mike Snyder's Acknowledgments xviii

Brad Bosak's Acknowledgments xviii

Corey O'Brien's Acknowledgments xviii

Philip Richardson's Acknowledgments xix

Introduction xxi

Who This Book Is For xxi

What This Book Is About xxii

Companion Content xxiii

System Requirements xxiv

Client xxiv

Server xxiv

Find Additional Content Online xxiv

Support for This Book xxv

Questions and Comments xxv

Part I **Overview**

1 Microsoft Dynamics CRM 4.0 SDK Overview 3

Software Development Kit Introduction 4

Hitchhiker's Guide to Common Questions 5

Can we alter the CRM database structure to add our custom tables and columns? 5

How do we write custom code that gets data into and out of Microsoft Dynamics CRM? 7

Can we change the current CRM form layouts and controls? 8

How do we implement our own custom business logic? 9
How much control do we have over the user interface and branding? 12
How do we deploy changes from one system to another? 13
Will our customizations upgrade when Microsoft releases a new version of the software? 14
Are role-based security permissions supported and configurable? 15
Does Microsoft Dynamics CRM support multiple languages and currencies?. 15
Will our programming customizations run offline?. 16
How do you recommend we set up a Microsoft Dynamics CRM development environment?. 16
Microsoft Dynamics CRM as a Business Application Platform 17
Summary 18

2 Development Overview and Environment 19

Microsoft Dynamics CRM 4.0 System Overview 19
Microsoft Dynamics CRM 4.0 Versions. 20
Microsoft Dynamics CRM Components 20
Licensing 21
Server Roles. 22
Hardware Requirements 23
Microsoft Dynamics CRM 4.0 Web Server Requirements 23
Virtualization 24
Development Environment. 24
Server Environment Development Options. 25
Developing Customizations Accessed From Outlook 32
Testing Environment Considerations 33
Migrating Data 34
Redeployment 36
Development Tools 40
Integrated Development Environment (IDE). 41
Source Control 41
Continuous Integration Tool 42
Installer 42
Additional Development Utilities 45
Additional Development Considerations 46
Summary 47

3 Communicating with Microsoft CRM APIs 49

Overview of the Microsoft Dynamics CRM 4.0 SDK 49
Accessing the APIs in Visual Studio 2008 50
CrmService Web Service 54
Authentication 56
Common Methods 58
Execute Method 62
Request and *Response* Classes 63
DynamicEntity Class 65
Attributes 66
MetadataService Web Service 66
CrmDiscoveryService Web Service 67
Connecting to Microsoft Dynamics CRM IFD 69
Connecting to Microsoft Dynamics CRM Offline 70
Connecting to Microsoft Dynamics CRM Online 70
Summary 75

Part II Extending Microsoft CRM

4 Security 79

Role-Based and Object-Based Security 80
Security Principals 81
Access Rights 81
Impersonation 82
Impersonation for Web applications 82
Impersonation for Plug-ins 83
Using the Microsoft Dynamics CRM SDK to Perform
Security-Related Operations 85
Programmatically Creating a Security Role and
Adding Privileges 85
Programmatically Assigning a Security Role 87
Retrieving Roles and Privileges 88
Sharing Records 90
Assigning Records 93
Using Direct SQL for Accessing Microsoft Dynamics CRM Data 94
Determining the Organization's Connection String 95
Filtered Views 96
Additional Programming Considerations with Filtered Views 98

Data Encryption 100
One-Way Encryption 100
Two-Way Encryption 103
Summary 107

5 Plug-ins. 109
Writing Your First Plug-in 110
Creating the Plug-in Project. 110
Implementing the Plug-in Class 111
Building the Registration Tool 113
Deploying the Plug-in. 113
The Event Execution Pipeline 120
Supported Messages and Entities 120
Parent and Child Pipelines 121
IPluginExecutionContext 122
Impersonation. 126
Impersonation During Registration 126
Impersonation During Execution 127
Exception Handling 127
Exceptions and the Event Processing Pipeline 127
Exception Feedback. 127
Deployment 128
Plug-in Entities 128
Programmatic Plug-in Registration 128
Images 138
Programmatic Image Registration 139
Custom Configuration. 147
Deploying Referenced Assemblies 147
Debugging Plug-ins. 149
Remote Debugging 149
Logging 149
Unit Testing 150
Mock Objects 150
Test Frameworks. 151
Sample Test. 151
Sample Plug-ins 156
Rolling Up Child Entity Attributes to a Parent Entity. 156
System View Hider. 161
Customization Change Notifier. 162
Summary 168

6 Programming Workflow 169
Overview 170
The Workflow Designer. 170
Testing the Workflow 176
Workflow Definitions. 178
Dynamic Values in Workflow 181
Windows Workflow Foundation 185
Activities 186
Dynamic Value Binding. 187
Custom Workflow Activities 188
Implementing a Custom Activity. 188
Deploying the Custom Assembly 193
Testing the Custom Activity 197
Investigating Custom Activity Errors 201
CRM Workflow Attributes. 203
Workflow Context 204
Entity Images in Custom Workflow Activities 206
Workflow Designer Limitations 213
Example Activities: Math Building Blocks. 214
Example Activity: Retrieve Most Available User 216
Example Activity: Calculate Related Aggregate 221
Summary 231

7 Form Scripting. 233
Form Scripting Overview 234
Understanding Client-Side Scripting with Microsoft Dynamics CRM 234
Referencing Microsoft Dynamics CRM Elements 234
Available Events 238
Calling Web Services from Script. 241
Using the CRM API SOAP Request from Script 241
Testing and Deployment. 244
Scripting from ISV.Config Buttons. 248
Advanced Topics. 253
Form Type Considerations 253
Cross-Site Scripting (XSS) 255
Using Script for Validation 257
Loading External JavaScript Files. 259

Scripting Examples 261
Display Customer Information in a ToolTip 262
Setting the Form's Title at Run Time 265
Enhancing the Form's Display 266
Summary 269

8 Developing with the Metadata Service 271

Connecting to the *MetadataService* 273
Referencing the *MetadataService* 273
Locating the Endpoints 273
Microsoft Dynamics CRM Security 277
Retrieving Metadata 278
Names Used in the *MetadataService* 279
Retrieving Entities 279
Retrieving Attributes and Relationships 282
Multilingual Strings 285
Remote Customization 288
Entities 288
Attributes 292
Relationships 298
Publishing Metadata 304
Caching the Metadata 304
Using the Timestamp 304
Example File System Cache 305
Handling Errors 308
Summary 309

Part III **Advanced Topics**

9 Deployment 313

Common Deployment Steps 313
Build 314
Install 315
Configure 317
Uninstall 319
Deploying Microsoft Dynamics CRM Components 320
Customizations 321
Workflow Rules, Templates, and Security Roles 323

User Interface Changes (Forms, ISV.Config, and Sitemap) 324
Custom Web Pages 325
Plug-in and Workflow Assemblies 326
Online Help 328
Custom Reports 329
Configuration Settings 330
Offline Application Deployment 330
Offline Web Applications 330
Plug-in Allow List 331
Testing Strategies 333
Server Topologies 334
Operating Systems. 334
Web Browsers. 334
Database Servers 335
Authentication 335
Multi-Tenant 335
Multilingual. 336
Accessibility. 336
Additional Deployment Considerations 340
Missing Prerequisites. 341
Resetting IIS 343
Customizations Management 343
Example Deployment Sequence 343
Component Inventory. 344
Build 344
Install 344
Configuration 345
Uninstall. 345
Summary 346

10 Developing Offline Solutions 347
Overview 348
Offline Development Environment 349
Workstation 349
Laptop 350
Virtual PC 350
Offline Navigation 350

Communicating with the Microsoft Dynamics CRM SDK API Offline 352
CrmService Offline 352
MetadataService Offline 353
Microsoft Dynamics Outlook SDK. 354
Scripting for Offline 355
Developing IFrames for Offline 356
Programming the IFrame 356
Deploying the IFrame Page 361
Testing the IFrame Page Offline 365
Developing an Offline Plug-in 367
Programming the Plug-in. 367
Deploying the Plug-in. 371
Testing the Plug-in. 371
Offline Development Considerations 372
Summary 373

11 Multilingual and Multi-Currency Applications 375
Programming for Multilingual Applications 375
CrmService Messages for Multilingual Support 376
Using the Metadata Service for Multilingual Applications. 385
Using Resource Assemblies for Multilingual Strings 395
Programming for Multi-Currency Applications 403
Summary 405

12 Advanced Workflow Programming 407
Custom Workflow Manager Tool. 407
Workflows as Entities. 409
The *workflow* Entity 409
Interacting with Workflows Through *CrmService*. 411
Workflow Publication 412
The *workflowdependency* Entity 413
Retrieving Workflow Dependencies 418
Declarative Workflows. 419
XAML Syntax. 419
XAML in Microsoft Dynamics CRM. 420
Creating Your First Declarative Workflow 421
Declarative Workflow Deployment. 429

Examining Native Workflow XAML. 442
Exporting Workflows Programmatically 444
Summary 448

13 Emulating User Interface with ASP.NET Development 449
IFrame Development 450
Programming the IFrame 451
Deploying and Testing the IFrame 460
Dialog Box Development 464
Programming the Dialog Box 465
Deploying and Testing the Dialog Box. 476
ISV Page Development 478
Programming the ISV Page 479
PopulateLeadTable and *PopulateOpportunityTable* 489
PopulateActivityTable Method. 490
Deploying and Testing the ISV Page 490
Summary 491

14 Developing Custom Microsoft CRM Controls 493
Overview 494
CrmPicklistControl 496
Programming the *CrmPicklistControl* 496
Testing the *CrmPicklistControl* 502
CrmBooleanControl 503
Programming the *CrmBooleanControl*. 503
Testing the *CrmBooleanControl*. 506
CrmDateTimeControl 507
Programming the *CrmDateTimeControl* 507
Testing the *CrmDateTimeControl* 515
CrmEntityPicklistControl 516
Programming the *CrmEntityPicklistControl* 516
Testing the *CrmEntityPicklistControl*. 521
CrmGridViewControl 523
Programming the *CrmGridViewControl*. 523
Testing the *CrmGridViewControl*. 545
Summary 548

15 Additional Samples and Utilities 549
Utility Classes 549
MetadataUtility Class 552
Customizations Utility 553
Additional Script Samples 556
Conditionally Enabling Attributes 556
Hiding Navigation Links on a Form 558
Field-Level Security 563
Hiding Tabs and Fields on a Form 564
Field-Level Security Script 566
Field-Level Security Script Example 572
Summary 580

Index 581

Foreword

Welcome to the world of developing business solutions with Microsoft Dynamics CRM!

For a long time, professional developers building business applications have been forced to choose between two equally unappealing alternatives when designing their solution—either buy an off-the-shelf package and have their hands tied with closed, proprietary designs; or build their own solution from scratch using commonly available technology and spend the majority of the project implementing the "basics" (again!) such as storage, security, and a user interface framework.

Microsoft Dynamics CRM is committed to providing a third way—a flexible architectural model that combines the power of the Microsoft platform with the appeal of a familiar Microsoft Office–style user experience and configurable business process. More important, although the power and value of Microsoft Dynamics CRM are most easily applied to sales, service, and marketing scenarios, the product's capabilities easily provide a platform for enabling a wide range of business processes and applications.

Put simply, Microsoft Dynamics CRM makes delivering the basics easy and lets you apply your energy and creativity where it matters the most—solving unique problems and helping the business succeed with intelligent solutions.

In the end, the most important element of any business application development project is *you*—the developer. With this book, the authors make it easy for you to benefit from their years of practical experience working with customers and other Microsoft partners to deliver high-value CRM solutions. They explain what you need (and want) to know before you start in on your Microsoft Dynamics CRM development project and how to get the most out of the time that you spend. Their ability to provide clear, concise guidance across the entire range of developer capabilities in CRM is a tremendous asset for anyone building custom solutions with the product.

If you're just getting started as a developer working with CRM, this book will give you the strong foundation in the core architecture, processes, and development capabilities to be a great Dynamics CRM developer.

If you've already had some experience with the product, this book is a handy reference that provides ideas and samples to stimulate your own creativity and help tackle common challenges.

I hope you find this guide both as informative and useful to read as I have during my collaboration with the authors. In the end, this book is just a first step. Whether you enjoy building business applications for the technical challenge or for the opportunity to help the world run a little smoother, I'm confident that Microsoft Dynamics CRM can help you reach your project goals faster and more effectively.

Welcome to the next generation of business application development—happy coding!

Andrew Bybee
Principal Program Manager
Microsoft Dynamics CRM
Microsoft Corporation

Acknowledgments

We want to thank all of the people who assisted us in writing this book. If we accidentally missed anyone, we apologize in advance. We would like to extend a special thanks to the following people:

- **Elliot Lewis** Elliot served as our technical reviewer for the book. Elliot's keen eye helped refine the book's approach and ensured the accuracy of its contents. We are all very appreciative of the effort and feedback Elliot provided.
- **Andy Bybee** Andy provided overall guidance and support for the book within Microsoft. He also was gracious enough to provide the book's foreword.

In addition, we want to thank these members of the Microsoft Dynamics CRM product team who helped us at one point or another during the book project:

Kam Baker	Steven Kaplan	Dominic Pouzin
Andrew Becraft	Jeff Kelleran	Manisha Powar
Rohit Bhatia	Amit Kumar	Michael Scott
Matt Cooper	Donald La	Nirav Shah
Jim Daly	Amy Langlois	John Song
Rich Dickinson	Chris Laver	Derik Stenerson
Ajith Gande	Patrick Le Quere	Craig Unger
Barry Givens	Dinesh Murthy	Praveen Upadhyay
Humberto Lezama Guadarrama	Kevin Nazemi	Mahesh Vijayaraghavan
Nishant Gupta	Michael Ott	Sumit Virmani
Allen Hafezipour	Ramanathan Pallassana	Brad Wilson
Peter Hecke	Irene Pasternack	Charlie Wood
Akezyt Janedittakarn	Dave Porter	Tobin Zerba

Thank you to the following Sonoma Partners colleagues who assisted with reviewing the content and providing feedback:

Brian Baseggio	Bob Lauer	Matt Spezzano
Matt "MattDawg" Dearing	Andy Meyers	Matt Weiler
Jeff Klosinski	Blake Scarlavai	

Of course, we also want to thank the folks at Microsoft Press who helped support us throughout the writing and publishing process:

- **Ben Ryan** Ben again championed the project and was an invaluable resource for the logistics and planning.

- **Devon Musgrave** Devon provided initial review for the book and provided insight and direction with the book's schedule.
- **Lynn Finnel** Lynn, our project editor, provided the day-to-day guidance and coordination of the editing process.

We also wanted to extend our thanks to the rest of the production team who provided editorial feedback.

Jim Steger's Acknowledgments

I wish to thank my wife, Heidi, for her patience and for continuing to support me during this arduous process again. I want to thank both of my children, who continue to grow, impress, and motivate me. I also received input from numerous members of the Microsoft Dynamics CRM product team, and I want to extend my thanks to them as well. Finally, I wish to express my gratitude to my associates at Sonoma Partners who really stepped up their efforts and understanding while I was forced to prioritize my writing over some of my day-to-day duties.

Mike Snyder's Acknowledgments

I want to thank my wife, Gretchen, who supported me during this project. Writing this book required a significant time commitment above and beyond my normal work responsibilities, but Gretchen remained supportive from start to finish. I want also to thank my children for not deleting my completed work as they learned to play games on daddy's computer! I want to recognize my parents and my wife's parents who assisted my family with various babysitting stints. Finally, thanks to all of my coworkers at Sonoma Partners who allowed me the time and understanding to work on this book.

Brad Bosak's Acknowledgments

I would like to thank my family and friends for being supportive and understanding during the writing process. I'd also like to thank my coworkers at Sonoma Partners for their patience during the busy work days and also for their input and ideas. Finally, I'd like to thank Mike and Jim for the opportunity to help write this book.

Corey O'Brien's Acknowledgments

I would like to thank my wife, Pilar, for supporting me during the writing of this book. She tirelessly took care of our newborn son, Dylan, throughout the late nights and weekends while I was writing. I'd also like to thank my parents and my wife's parents for happily helping with babysitting duties whenever we asked. I'd also like to thank all of my coworkers at

Sonoma Partners for understanding that the growling was due to lack of sleep and not anything they'd done wrong.

Philip Richardson's Acknowledgments

I'd like to personally thank the CRM customers and partners who are a constant source of inspiration for the Dynamics CRM team at Microsoft. During my tenure on the CRM team they helped fuel my passion for the product with constant e-mails, instant messages, and meetings at various conferences. On a personal note, I'd like to thank my wife, Ellie, who is ever supportive of my profession regardless of the unusual working hours which it demands. Finally, I'm ever appreciative of Sonoma Partners for asking me to contribute to this book.

Introduction

If your organization has customers, you need a software system to help you manage your customer information. Unfortunately, many companies today are stuck using antiquated customer systems that don't integrate with Microsoft Office Outlook, aren't available from the Web, and can't be accessed via mobile devices. Even worse, some companies rely on Outlook contacts and Microsoft Office Excel files for precious customer data, making team collaboration on these records very difficult.

You probably already know that Microsoft offers a Customer Relationship Management (CRM) software solution as part of its Dynamics family. Microsoft Dynamics CRM is an easy-to-use application that businesses of all sizes and types can utilize. One of Microsoft Dynamics CRM's most important benefits is its native integration with other Microsoft productivity tools: Outlook, Excel, and Microsoft Office Word. Microsoft Dynamics CRM allows organizations to manage their sales, marketing, and customer service information more efficiently, leading to higher sales revenue and improved customer satisfaction.

But just as important as Microsoft Dynamics CRM's integration with other Microsoft tools, Microsoft Dynamics CRM offers developers a powerful customization and programming platform that you can use to satisfy almost any business requirement. This book provides a detailed explanation of the key areas in the Software Development Kit (SDK) and the Web-service–based Application Programming Interfaces (APIs). This book includes plenty of code samples and examples on topics such as form scripting, plug-ins, workflow assemblies, customizing the user interface, and more.

Programming Microsoft Dynamics CRM 4.0 was written by the consulting firm Sonoma Partners. Our firm has written several other successful titles for Microsoft Press, such as *Microsoft Dynamics CRM 4.0 Step by Step* (2008) and *Working with Microsoft Dynamics CRM 4.0, Second Edition* (2008). We tried to bring our real-world customer experiences to the writing process and share the most relevant information we think you'll need to program with the latest version of Microsoft Dynamics CRM 4.0.

Who This Book Is For

We wrote this book for professional developers who want to use the Microsoft Dynamics CRM SDK and its APIs to extensively customize the software application. We assume that you're comfortable working with .NET solutions and Web services. In addition, we also assume that you have a basic understanding of how to navigate the Microsoft Dynamics CRM interface and you understand its configuration capabilities. If you're looking for a detailed explanation of the Web-based configuration tools that Microsoft Dynamics CRM offers, please refer to *Working with Microsoft Dynamics CRM 4.0, Second Edition*, which explains these topics in great detail. If you're brand new to Microsoft Dynamics CRM, and you want to learn how to navigate through the user interface (from an end-user perspective), you can refer to *Microsoft Dynamics CRM 4.0 Step by Step*, which explains various day-to-day tasks such as creating accounts, logging a phone call, tracking an e-mail, and so on.

What This Book Is About

We divided this book into 15 chapters:

Chapter 1, "Microsoft Dynamics CRM 4.0 SDK Overview," introduces the Microsoft Dynamics CRM Software Development Kit (SDK) and outlines the most common questions that developers might ask about developing within Microsoft Dynamics CRM.

Chapter 2, "Development Overview and Environment," provides information about the various software editions and looks at the unique Microsoft Dynamics CRM issues related to setting up a development environment.

Chapter 3, "Communicating with Microsoft CRM APIs," explains how to programmatically connect with the Microsoft Dynamics CRM APIs. This chapter also covers how you connect to the APIs in the various deployment options: on-premise, Internet-facing, and Microsoft Dynamics CRM Online.

Chapter 4, "Security," supplies information about how your custom code interacts with the Microsoft Dynamics CRM security model. This chapter also takes a look at using custom code to encrypt specific data attributes.

Chapter 5, "Plug-ins," offers a detailed look at the Microsoft Dynamics CRM plug-in model. This includes creating the project, registering the plug-in, deploying the plug-in, and then working with the *IPluginExecutionContext*.

Chapter 6, "Programming Workflow," examines the Microsoft Dynamics CRM workflow module and how it takes advantage of the Windows Workflow Foundation. More important, this chapter explains how you can create your own custom workflow activities that you can reference in Microsoft Dynamics CRM workflow rules.

Chapter 7, "Form Scripting," explains the client-side scripting model. The chapter also provides examples of how you can create custom client-side code that calls Web services, run scripts from ISV.Config buttons, and so on.

Chapter 8, “Developing with the Metadata Service,” explains the Microsoft Dynamics CRM *MetadataService*, and how you can use this API to programmatically retrieve and modify data about the system schema.

Chapter 9, “Deployment,” explains various topics related to deploying your Microsoft Dynamics CRM solution from one environment to another.

Chapter 10, “Developing Offline Solutions,” outlines the nuances of writing custom code that works properly using Microsoft Dynamics CRM for Outlook with Offline Access.

Chapter 11, “Multilingual and Multi-Currency Applications,” offers a look at how to use Microsoft Dynamics CRM’s multilingual and multi-currency functionality within your custom code to support global deployments.

Chapter 12, “Advanced Workflow Programming,” goes deeper into programming the Microsoft Dynamics CRM workflow functionality, and explains how you can create custom workflow activities with XAML.

Chapter 13, “Emulating User Interface with ASP.NET Development,” shows how you can create custom Web pages and user interfaces that blend seamlessly into the out-of-the-box Microsoft Dynamics CRM user interface, which provides a better end-user experience for your organization.

Chapter 14, “Developing Custom Microsoft CRM Controls,” provides examples of creating custom user controls that reference Microsoft Dynamics CRM data.

Chapter 15, “Additional Samples and Utilities,” discusses some of the utility classes and code used in the previous chapters as well as providing additional examples using the Microsoft Dynamics CRM technologies.

Companion Content

This book features a companion Web site that makes available to you all the code used in the book. This code is organized by chapter, and you can download it from the companion site at the following URL: *http://www.microsoft.com/mspress/companion/9780735625945*.

System Requirements

We recommend that you refer to the Microsoft Dynamics CRM Implementation Guide for detailed system requirements. From a high level, you'll need the following hardware and software to run the code samples in this book:

Client

- Microsoft Windows XP with Service Pack 2 (SP2) or the Windows Vista operating system
- Microsoft Internet Explorer 6 SP1 or Internet Explorer 7
- Microsoft Visual Studio 2005 or Microsoft Visual Studio 2008 (for the code samples)
- Microsoft Office 2003 with SP3 or the 2007 Microsoft Office System with SP1 (if you want to use Microsoft Dynamics CRM for Microsoft Office Outlook)

Server

- Microsoft Windows Server 2003 or Microsoft Windows Small Business Server 2003
- Microsoft SQL Server 2005
- Microsoft Dynamics CRM 4.0 Server license (Workgroup, Professional, or Enterprise edition)
- Computer/processor: Dual 1.8-gigahertz (GHz) or higher Pentium (Xeon P4) or compatible CPU
- Memory: 1 gigabyte (GB) of RAM minimum, 2 GB or more of RAM recommended
- Hard disk: 400 megabytes (MB) free space
- Network card: 10/100 Mbps minimum, dual 10/100/1000 Mbps recommended

Find Additional Content Online

As new or updated material becomes available that complements your book, it will be posted online on the Microsoft Press Online Developer Tools Web site. The type of material you might find includes updates to book content, articles, links to companion content, errata, sample chapters, and more. This Web site is available at *www.microsoft.com/learning/books/online/developer*, and is updated periodically.

Support for This Book

Every effort has been made to ensure the accuracy of this book and the contents of the companion Web site. As corrections or changes are collected, they will be added to a Microsoft Knowledge Base article.

Microsoft Press provides support for books and companion content at the following Web site:

http://www.microsoft.com/learning/support/books/

Questions and Comments

If you have comments, questions, or ideas regarding the book or the companion Web site, or questions that are not answered by visiting the sites above, please send them to Microsoft Press via e-mail to

mspinput@microsoft.com

Or via postal mail to

Microsoft Press
Attn: *Programming Microsoft Dynamics CRM 4.0* Editor
One Microsoft Way
Redmond, WA 98052-6399

Please note that Microsoft software product support is not offered through the above addresses.

Chapter 1

Microsoft Dynamics CRM 4.0 SDK Overview

You are probably reading this book because your organization recently purchased Microsoft Dynamics CRM or because your organization is evaluating it. As a developer, you want to know what this new software application will mean to your day-to-day life. Will it cause you nightmares and sleepless nights? Or will it be a dream to work with and solve all your current development headaches? As you might guess, the true answer lies somewhere in between. However, we strongly believe that if you take the time to learn the Microsoft Dynamics CRM application, you will find yourself much closer to the latter. If you're new to Microsoft Dynamics CRM, your initial questions might include the following:

- Will the software limit what I can do?
- How do I customize and extend the software?
- What types of resources are available to help me with the software?

We wrote this book to explain how professional software developers can extend the Microsoft Dynamics CRM software application to meet their business needs. To create customizations and integrations outlined in this book, you must be comfortable developing Web-based applications using tools such as Microsoft Visual Studio. We assume you have working knowledge of Visual Studio and Web application configuration with Microsoft Internet Information Services (IIS). Even if you're not a developer, you might benefit from reading these chapters to understand the different types of customizations that the Microsoft Dynamics CRM programming model makes possible.

From a very high level, Microsoft Dynamics CRM is just a large and sophisticated Web application. The application serves Web pages through IIS while accessing data from a Microsoft SQL Server database. Consequently, users access data through a Web browser, in addition to having the option to install Microsoft Office Outlook integration software. For most developers, we recommend that they simply think of Microsoft Dynamics CRM as a typical Web application.

This chapter introduces three topics regarding programming Microsoft Dynamics CRM:

- The Software Development Kit
- A hitchhiker's guide to common questions
- Microsoft Dynamics CRM as a business-application platform

The subsequent chapters dive into the Microsoft Dynamics CRM software architecture and provide programming examples.

Software Development Kit Introduction

Like many commercial software applications, Microsoft Dynamics CRM offers a Software Development Kit (SDK) that documents how you can customize and extend the system. The SDK consists of many different components related to extending the software:

- A compiled Help file that documents the application's architecture and programming interfaces, provides a report writer's guide, and offers additional development information
- Microsoft Dynamics CRM 4.0 user interface style guide
- Code samples (walkthroughs)
- Helper classes and utilities
- Graphic images
- The SDKreadme.htm file, which documents any known issues

Sometimes people refer to just the compiled help file as the SDK, but you can see all of these documents when you download the SDK and extract the files.

Important Microsoft updates the SDK on a periodic basis (approximately once every two or three months), so be sure to obtain the latest version. You can download the Microsoft Dynamics CRM 4.0 Software Development Kit at *http://www.microsoft.com/downloads/details.aspx?FamilyID=82E632A7-FAF9-41E0-8EC1-A2662AAE9DFB*.

As part of the SDK, Microsoft documents all of the supported interaction points—also known as application programming interfaces (APIs)—that you can use when writing code that integrates with Microsoft Dynamics CRM. Using the APIs for your customizations provides several significant benefits:

- **Ease of use** The APIs include hundreds of pages of documentation complete with real-world examples, code samples, and helper classes to help you write code that works with Microsoft Dynamics CRM.
- **Supportability** If you encounter technical problems or issues using the APIs, you can contact Microsoft technical support or use the Microsoft Dynamics CRM public newsgroup for assistance.
- **Upgrade support** Microsoft makes every effort to ensure that the code you create for Microsoft Dynamics CRM using the APIs upgrades smoothly to future versions of

the product, even if the underlying Microsoft SQL Server database changes radically. This is also true for any updates and hotfixes that Microsoft might release for Microsoft Dynamics CRM.

- **Certification** By following the documented APIs, you can submit your customizations to a third-party testing vendor to certify that your application works within the confines of the SDK. This certification provides comfort and reassurance for people evaluating your customizations.

Hitchhiker's Guide to Common Questions

Throughout the years we've worked with Microsoft Dynamics CRM, we find that a common set of developer questions pop up again and again. This section lists some of these questions and points you to the chapters in this book where you can find additional details about what you're trying to accomplish.

Can we alter the CRM database structure to add our custom tables and columns?

Yes, you can extend the Microsoft Dynamics CRM database with new entities (tables), attributes (columns), and relationships (keys). You can also add new attributes to the out-of-box entities. However, you do not make these modifications to SQL Server directly. Instead you use one of two different tools to modify the database:

- A Web-based customization tool
- The metadata API

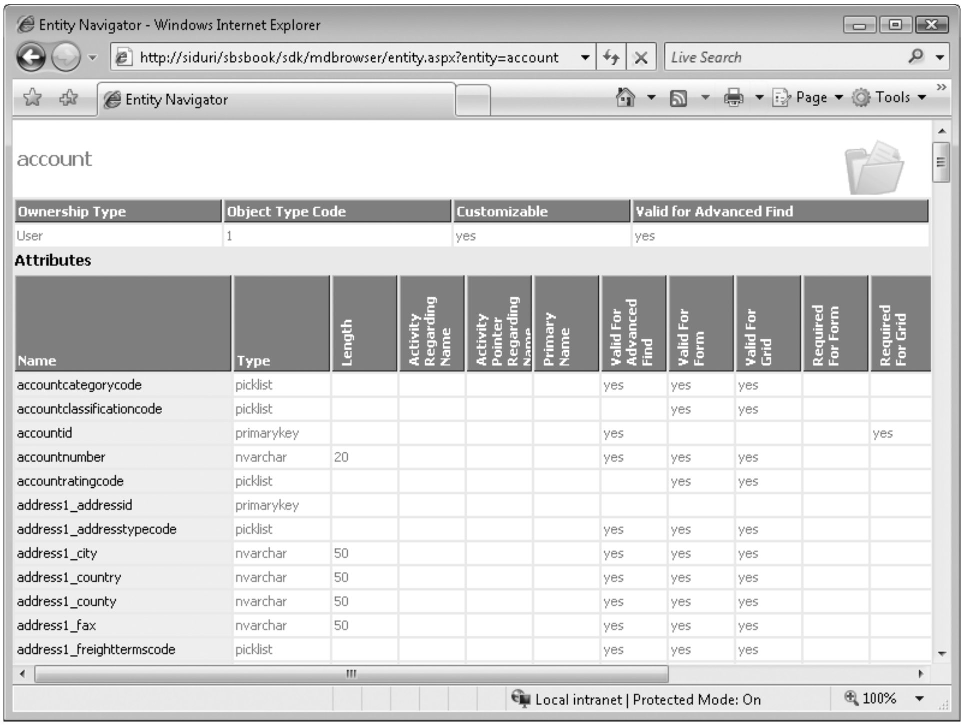
For more information about using the Web-based customization tool, please refer to the book *Working with Microsoft Dynamics CRM* by Mike Snyder and Jim Steger (Microsoft Press, 2008). That book includes several chapters on using the Web-based customization tools to modify the data structure.

The metadata API allows you to programmatically modify the database, including adding new attributes, entities, and so on. In this book, please refer to Chapter 8, "Developing with the Metadata Service," for more information about programmatically modifying the database.

Important Even though you can technically modify the database structure directly within SQL Server, you should not attempt to do so because the modifications might cause unintended consequences in your application, including possible data loss or system corruption. The Microsoft Dynamics CRM customization tools and the metadata API provide all of the resources you need to modify the database structure.

Another related question we frequently hear is "What does the database structure look like?" Although Microsoft Dynamics CRM does use a SQL Server database, theoretically you should not need to poke around the database structure or examine it. You can access data about the entities through the user interface or the metadata API. To further emphasize this idea, we want to point out that Microsoft released *logical* database diagrams for Microsoft Dynamics CRM 4.0. These logical database diagrams do not include the actual table structure; instead, they list the abstracted logical structure just as you utilize it through the user interface and API. You can download and view the Microsoft Dynamics CRM 4.0 logical database diagrams from *http://www.microsoft.com/downloads/details.aspx?FamilyID=b73912e8-861e-43ae-97b4-72b3e809f287&DisplayLang=en*. These database diagrams show the logical data relationships and the linked attributes between entities in Microsoft Office Visio format.

In addition to the logical database diagrams, you can also view information about the entities and entity relationships through the Metadata browser at *http://*<yourcrmserver>*/*<yourorganizationname>*/sdk/list.aspx* (see Figure 1-1).


Entity Navigator - Windows Internet Explorer
http://siduri/sbsbook/sdk/mdbrowser/entity.aspx?entity=account
Live Search
Entity Navigator
Page
Tools
Local intranet | Protected Mode: On
100%

account

Ownership Type	Object Type Code	Customizable	Valid for Advanced Find
User	1	yes	yes

Attributes

Name	Type	Length	Activity Regarding Name	Activity Pointer Regarding Name	Primary Name	Valid For Advanced Find	Valid For Form	Valid For Grid	Required For Form	Required For Grid
accountcategorycode	picklist					yes	yes	yes		
accountclassificationcode	picklist						yes	yes		
accountid	primarykey					yes				yes
accountnumber	nvarchar	20				yes	yes	yes		
accountratingcode	picklist						yes	yes		
address1_addressid	primarykey									
address1_addresstypecode	picklist					yes	yes	yes		
address1_city	nvarchar	50				yes	yes	yes		
address1_country	nvarchar	50				yes	yes	yes		
address1_county	nvarchar	50				yes	yes	yes		
address1_fax	nvarchar	50				yes	yes	yes		
address1_freighttermscode	picklist					yes	yes	yes		

FIGURE 1-1 The Microsoft Dynamics CRM Metadata browser

Lastly, you can also use the metadata service API to programmatically view data about the database schema, attribute values, relationships, and so on.

If you're just dying to see the underlying database structure, of course you can simply open SQL Server and examine it for yourself. You will find that Microsoft Dynamics CRM uses a normalized underlying database structure with clearly named tables such as *account_base* and *account_extensionbase*.

How do we write custom code that gets data into and out of Microsoft Dynamics CRM?

When you create custom code that needs to interact with Microsoft Dynamics CRM data, you should use one of two techniques:

- ***CrmService* Web Service** An API that performs authentication and supports common data requests such as create, read, update, and delete. This API uses a Web service interface.
- **Filtered views** Filtered views are SQL Server database views that your custom application can query to obtain read-only information about records.

You should *avoid* creating custom code that accesses the SQL Server database tables directly—please stick to one of these two techniques. Both of these interfaces abstract the underlying database from your code so that if necessary Microsoft can modify the SQL Server database for hotfixes, new versions, and so on. If your custom code accesses a database table directly and then Microsoft needs to modify it, your custom code will probably break. However, if your code accesses the *CrmService* Web service or a filtered view, Microsoft updates these interfaces with the corresponding database changes so that your code continues to run as-is.

While the *CrmService* Web service provides access to data about records, Microsoft Dynamics CRM includes two additional Web services that you can utilize:

- ***MetadataService* Web Service** This Web service provides an API that allows you to query and manipulate the data structure.
- ***CrmDiscoveryService* Web Service** This Web service provides an API that allows you to query for information about the Microsoft Dynamics CRM installation.

Refer to Chapter 3, "Communicating with Microsoft Dynamics CRM APIs," for information about connecting to the APIs. Chapter 8 includes a deeper look at retrieving and modifying the database schema programmatically.

Can we change the current CRM form layouts and controls?

Yes, Microsoft Dynamics CRM offers multiple tools to modify the existing forms. The Web-based customization tools allow you to:

- Add, remove, and modify form fields.
- Add, remove, and move tabs.
- Change field and tab labels.

Figure 1-2 shows the form editor for the contact entity.

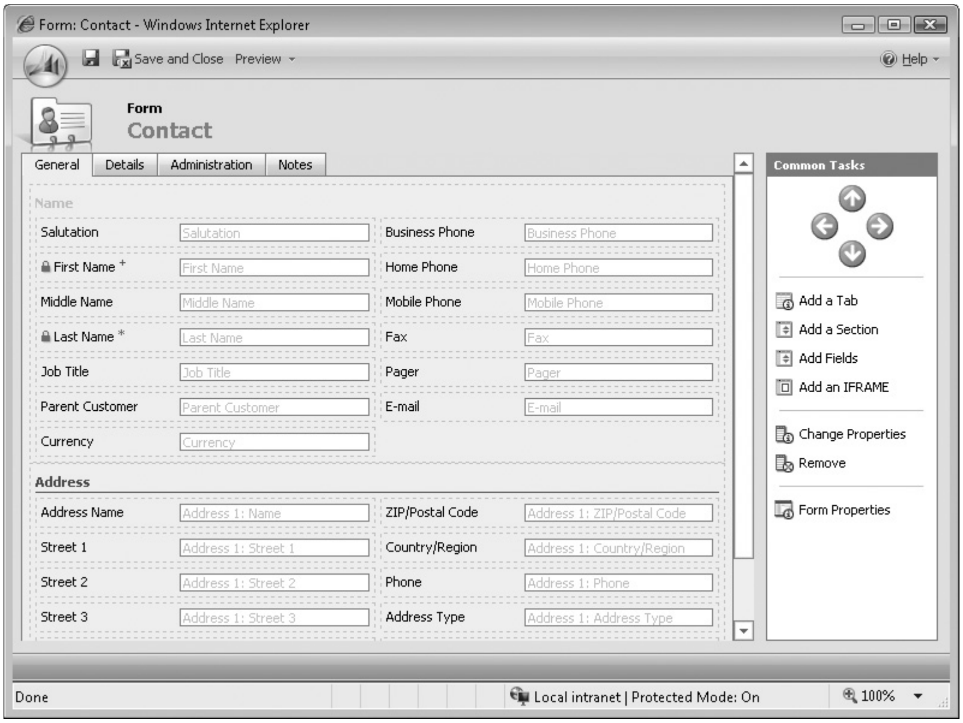


FIGURE 1-2 The contact form editor

Many of the attributes on the form include built-in controls such as a calendar for date fields, drop-down menus for picklist fields, check boxes for bit fields, and so on. Obviously this form editor provides great convenience for you to add and remove fields, in addition to changing the form layout. The book *Working with Microsoft Dynamics CRM* includes several chapters explaining how to modify form layouts.

However, if you want to use different controls than the ones included by default, Microsoft Dynamics CRM does not include a tool to swap out the default controls with your controls. However, you can implement your own custom controls by using a combination of IFrames and your own custom Web pages. An IFrame allows you to embed a custom Web page into a Microsoft Dynamics CRM form so that it appears in the context of other Microsoft Dynamics CRM fields. For more information on creating custom user controls, please refer to Chapter 14, "Developing Custom Microsoft Dynamics CRM Controls."

How do we implement our own custom business logic?

Microsoft Dynamics CRM includes several different options for implementing your custom business logic:

- **Form scripting events** Microsoft Dynamics CRM offers *onSave*, *onLoad*, and *onChange* form events that you can use to trigger form scripting code.
- **Server-side events** You can register Microsoft .NET assemblies that contain your custom code, and Microsoft Dynamics CRM will trigger these assemblies based on the user operations you configure, such as creating a record, deleting a record, assigning a record, and so on. These .NET assemblies are known as plug-ins in Microsoft Dynamics CRM, and you can run them either synchronously or asynchronously.
- **Microsoft Dynamics CRM Workflow** This option uses the Windows Workflow Foundation framework to create business automation processes triggered by the actions you configure. Sample workflow rules include e-mail alerts, task creation, record assignment, and so on.
- **Custom Web pages** You can embed your own custom Web pages directly within the Microsoft Dynamics CRM application and user interface. These pages can contain any type of business logic that you deem necessary.

As you would expect, you configure form scripting events on a record's form that Microsoft Dynamics CRM can trigger when a user saves a record, loads a form, or changes a field's data value. Form scripting events allow you to perform conditional form manipulation such as updating one field's value based on the value of a different field, or changing the form layout that a user sees based on the security role of the user viewing the record. You use JavaScript as your form scripting language. Figure 1-3 shows where you can load script onto a form. Please refer to Chapter 7, "Form Scripting," for a detailed look at the client script programming model. Microsoft Dynamics CRM executes form scripting both online and offline (within Microsoft Dynamics CRM for Outlook with Offline Access).

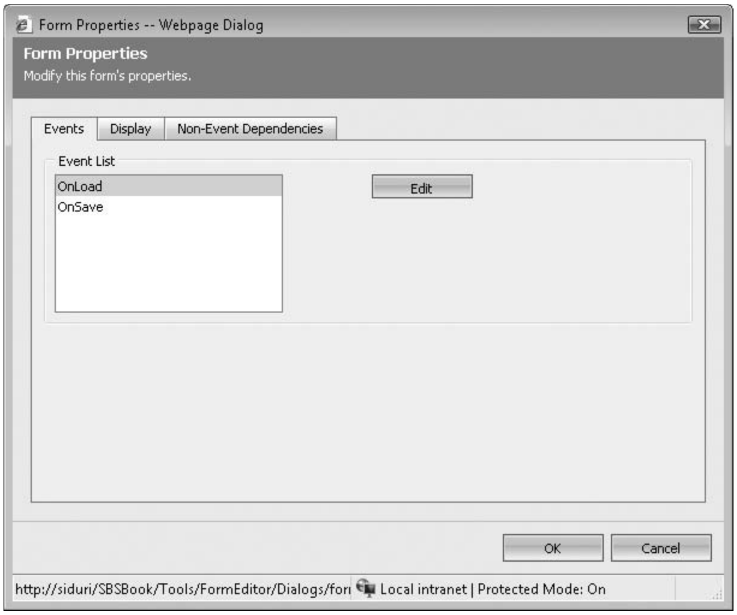
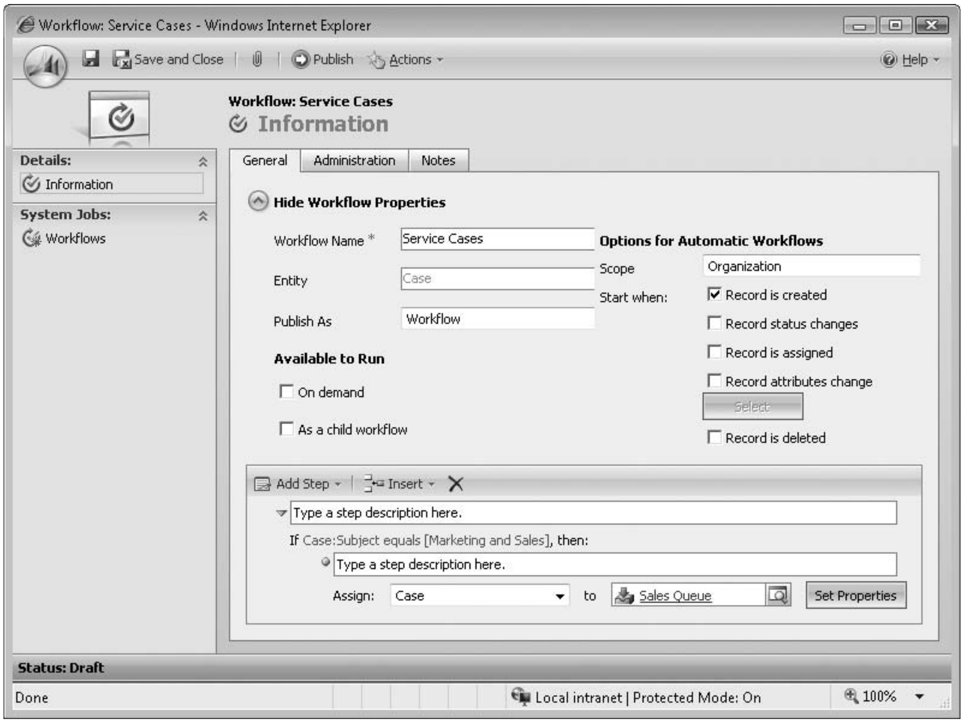

Form Properties -- Webpage Dialog
Form Properties
Modify this form's properties.
Events
Display
Non-Event Dependencies
Event List
OnLoad
OnSave
Edit
OK
Cancel
http://siduri/SBSBook/Tools/FormEditor/Dialogs/for
Local intranet | Protected Mode: On

FIGURE 1-3 A dialog box for adding client-side scripts to a form

For server-side logic, Microsoft Dynamics CRM offers a plug-in model where you can create custom .NET assemblies that Microsoft Dynamics CRM executes based upon the defined trigger operations. For example, you can create an assembly that runs every time a user deactivates a lead or closes an opportunity. Because the plug-in model accepts .NET assemblies, developers can take advantage of the .NET Framework to accommodate almost any type of customization your organization might require. You can configure plug-ins to run either synchronously or asynchronously. In addition, you can even create plug-ins that run offline (disconnected from the server) in the Microsoft Dynamics CRM for Outlook software. Chapter 5, “Plug-ins,” explains how to write plug-ins in exhaustive detail.

Microsoft Dynamics CRM Workflow offers another option for implementing your own business logic. Unlike form scripting events and plug-ins, Microsoft Dynamics CRM Workflow includes a user interface that nondevelopers can use to set up and create their own automation processes. As a developer, this frees you from simple and common requests such as creating e-mail alerts and notifications. Figure 1-4 shows an example of a workflow rule created in the Web interface. Please refer to *Working with Microsoft Dynamics CRM* for an explanation of the Workflow Web interface.



Workflow Name * Service Cases
Entity Case
Publish As Workflow

Available to Run
☐ On demand
☐ As a child workflow

Options for Automatic Workflows
Scope Organization
Start when:
☑ Record is created
☐ Record status changes
☐ Record is assigned
☐ Record attributes change
☐ Record is deleted

Type a step description here.
If Case:Subject equals [Marketing and Sales], then:
Type a step description here.
Assign: Case to Sales Queue

Status: Draft

FIGURE 1-4 The Web-based workflow rule designer

Even though the Workflow Web interface is quite powerful, undoubtedly your users will encounter scenarios where they can't design their business logic within the existing Web-based tools. Fortunately, Microsoft Dynamics CRM allows you to create custom workflow assemblies that your users can reference in the Web workflow designer to utilize in their rules. Just like plug-ins, workflow assemblies are fully .NET-compliant so that you have almost unlimited programming options to create complex and sophisticated business logic within workflow. Chapter 6, "Programming Workflow," explains the process for creating workflow assemblies within Microsoft Dynamics CRM. Chapter 12, "Advanced Workflow Programming," contains additional information about more complex programming customizations within workflow.

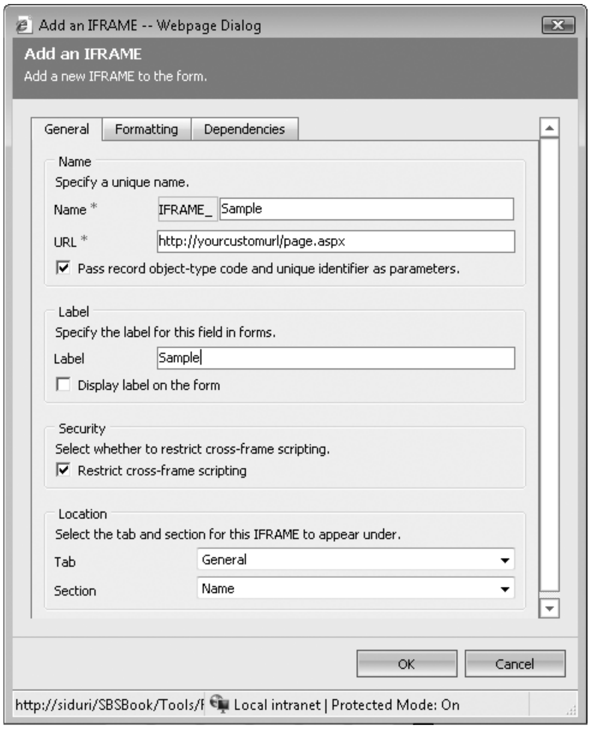
Caution Many people assume that because Microsoft Dynamics CRM uses SQL Server, they can use database triggers for their business logic. This is not the case. If you want to create custom business logic related to database activity, you should plan to use one of the supported mechanisms such as form scripting events, plug-ins, or workflow instead of database triggers.

Another powerful option to implement your custom business logic in Microsoft Dynamics CRM is to create custom Web pages that you embed in the user interface. You can create these pages using any technology that you prefer—Microsoft Dynamics CRM simply references your pages.

How much control do we have over the user interface and branding?

As we already mentioned, Microsoft Dynamics CRM offers Web-based customization tools that allow you to modify the various forms with your custom attributes and relationships. This form-customization tool is nice because nondevelopers can use it to make modifications to your system.

However, you can perform more complex modifications to the user interface through the use of IFrames to implement your own custom user interface. While IFrames allow you to embed your custom Web pages within a Microsoft Dynamics CRM form, you can also modify the user interface by creating entirely new Web pages within the application. Users can access these custom Web pages through the primary navigation, or from buttons or links that you can add to existing records. Figure 1-5 shows the dialog to add an IFrame to a Microsoft Dynamics CRM form.


Add an IFRAME -- Webpage Dialog

Add an IFRAME

Add a new IFRAME to the form.

Name

Specify a unique name.

Name * IFRAME_ Sample

URL * http://yourcustomurl/page.aspx

☑ Pass record object-type code and unique identifier as parameters.

Label

Specify the label for this field in forms.

Label Sample

☐ Display label on the form

Security

Select whether to restrict cross-frame scripting.

☑ Restrict cross-frame scripting

Location

Select the tab and section for this IFRAME to appear under.

Tab General

Section Name

http://siduri/SBSBook/Tools/F Local intranet | Protected Mode: On

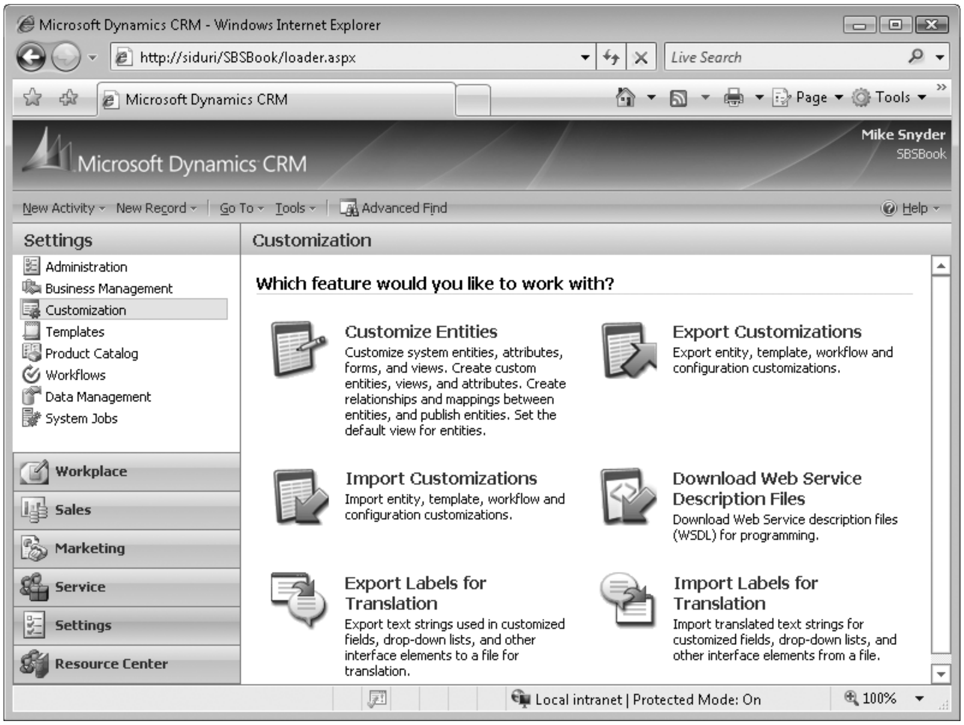
FIGURE 1-5 Adding an IFrame to a form

Please refer to Chapter 13, "Emulating the User Interface with ASP.NET Development," for information about creating new Web pages that work within Microsoft Dynamics CRM. Please refer to the book *Working with Microsoft Dynamics CRM* for an explanation of using the SiteMap and ISV.Config to modify the navigation model.

Warning Even though you can technically modify the .aspx Web pages and the .js files in the Microsoft Dynamics CRM Web application, Microsoft considers these types of modifications unsupported. Instead you should use the other techniques outlined above to implement your custom business logic and user interface. Modifying the .aspx or .js files will probably cause unexpected (bad) behavior within your system.

How do we deploy changes from one system to another?

Microsoft Dynamics CRM includes a customization import and export utility in the Web interface so that you can easily move customizations from one system to another (such as moving from development to staging to production). Figure 1-6 shows some of the customization import and export utilities.



Which feature would you like to work with?

Customize Entities
Customize system entities, attributes, forms, and views. Create custom entities, views, and attributes. Create relationships and mappings between entities, and publish entities. Set the default view for entities.

Export Customizations
Export entity, template, workflow and configuration customizations.

Import Customizations
Import entity, template, workflow and configuration customizations.

Download Web Service Description Files
Download Web Service description files (WSDL) for programming.

Export Labels for Translation
Export text strings used in customized fields, drop-down lists, and other interface elements to a file for translation.

Import Labels for Translation
Import translated text strings for customized fields, drop-down lists, and other interface elements from a file.

FIGURE 1-6 Import and export customizations in Microsoft Dynamics CRM

When you export customizations, Microsoft Dynamics CRM creates an XML file that contains all of the details of your entities. You can then take that customization file and import it into your target system. If you plan on frequent updates from one system to another, you can write code using the Microsoft Dynamics CRM Metadata API that will automatically export customizations from one system, import them into another system, and then publish those changes on the target system. If your system includes custom Web pages, you are responsible for deploying those files. Microsoft Dynamics CRM will not include your custom Web pages in the customizations import/export process. Chapter 9, "Deployment," takes a closer look at deploying your Microsoft Dynamics CRM customizations.

Will our customizations upgrade when Microsoft releases a new version of the software?

This question appears more frequently than probably all of the other questions combined, and understandably so! If you invest hundreds or thousands of hours customizing Microsoft Dynamics CRM, you want to know that you won't lose that investment when Microsoft releases the next version of the software. The key to answering this question is understanding what Microsoft means when they talk about "supported customizations." If Microsoft considers a customization supported, you can pretty safely assume that the customization will upgrade smoothly. We like to think of the SDK as the authoritative list of supported customizations, so if you follow the guidance outlined in that document you should not experience a problem.

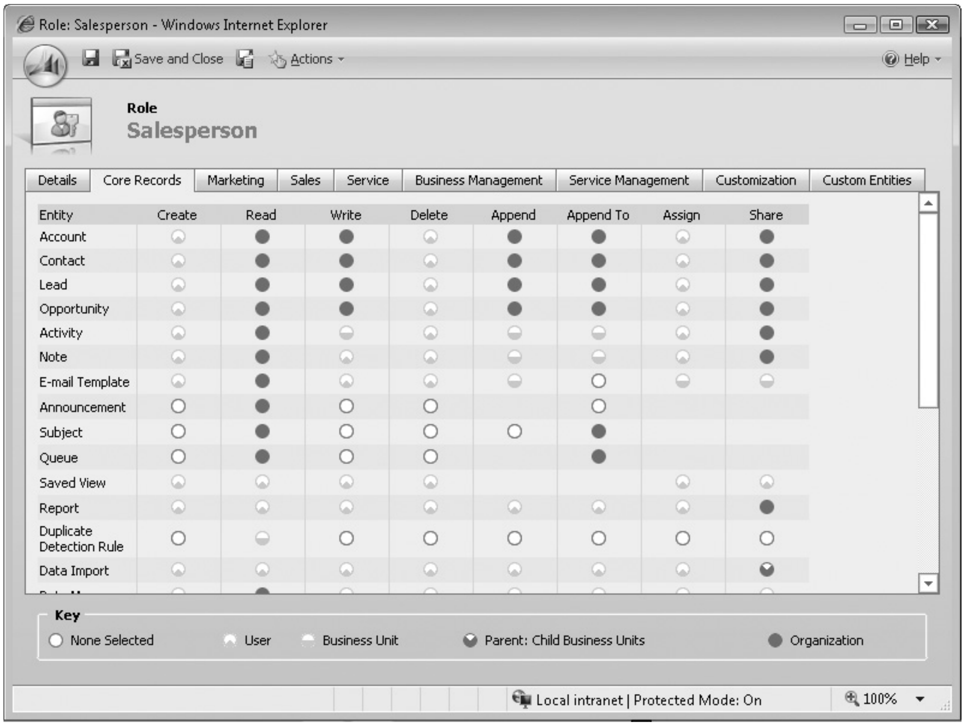
Caution While most supported customizations upgrade to future versions, Microsoft cannot guarantee this. For example, upgrading from Microsoft Dynamics CRM 1.2 to Microsoft Dynamics CRM 3.0 included a few breaking changes related to activities. However, Microsoft only makes these types of changes when the benefit of the new functionality clearly outweighs the cost of re-creating a customization.

Having experienced multiple upgrades of Microsoft Dynamics CRM, we feel that Microsoft demonstrates a good track record of supporting customizations. For example, Microsoft completely revamped the asynchronous service for Microsoft Dynamics CRM 4.0, replacing 3.0 callouts with plug-ins in 4.0. The new 4.0 plug-in model included a large number of new benefits for developers and administrators, so plug-ins were a great architecture improvement over callouts. However, Microsoft included backward-compatibility support for Microsoft Dynamics CRM 3.0 callouts so that they can run in Microsoft Dynamics CRM 4.0 without any code changes.

More Info Microsoft stated that they plan to release a new major release of Microsoft Dynamics CRM once every two years. In the interim, Microsoft will release smaller updates, hotfixes, and security updates along the way. However, many customers find it comforting that the major updates follow a periodic update schedule at a reasonable interval.

Are role-based security permissions supported and configurable?

Microsoft Dynamics CRM uses a role-based security model to determine the various privileges with the system. Each user can possess one or more security roles, and each security role defines the various privileges within the system. Administrators can configure and assign security roles through a Web interface. Figure 1-7 shows how an administrator can configure a security role in the application.



Entity	Create	Read	Write	Delete	Append	Append To	Assign	Share
Account	◔	●	●	◔	●	●	◔	●
Contact	◔	●	●	◔	●	●	◔	●
Lead	◔	●	●	◔	●	●	◔	●
Opportunity	◔	●	●	◔	●	●	◔	●
Activity	◔	●	◑	◔	◑	◑	◔	●
Note	◔	●	◔	◔	◑	◑	◔	●
E-mail Template	◔	●	◔	◔	◑	○	◑	◑
Announcement	○	●	○	○		○		
Subject	○	●	○	○	○	●		
Queue	○	●	○	○		●		
Saved View	◔	◔	◔	◔			◔	◔
Report	◔	◔	◔	◔	◔	◔	◔	●
Duplicate Detection Rule	○	◑	○	○	○	○	○	○
Data Import	◔	◔	◔	◔	◔	◔	◔	◕

Key
○ None Selected ◔ User ◑ Business Unit ◕ Parent: Child Business Units ● Organization

FIGURE 1-7 The security role editor in Microsoft Dynamics CRM

Please refer to *Working with Microsoft Dynamics CRM* for more information on setting up user security. For information about security within your programming customizations, please refer to Chapter 4, "Security," in this book.

Does Microsoft Dynamics CRM support multiple languages and currencies?

Yes, Microsoft Dynamics CRM is a truly global product that supports multiple languages and multiple currencies within a single deployment. Suppose that a sample organization has 500 users using Microsoft Dynamics CRM. That organization could theoretically set up their users as follows:

- 100 users with US English and US Dollars
- 100 users with Spanish and US Dollars
- 100 users with French and Euros
- 200 users with Spanish and Euros

As a developer, you must understand how your custom code needs to accommodate these types of multiple language and multiple currency scenarios. Chapter 11, "Multilingual and Multi-Currency Applications," takes a look at programming for these situations within Microsoft Dynamics CRM.

Will our programming customizations run offline?

As we previously mentioned, Microsoft Dynamics CRM includes optional add-in software for Microsoft Office Outlook. This add-in software comes in two different versions:

- Microsoft Dynamics CRM for Outlook
- Microsoft Dynamics CRM for Outlook with Offline Access

The offline-enabled version of this software allows your users to work while disconnected from the Microsoft Dynamics CRM server. As a developer, you have the option to create customizations that also run offline within Microsoft Dynamics CRM for Outlook. The SDK includes support for offline programming interfaces. Even if your customizations don't need to run offline, you should take some time to understand how users with the offline version of Microsoft Dynamics CRM for Outlook might interact with your server-based customizations. Please refer to Chapter 10, "Developing Offline Solutions," for more information on this topic.

How do you recommend we set up a Microsoft Dynamics CRM development environment?

When you're creating your Microsoft Dynamics CRM customizations, of course you don't want to develop and test your code in a production environment. You want to work in a sandbox system and then push your completed customizations to a different environment upon completion. Chapter 2, "Development Overview and Environment," examines different options for setting up a development system for your team of developers.

Microsoft Dynamics CRM as a Business Application Platform

If you're new to Microsoft Dynamics CRM, you might think of the application as just a sales, marketing, and service tool. However, we encourage you to think of new and creative ways to use your programming skills and the Microsoft Dynamics CRM platform to tackle new business challenges. We believe that Microsoft Dynamics CRM is an excellent development platform for many reasons, including:

- Metadata architecture that allows for easy extensions to the database model
- Web-based customization tools that allow nondevelopers to make application changes
- Built-in workflow capability
- Documented and easy-to-use software development kit
- Service-orientated architecture
- Native support for online and offline use
- Native support for multiple currencies and multiple languages
- Enterprise-class capabilities with SQL Server database
- Out–of-the-box integration with the common end-user applications Microsoft Office Outlook, Microsoft Office Excel, Microsoft Office Word, and Microsoft Office Communication Server
- Common user authentication with Microsoft Active Directory for single sign-on with Microsoft Office SharePoint Server

Having worked with many different customers implementing Microsoft Dynamics CRM, our company Sonoma Partners has helped many organizations use Microsoft Dynamics CRM as a business application platform to tackle nontraditional CRM business issues. Examples include:

- Helping a large national franchise to use Microsoft Dynamics CRM to scout, rank, and identify potential restaurant locations.
- Working with a national real-estate company to track condominium developments and condominium inventory in Microsoft Dynamics CRM. The company also tracked each buyer's preferences and upgrades such as appliances, paint color, furnishings, and so on.
- Developing a system for a nonprofit organization to qualify applicants of oil and heat subsidies, including tracking applications, receipts, and vendor payment status.

- Designing a database of hospitals and physicians for a long-term care management company to help them better understand the patient referral and new patient setup process.

Most people would not consider any of these examples as traditional CRM, yet all of them work excellently on the Microsoft Dynamics CRM platform! If your organization is considering building a custom software application from scratch, or if you have an existing home-grown custom application, we strongly urge you to consider using Microsoft Dynamics CRM as a platform to replace custom software applications. We hope that the chapters and examples in this book will give you the confidence that Microsoft Dynamics CRM is truly easy to program with, and offers an unbelievable amount of flexibility.

Tip Sometimes people use the term xRM to describe using a CRM software application as the business application platform to solve nontraditional business challenges. We've seen different definitions for the acronym xRM, but we like to think of the letter *X* as a variable just like you might remember from your algebra class. You can plug in almost any value for *X*, but it always includes the relationship management.

Summary

Microsoft Dynamics CRM includes many different software development tools that professional developers can use to create complex system customizations. The Microsoft Dynamics CRM SDK is the primary development documentation for developers, as the SDK includes architecture information, helper classes, and definitions of supported customizations. When developing Microsoft Dynamics CRM customizations, you should not access the SQL Server database directly. Instead you should connect to system data through the *CrmService* Web service or the *MetadataService* Web service. The *CrmService* provides basic create, read, update, and delete functionality, and the *MetadataService* provides a programmatic interface to the data schema. You can implement your own business logic in Microsoft Dynamics CRM using a combination of techniques, such as form scripting, server-side assemblies, workflow assemblies, and custom Web pages. Because of the flexibility of the Microsoft Dynamics CRM programming platform, the software offers an ideal development platform for tracking nontraditional CRM data beyond sales, marketing, and customer service.

Chapter 5

Plug-ins

Plug-ins provide one of the most powerful customization points within Microsoft Dynamics CRM. As users work in the application, their actions cause Microsoft Dynamics CRM to trigger events that developers can use to execute custom business logic through the use of plug-ins. For example, you can register plug-ins to run business logic every time a user creates an account or deletes an activity. You can create plug-ins to run in response to a vast number of events, including plug-ins for custom entities. You can use plug-ins for a variety of features, such as synchronizing data to an external database, tracking changes in an audit log, or simply creating follow-up tasks for a newly created account.

Note At the time this book went to press, Microsoft Dynamics CRM Online (the Microsoft hosted version of Microsoft Dynamics CRM) does not support custom plug-ins.

Some of the tasks you can accomplish with plug-ins—such as populating fields with default values or specific field formatting—you can also accomplish with form JavaScript. Plug-ins have the advantage of running on the server, so you are guaranteed that these types of tasks will run even if the entity is created or updated from a bulk import or through the Web service API.

Note If you are familiar with Microsoft Dynamics CRM 3.0, you are probably thinking that plug-ins sound very similar to callouts. Plug-ins are in fact the replacement for callouts, but they offer a much more robust programming model. Microsoft Dynamics 4.0 does support version 3.0 callouts if you still need to use them, but they access CRM via the version 3.0 endpoint and do not support version 4.0 features such as multi-tenancy.

In this chapter, we will explore the following topics in detail:

- Writing your first plug-in
- The event execution pipeline
- Details of the *IPluginExecutionContext* interface
- Impersonation
- Exception handling

- Deploying plug-ins
- Debugging plug-ins
- Unit testing plug-ins
- Real-world plug-in samples

Writing Your First Plug-in

When working with a new framework or technology, we find it easiest to start with a simple hands-on example and then dig deeper into real-world examples. We'll start by implementing a simple plug-in to provide a more concrete foundation for the remainder of the chapter. This plug-in verifies that an account's *accountnumber* follows a specific format. In this example, Microsoft Dynamics CRM executes the plug-in when a new account is created or modified to verify that the account number starts with two letters followed by six numbers.

As mentioned earlier, you could accomplish this same type of account validation through scripting with the form's *onsave* event. However, enforcing business logic on the form might not be ideal because modifications to the account number through workflow or through an external application would bypass the *onsave* event script and possibly allow an invalid account number format. By using a plug-in, we can guarantee that Microsoft Dynamics CRM enforces our business logic regardless of the method used to create the account.

Creating the Plug-in Project

Plug-ins are implemented as classes that implement a specific interface and are contained within a signed Microsoft .NET assembly. The assembly needs to target the Microsoft .NET runtime version 2.0, which can be accomplished by creating a class library in Microsoft Visual Studio 2008 targeting the .NET Framework 2.0, 3.0, or 3.5. However, installing Microsoft Dynamics CRM 4.0 only guarantees that Microsoft .NET Framework 3.0 is installed on the server. If you need assemblies included in the Microsoft .NET Framework 3.5, you have to install that version of the framework yourself. Before we can create our first plug-in, we need to create a class library project. Follow these steps to set up your first plug-in project.

Creating the plug-in project in Microsoft Visual Studio 2008

1. Open Microsoft Visual Studio 2008.
2. On the File Menu, select New and then click Project.
3. In the New Project dialog box, select the Other Project Types > Visual Studio Solutions type, and then select the Blank Solution template.
4. Type the name **ProgrammingWithDynamicsCrm4** in the Name box. Click OK.

5. On the File Menu, select Add and then click New Project.
6. In the New Project dialog box, select the Visual C# project type targeting the .NET Framework 3.0 and then select the Class Library template.
7. Type the name **ProgrammingWithDynamicsCrm4.Plugins** in the Name box. Click OK.
8. Delete the default Class.cs file.
9. Right-click the ProgrammingWithDynamicsCrm4.Plugins project in Solution Explorer and then click Add Reference.
10. On the Browse tab, navigate to the CRM SDK's bin folder and select microsoft.crm.sdk.dll and microsoft.crm.sdktypeproxy.dll. Click OK.
11. Right-click the ProgrammingWithDynamicsCrm4.Plugins project in Solution Explorer and then click Add Reference.
12. On the .NET tab, select System.Web.Services. Click OK.
13. Right-click the ProgrammingWithDynamicsCrm4.Plugins project in Solution Explorer and then click Properties.
14. On the Signing tab, select the Sign The Assembly box and then select <New...> from the list below it.
15. Type the key file name ProgrammingWithDynamicsCrm4.Plugins, and then clear the Protect My Key File With A Password check box. Click OK.
16. Close the project properties window.

Implementing the Plug-in Class

After setting up our project, we are ready to implement our first plug-in. Let's start by adding a class to our newly created project.

Adding the AccountNumberValidator class

1. Right-click the ProgrammingWithDynamicsCrm4.Plugins project in Solution Explorer. Under Add, click Class.
2. Type **AccountNumberValidator.cs** in the Name box. Click Add.

Replace the generated code in the *AccountNumberValidator* class with the code displayed in Listing 5-1.

LISTING 5-1 The *AccountNumberValidator* plug-in source code

```
using System;
using Microsoft.Crm.Sdk;
using System.Text.RegularExpressions;

namespace ProgrammingWithDynamicsCrm4.Plugins
{
    public class AccountNumberValidator: IPlugin
    {
        public void Execute(IPluginExecutionContext context)
        {
            DynamicEntity target =
                (DynamicEntity)context.InputParameters[ParameterName.Target];

            if (target.Properties.Contains("accountnumber"))
            {
                string accountNumber = (string)target["accountnumber"];
                Regex validFormat = new Regex("[A-Z]{2}-[0-9]{6}");

                if (!validFormat.IsMatch(accountNumber))
                {
                    string message =
                        "Account number does not follow the required format. " +
                        "(AA-######)";

                    throw new InvalidPluginExecutionException(message);
                }
            }

        }

    }
}
```

AccountNumberValidator, a very simple plug-in, extracts the target account as a *DynamicEntity* and validates that the *accountnumber* property follows a specific pattern (two capital letters followed by a dash and then six numbers). We know the target input parameter will be a *DynamicEntity* representing the account because we will be registering this plug-in with the Create and Update messages for the account entity.

Notice that the only requirement at the class level for a plug-in is that it must implement the *Microsft.Crm.Sdk.IPlugin* interface. *IPlugin* has only a single method, named *Execute*, which takes a single argument of type *IPluginExecutionContext*. We will be exploring the *IPluginExecutionContext* interface in detail—as well as how Microsoft Dynamics CRM 4.0 handles exceptions thrown by plug-ins—later in this chapter. For more information on the *DynamicEntity* class and its use, refer to Chapter 3, “Using the Web Service APIs.”

Building the Registration Tool

Unlike for workflows, form changes, and other customizations to Microsoft Dynamics CRM, no Web-based interface is included to register plug-ins. However, the Microsoft Dynamics CRM SDK includes two utilities to help you register plug-ins, and you can also register plug-ins using the API.

Later in the chapter we will explore using the API to write your own plug-in registration tools, but for this first example we will use one of the CRM SDK's registration tools, PluginRegistration. PluginRegistration is a Windows desktop application that has an intuitive graphical user interface for registering plug-ins and configuring which messages cause the plug-in to execute. You can find the PluginRegistration tool in the Tools folder within the CRM SDK.

The Microsoft Dynamics CRM SDK distributes PluginRegistration as source code only, so you will need to compile it before you can run it. Follow the guidelines in the readme.doc included in the tools\PluginRegistration folder to compile the application. PluginRegistration is distributed as a Visual Studio 2005 project, but Visual Studio 2008 can automatically upgrade it without problems.

Deploying the Plug-in

After compiling our plug-in registration tool, we are ready to register our first plug-in. During registration you specify which messages for specific entities will cause the plug-in to execute. Depending on the message, you can specify additional filtering or request more information to be provided to your plug-in during execution.

Important To register a plug-in you must be listed as a Deployment Administrator on the CRM server. To verify that you are a Deployment Administrator, log on to the CRM server and launch the Deployment Manager tool, which is located in the Microsoft Dynamics CRM group on the Start menu. If you are not a Deployment Administrator, the tool will show an error indicating so and then exit. If this is the case, you need to have a Deployment Administrator use this tool and add you to the list of Deployment Administrators.

When you register a plug-in, Microsoft Dynamics CRM offers you multiple registration properties:

- **Mode** A plug-in can execute either synchronously or asynchronously.
- **Stage** This option specifies whether the plug-in will respond to pre-events or post-events.
- **Deployment** A plug-in can execute only on the server, within the Outlook client, or both.

- **Messages** This option determines which Microsoft Dynamics CRM events should trigger your logic, such as *Create*, *Update*, and even *Retrieve*.
- **Entity** A plug-in can execute against most of the entities, including custom entities.
- **Rank** This option is an integer that specifies the order in which all plug-in steps should be executed.
- **Assembly Location** This option tells Microsoft Dynamics CRM whether the assemblies are stored in the database or on the Web server's file system.
- **Images** You can pass attribute values from the record as either pre-images or post-images for certain message types.

You configure these plug-in properties when you register the plug-in with Microsoft Dynamics CRM.

Mode

Microsoft Dynamics CRM allows you to execute plug-ins synchronously or asynchronously. Asynchronous plug-ins are loaded into the Microsoft CRM Asynchronous Service and executed at some point after the main event processing is complete. Asynchronous plug-ins are ideal for handling situations that are not critical to complete immediately, such as audit logging. Because the plug-in executes asynchronously, it does not negatively affect the response time for an end user who initiates the core operation.

Real World In practice, most plug-ins perform tasks that users expect to see feedback on as soon as they save their changes within the CRM application. Because of this, you will probably find that most plug-ins are registered to execute synchronously. When it is determined that a plug-in can be registered to execute asynchronously, implementing a custom workflow step instead is frequently more beneficial because business users can more easily maintain the workflow. Scenarios still exist in which an asynchronous plug-in is the right answer, but they are not very common. Microsoft Dynamics CRM does not support pre-event plug-ins configured for asynchronous operation.

Stage

When you register a plug-in, you can configure the plug-in to run before or after the core operation takes place. A plug-in that executes before the core operation is referred to as a *pre-event* plug-in, while a plug-in that executes after the core operation is a *post-event* plug-in. Pre-event plug-ins are useful when you want to validate or alter data prior to submission. With post-event plug-ins, you can execute additional logic or integration after the data has been safely stored in the database.

Important How do you know which stage to register for? If a plug-in needs to interrupt or modify values before they are committed to the database, you should register it as a pre-event plug-in. Otherwise, you end up needing to execute an additional message to apply your change when you could have accomplished this by just modifying the data before the original message's core operation executed. On the other hand, if your plug-in needs to create a child entity whenever the parent entity type is created, you need to register it to execute during the post-event stage to have access to the newly created parent's ID.

Deployment

One of the great new features of Microsoft Dynamics CRM 4.0 is the ability to have your plug-in logic execute offline with the Outlook client, further extending your existing solution. You can choose to have the plug-in execute only against the server, run offline with the Outlook client, or both.

Remember that when a client goes offline and then returns online, any plug-in calls are executed after the data synchronizes with the server. If you choose to have your logic execute both with the server and offline, be prepared for Microsoft Dynamics CRM to execute your plug-in code twice.

Caution Microsoft Dynamics CRM does not support an asynchronous implementation of a plug-in with offline deployment. If you want to have your plug-in work offline, you need to register it in synchronous mode.

For more information about developing offline solutions and using plug-ins offline, please refer to Chapter 10, "Developing Offline Solutions."

Messages

In the documentation, Microsoft Dynamics CRM 4.0 refers to server-based trigger events as *messages*. The Microsoft Dynamics CRM 4.0 SDK also supports all the events from Microsoft Dynamics CRM 3.0, such as *Create*, *Update*, *Delete*, and *Merge*. In addition, Microsoft Dynamics CRM 4.0 includes some new messages such as *Route*, *Retrieve*, and *RetrieveMultiple*.

See the "Supported Messages and Entities" section later in this chapter for more information about the available messages. You can also use the API to write code to see whether Microsoft Dynamics CRM supports a particular message.

Entities

Most system and all custom entities are available for plug-in execution. Please refer to the "Supported Messages and Entities" section for more information on the supported entities.

Rank

Rank merely denotes the order in which a plug-in should fire. Rank is simply an integer, and Microsoft Dynamics CRM starts with the plug-in with the lowest rank and then cycles through all available plug-ins. You should definitely consider the order of plug-ins, depending on the logic they perform.

Assembly Location

You can deploy plug-in assemblies to the database, to a folder on the Microsoft Dynamics CRM server, or to the Global Assembly Cache (GAC) on the server. Typically the database is the best option because you do not need to manually copy the file to the server before registering the plug-in. Unless you have a specific need to do otherwise, we recommend that you leave the default option and deploy your plug-ins to the database.

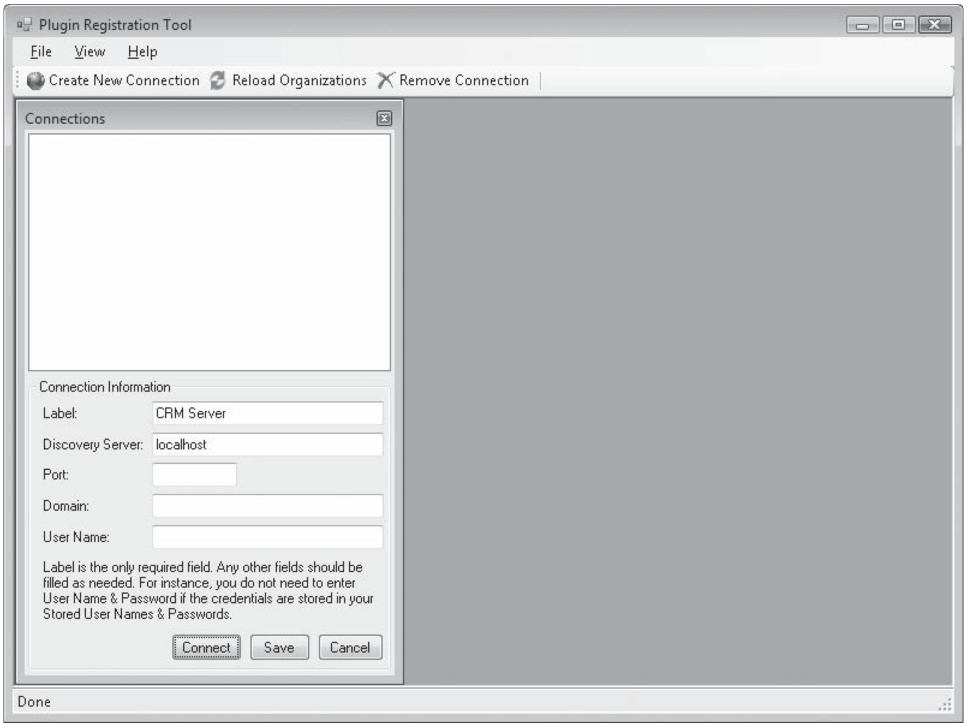
Images

Images provide you with the record attribute values. Images exist as pre-values (before the core platform operation) and post-values. Not all messages allow images.

Now that you understand a little more background about the plug-in registration process, use the following steps to register the *AccountNumberValidator* plug-in.

Connecting to the server with the PluginRegistration tool

1. Launch the PluginRegistration tool that you compiled in the previous section. You will see the New Connection screen first (Figure 5-1).
2. Type any name you want for the Label. It is only used for display purposes in the PluginRegistration tool.
3. Type the name of your CRM server for the Discovery Server.
4. Optionally, specify the port your CRM server is running on if it is not port 80.
5. Optionally, specify the domain and user name you want to use to connect to the CRM server. If you specify a domain and user name, you will be prompted for a password when you connect. If you leave these fields blank, the tool connects as the currently logged on user.
6. Click Connect. You should now see a list of organizations under your connection.
7. Double-click the organization you want to register the plug-in with.



Label: CRM Server
Discovery Server: localhost
Port:
Domain:
User Name:

Label is the only required field. Any other fields should be filled as needed. For instance, you do not need to enter User Name & Password if the credentials are stored in your Stored User Names & Passwords.

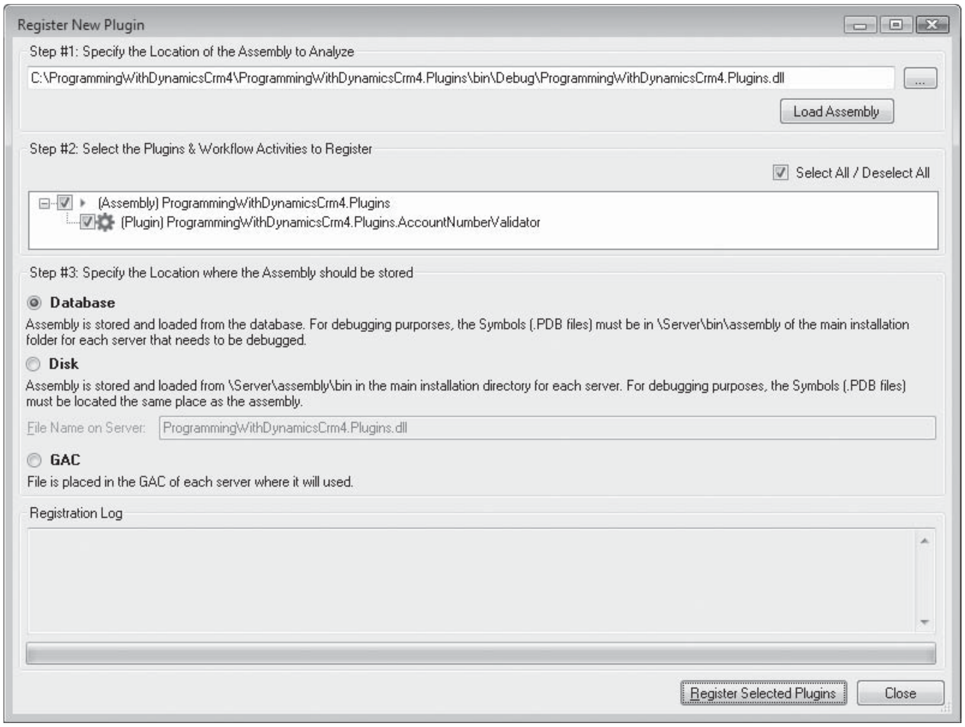
FIGURE 5-1 The New Connection screen

Now that you are connected to the server, next you will register the assembly on the server.

Registering the assembly

1. Select Register New Assembly from the Register toolbar drop-down list to open the Register New Assembly dialog box (Figure 5-2).
2. Click the ellipsis button to browse and select the plug-in DLL. Note that the assembly and plug-in classes are selected by default in the selection tree.
3. Leave Database selected as the deployment location.

   While you can deploy plug-ins to a folder or to the GAC on the Microsoft Dynamics CRM server, it is typically better to deploy to the database because you do not need to manually set up the assembly on the server. This point becomes even more valid if you are dealing with a web farm environment because you would need to copy the assembly to each server if you don't specify database deployment.
4. Click Register Selected Plugins.



Register New Plugin

Step #1: Specify the Location of the Assembly to Analyze

C:\ProgrammingWithDynamicsCrm4\ProgrammingWithDynamicsCrm4.Plugins\bin\Debug\ProgrammingWithDynamicsCrm4.Plugins.dll

Load Assembly

Step #2: Select the Plugins & Workflow Activities to Register

☑ Select All / Deselect All

☑ (Assembly) ProgrammingWithDynamicsCrm4.Plugins
☑ (Plugin) ProgrammingWithDynamicsCrm4.Plugins.AccountNumberValidator

Step #3: Specify the Location where the Assembly should be stored

● **Database**
Assembly is stored and loaded from the database. For debugging purposes, the Symbols (.PDB files) must be in \Server\bin\assembly of the main installation folder for each server that needs to be debugged.

○ **Disk**
Assembly is stored and loaded from \Server\assembly\bin in the main installation directory for each server. For debugging purposes, the Symbols (.PDB files) must be located the same place as the assembly.

File Name on Server: ProgrammingWithDynamicsCrm4.Plugins.dll

○ **GAC**
File is placed in the GAC of each server where it will used.

Registration Log

FIGURE 5-2 The Register New Assembly dialog box

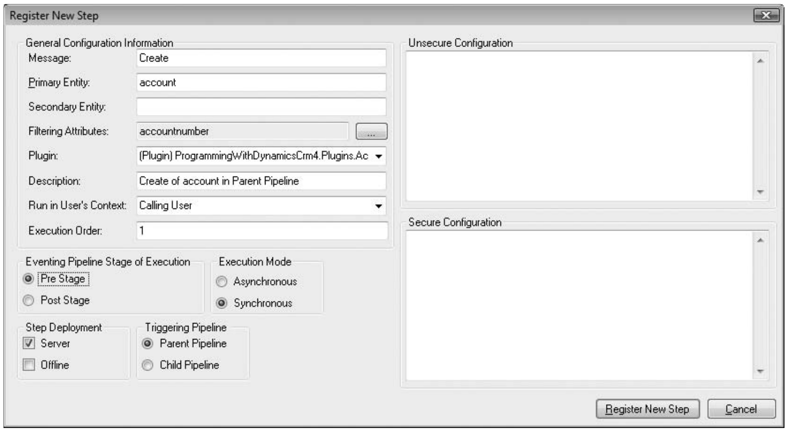
After this step you should see a series of messages in the Registration log. If all goes well, a confirmation dialog box will pop up to tell you that one assembly and one plug-in were registered.

Last you need to configure when the plug-in should run. You do this by registering steps with the PluginRegistration tool. Steps contain information such as the entity and message that will cause a plug-in to execute, as well as the stage it will execute in. Each plug-in can have multiple steps, allowing it to execute for different entities and messages.

Registering plug-in steps

1. Right-click the *AccountNumberValidator* plug-in in the Registered Plugins & Custom Workflow Activities tree, and then select Register New Step to open the Register New Step dialog box (Figure 5-3).
2. Type **Create** in the Message box.
3. Type **account** in the Primary Entity box.
4. Specify **accountnumber** for the Filtering Attributes by clicking on the ellipsis button and then clearing all but the Account Number check box in the resulting dialog box.

5. Select the Pre Stage option.
6. Leave the rest of the settings at their default values.
7. Click Register New Step.
8. Repeat steps 1 through 7, but type **Update** in the Message box.


Register New Step

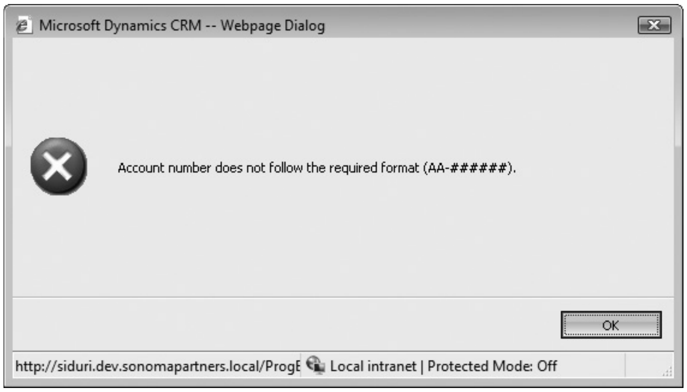
General Configuration Information
Message: Create
Primary Entity: account
Secondary Entity:
Filtering Attributes: accountnumber
Plugin: (Plugin) ProgrammingWithDynamicsCrm4.Plugins.Ac
Description: Create of account in Parent Pipeline
Run in User's Context: Calling User
Execution Order: 1

Eventing Pipeline Stage of Execution: [✓ Pre Stage] Post Stage
Execution Mode: Asynchronous [✓ Synchronous]
Step Deployment: [✓ Server] Offline
Triggering Pipeline: [✓ Parent Pipeline] Child Pipeline

Unsecure Configuration
Secure Configuration

FIGURE 5-3 The Register New Step dialog box

Now that you've registered the plug-in, it will verify that all newly created or modified account numbers match the specified format. If a user tries to create or update an account using an invalid account number, the error shown in Figure 5-4 appears. Likewise, if a workflow or service call tries to create or modify an account with an invalid account number, Microsoft Dynamics CRM will not update the account and will bubble up an exception to the caller.


Microsoft Dynamics CRM -- Webpage Dialog

Account number does not follow the required format (AA-######).

http://siduri.dev.sonomapartners.local/ProgE Local intranet | Protected Mode: Off

FIGURE 5-4 An error presented to the user by the *AccountNumberValidator* plug-in

The Event Execution Pipeline

Now that we implemented a basic plug-in, let's step back and look at the bigger picture. Plug-ins run within an execution pipeline specific to the message being executed. Also executing within the pipeline is the core operation, which is implemented by Microsoft Dynamics CRM 4.0. The core operation typically consists of a database operation—either retrieving, updating, inserting, or deleting records. For example, when a RetrieveMultiple request is executed, the core operation is the selection of data from the database. Figure 5-5 illustrates the various stages of the event execution pipeline.

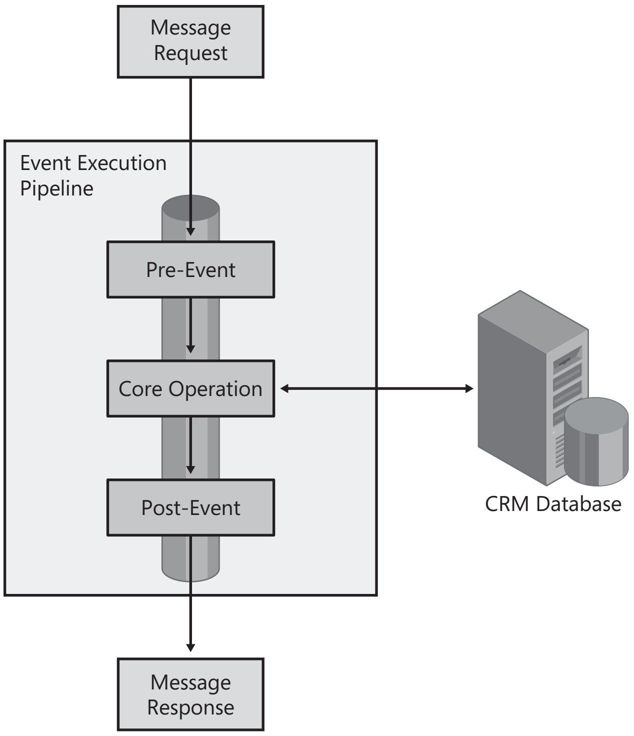
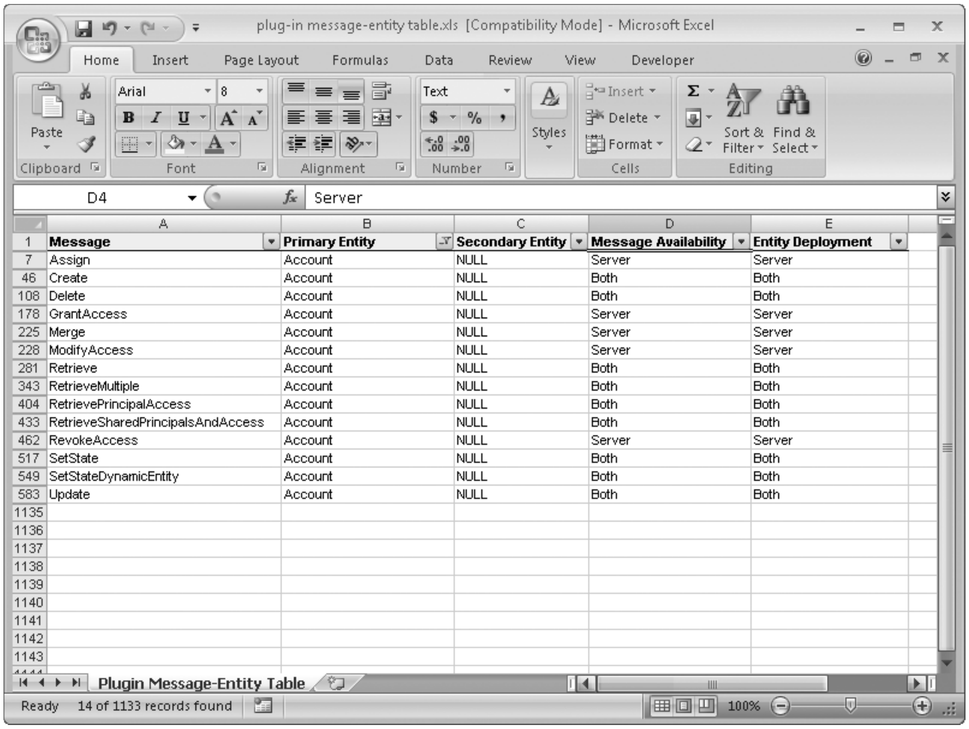


FIGURE 5-5 The event execution pipeline

Supported Messages and Entities

When trying to determine how to register a plug-in or even what is possible to hook into, you often find yourself wondering which messages exist for any given entity. The CRM SDK includes a Microsoft Office Excel spreadsheet that lists all the events that can be registered for and their corresponding entities. The file is named Plug-in Message-Entity Table.xls and is located in the Tools subfolder of the SDK.

This spreadsheet includes filterable columns and can be an excellent tool when you are trying to determine which messages an entity supports—or when you are just trying to brainstorm creative solutions. From this list you can see that several messages, such as Import, Export, and Publish, are not even tied to an entity. Figure 5-6 shows the spreadsheet filtered to only display messages supported by the Account entity.



	Message	Primary Entity	Secondary Entity	Message Availability	Entity Deployment
7	Assign	Account	NULL	Server	Server
46	Create	Account	NULL	Both	Both
108	Delete	Account	NULL	Both	Both
178	GrantAccess	Account	NULL	Server	Server
225	Merge	Account	NULL	Server	Server
228	ModifyAccess	Account	NULL	Server	Server
281	Retrieve	Account	NULL	Both	Both
343	RetrieveMultiple	Account	NULL	Both	Both
404	RetrievePrincipalAccess	Account	NULL	Both	Both
433	RetrieveSharedPrincipalsAndAccess	Account	NULL	Both	Both
462	RevokeAccess	Account	NULL	Server	Server
517	SetState	Account	NULL	Both	Both
549	SetStateDynamicEntity	Account	NULL	Both	Both
583	Update	Account	NULL	Both	Both

FIGURE 5-6 The plug-in message entity spreadsheet

Parent and Child Pipelines

Some events will in turn cause other events to be executed. When this happens, a secondary pipeline is created for this event and is referred to as a child pipeline. For example, when an Opportunity is converted to an Account, the Create event is executed in a child pipeline. If you want to handle the creation of an account in this scenario, you need to specify Child as the *InvocationSource* when registering your plug-in step.

Typically, plug-ins only execute outside the main database transaction and cannot cause a rollback to occur. However, when a plug-in is running inside a child pipeline, it is executing inside the parent pipeline's transaction, and if the plug-in throws an exception, the parent's transaction will be rolled back.

Caution One additional point to be aware of is that when you run a plug-in inside a child pipeline, you cannot use the *IPluginExecutionContext* interface's *CreateCrmService* method. If you do, an exception is thrown. The use of the *CreateCrmService* method was intentionally disabled in child pipelines because it would be too easy to cause an infinite loop or a database deadlock if it were enabled. If you absolutely need to talk back to the CRM services inside a child pipeline, you can manually create a *CrmService*, but be sure to use it with caution. Additionally, any calls you make with your own *CrmService* run within their own thread and are outside transactions in which the plug-in executes. This means that if the transaction is rolled back for any reason, changes made with your instance of *CrmService* will not be undone.

IPluginExecutionContext

As stated earlier, every plug-in must implement the *IPlugin* interface, which includes the *Execute* method in its definition. The *Execute* method takes a single argument of type *IPluginExecutionContext*, which provides the plug-in with the state of the current execution pipeline and a means to communicate with the Microsoft Dynamics CRM Web service API. *IPluginExecutionContext* has twenty-two properties and two methods, all of which are described in the following list.

- ***BusinessUnitId* property** *BusinessUnitId* is a *Guid* that represents the business unit that the primary entity belongs to.
- ***CallerOrigin* property** *CallerOrigin* is an in*stance of on*e of the following classes: *ApplicationOrigin*, *AsyncServiceOrigin, OfflineOrigin, or WebServiceApiOrigin*. You can use this property to determine who initiated the pipeline. The following code determines whether the pipeline was initiated from the CRM Web service.

```
public bool IsOriginatingFromWebServiceApi(IPluginExecutionContext context)
{
    return context.CallerOrigin is WebServiceApiOrigin;
}
```

- ***CorrelationId, CorrelationUpdatedTime, and Depth* properties** These three properties are combined to detect infinite loops in plug-ins. If you only use the *IPluginExecutionContext* interface's *CreateCrmService* method *to create CrmService* instances, you don't need to worry about these three properties, as they will be set on the returned *CrmService* for you. However, if you need to create your own instance of a *CrmService* class, you can use these properties to initialize its *CorrelationTokenValue* property, which ensures safety from infinite loops. The code shown here demonstrates how to use the correlation properties when creating your own *CrmService* instances.

```
public CrmService GetSafeCrmService(IPluginExecutionContext context)
{
    CrmService crmService = new CrmService();
    crmService.CorrelationTokenValue = new CorrelationToken(
        context.CorrelationId,
        context.CorrelationUpdatedTime,
        context.Depth
    );

    // finish initializing crmService here...

    return crmService;
}
```

More Info One additional use for *CorrelationId* is as a unique value for logging. In a production environment you will likely have multiple plug-ins executing at the same time, and the unique ID can be useful in determining which plug-in instance is generating the log messages.

- ***InitiatingUserId* property** This property is always the *Guid* of the user that caused the event to execute, regardless of whether the plug-in was registered to impersonate another user. See the *UserId* property later in this section for more information.
- **InputParameters property** This property is an instance of *Microsoft.Crm.Sdk.PropertyBag*. Each value contained in *PropertyBag* corresponds with a property on the *Request* that caused this event to execute. For example, *CreateRequest* has a property named *Target*, so you would find a value in *InputParameters* with a key of "*Target*".

Tip When accessing the values in *InputParameters,* you should use the *ParameterNames* static class, instead of typing keys, to avoid run-time errors caused by typos.

```
if (context.InputParameters.Contains(ParameterName.Target))
{
    DynamicEntity target = (DynamicEntity)
        context.InputParameters[ParameterName.Target];
    // ...
}
```

- ***InvocationSource* property** The *InvocationSource* property is an integer value that you can use to determine whether the current plug-in is running in a child pipeline. Table 5-1 lists the valid values as defined by the *MessageInvocationSource* class.

TABLE 5-1 ***MessageInvocationSource* Values**

Field	Value	Description
Child	1	Specifies a child pipeline
Parent	0	Specifies a parent pipeline

- ***IsExecutingInOfflineMode* property** You can register plug-ins to run offline with Microsoft CRM for Outlook with Offline Access. If a plug-in is running in such a state, this Boolean property is set to true. See Chapter 10 for more information on offline plug-ins.

- ***MesssageName* property** *MessageName* is a string property that allows the current plug-in to know the name of the message that is being executed (*Create*, *Update*, *Assign*, and so on).
- ***Mode* property** *Mode* is an integer property that you can use to determine whether the plug-in is executing synchronously or asynchronously. The valid values are from the *MesssageProcessingMode* class, as listed in Table 5-2.

TABLE 5-2 *MessageProcessingMode* Values

Field	Value	Description
Asynchronous	1	Specifies asynchronous processing
Synchronous	0	Specifies synchronous processing

- ***OrganizationId* and *OrganizationName* properties** These properties contain information about the organization that the current entity belongs to and that the current pipeline is executing within.

Caution The initial release of Microsoft Dynamics CRM 4.0 had a bug that caused the friendly organization name to be passed into the plug-in execution context instead of the actual name. When you create an organization, these two values are the same by default, but if they are different you can run into issues quickly. The main problem is that when you use the *CreateCrmService* method, an invalid organization is specified for the *ICrmService* proxy and any calls you make with it result in an unauthorized exception. At the time this book went to press, Microsoft was aware of the defect and was implementing a fix, but until the fix is released you can just keep the organization name and the friendly name identical.

- ***OutputParameters* property** Similar to the *InputParameters* property, this property is an instance of a *PropertyBag*. The values in the *OutputParameters* property correspond with the properties on the *Response* for the message being executed. For example, a *CreateResponse* has an *Id* property, so a post-event plug-in could expect the corresponding value in the *OutputParameters* property using a key value of "Id".

Tip Using the static *ParameterNames* class instead of string keys is encouraged so that you'll discover errors at compile time instead of at run time.

```
// Getting the entity id in a Post-Event for a Create message
Guid contactId = (Guid)context.OutputParameters[ParameterName.Id];
```

- ***ParentContext* property** *ParentContext* is another instance of *IPluginExecutionContext*. If the current plug-in is executing in a child pipeline, *ParentContext* will contain the context of the parent pipeline; otherwise, *ParentContext* will be null.

- ***PreEntityImages* and *PostEntityImages* properties** *PreEntityImages* and *PostEntityImages* are both *PropertyBag* properties. When registering a plug-in, you can specify for certain messages that you want a snapshot of the entity before or after the core operation has completed. You also specify the alias you would like to give that snapshot. Those snapshots, or images, show up in these two collections with the alias as the key. *PreEntityImages* contains the images from before the core operation, and *PostEntityImages* contains the images from after the core operation.
- ***PrimaryEntityName* property** *PrimaryEntityName* is a string property that contains the name of the primary entity for which the pipeline is executing.
- ***SecondaryEntityName* property** *SecondaryEntityName* is a string property that contains the name of the secondary entity for which the pipeline is executing, if one exists. A majority of the messages deal with a single entity, so this property will almost always be set to "none". However some messages, like SetRelated, refer to two entities. In this case, you can use *SecondEntityName* to find out the type of the second entity.
- ***SharedVariables* property** *SharedVariables* is a *PropertyBag* property that is meant to be used by plug-in developers to pass information between plug-ins. Using *SharedVariables*, a pre-event plug-in can pass along information to a post-event plug-in. Another potential use is to look up data in a parent pipeline step and then later access it in a child pipeline through the child's *ParentContext* property's *SharedVariables* property.
- ***Stage* property** *Stage* is an integer property that a plug-in can use to determine whether it is running as a pre-event or a post-event plug-in. The valid values are from the *MessageProcessingStage* class, as listed in Table 5-3.

TABLE 5-3 ***MessageProcessingStage* Values**

Field	Value	Description
AfterMainOperationOutsideTransaction	50	Specifies to process after the main operation, outside the transaction
BeforeMainOperationOutsideTransaction	10	Specifies to process before the main operation, outside the transaction

More Info There are, in fact, three other values for Stage, but they are for internal use only by Microsoft Dynamics CRM and you will receive an error if you try to register your plug-in to run in one of these stages. Just in case you run into one of these values while trying to debug an issue, they are BeforeMainOperationInsideTransaction (20), MainOperation (30), and AfterMainOperationInsideTransaction (40).

- ***UserId* property** *UserId* is a *Guid* property that represents the user that the plug-in is running as for any *CrmService* calls. This value is typically the user that initiated the event, but if a plug-in is registered to impersonate another user, this value contains the impersonated user's ID. See the *InitiatingUserId* property for more information.
- ***CreateCrmService* method** This is an overloaded method that you can use to create an instance of an *ICrmService* interface that has the same methods as the *CrmService* class, which is explained in detail in Chapter 3. The arguments control impersonation within the plug-in and are explored in more depth in the "Impersonation" section later in this chapter.
- ***CreateMetadataService* method** You use the *CreateMetadataService* method to get an instance of the *IMetadataService* interface that has the same methods as the *MetadataService* class, which is explained in detail in Chapter 3. The method accepts a single Boolean named *useCurrentUserId* and is used for impersonation within the plug-in. See the next section, "Impersonation," for more details.

Impersonation

Impersonation in Microsoft Dynamics CRM occurs when a *CrmService* or *MetadataService* call is made on behalf of another user. Plug-ins have two options for impersonation. First, they can be registered to impersonate a specific user by default. Second, they can specify a user ID to impersonate on the fly during execution.

Important Plug-in impersonation does not work offline. Actions offline are always taken by the logged-on user.

Impersonation During Registration

When you register a plug-in, you can specify an *impersonatinguserid* value. In this situation, any calls to the *IPluginExecutionContext* interface's *CreateCrmService* or *CreateMetadataService* methods with a value of *true* for the *useCurrentUser* argument result in a service that is impersonating the user specified at registration. Passing *false* for the *useCurrentUser* argument results in a service that is executing as the "system" user. In addition, the *IPluginExecutionContext* interface's *UserId* property contains the user ID specified during registration.

Impersonation During Execution

A plug-in's second option for impersonation is to specify a user ID when calling the *IPluginExecutionContext* interface's *CreateCrmService* method. This allows the plug-in to determine on the fly which user to impersonate, possibly pulling a value from a registry setting or configuration file.

Best Practices You may be wondering which method of impersonation you should use. Unless you know that you need to impersonate another user, you should simply pass in *true* to the *useCurrentUser* argument and create service instances that will behave as determined by the plug-in registration. Most often, plug-ins will be registered without an *impersonatinguserid* specified and you will run as the user that initiated the event. If at a later point it is determined that you need a plug-in to run with impersonation, you can change the plug-in step without needing to recompile the plug-in assembly. Avoid passing in *false* for *useCurrentUser* unless you need to because this value means that calls into the *CrmService* effectively run as an administrator, possibly elevating the privilege of the user who caused the plug-in to execute.

Exception Handling

We frequently receive questions regarding exceptions when writing plug-ins. How are exceptions handled? Should all inner exceptions be handled by the plug-in? Does Microsoft Dynamics CRM automatically log exceptions? What does an end user see when an exception goes unhandled? Fortunately these questions have fairly straightforward answers, as detailed in the following sections.

Exceptions and the Event Processing Pipeline

The impact of an unhandled exception within a plug-in on the event processing pipeline is fairly intuitive. If you registered your plug-in as a pre-event plug-in and it throws an exception or lets an exception go unhandled, no further plug-ins will execute and the core operation will not occur. If you registered your plug-in as a post-event and it throws an exception, no further plug-ins will execute, and since the core operation already occurred Microsoft Dynamics CRM will not roll it back. However, if the plug-in is executing in a child pipeline, an unhandled exception results in the parent pipeline's core operation being rolled back.

Exception Feedback

Microsoft Dynamics CRM logs all unhandled exceptions in the Event Viewer on the server where they occurred. In addition, if the exception generating event was initiated by the user through the Microsoft Dynamics CRM user interface, the user is presented with an error message. To control the message that the user sees, you should throw an *InvalidPluginExecution-Exception*. In this case, the *Message* property for the exception is displayed. If you let an exception of another type go unhandled, a generic error message may be used.

Deployment

At the beginning of the chapter we briefly touched on one of the tools used to deploy plug-ins. In this section we'll take a deeper look at what happens during plug-in registration and how you can write your own registration tools.

Plug-in Entities

Microsoft Dynamics CRM stores plug-in information in a series of entities as listed in Table 5-4.

TABLE 5-4 Plug-in Entities

Entity Name	Description
pluginassembly	Represents the registered plug-in assembly. Can have multiple *plugintype* entities associated with it.
plugintype	Represents the class in the plug-in assembly that implements *IPlugin*. Can have multiple steps associated with it.
sdkmessageprocessingstep	Represents a step in the event execution pipeline when a plug-in type should be executed. Can have multiple *sdkmessageprocessingimage* entities and multiple *sdkmessageprocessingstepsecureconfig* entities associated with it.
sdkmessageprocessingstepimage	Represents the definition of which types of images should be provided to a plug-in for a particular step. Images are essentially snapshots of the entity before or after the core operation has taken place.
sdkmessageprocessingstepsecure-config	Represents secure configuration information for a particular plug-in step. Passed to the plug-in constructor if provided.

Programmatic Plug-in Registration

You can register and deploy plug-ins programmatically through the API, which allows you to implement your own deployment tools without a lot of code. To demonstrate this, we will implement a plug-in registration tool that uses custom .NET attributes to specify how to register our plug-ins. This approach offers the benefit of letting the developer implement the plug-in to specify its use as he codes it.

Because both the plug-in assembly and our installation tool reference our custom .NET attributes, we need to put them in their own class library. Follow these steps to add the project to our existing solution.

Adding the custom attribute project

1. On the File Menu, select Add and then click New Project.
2. In the New Project dialog box, select the Visual C# project type targeting the .NET Framework 3.0, and then select the Class Library template.
3. Type the name **ProgrammingWithDynamicsCrm4.Plugins.Attributes** in the Name box, and then click OK.
4. Delete the default Class.cs file.
5. Right-click the ProgrammingWithDynamicsCrm4.Plugins.Attributes project in Solution Explorer and then click Properties.
6. On the Signing tab, select the Sign The Assembly box and select <New...> from the drop-down list below it.
7. Type the key file name **ProgrammingWithDynamicsCrm4.Plugins.Attributes** and then clear the Protect My Key File With A Password check box. Click OK.
8. Close the project properties window.

Adding the PluginStepAttribute class

Next we need to define the custom attribute class.

1. Right-click the ProgrammingWithDynamicsCrm4.Plugins.Attributes project in Solution Explorer. Under Add, click Class.
2. Type **PluginStepAttribute.cs** in the Name box and click Add.

Listing 5-2 shows the full source code for the *PluginStepAttribute* class.

LISTING 5-2 *PluginStepAttribute* source code

```
using System;

namespace ProgrammingWithDynamicsCrm4.Plugins.Attributes
{
    [AttributeUsage(AttributeTargets.Class, AllowMultiple=true)]
    public class PluginStepAttribute: Attribute
    {
        public PluginStepAttribute(string message, PluginStepStage stage)
        {
            this.Message = message;
            this.Stage = stage;
        }

        public PluginStepStage Stage { get; private set; }
        public string Message { get; private set; }

        public string PrimaryEntityName { get; set; }
        public string SecondaryEntityName { get; set; }

        public PluginStepMode Mode { get; set; }
        public int Rank { get; set; }
        public string Description { get; set; }
        public string FilteringAttributes { get; set; }
        public PluginStepInvocationSource InvocationSource { get; set; }
        public PluginStepSupportedDeployment SupportedDeployment { get; set; }
    }
}
```

Custom attributes like the one we have defined here allow us to embed extra information into our compiled types and assemblies. We can use this attribute to attach information about our plug-in registration to the plug-in class itself. The following example demonstrates how this attribute might be applied to a plug-in class.

```
[PluginStep("Update", PluginStepStage.PreEvent, PrimaryEntityName = "account")]
public class MyPluginClass: IPlugin
{
    ...
}
```

Notice that even though the class name is *PluginStepAttribute*, we can omit the trailing *Attribute*—which is just a shortcut supplied by .NET—when applying it to a class. Also worth noticing is that we've exposed some of the arguments as constructor arguments and others as public properties. In general, when you work with custom attributes you want to make anything required a constructor argument and anything optional a public property. You might argue that *PrimaryEntityName* should have been in the list of required attributes, but you can register for a few messages that do not have an entity associated with them.

If you have a sharp eye, you probably noticed that we still need to define the types for a few properties in *PluginStepAttribute*. These four types are all defined as enums so that you will receive IntelliSense in Visual Studio 2008 when you apply this attribute to a plug-in class. The enums are defined as shown in Listing 5-3.

LISTING 5-3 Enum type definitions

```
namespace ProgrammingWithDynamicsCrm4.Plugins.Attributes
{
    public enum PluginStepInvocationSource
    {
        ParentPipeline = 0,
        ChildPipeline = 1,
    }

    public enum PluginStepMode
    {
        Synchronous = 0,
        Asynchronous = 1,
    }

    public enum PluginStepStage
    {
        PreEvent = 10,
        PostEvent = 50,
    }

    public enum PluginStepSupportedDeployment
    {
        ServerOnly=0,
        OutlookClientOnly=1,
        Both=2,
    }
}
```

You can place these definitions in their own files or just after the *PluginStepAttribute* class in PluginStepAttribute.cs.

Now we should be able to go back to our original ProgrammingWithDynamicsCrm4.Plugins project and add a reference to our new attributes project.

Adding a reference to the attributes project

1. Right-click the ProgrammingWithDynamicsCrm4.Plugins project in Solution Explorer and then click Add Reference.
2. On the Projects tab, select ProgrammingWithDynamicsCrm4.Plugins.Attributes. Click OK.

Now we can add our attribute to the *AccountNumberValidator* plug-in. You will need to add the following using statement at the top of AccountNumberValidator.cs:

```
using ProgrammingWithDynamicsCrm4.Plugins.Attributes;
```

Then you can add the following two attributes to the class definition:

```
[PluginStep("Create", PluginStepStage.PreEvent, PrimaryEntityName = "account",
    FilteringAttributes = "accountnumber")]
[PluginStep("Update", PluginStepStage.PreEvent, PrimaryEntityName = "account",
    FilteringAttributes = "accountnumber")]
public class AccountNumberValidator: IPlugin
{
    ...
}
```

At this point, the only remaining step is creating the actual tool to register the plug-in.

Creating the ProgrammingWithDynamicsCrm4.PluginDeploy project

1. On the File Menu, select Add and then click New Project.
2. In the New Project dialog box, select the Visual C# project type targeting the .NET Framework 3.0 and then select the Console Application template.
3. Type the name **ProgrammingWithDynamicsCrm4.PluginDeploy** in the Name box and click OK.
4. Right-click the ProgrammingWithDynamicsCrm4.PluginDeploy project in Solution Explorer and then click Add Reference.
5. On the Browse tab, navigate to the CRM SDK's bin folder and select microsoft.crm.sdk.dll and microsoft.crm.sdktypeproxy.dll. Click OK.
6. Right-click the ProgrammingWithDynamicsCrm4.PluginDeploy project in Solution Explorer and then click Add Reference.
7. On the .NET tab, select System.Web.Services and System.Configuration. Click OK.
8. Right-click the ProgrammingWithDynamicsCrm4.PluginDeploy project in Solution Explorer and then click Add Reference.
9. On the Projects tab, select ProgrammingWithDynamicsCrm4.Plugins.Attributes and click OK.

Now we can proceed to the *Main* method. Replace the generated code in Program.cs with the code shown in Listing 5-4.

LISTING 5-4 PluginDeploy's *Main* method

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.IO;
using System.Net;
using System.Reflection;
using System.Text;
using System.Web.Services.Protocols;
using Microsoft.Crm.Sdk;
using Microsoft.Crm.Sdk.Query;
using Microsoft.Crm.SdkTypeProxy;
using ProgrammingWithDynamicsCrm4.Plugins.Attributes;

namespace ProgrammingWithDynamicsCrm4.PluginDeploy
{
    static void Main(string[] args)
    {
        if (args.Length != 3)
        {
            string exeName = Path.GetFileName(Environment.GetCommandLineArgs()[0]);
            Console.WriteLine(
                "Usage: {0} <pluginAssembly> <crmServerUrl> <organizationName>",
                exeName);
            Environment.Exit(1);
        }

        try
        {
            string pluginAssemblyPath = args[0];
            string crmServer = args[1];
            string organizationName = args[2];

            DeployPlugin(pluginAssemblyPath, crmServer, organizationName);
        }
        catch (SoapException e)
        {
            Console.WriteLine(e.Detail.InnerText);
        }
        catch (Exception e)
        {
            Console.WriteLine(e.Message);
        }
    }
}
```

The *Main* method doesn't do much more than look for simple usage errors and display unhandled exceptions to the user. All of the real work is left to the *DeployPlugin* method, which is shown in Listing 5-5.

LISTING 5-5 The *DeployPlugin* method

```
private static void DeployPlugin(
    string pluginAssemblyPath,
    string crmServer,
    string organizationName)
{
    Console.Write("Initializing CrmService... ");
    CrmService crmService = CreateCrmService(crmServer, organizationName);
    Console.WriteLine("Complete");

    pluginassembly pluginAssembly = LoadPluginAssembly(pluginAssemblyPath);

    UnregisterExistingSolution(crmService, pluginAssembly.name);

    SdkMessageProcessingStepRegistration[] steps =
        LoadPluginSteps(pluginAssemblyPath);

    RegisterSolution(crmService, pluginAssembly, steps);
}
```

The first thing *DeployPlugin* does is create an instance of the *CrmService*. *CreateCrmService* is a helper method that creates a *CrmService* in a fairly straightforward way. Listing 5-6 shows the implementation of *CreateCrmService*.

LISTING 5-6 The *CreateCrmService* method

```
private static CrmService CreateCrmService(
    string crmServer, string organizationName)
{
    UriBuilder crmServerUri = new UriBuilder(crmServer);
    crmServerUri.Path = "/MSCRMServices/2007/CrmService.asmx";

    string userName = ConfigurationManager.AppSettings["crmUserName"];
    string password = ConfigurationManager.AppSettings["crmPassword"];
    string domain = ConfigurationManager.AppSettings["crmDomain"];

    CrmService crmService = new CrmService();
    if (String.IsNullOrEmpty(userName))
    {
        crmService.UseDefaultCredentials = true;
    }
    else
    {
        crmService.Credentials = new NetworkCredential(userName, password, domain);
    }

    crmService.Url = crmServerUri.ToString();
    crmService.CrmAuthenticationTokenValue = new CrmAuthenticationToken();
    crmService.CrmAuthenticationTokenValue.AuthenticationType =
        AuthenticationType.AD;
    crmService.CrmAuthenticationTokenValue.OrganizationName = organizationName;

    return crmService;
}
```

CreateCrmService checks in the application configuration file to see whether any credentials are specified to use when communicating with Microsoft Dynamics CRM. If it does not find any, it uses the credentials of the user that started the process.

After *DeployPlugin* aquires a *CrmService*, it calls *LoadPluginAssembly* to load an instance of the *pluginassembly* class from the plug-in DLL. The source for *LoadPluginAssembly* is shown in Listing 5-7.

LISTING 5-7 The *LoadPluginAssembly* method

```
private static pluginassembly LoadPluginAssembly(string pluginAssemblyPath)
{
    Assembly assembly = Assembly.LoadFile(pluginAssemblyPath);
    pluginassembly pluginAssembly = new pluginassembly();
    pluginAssembly.name = assembly.GetName().Name;
    pluginAssembly.sourcetype = new Picklist(AssemblySourceType.Database);
    pluginAssembly.culture = assembly.GetName().CultureInfo.ToString();
    pluginAssembly.version = assembly.GetName().Version.ToString();

    if (String.IsNullOrEmpty(pluginAssembly.culture))
    {
        pluginAssembly.culture = "neutral";
    }

    byte[] publicKeyToken = assembly.GetName().GetPublicKeyToken();
    StringBuilder tokenBuilder = new StringBuilder();
    foreach (byte b in publicKeyToken)
    {
        tokenBuilder.Append(b.ToString("x").PadLeft(2, '0'));
    }
    pluginAssembly.publickeytoken = tokenBuilder.ToString();

    pluginAssembly.content = Convert.ToBase64String(
        File.ReadAllBytes(pluginAssemblyPath));

    return pluginAssembly;
}
```

Most of the *pluginassembly* class's properties are populated using reflection on the assembly after it is loaded. The *publickeytoken* property is a little bit more work because we need to convert the byte array to a hexadecimal string. The *content* property is a Base64-formatted string that contains the raw bytes from the assembly DLL. Also note that we have just hard-coded *sourcetype* to be a database deployment.

After *PluginDeploy* receives *pluginassembly*, it calls *UnregisterExistingSolution* to make sure that no pre-existing version of this assembly is registered on the CRM server. The *Unregister-ExistingSolution* source code is shown in Listing 5-8.

LISTING 5-8 The *UnregisterExistingSolution* method

```
private static void UnregisterExistingSolution(
    CrmService crmService,
    string assemblyName)
{
    QueryByAttribute query = new QueryByAttribute();
    query.EntityName = EntityName.pluginassembly.ToString();
    query.ColumnSet = new ColumnSet(new string[] { "pluginassemblyid" });
    query.Attributes = new string[] { "name" };
    query.Values = new object[] { assemblyName };

    RetrieveMultipleRequest request = new RetrieveMultipleRequest();
    request.Query = query;

    RetrieveMultipleResponse response;
    Console.Write("Searching for existing solution... ");
    response = (RetrieveMultipleResponse)crmService.Execute(request);
    Console.WriteLine("Complete");
    if (response.BusinessEntityCollection.BusinessEntities.Count > 0)
    {
        pluginassembly pluginAssembly = (pluginassembly)
            response.BusinessEntityCollection.BusinessEntities[0];
        Console.Write("Unregistering existing solution {0}... ",
            pluginAssembly.pluginassemblyid.Value);

        UnregisterSolutionRequest unregisterRequest =
            new UnregisterSolutionRequest();
        unregisterRequest.PluginAssemblyId = pluginAssembly.pluginassemblyid.Value;

        crmService.Execute(unregisterRequest);
        Console.WriteLine("Complete");
    }
}
```

The *UnregisterExistingSolution* method starts by querying *CrmService* to see whether any *pluginassembly* entities are already registered with the same name. If it finds one, it executes an *UnregisterSolutionRequest*, passing in the *Guid* of the assembly that was determined to be a match.

DeployPlugin is now ready to use our custom attribute and create an array of *SdkMessageProcessingStepRegistration* instances. *SdkMessageProcessingStepRegistration* is a part of the *Microsoft.Crm.Sdk* namespace and is used to simplify the registration of plug-ins. Listing 5-9 shows the source code for *LoadPluginSteps*.

LISTING 5-9 The *LoadPluginSteps* method

```
private static SdkMessageProcessingStepRegistration[] LoadPluginSteps(string
pluginAssemblyPath)
{
    List<SdkMessageProcessingStepRegistration> steps =
        new List<SdkMessageProcessingStepRegistration>();

    Assembly assembly = Assembly.LoadFile(pluginAssemblyPath);
    foreach (Type pluginType in assembly.GetTypes())
    {
        if (typeof(IPlugin).IsAssignableFrom(pluginType) && !pluginType.IsAbstract)
        {
            object[] stepAttributes =
                pluginType.GetCustomAttributes(typeof(PluginStepAttribute), false);

            foreach (PluginStepAttribute stepAttribute in stepAttributes)
            {
                steps.Add(CreateStepFromAttribute(pluginType, stepAttribute));
            }
        }
    }

    return steps.ToArray();
}
```

LoadPluginSteps loads the assembly from the disk and then uses reflection to iterate through all the types defined in the assembly. If it finds a concrete implementation of *IPlugin*, it determines whether our *PluginStepAttribute* is associated with that type. For each *PluginStepAttribute* associated with the *plugin* type, it calls *CreateStepFromAttribute* to create an instance of *SdkMessageProcessingStepRegistration*. The *CreateStepFromAttribute* source code is shown in Listing 5-10.

LISTING 5-10 The *CreateStepFromAttribute* method

```
private static SdkMessageProcessingStepRegistration CreateStepFromAttribute(
    Type pluginType,
    PluginStepAttribute stepAttribute)
{
    SdkMessageProcessingStepRegistration step =
        new SdkMessageProcessingStepRegistration();
    step.Description = stepAttribute.Description;
    step.FilteringAttributes = stepAttribute.FilteringAttributes;
    step.InvocationSource = (int)stepAttribute.InvocationSource;
    step.MessageName = stepAttribute.Message;
    step.Mode = (int)stepAttribute.Mode;
    step.PluginTypeName = pluginType.FullName;
    step.PluginTypeFriendlyName = pluginType.FullName;
    step.PrimaryEntityName = stepAttribute.PrimaryEntityName;
    step.SecondaryEntityName = stepAttribute.SecondaryEntityName;
    step.Stage = (int)stepAttribute.Stage;
    step.SupportedDeployment = (int)stepAttribute.SupportedDeployment;

    if (String.IsNullOrEmpty(step.Description))
    {
        step.Description = String.Format("{0} {1} {2}",
            step.PrimaryEntityName, step.MessageName, stepAttribute.Stage);
    }

    return step;
}
```

Almost all the *SdkMessageProcessingStepRegistration* values are assigned directly from corresponding values on our *PluginStepAttribute* class. The *PluginTypeName* property comes from the actual plug-in type (and we use that for the *PluginTypeFriendlyName* too). If no *Description* is provided, we derive one from the *PrimaryEntityName*, *MessageName,* and *Stage* properties.

Finally, *DeployPlugin* is ready to send all this information over to the CRM server. *RegisterSolution* is called, passing our previously loaded *pluginassembly* and our newly initialized array of *SdkMessageProcessingStepRegistrations*. The *RegisterSolution* source code is shown in Listing 5-11.

LISTING 5-11 The *RegisterSolution* method

```
private static void RegisterSolution(CrmService crmService, pluginassembly
pluginAssembly, SdkMessageProcessingStepRegistration[] steps)
{
    RegisterSolutionRequest registerRequest = new RegisterSolutionRequest();
    registerRequest.PluginAssembly = pluginAssembly;
    registerRequest.Steps = steps;
    Console.Write("Registering solution... ");
    crmService.Execute(registerRequest);
    Console.WriteLine("Complete");
}
```

RegisterSolution is a straightforward method that simply creates a *RegisterSolutionRequest* and executes it with the *CrmService*.

At this point you should be able to compile the solution and use ProgrammingWithDynamics Crm4.PluginDeploy.exe to deploy ProgrammingWithDynamicsCrm4.Plugins to your CRM server. The command line used to deploy a plug-in is:

```
ProgrammingWithDynamicsCrm4.PluginDeploy.exe <pathToAssembly> <crmServerUrl>
<organizationName>
```

Images

One concept that our previous example did not touch on is the ability to request entity images during registration. An image is essentially a snapshot of an entity and it can be taken either before or after the core operation is performed. Images allow a plug-in access to attribute values that would otherwise not be available. For example, an audit log plug-in could be provided with an image that contains the original attribute values for an entity that has just been modified. Using this image, the plug-in could record both the new and old attribute values in a log. Another example of using images would be a plug-in that needs to keep a calculated value on a parent entity up to date. When the child entity is associated with a new parent, a plug-in can use a pre-image to retrieve the previous parent and ensure that both the new and the old parent are kept up to date.

We refer to images that are taken before the core operation as *pre-images* and images taken after the core operation as *post-images*. These images are then passed to a plug-in through the *IPluginExecutionContext* interface's *PreEntityImages* and *PostEntityImages* properties.

When your plug-in requires an image, you need to specify the type of image and the name of the message property that contains the entity you want an image of. In addition, not all messages can produce images. Table 5-5 lists all supported messages and their corresponding message property names.

TABLE 5-5 **Messages That Support Images**

Message	Message Property Name	Notes
Assign	Target	
Create	Id	Does not support pre-images
Delete	Target	Does not support post-images
DeliverIncoming	EmailId	
DeliverPromote	EmailId	
Merge	Target	
Merge	SubordinateId	
Route	Target	
Send	EmailId	
SetState	EntityMoniker	
SetStateDynamicEntity	EntityMoniker	
Update	Target	

Programmatic Image Registration

To add image support to ProgrammingWithDynamicsCrm4.PluginDeploy, we need an additional attribute class. Using the steps outlined earlier, add a new class to the ProgrammingWithDynamicsCrm4.Plugins.Attributes project named *PluginImageAttribute*. The source code for this class is shown in listing 5-12.

LISTING 5-12 *PluginImageAttribute* source code

```
using System;

namespace ProgrammingWithDynamicsCrm4.Plugins.Attributes
{
    [AttributeUsage(AttributeTargets.Class, AllowMultiple=true)]
    public class PluginImageAttribute: Attribute
    {
        public PluginImageAttribute(
            ImageType imageType,
            string stepId,
            string messagePropertyName,
            string entityAlias)
        {
            this.ImageType = imageType;
            this.StepId = stepId;
            this.MessagePropertyName = messagePropertyName;
            this.EntityAlias = entityAlias;
        }

        public ImageType ImageType { get; private set; }
        public string StepId { get; private set; }
        public string MessagePropertyName { get; private set; }
        public string EntityAlias { get; private set; }
        public string Attributes { get; set; }
    }

    public enum ImageType
    {
        PreImage = 0,
        PostImage = 1,
        Both = 2,
    }
}
```

All the properties except *Attributes* are required for *PluginImageAttribute*, so they are all passed in to the constructor. If no attributes are specified for an image, it is populated with all of the entity's attributes that have values.

One property that might not have a readily apparent use is *StepId*. Because an image is associated with a particular step and we can have multiple steps per plug-in, we need a way to tie the two attributes together. To do this, we assign a unique value to the *StepId* property on both the *PluginStepAttribute* and *PluginImageAttribute* classes. We need to modify the *PluginStepAttribute* class to include the following new property:

```
public string StepId { get; set; }
```

Notice that *StepId* is optional on the *PluginStepAttribute* class because it is only needed if the developer wants to specify an image for that step.

Now a developer can register for an image on her plug-in class by using two attributes together:

```
[PluginStep("Update", PluginStepStage.PreEvent, PrimaryEntityName = "account",
    StepId="AccountPreUpdate")]
[PluginImage(ImageType.PreImage, "AccountPreUpdate", "Target", "Account")]
public class MyPlugin: IPlugin
{
    ...
}
```

Notice that the *StepId* for this example is arbitrarily set to *AccountPreUpdate*. It doesn't matter what value you use as long as the values for the step and the image match.

Finally, we need to modify our console application to use the new *PluginImageAttribute* type. We need to insert the code from Listing 5-13 into the *CreateStepFromAttribute* method just before the *return* statement.

LISTING 5-13 Modifications to the *CreateStepFromAttribute* method

```
if (!String.IsNullOrEmpty(stepAttribute.StepId))
{
    List<SdkMessageProcessingStepImageRegistration> images =
        new List<SdkMessageProcessingStepImageRegistration>();
    object[] imageAttributes = pluginType.GetCustomAttributes(
        typeof(PluginImageAttribute), false);

    foreach (PluginImageAttribute imageAttribute in imageAttributes)
    {
        if (imageAttribute.StepId == stepAttribute.StepId)
        {
            images.Add(CreateImageFromAttribute(imageAttribute));
        }
    }

    if (images.Count > 0)
    {
        step.Images = images.ToArray();
    }
}
```

This change checks whether the current step attribute has a step ID assigned. If it does, the method looks for image attributes on the plugin type. If it finds any image attributes, it calls the *CreateImageFromAttribute* method, which is shown in Listing 5-14.

LISTING 5-14 The *CreateImageFromAttribute* method

```
private static SdkMessageProcessingStepImageRegistration
CreateImageFromAttribute(PluginImageAttribute imageAttribute)
{
    SdkMessageProcessingStepImageRegistration image =
            new SdkMessageProcessingStepImageRegistration();

    if (!String.IsNullOrEmpty(imageAttribute.Attributes))
    {
        image.Attributes = imageAttribute.Attributes.Split(',');
    }

    image.EntityAlias = imageAttribute.EntityAlias;
    image.ImageType = (int)imageAttribute.ImageType;
    image.MessagePropertyName = imageAttribute.MessagePropertyName;

    return image;
}
```

CreateImageFromAttribute creates a new instance of the *SdkMessageProcessingStepImage-Registration* class and populates it from the image attribute. The appropriate images are assigned back to the step's *Images* property and are automatically registered when the call to *RegisterSolution* is made.

The final code for Program.cs is shown in Listing 5-15.

LISTING 5-15 Source code for ProgrammingWithDynamicsCrm4.PluginDeploy's Program.cs

```
using System;
using System.Collections.Generic;
using System.Configuration;
using System.IO;
using System.Net;
using System.Reflection;
using System.Text;
using System.Web.Services.Protocols;
using Microsoft.Crm.Sdk;
using Microsoft.Crm.Sdk.Query;
using Microsoft.Crm.SdkTypeProxy;
using ProgrammingWithDynamicsCrm4.Plugins.Attributes;

namespace ProgrammingWithDynamicsCrm4.PluginDeploy
{
    class Program
    {
        static void Main(string[] args)
        {
            if (args.Length != 3)
            {
                string exeName = Path.GetFileName(
                    Environment.GetCommandLineArgs()[0]);
                Console.WriteLine(
                    "Usage: {0} <pluginAssembly> <crmServerUrl> <organizationName>",
                    exeName);
                Environment.Exit(1);
            }

            try
            {
                string pluginAssemblyPath = args[0];
                string crmServer = args[1];
                string organizationName = args[2];

                DeployPlugin(pluginAssemblyPath, crmServer, organizationName);
            }
            catch (SoapException e)
            {
                Console.WriteLine(e.Detail.InnerText);
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }

        private static void DeployPlugin(
            string pluginAssemblyPath,
            string crmServer,
            string organizationName)
        {
            Console.Write("Initializing CrmService... ");
            CrmService crmService = CreateCrmService(crmServer, organizationName);
            Console.WriteLine("Complete");

            pluginassembly pluginAssembly = LoadPluginAssembly(pluginAssemblyPath);

            UnregisterExistingSolution(crmService, pluginAssembly.name);

            SdkMessageProcessingStepRegistration[] steps =
                LoadPluginSteps(pluginAssemblyPath);

            RegisterSolution(crmService, pluginAssembly, steps);
        }

        private static pluginassembly LoadPluginAssembly(string pluginAssemblyPath)
        {
            Assembly assembly = Assembly.LoadFile(pluginAssemblyPath);
            pluginassembly pluginAssembly = new pluginassembly();
            pluginAssembly.name = assembly.GetName().Name;
            pluginAssembly.sourcetype = new Picklist(AssemblySourceType.Database);
            pluginAssembly.culture = assembly.GetName().CultureInfo.ToString();
            pluginAssembly.version = assembly.GetName().Version.ToString();

            if (String.IsNullOrEmpty(pluginAssembly.culture))
            {
                pluginAssembly.culture = "neutral";
            }

            byte[] publicKeyToken = assembly.GetName().GetPublicKeyToken();
            StringBuilder tokenBuilder = new StringBuilder();
            foreach (byte b in publicKeyToken)
            {
                tokenBuilder.Append(b.ToString("x").PadLeft(2, '0'));
            }
            pluginAssembly.publickeytoken = tokenBuilder.ToString();

            pluginAssembly.content =
                Convert.ToBase64String(File.ReadAllBytes(pluginAssemblyPath));

            return pluginAssembly;
        }

        private static void UnregisterExistingSolution(
            CrmService crmService,
            string assemblyName)
        {
            QueryByAttribute query = new QueryByAttribute();
            query.EntityName = EntityName.pluginassembly.ToString();
            query.ColumnSet = new ColumnSet(new string[] { "pluginassemblyid" });
            query.Attributes = new string[] { "name" };
            query.Values = new object[] { assemblyName };
            RetrieveMultipleRequest request = new RetrieveMultipleRequest();
            request.Query = query;

            RetrieveMultipleResponse response;
            Console.Write("Searching for existing solution... ");
            response = (RetrieveMultipleResponse)crmService.Execute(request);
            Console.WriteLine("Complete");

            if (response.BusinessEntityCollection.BusinessEntities.Count > 0)
            {
                pluginassembly pluginAssembly = (pluginassembly)
                    response.BusinessEntityCollection.BusinessEntities[0];
                Console.Write("Unregistering existing solution {0}... ",
                    pluginAssembly.pluginassemblyid.Value);

                UnregisterSolutionRequest unregisterRequest =
                    new UnregisterSolutionRequest();
                unregisterRequest.PluginAssemblyId =
                    pluginAssembly.pluginassemblyid.Value;

                crmService.Execute(unregisterRequest);
                Console.WriteLine("Complete");
            }
        }

        private static SdkMessageProcessingStepRegistration[] LoadPluginSteps(
            string pluginAssemblyPath)
        {
            List<SdkMessageProcessingStepRegistration> steps =
                new List<SdkMessageProcessingStepRegistration>();

            Assembly assembly = Assembly.LoadFile(pluginAssemblyPath);
            foreach (Type pluginType in assembly.GetTypes())
            {
                if (typeof(IPlugin).IsAssignableFrom(pluginType)
                    && !pluginType.IsAbstract)
                {
                    object[] stepAttributes = pluginType.GetCustomAttributes(
                        typeof(PluginStepAttribute), false);
                    foreach (PluginStepAttribute stepAttribute in stepAttributes)
                    {
                        steps.Add(CreateStepFromAttribute(pluginType,
                            stepAttribute));
                    }
                }
            }

            return steps.ToArray();
        }

        private static SdkMessageProcessingStepRegistration CreateStepFromAttribute(
            Type pluginType,
            PluginStepAttribute stepAttribute)
        {
            SdkMessageProcessingStepRegistration step =
                new SdkMessageProcessingStepRegistration();
            step.Description = stepAttribute.Description;
            step.FilteringAttributes = stepAttribute.FilteringAttributes;
            step.InvocationSource = (int)stepAttribute.InvocationSource;
            step.MessageName = stepAttribute.Message;
            step.Mode = (int)stepAttribute.Mode;
            step.PluginTypeName = pluginType.FullName;
            step.PluginTypeFriendlyName = pluginType.FullName;
            step.PrimaryEntityName = stepAttribute.PrimaryEntityName;
            step.SecondaryEntityName = stepAttribute.SecondaryEntityName;
            step.Stage = (int)stepAttribute.Stage;

            if (String.IsNullOrEmpty(step.Description))
            {
                step.Description = String.Format("{0} {1} {2}",
                    step.PrimaryEntityName, step.MessageName, stepAttribute.Stage);
            }

        if (!String.IsNullOrEmpty(stepAttribute.StepId))
        {
            List<SdkMessageProcessingStepImageRegistration> images =
                new List<SdkMessageProcessingStepImageRegistration>();
            object[] imageAttributes = pluginType.GetCustomAttributes(
                typeof(PluginImageAttribute), false);
            foreach (PluginImageAttribute imageAttribute in imageAttributes)
            {
                if (imageAttribute.StepId == stepAttribute.StepId)
                {
                    images.Add(CreateImageFromAttribute(imageAttribute));
                }
            }

            if (images.Count > 0)
            {
                step.Images = images.ToArray();
            }
        }

            return step;
        }

        private static SdkMessageProcessingStepImageRegistration
            CreateImageFromAttribute(PluginImageAttribute imageAttribute)
        {
            SdkMessageProcessingStepImageRegistration image =
                new SdkMessageProcessingStepImageRegistration();

            if (!String.IsNullOrEmpty(imageAttribute.Attributes))
            {
                image.Attributes = imageAttribute.Attributes.Split(',');
            }
            image.EntityAlias = imageAttribute.EntityAlias;
            image.ImageType = (int)imageAttribute.ImageType;
            image.MessagePropertyName = imageAttribute.MessagePropertyName;

            return image;
        }

        private static void RegisterSolution(
            CrmService crmService,
            pluginassembly pluginAssembly,
            SdkMessageProcessingStepRegistration[] steps)
        {
            RegisterSolutionRequest registerRequest = new RegisterSolutionRequest();
            registerRequest.PluginAssembly = pluginAssembly;
            registerRequest.Steps = steps;
            Console.Write("Registering solution... ");
            crmService.Execute(registerRequest);
            Console.WriteLine("Complete");
        }

        private static CrmService CreateCrmService(
            string crmServer,
            string organizationName)
        {
            UriBuilder crmServerUri = new UriBuilder(crmServer);
            crmServerUri.Path = "/MSCRMServices/2007/CrmService.asmx";

            string userName = ConfigurationManager.AppSettings["crmUserName"];
            string password = ConfigurationManager.AppSettings["crmPassword"];
            string domain = ConfigurationManager.AppSettings["crmDomain"];

            CrmService crmService = new CrmService();
            if (String.IsNullOrEmpty(userName))
            {
                crmService.UseDefaultCredentials = true;
            }
            else
            {
                crmService.Credentials = new NetworkCredential(
                    userName, password, domain);
            }

            crmService.Url = crmServerUri.ToString();
            crmService.CrmAuthenticationTokenValue = new CrmAuthenticationToken();
            crmService.CrmAuthenticationTokenValue.AuthenticationType =
                AuthenticationType.AD;
            crmService.CrmAuthenticationTokenValue.OrganizationName =
                organizationName;

            return crmService;
        }
    }
}
```

Custom Configuration

We have not yet touched on one entity tied to plug-in registration: *sdkmessageprocessing-stepsecureconfig*. You use this entity to pass a step-specific configuration value to the plug-in. The data in these entities is secure because only users with high security roles (for example, System Administrator or System Customizer) have permission to read these entities; therefore, you can safely use them to store sensitive information such as database connection strings. If you don't need the security, you can also specify a value to the *sdkmessageprocessingstep* class's *configuration* property. In our previous example you would specify the value to the *CustomConfiguration* property on the *SdkMessageProcessingStepRegistration* class.

For a plug-in to get the custom configuration value that you registered it with it must implement a constructor that takes one or two string arguments. If the version that takes two arguments exists, it will be called with the nonsecure configuration and the secure configuration as the two values. If the single argument version is implemented, it will be called with the nonsecure configuration value. Listing 5-16 shows an example of the two argument version.

LISTING 5-16 A plug-in constructor accepting custom configuration values

```
public class MyPlugin: IPlugin
{
    private string _connectionString;
    private Guid _defaultAccount;

    public MyPlugin(string unsecureConfig, string secureConfig)
    {
        if (!String.IsNullOrEmpty(unsecureConfig))
        {
            _defaultAccount = new Guid(unsecureConfig);
        }
        _connectionString = secureConfig;
    }

    ...
}
```

Deploying Referenced Assemblies

Frequently a plug-in includes dependencies on other assemblies. If those assemblies are not a part of the .NET Framework or the CRM SDK, you need to consider how to deploy them. The simplest option is to deploy them into the GAC on the CRM server. Depending on how frequently those referenced assemblies change, keeping the server's GAC up to date can be a hassle. The GAC is not easily maintained remotely, and you usually end up using Remote Desktop or something equivalent to manually copy the files into the GAC.

Tip You might recall the reference to our custom attribute class library that we added to our plug-in assembly. Because the plug-in does not reference any of the classes in the custom attribute assembly at run time, you don't need to deploy the custom attribute DLL to the Microsoft Dynamics CRM server at all!

An alternative is to use a tool called ILMerge. You use ILMerge to combine multiple .NET assemblies into a single one. This allows you to merge your plug-in DLL with any of the DLLs it references and then deploy the single DLL to the CRM database. We frequently create a post-build step on our plug-in class library project to merge the output with the dependencies.

To add a post-build step in Visual Studio, right-click the project in Solution Explorer and select Properties. Then click the Build Events tab. You can then enter command-line commands into the post-build event command line.

Here is an example of what the post-build command line might look like:

```
if not exist PreMerge mkdir PreMerge
del /Q PreMerge\*.*

move ProgrammingWithDynamicsCrm4.Plugins.dll PreMerge
move ProgrammingWithDynamicsCrm4.Plugins.pdb PreMerge
move <referencedDll> PreMerge

$(SolutionDir)Tools\ILMerge.exe /keyfile:$(ProjectDir)
ProgrammingWithDynamicsCrm4.Plugins.snk /lib:PreMerge /out:
ProgrammingWithDynamicsCrm4.Plugins.dll ProgrammingWithDynamicsCrm4.Plugins.dll
<referencedDll>
```

In this example, we want the final DLL to be in the same folder and have the same name as the original DLL, so we create a subfolder called PreMerge within the output folder. We then proceed to copy the recently compiled DLL and its dependencies into the PreMerge folder. Notice that we do not include Microsoft.Crm.Sdk.dll or Microsoft.Crm.SdkTypeProxy.dll. Because those files will be on the server, we do not need to merge them into our DLL. The final step is to execute ILMerge.exe specifying the keyfile to use to sign the assembly, the folder where it can find the DLLs to include, the name of the output file, and the list of DLLs to include in the merge.

More Info For more information on ILMerge, see *http://research.microsoft.com/~mbarnett/ILMerge.aspx*.

Debugging Plug-ins

The first thing you will probably do after deploying a plug-in is attempt to execute it to see whether it works. If you are greeted with a vague error message, you can check the Event Viewer on the CRM server for more information, but eventually you will find that you need more information, especially for more advanced plug-ins. Remote debugging and logging are two common techniques used to chase down errors in plug-ins.

Remote Debugging

By far the most powerful option, remote debugging allows you to set breakpoints in your plug-in code and step through the process in Visual Studio. The steps for setting up remote debugging are detailed in Chapter 9 in the companion book to this one: *Working With Microsoft Dynamics CRM 4.0* by Mike Snyder and Jim Steger. The CRM SDK also has information to help you set up remote debugging.

The downside to remote debugging is that it blocks other calls to the CRM application while you are stepping through your code. This can be a problem if you have multiple developers working with the same environment at the same time, and it will definitely be a problem if you are trying to debug something in a production environment.

Logging

The next-best option to discovering errors is to include extensive logging code in your plug-ins. Plug-ins typically execute in a process that is running as the Network Service user and should have rights to access the file system. You could then write some simple logging logic to output to a text file. Listing 5-17 demonstrates some simple logging code.

LISTING 5-17 Simple logging code

```
private static readonly object _logLock = new Object();
protected static void LogMessage(string message)
{
    try
    {
        if (IsLoggingEnabled)
        {
            lock (_logLock)
            {
                File.AppendAllText(LogFilePath, String.Format("[{0}] {1} {2}",
                    DateTime.Now, message, Environment.NewLine));
            }
        }
    }
    catch
    {
    }
}
```

The *IsLoggingEnabled* and *LogFilePath* properties could be initialized once at startup or be implemented to check the registry at a certain frequency to determine whether logging should occur and where the log file should be created. With this method implemented, you can add logging messages to your plug-ins to help chase down those hard-to-interpret errors:

```
if (IsLoggingEnabled)
{
    StringBuilder message = new StringBuilder();
    message.Append("InputParameters: ");
    foreach (PropertyBagEntry entry in context.InputParameters.Properties)
    {
        message.Append(entry.Name);
        message.Append(" ");
    }

    LogMessage(message.ToString());
}
```

Warning Be sure that you restrict directory access to only those users who need access to the log data, especially if the logs might contain sensitive customer data. Plug-ins execute as the user who the Microsoft Dynamics CRM Web application pool is configured to run as. By default this is the special Network Service user. This user will, of course, need write access to the log folder.

Unit Testing

Automated unit testing, used to validate the individual units of functionality in a program, continues to gain momentum and popularity in the software development community. Unit testing can improve the quality of an application and reduce the risk of breaking functionality when changes are made to the code.

Taken a step further, you can use unit tests as a design tool. Test-driven design (TDD) is a methodology that dictates that unit tests should be written before implementing the feature. The developer then implements the functionality in the simplest way possible to satisfy the unit test.

Mock Objects

Writing unit tests that depend on an external data source, such as the CRM Web Services, introduces additional challenges. Every time a test runs, the state of the server impacts the test results, causing tests that previously passed to unexpectedly fail. Because tests should only start to fail when the code changes, this server dependency needs to be removed.

Fortunately, nothing in the plug-in definition dictates that it must communicate with a live server. A plug-in only references a single type in its definition, *IPluginExecutionContext*. Because *IPluginExecutionContext* is an interface, we can provide our own implementation during testing and remove the dependency on the server. This concept of providing a "fake" implementation of an abstract type is commonly called *mocking* in automated unit testing.

Test Frameworks

In our sample test, we will take advantage of two testing frameworks. The Microsoft Unit Testing Framework, commonly referred to as MSTest, is now included in all editions of Visual Studio 2008, with the exception of the Express edition. This framework provides custom attributes used to decorate test classes and a library of assertions that you can use within your tests to validate the actual results against the expected results. In addition, MSTest integrates with the Visual Studio 2008 user interface and allows you to execute your tests without leaving the development environment.

A framework called Rhino Mocks provides our mock *IPluginExecutionContext* implementation. Rhino Mocks works by generating classes on the fly that can implement a specific interface or extend a base class. As the test authors, we define which methods the tested functionality will call and what should be returned when those calls are made.

More Info You can find more information and download instructions for Rhino Mocks at *http://www.ayende.com/projects/rhino-mocks.aspx*.

Sample Test

Now we will walk through the implementation of an automated unit test that verifies that our *AccountNumberValidator* plug-in is implemented correctly. Before we can write our first test, we need to include a test project in our solution.

Adding the test project

1. On the File Menu, select Add and then click New Project.
2. In the New Project dialog box, within the Visual C# > Test project types, select the Test Project template targeting the .Net Framework 3.0.
3. Type the name **ProgrammingWithDynamicsCrm4.PluginTests** into the Name box and click OK.
4. Delete the default UnitTest1.cs file.
5. Right-click the ProgrammingWithDynamicsCrm4.PluginTests project in Solution Explorer and then click Add Reference.

6. On the Browse tab, navigate to the CRM SDK's bin folder and select microsoft.crm.sdk.dll and microsoft.crm.sdktypeproxy.dll. Click OK.
7. Right-click the ProgrammingWithDynamicsCrm4.PluginTests project in Solution Explorer and then click Add Reference.
8. On the Project tab, select the ProgrammingWithDynamicsCrm4.Plugins project and click OK.

Now we can add our test class. Typically you would add a unit test to your project, which already contains sample code, but to introduce things one at a time, we will build the class from scratch. Create a class named *AccountNumberValidatorTests* and enter the code from Listing 5-18.

LISTING 5-18 The empty *AccountNumberValidatorTests* class

```
using System;
using Microsoft.Crm.Sdk;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using ProgrammingWithDynamicsCrm4.Plugins;

namespace ProgrammingWithDynamicsCrm4.PluginTests
{
    [TestClass]
    public class AccountNumberValidatorTests
    {
    }
}
```

Note the inclusion of the *TestClass* attribute on the *AccountNumberValidatorTests* class. This attribute, defined by the MSTest framework, indicates that the *AccountNumberValidatorTests* class contains tests and should be included when tests are run.

To define our first test, add the following code to the *AccountNumberValidatorTests* class:

```
[TestMethod]
public void TestInvalidFormat()
{
    AccountNumberValidator validator = new AccountNumberValidator();
    validator.Execute(null);
}
```

Similar to the *TestClass* attribute previously discussed, the *TestMethod* attribute identifies a method that represents a test within the test class. When all tests are run, MSTest will iterate through all the classes marked with a *TestClass* attribute and execute the methods marked with a *TestMethod* attribute individually.

You can run this test by selecting Test > Run > Tests in Current Context from the menu, but at this point it will always fail with the message "Test method ProgrammingWith-

DynamicsCrm4.PluginTests.AccountNumberValidatorTests.TestInvalidFormat threw exception: System.NullReferenceException: Object reference not set to an instance of an object." This makes sense because the *AccountNumberValidator* plug-in expects a valid (non-null) *IPluginExecutionContext* to be passed in to the *Execute* method.

To provide an implementation of the *IPluginExecutionContext* interface to the *AccountNumberValidator* class, we need to add a reference to the Rhino Mocks framework.

Adding the Rhino Mocks Reference

1. Download and extract the latest stable build of the Rhino Mocks framework that targets .NET 2.0 from *http://www.ayende.com/projects/rhino-mocks/downloads.aspx.*
2. Right-click the ProgrammingWithDynamicsCrm4.PluginTests project in Solution Explorer and then click Add Reference.
3. On the Browse tab, navigate to the Rhino Mocks framework folder and select Rhino. Mocks.dll. Click OK.

Before we define our mock implementation, we should add a using statement to the top of the AccountNumberValidatorTests.cs file to make references to the framework easier:

```
using Rhino.Mocks;
```

With the Rhino Mocks framework properly referenced, we can modify the *TestInvalidFormat* method to match Listing 5-19.

LISTING 5-19 The *TestInvalidFormat* method updated with a mock *IPluginExecutionContext*

```
[TestMethod]
public void TestInvalidFormat()
{
    MockRepository mocks = new MockRepository();
    IPluginExecutionContext context = mocks.CreateMock<IPluginExecutionContext>();

    PropertyBag inputParameters = new PropertyBag();

    DynamicEntity account = new DynamicEntity();
    account["accountnumber"] = "123456";
    inputParameters[ParameterName.Target] = account;

    using (mocks.Record())
    {
        Expect.On(context).Call(context.InputParameters).Return(inputParameters);
    }

    using (mocks.Playback())
    {
        AccountNumberValidator validator = new AccountNumberValidator();
        validator.Execute(context);
    }
}
```

The first difference we notice is the inclusion of the *mocks* variable. This instance of the *MockRepository* class is responsible for creating instances of our mock classes, as well as switching between record and playback modes, which we will discuss shortly. Creating an instance of a mock object is as simple as calling the *CreateMock* method and passing in the type you want to mock in the generics argument.

The next steps revolve around defining the expected use of the mock object. The *AccountNumberValidator* plug-in only accesses the *InputParameters* property on the *IPluginExecutionContext*. To avoid an error during test execution, we need to let Rhino Mocks know how it should respond when the *InputParameters* property is accessed. We begin by creating an instance of a *PropertyBag* and setting up the target property in it to contain a simple instance of a *DynamicEntity* with a short name.

With the local version of *inputParameters* set up and ready to go, we tell our *MockRepository* to switch to record mode. Record mode allows us to define our expectations on any mock objects. The next line might look a little odd if you are not used to dealing with mock frameworks. It reads more like English than typical C# code and tells the *MockRepository* to expect a call for the *InputParameters* property on the mock *IPluginExecutionContext*. It goes on to say that when this call is made, return our local *inputParameters* variable.

Implementing a Simple Mock Object

Looking at the overhead involved with setting up a mock framework you might find yourself wondering if it would be easier to implement the *IPluginExecutionContext* interface in your own test class. Such a class might look like this:

```
public class MockPluginExecutionContext : IPluginExecutionContext
{
    private PropertyBag _inputParameters;
    public PropertyBag InputParameters
    {
        get { return _inputParameters; }
        set { _inputParameters = value; }
    }

    // remaining IPluginExecutionContext members here...
}
```

This would allow the previous *TestShortName* method shown in Listing 5-19 to be simplified to this:

```
public void TestShortName()
{
    MockPluginExecutionContext context = new MockPluginExecutionContext();
    context.InputParameters = new PropertyBag();

    DynamicEntity account = new DynamicEntity();
    account["name"] = "ABC";
    context.InputParameters[ParameterName.Target] = account;

    AccountNumberValidator validator = new AccountNumberValidator();
    validator.Execute(context);
}
```

For simple tests such as *TestInvalidFormat*, this is a perfectly valid and simple choice for implementing a mock object. The challenges arrive when the plug-ins become more complex and start to use the *CreateCrmService* and *CreateMetadataService* methods exposed on the *IPluginExecutionContext* interface. With a mock framework you can specify that the context should return another mock implementation of *ICrmService* or *IMetadataService* when these methods are called and then further define your expectations on those mock implementations. Using your own library of mock classes, you will find it increasingly difficult to specify the expected behavior for the functionality being tested.

With the expectations defined on our mock object, we switch the *MockRepository* to playback mode, in which all the expectations must be met as defined during the record mode.

Finally, we pass our mock *IPluginExecutionContext* in to the *AccountNumberValidator*'s *Execute* method. If we run our test at this point, however, we still get a failure with the message: "Test method ProgrammingWithDynamicsCrm4.PluginTests.AccountNumberValidatorTests.TestShortName threw exception: Microsoft.Crm.Sdk.InvalidPluginExecutionException: Account number does not follow the required format (AA-######)." This, of course, is the expected behavior for our plug-in and means that it is validating as expected.

Tests that require an exception to be thrown in order for the test to pass have an additional attribute at their disposal. The *ExpectedException* attribute is applied at the method level and notifies MSTest that for this test to pass, the specific exception must be thrown. An example of the *ExpectedException* attribute applied to our *TestInvalidFormat* method can be seen here:

```
[TestMethod]
[ExpectedException(typeof(InvalidPluginExecutionException),
    "Account number does not follow the required format (AA-######).")]
public void TestInvalidFormat()
{
    ...
}
```

With this addition our test will run and pass every time, unless the *AccountNumberValidator* code is modified to change the behavior. If the test fails, it is up to the developer to modify the test accordingly—to include the new functionality or determine whether the new code has inadvertently broken something that was previously working.

For this test class to be complete, it should minimally test account numbers that are in a valid format as well. It could additionally test for a null *IPluginExecutionContext* or an *InputParameters* property that does not include a value for the "target" key. All these scenarios would be simple to include using the techniques already demonstrated.

Sample Plug-ins

Now that you have a good understanding of how plug-ins work, let's dig into some real-world examples. We will cover three different plug-in examples:

- Rolling up child entity attributes to a parent entity
- System view hider
- Customization change notifier

All these examples include source code that you can examine and use in your Microsoft Dynamics CRM deployment.

Rolling Up Child Entity Attributes to a Parent Entity

Frequently you will encounter a request to include some information in a view, such as the number of active contacts for a particular account or the next activity due date on a lead. You can accomplish this by writing a plug-in in a generic way so that it can handle all the messages involved in modifying a child record. The next example keeps track of the next scheduled phone call's *scheduledstart* value and stores it in a custom attribute on the related lead.

Start by adding a new class named *NextPhoneCallUpdater* to the ProgrammingWithDynamics-Crm4.Plugins project. Then stub out the class to match Listing 5-20.

LISTING 5-20 The start of the *NextPhoneCallUpdater* plug-in

```
using System;
using Microsoft.Crm.Sdk;
using Microsoft.Crm.Sdk.Query;
using Microsoft.Crm.SdkTypeProxy;
using ProgrammingWithDynamicsCrm4.Plugins.Attributes;

namespace ProgrammingWithDynamicsCrm4.Plugins
{
    public class NextPhoneCallUpdater : IPlugin
    {
        public void Execute(IPluginExecutionContext context)
        {
        }
    }
}
```

The first thing we need to think about is which messages our plug-in needs to register for. It needs to listen to Create and Delete messages for a *phonecall*. It also needs to listen to Update messages in case the *scheduledstart* attribute is changed or the *regardingobjectid* is changed. Finally, it needs to listen to the SetState and SetStateDynamicEntity messages to detect when the *phonecall* is marked as Complete or Cancelled. SetState and SetStateDynamicEntity are two different messages that accomplish the same thing, but you need to listen for both if you want to handle updates from the Web service API and from the CRM application. Based on this information we can add our *PluginStep* and *PluginImage* attributes to our class definition as shown in Listing 5-21.

LISTING 5-21 The *PluginStep* and *PluginImage* attributes for the *NextPhoneCallUpdater* plug-in

```
[PluginStep("Create", PluginStepStage.PostEvent,
    PrimaryEntityName = "phonecall", StepId = "PhoneCallPostCreate")]
[PluginImage(ImageType.PostImage, "PhoneCallPostCreate", "Id", "PhoneCall")]

[PluginStep("Update", PluginStepStage.PostEvent,
    PrimaryEntityName = "phonecall", StepId = "PhoneCallPostUpdate")]
[PluginImage(ImageType.Both, "PhoneCallPostUpdate", "Target", "PhoneCall")]

[PluginStep("Delete", PluginStepStage.PostEvent,
     PrimaryEntityName = "phonecall", StepId = "PhoneCallPostDelete")]
[PluginImage(ImageType.PreImage, "PhoneCallPostDelete", "Target", "PhoneCall")]

[PluginStep("SetState", PluginStepStage.PostEvent,
     PrimaryEntityName = "phonecall", StepId = "PhoneCallPostSetState")]
[PluginImage(ImageType.Both, "PhoneCallPostSetState", "EntityMoniker", "PhoneCall")]

[PluginStep("SetStateDynamicEntity", PluginStepStage.PostEvent,
    PrimaryEntityName = "phonecall", StepId = "PhoneCallPostSetStateDynamicEntity")]
[PluginImage(ImageType.Both, "PhoneCallPostSetStateDynamicEntity", "EntityMoniker",
"PhoneCall")]
public class NextPhoneCallUpdater : IPlugin
{
    ...
}
```

This probably looks like a lot of code to register for the appropriate messages, and it is. However, when you are keeping track of information on a child entity you need to account for all of the scenarios that could change your calculated value, and register messages accordingly. Therefore, you often register for these same messages whenever you need to populate one of these rolled-up attributes.

Also note the images that we set up for each step. Create gets a post-image, Delete gets a pre-image, and the rest get both types of images. The values we pass in for *MessagePropertyName* on the images come from Table 5-5.

The *Execute* method needs to determine which lead the *phonecall* is associated with in the pre-image and which it is associated with in the post-image. If they are different, both need to be updated. If they are the same, only that single lead will be updated. The *Execute* method source code is shown in Listing 5-22.

LISTING 5-22 *NextPhoneCallUpdater's Execute* method

```
public void Execute(IPluginExecutionContext context)
{
    Guid preLeadId = GetRegardingLeadId(context.PreEntityImages, "PhoneCall");
    Guid postLeadId = GetRegardingLeadId(context.PostEntityImages, "PhoneCall");

    ICrmService crmService = context.CreateCrmService(true);
    UpdateNextCallDueDate(crmService, preLeadId);

    if (preLeadId != postLeadId)
    {
        UpdateNextCallDueDate(crmService, postLeadId);
    }
}
```

The *Execute* method is fairly easy to understand, but it passes off most of the work to two additional methods, *GetRegardingLeadId* and *UpdateNextCallDueDate*. Let's start by taking a look at *GetRegardingLeadId* in Listing 5-23.

LISTING 5-23 The *GetRegardingLeadId* method

```
private Guid GetRegardingLeadId(PropertyBag images, string entityAlias)
{
    Guid regardingLeadId = Guid.Empty;

    if (images.Contains(entityAlias))
    {
        DynamicEntity entity = (DynamicEntity)images[entityAlias];

        if (entity.Properties.Contains("regardingobjectid"))
        {
            Lookup regardingObjectId = (Lookup)entity["regardingobjectid"];
            if (regardingObjectId.type == "lead")
            {
                regardingLeadId = regardingObjectId.Value;
            }
        }
    }

    return regardingLeadId;
}
```

Because not all messages have a pre-image and a post-image, we wrote this method to be forgiving if the image is not found. If the phone call image is found and the *regardingobjectid* attribute is associated with a lead, the method returns the *Guid* from *regardingobjectid*. Otherwise, it returns an empty *Guid*.

Once the lead IDs are identified, we need to update the attribute on the corresponding leads. *UpdateNextCallDueDate* is responsible for performing this functionality. Listing 5-24 is the full source code for the *UpdateNextCallDueDate* method.

LISTING 5-24 The *UpdateNextCallDueDate* method

```
private void UpdateNextCallDueDate(ICrmService crmService, Guid leadId)
{
    if (leadId != Guid.Empty)
    {
        DynamicEntity lead = new DynamicEntity("lead");
        lead["leadid"] = new Key(leadId);

        DynamicEntity phoneCall = GetNextScheduledPhoneCallForLead(crmService,
            leadId);
        if (phoneCall != null)
        {
            lead["new_nextphonecallscheduledat"] = phoneCall["scheduledstart"];
        }
        else
        {
            lead["new_nextphonecallscheduledat"] = CrmDateTime.Null;
        }

        crmService.Update(lead);

    }
}
```

UpdateNextCallDueDate is responsible for updating the custom *new_nextphonecallesched-uledat* attribute on the lead. It sets it to the earliest *scheduledstart* value for phone calls associated with this lead. If no phone calls are associated with the lead (or they do not have *scheduledstart* values), it nulls out the *new_nextphonecallscheduledat* attribute on the lead.

UpdateNextCallDueDate calls one additional method, *GetNextScheduledPhoneCallForLead*, to determine which phone call it should use. The source code for *GetNextScheduledPhoneCall-ForLead* is displayed in Listing 5-25.

LISTING 5-25 The *GetNextScheduledPhoneCallForLead* method

```
private DynamicEntity GetNextScheduledPhoneCallForLead(
    ICrmService crmService, Guid leadId)
{
    QueryExpression query = new QueryExpression();
    query.EntityName = EntityName.phonecall.ToString();

    ColumnSet cols = new ColumnSet(new string[] { "scheduledstart" });
    query.ColumnSet = cols;

    FilterExpression filter = new FilterExpression();
    query.Criteria = filter;

    ConditionExpression regardingCondition = new ConditionExpression();
    regardingCondition.AttributeName = "regardingobjectid";
    regardingCondition.Operator = ConditionOperator.Equal;
    regardingCondition.Values = new object[] { leadId };
    filter.Conditions.Add(regardingCondition);

    ConditionExpression activeCondition = new ConditionExpression();
    activeCondition.AttributeName = "statecode";
    activeCondition.Operator = ConditionOperator.Equal;
    activeCondition.Values = new object[] { "Open" };
    filter.Conditions.Add(activeCondition);

    ConditionExpression scheduledCondition = new ConditionExpression();
    scheduledCondition.AttributeName = "scheduledstart";
    scheduledCondition.Operator = ConditionOperator.NotNull;
    filter.Conditions.Add(scheduledCondition);

    query.PageInfo = new PagingInfo();
    query.PageInfo.Count = 1;
    query.PageInfo.PageNumber = 1;
    query.AddOrder("scheduledstart", OrderType.Ascending);

    RetrieveMultipleRequest request = new RetrieveMultipleRequest();
    request.Query = query;
    request.ReturnDynamicEntities = true;

    RetrieveMultipleResponse response;
    response = (RetrieveMultipleResponse)crmService.Execute(request);

    DynamicEntity phoneCall = null;
    if (response.BusinessEntityCollection.BusinessEntities.Count == 1)
    {
        phoneCall = (DynamicEntity)
            response.BusinessEntityCollection.BusinessEntities[0];
    }
    return phoneCall;
}
```

GetNextScheduledPhoneCallForLead constructs a *QueryExpression* that filters for active *phonecall* entities that are associated with the specified lead and have a *scheduledstart* value. The query is set to return only one record and is sorted by the *scheduledstart* attribute in ascending order. If no matching *phonecall* is found, it returns *null*.

The end result is that regardless of how a *phonecall* entity is created, updated, or deleted, the parent entity's attribute will always be recalculated.

System View Hider

Microsoft Dynamics CRM includes default system views for entities such as accounts, contacts, and others. You might find that your organization does not want to use all these system views, and therefore you'd like to remove one or more of them. Unfortunately, if you try to customize the entity with the Web interface to delete a system view, you will receive an error message stating that you cannot remove a system view.

Fortunately, however, you can use a plug-in to hide specific system views from your users. For this sample, let's assume we want to hide the No Orders in Last 6 Months system view on both the account and contact entities. The plug-in can do this because CRM queries for the list of *systemview* entities associated with a particular entity when displaying the picklist of views. This plug-in example is fairly straightforward to implement, so let's start by looking at the completed code in Listing 5-26.

LISTING 5-26 *SystemViewHider* plug-in source code

```
using System;
using Microsoft.Crm.Sdk;
using Microsoft.Crm.Sdk.Query;
using Microsoft.Crm.SdkTypeProxy;
using ProgrammingWithDynamicsCrm4.Plugins.Attributes;

namespace ProgrammingWithDynamicsCrm4.Plugins
{
    [PluginStep("RetrieveMultiple", PluginStepStage.PreEvent,
        PrimaryEntityName="savedquery")]
    public class SystemViewHider : IPlugin
    {
        public void Execute(IPluginExecutionContext context)
        {
            object[] systemViews = new object[]
            {
                //Contacts: No Orders in Last 6 Months
                new Guid("9818766E-7172-4D59-9279-013835C3DECD"),

                //Accounts: No Orders in Last 6 Months
                new Guid("C147F1F7-1D78-4D10-85BF-7E03B79F74FA"),
            };

            if (context.InputParameters != null && systemViews.Length > 0)
            {
                if (context.InputParameters.Properties.Contains(
                    ParameterName.Query))
                {
                    QueryExpression query;
                    query = (QueryExpression)
                        context.InputParameters[ParameterName.Query];
                    if (query.EntityName == EntityName.savedquery.ToString())
                    {
                        if (query.Criteria != null)
                        {
                            if (query.Criteria.Conditions != null)
                            {
                                ConditionExpression condition =
                                    new ConditionExpression();
                                condition.AttributeName = "savedqueryid";
                                condition.Operator = ConditionOperator.NotIn;
                                condition.Values = systemViews;

                                query.Criteria.Conditions.Add(condition);

                                context.InputParameters[ParameterName.Query] =
                                    query;
                            }
                        }
                    }
                }
            }
        }
    }
}
```

The first thing to notice is this plug-in takes advantage of a message that might not be as obvious as some that we have dealt with in the past. The RetrieveMultiple message is a valid message to register for, and as is shown here you can manipulate the *QueryExpression* being passed to it before the core operation is executed.

The other factor that allows this plug-in to work is that the system view IDs for native entities are always the same across CRM installations. If this were not the case, we would need to specify the view IDs during registration in the *customconfiguration* attribute for the plug-in step or perform a query within the plug-in to find the right view ID to exclude.

Customization Change Notifier

Customers often ask how they can keep track of customization changes in a development environment, or even for auditing in a production environment. If multiple people possess system administrator privileges, they could be making customization changes to the system at the same time. This might cause confusion or problems, especially if multiple users work with the same entity at the same time.

Obviously a good software development process dictates that developers and system administrators should communicate and follow a strict process when making changes to any environment. However, we created a sample plug-in that records customization changes. The plug-in presented in this sample doesn't prevent two users from working on the same records at the same time, but you can use it in conjunction with your development process as a safety net.

The *CustomizationChangeNotifier* plug-in listens for the Publish and PublishAll messages. To specify which users to notify of customization changes, we have added a custom Boolean attribute named *new_receivecustomizationnotifications* on the *systemuser* entity. By checking the corresponding check box on the *systemuser* form, the user is included in the notification e-mails. This plug-in differs from previous samples because it subscribes to both the pre-event and post-event steps and passes information between the two steps. Listing 5-27 shows the start of the *CustomizationChangeNotifier* code.

LISTING 5-27 *CustomizationChangeNotifier*

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Text;
using System.Xml;
using System.Xml.Xsl;
using Microsoft.Crm.Sdk;
using Microsoft.Crm.Sdk.Query;
using Microsoft.Crm.SdkTypeProxy;
using ProgrammingWithDynamicsCrm4.Plugins.Attributes;

namespace ProgrammingWithDynamicsCrm4.Plugins
{
    [PluginStep("Publish", PluginStepStage.PreEvent)]
    [PluginStep("Publish", PluginStepStage.PostEvent)]
    [PluginStep("PublishAll", PluginStepStage.PreEvent)]
    [PluginStep("PublishAll", PluginStepStage.PostEvent)]
    public class CustomizationChangeNotifier : IPlugin
    {
        public void Execute(IPluginExecutionContext context)
        {
        }
    }
}
```

So far everything looks pretty normal, with the exception of the already mentioned fact that we registered this plug-in for both the pre-event and post-event steps. Listing 5-28 fills out the *Execute* method, and things start to get interesting.

LISTING 5-28 *CustomizationChangeNotifier's Execute* method

```
public void Execute(IPluginExecutionContext context)
{
    if (context.Stage ==
        MessageProcessingStage.BeforeMainOperationOutsideTransaction)
    {
        byte[] preXml = GetCustomizationSnapshot(context);
```

```
            context.SharedVariables["CustomizationChangeNotifier.PreXml"] = preXml;
        }
        else
        {
            SendNotification(context,
                (byte[])context.SharedVariables["CustomizationChangeNotifier.PreXml"]);
        }

    }
```

Because this plug-in executes in two different steps it needs to determine which step it is executing in right away and call the appropriate method. During the pre-event step, this plug-in grabs a snapshot of the customizations and stores them in the context's *SharedVariables PropertyBag*. Then, during the post-event step, it gets that customization snapshot out of *SharedVariables* and passes it on to the *SendNotification* method.

SharedVariables is shared by all plug-ins within a pipeline. Because of this, you should be sure the keys you use are likely to be unique. The only reason we use a byte array here is to deal with the compressed version of the customization data. It is also worth mentioning that we could have implemented this plug-in as two different plug-ins, each registered for its own step, but both steps have enough shared functionality that it made sense to use a single plug-in class. Let's examine the source code for *GetCustomizationSnapshot* in Listing 5-29.

LISTING 5-29 *GetCustomizationSnapshot*

```
private byte[] GetCustomizationSnapshot(IPluginExecutionContext context)
{
    ICrmService crmService = context.CreateCrmService(true);

    if (context.MessageName == "Publish")
    {
        ExportCompressedXmlRequest request = new ExportCompressedXmlRequest();
        string parameterXml = (string)context.InputParameters["ParameterXml"];
        request.ParameterXml = TransformParameterXmlToExportXml(parameterXml);
        request.EmbeddedFileName = "customizations.xml";

        ExportCompressedXmlResponse response =
            (ExportCompressedXmlResponse)crmService.Execute(request);
        return response.ExportCompressedXml;
    }
    else
    {
        ExportCompressedAllXmlRequest request =
            new ExportCompressedAllXmlRequest();
        request.EmbeddedFileName = "customizations.xml";
        ExportCompressedAllXmlResponse response =
            (ExportCompressedAllXmlResponse)crmService.Execute(request);
        return response.ExportCompressedXml;
    }
}
```

If you recall, not only did we register this plug-in for two steps, but we also registered it for two messages. Depending on whether the message is Publish or PublishAll, the plug-in will either get a subset of the current customizations or all of them. The two messages, ExportCompressedXml and ExportCompressedAllXml, are used to get the customization changes from CRM. The *EmbeddedFileName* property is used to specify the name of the file that is embedded in the zip file that is returned.

Unfortunately, the *ParameterXml* passed in through the context's *InputParameters PropertyBag* is not quite the same as what is required by the ExportCompressedXml message. ExportCompressedXml requires all of the root node's children (entities, nodes, security roles, workflows, and settings) even if you are not asking for any of those customization types. The *ParameterXml* only contains the customizations that are being published. Because of this slight difference, we need to do a transformation on the XML as shown in Listing 5-30.

LISTING 5-30 The *TransformParameterXmlToExportXml* method

```
private string TransformParameterXmlToExportXml(string parameterXml)
{
    string xsl = @"
        <xsl:transform version='1.0'
          xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
          <xsl:template match='/'>
            <importexportxml>
              <entities>
                <xsl:apply-templates select='publish/entities/entity' />
              </entities>
              <nodes>
                <xsl:apply-templates select='publish/nodes/node' />
              </nodes>
              <securityroles>
                <xsl:apply-templates select='publish/securityroles/securityrole' />
              </securityroles>
              <workflows>
                <xsl:apply-templates select='publish/workflows/workflow' />
              </workflows>
              <settings>
                <xsl:apply-templates select='publish/settings/setting' />
              </settings>
            </importexportxml>
          </xsl:template>

          <xsl:template match='@*|node()'>
            <xsl:copy>
              <xsl:apply-templates select='@*|node()'/>
            </xsl:copy>
          </xsl:template>

        </xsl:transform>";

    XslCompiledTransform transform = new XslCompiledTransform();
    transform.Load(XmlReader.Create(new StringReader(xsl)));
    XmlTextReader publishXmlReader =
        new XmlTextReader(new StringReader(parameterXml));
    publishXmlReader.Namespaces = false;

    StringBuilder results = new StringBuilder();
    XmlWriter resultsWriter = XmlWriter.Create(results);
    transform.Transform(publishXmlReader, null, resultsWriter);

    return results.ToString();
}
```

Most of this method is just the declaration of the XSLT. While the specific details of the XSLT are outside the scope of this book, an abundance of information is available about XSLT both in books and on the Internet. The rest of the code in this method is simply using the XSLT to transform the *ParameterXml* passed in to the return value, which is passed to the *ExportCompressedXmlRequest*.

As shown back in Listing 5-28, when the *Execute* method is called for the post-event step, it passes along the plug-in context and the compressed XML from the pre-event step to the *SendNotification* method. The source code for the *SendNotification* method is displayed in Listing 5-31.

LISTING 5-31 The *SendNotification* method

```
private void SendNotification(IPluginExecutionContext context, byte[] preXml)
{
    ICrmService crmService = context.CreateCrmService(true);
    activityparty[] recipients = GetNotifcationRecipients(crmService);

    if (recipients.Length > 0)
    {
        byte[] postXml = GetCustomizationSnapshot(context);

        email email = new email();
        email.from = new activityparty[1];
        email.from[0] = new activityparty();
        email.from[0].partyid = new Lookup("systemuser", context.UserId);
        email.subject = "Customization Notification";
        email.description = @"You are receiving this email
            because a customization change has been published.";
        email.to = recipients;

        Guid emailId = crmService.Create(email);

        emailId = CreateEmailAttachment(crmService, emailId, preXml,
            "PreCustomizations.zip", "application/zip", 1);
        emailId = CreateEmailAttachment(crmService, emailId, postXml,
            "PostCustomizations.zip", "application/zip", 2);
        SendEmailRequest sendRequest = new SendEmailRequest();
        sendRequest.EmailId = emailId;
        sendRequest.IssueSend = true;
        sendRequest.TrackingToken = String.Empty;
        crmService.Execute(sendRequest);
    }
}
```

SendNotification starts by getting a list of recipients that have indicated they want to be notified of customization changes. As long as at least one user has indicated that he or she would like to receive change notifications, another snapshot of the customizations is taken that can be used to compare against the customizations that are captured in the pre-event step. An e-mail message is then prepared, including both snapshots of the customization files as attachments, and sent to the list of recipients.

GetNotificationRecipients, as shown in Listing 5-32, queries to find which system users have the custom attribute *new_receivecustomizationnotifications* set to true and returns an array of *activityparty* instances that reference them.

LISTING 5-32 The *GetNotificationRecipients* method

```
private activityparty[] GetNotifcationRecipients(ICrmService crmService)
{
    QueryByAttribute query = new QueryByAttribute();
    query.EntityName = "systemuser";
    query.ColumnSet = new ColumnSet(new string[] { "systemuserid" });
    query.Attributes = new string[] { "new_receivecustomizationnotifications" };
    query.Values = new object[] { true };

    RetrieveMultipleRequest request = new RetrieveMultipleRequest();
    request.Query = query;
    request.ReturnDynamicEntities = true;

    RetrieveMultipleResponse response;
    response = (RetrieveMultipleResponse)crmService.Execute(request);
    List<BusinessEntity> systemUsers =
        response.BusinessEntityCollection.BusinessEntities;

    List<activityparty> recipients = new List<activityparty>();
    foreach (DynamicEntity systemUser in systemUsers)
    {
        activityparty recipient = new activityparty();
        Guid systemUserId = ((Key)systemUser["systemuserid"]).Value;
        recipient.partyid = new Lookup("systemuser", systemUserId);
        recipients.Add(recipient);
    }

    return recipients.ToArray();
}
```

The last remaining piece of code is the *CreateEmailAttachment* method, which is displayed in Listing 5-33. As the name implies, this method creates an attachment for an e-mail message in CRM.

LISTING 5-33 The *CreateEmailAttachment* method

```
private static Guid CreateEmailAttachment(ICrmService crmService, Guid emailId,
    byte[] data, string filename, string mimeType, int attachmentNumber )
{
    activitymimeattachment emailAttachment = new activitymimeattachment();
    emailAttachment.activityid = new Lookup("email", emailId);
    emailAttachment.attachmentnumber = new CrmNumber(attachmentNumber);
    emailAttachment.mimetype = mimeType;
    emailAttachment.body = Convert.ToBase64String(data);
    emailAttachment.filename = filename;

    crmService.Create(emailAttachment);
    return emailId;
}
```

This sample demonstrates some of the more creative and powerful uses of plug-ins and *SharedVariables* as well as illustrating how to send an e-mail message with attachments using the CRM service.

Summary

Microsoft Dynamics CRM 4.0 offers a powerful means of extensibility through plug-ins, which you can register directly into the event execution pipeline. You can register and deploy plug-ins by using existing tools or by implementing your own deployment tools using an API. Because Microsoft Dynamics CRM implements plug-ins with no dependencies on specific class implementations, they are a good target for automated unit tests. The number of messages that can be registered for is significantly larger than any previous version of Microsoft Dynamics CRM and allows developers to extend CRM much further than was possible in the past.

Index

A

access rights, 81–82
accessibility, testing, 336–340
AccessRights enumeration, 91–92
Active Directory
  authentication support, 39, 56
  description, 20
activeworkflowid attribute, 409, 418
activities, workflow
  binding, 187
  implementing, 188–193
  investigating errors, 201–203
  overview, 186–187
  testing, 197–201
activities attribute, 409
ActivityExecutionContext class, 204
Add-Remove Programs (ARP), 317
Advanced Find technique, 202
AllowList registry key, 371–372
allow lists, 331–332
APIs (application programming interfaces). *See also* specific APIs
  accessing in Visual Studio 2008, 50–54
  offline access, 70, 347
  offline deployment, 330–332
  online access, 70–73
Application server role, 22
ApplicationOrigin class, 122
ARP (Add-Remove Programs), 317
ASP.NET
  dialog box development, 464–478
  IFrame development, 450–464
  installing Microsoft ASP.NET Cassini, 348
  ISV page development, 478–491
ASPX extension, 13, 372
assemblies
  adding references, 495
  creating custom, 188–193
  declarative workflows, 422–423
  deploying, 147–148, 193–196, 326–328
  multilingual strings, 395–402
  registering, 117–118, 193–195
Assembly Location registration property, 114, 116
assign record (workflow step), 180, 186
AssignActivity class, 186
AssignUserRolesRoleRequest, 88, 323
AsyncServiceOrigin class, 122
AttributeDefinitionDependency class, 433–434
AttributeMetadata class
  DisplayMask property, 293–295, 390
  functionality, 292
attributes
  altering database structure, 5
  conditionally enabling, 556–557
  CrmService Web service, 66
  display masks, 293–295
  remote customization, 292–298
  retrieving, 282
  rolling up, 156–160
  Searchable property, 293
  workflow, 203
  workflow entity, 409–411
  workflowdependency entity, 414–415
  XAML syntax, 420
authentication
  Active Directory support, 39
  API services, 56
  configuring security tokens, 57
  CrmImpersonator class, 325
  CrmService Web service, 56–58
  deployment considerations, 317
  filtered views, 98–99
  method calls, 57–58
  testing deployment, 335
  Windows Live ID, 71
automated installation, 316
automatic workflows, 178–179
availability attribute, 352
AvailableOffline attribute, 351

B

binding
  CrmEntityPicklistControl, 520
  dynamic values in workflows, 187
Boolean attribute type, 494
Browsable attribute, 501
Bulk edit form type, 254
business logic, implementing, 9–11

C

caching
  deployment considerations, 318
  example file system, 305
  external files, 259
calendar pop-up, 512–514
CallerId property, 83
CALs (Client Access Licenses)
  license types, 22
  overview, 21
CanBeManyToManyRequest, 303
CanBeReferencedRequest, 303
CanBeReferencingRequest, 303
CassiniPort registry entry, 353
change notifier, customizations, 162–168
change status (workflow step), 181, 186
check condition (workflow step), 180, 186
child controls, 512
child entities, 156–160, 221–231
child pipelines, 121
child workflows, 179
ChildWorkflowActivity class, 186
Clear operator, 182
click event, 473
Client Access Licenses (CALs)
  license types, 22
  overview, 21
Client attribute, 350–351
ClientAuthOrganizationName registry entry, 353
client-side processing
  events supported, 238–241
  form scripting, 234–238
  offline development environment, 349–350
close_click function, 473
columns. *See* attributes
conditional branch (workflow step), 180, 186
configuration settings, 330
connection strings, 95, 318
Content controls, 472
continuous integration, 40, 42
Convert class, 103
Create form type, 253
create record (workflow step), 180, 186
CreateActivity class, 186
CreateDataSource method, 544
createdby attribute
  workflow entity, 409
  workflowdependency attribute, 414
createdon attribute
  workflow entity, 409
  workflowdependency entity, 414
CreateEntityRequest, 288
CreateOneToManyRequest, 299
credentials, authenticating, 56
CRM queries, 203
CRM Web site, 326
CrmAppPool Properties dialog box, 83–84
CrmAttributeTarget attribute, 203
CrmAuthenticationToken, 242
CrmBooleanControl
  adding options, 506
  AttributeName property, 506
  EntityName property, 506
  functionality, 503
  programming, 503–504
  testing, 506
CrmDateTimeControl
  adding child controls, 512
  AttributeName property, 512
  CreateControlHierarchy method, 512
  creating calendar pop-up, 512–514
  EntityName property, 512
  functionality, 507
  programming, 507
  testing, 515
  Value property, 512
CrmDefault attribute, 203

CrmDiscoveryService Web service
  additional information, 67
  functionality, 7, 67–68
CrmEntityPicklistControl
  EntityName property, 520
  Filter property, 520
  functionality, 516
  IncludeNullOption property, 520
  programming, 516–517
  testing, 521–522
crmForm
  attributes supported, 237–238
  collection methods, 236
  collection properties, 236
  FormType property, 253–255
  methods supported, 236, 238
  properties supported, 235, 238
CrmGridViewControl
  adding columns, 535–536
  AutoGenerateColumns property, 535
  Columns property, 535
  creating DataSource, 544
  creating template, 537–543
  DataSource property, 535
  DataSourceID property, 535
  functionality, 523
  IsUserQuery property, 535
  onRowCancelingEdit method, 544
  OnRowEditing method, 544
  OnRowUpdating method, 544
  programming, 523–544
  testing, 545–548
  updating records, 544
  ViewId property, 535
CrmGridViewTemplate
  control mapping, 543
  creating, 537–542
  EntityPicklistEntityName property, 543
  EntityPicklistTextField property, 543
  EntityPicklistValueField property, 543
CrmImpersonator class
  deploying custom Web pages, 325
  IFD support, 69
  MetadataService Web service, 276
CrmInput attribute, 203
CrmLabel class, 285–287, 291, 390
CrmMetadata class, 281, 292
CrmOutlookService class
  GoOffline method, 354
  GoOnline method, 354
  IsCrmClientLoaded property, 354
  IsCrmClientOffline property, 354
  IsCrmDesktopClient property, 355
  OutlookSyncType enumeration, 354
  ServerUri property, 355
  SetOffline method, 354
  State property, 355
  Sync method, 354
CrmOutput attribute, 203
CrmPicklistControl
  adding options, 501–502
  AttributeName property, 500
  DataSource property, 501
  DataSourceID property, 501
  DataTextField property, 501
  DataValueField property, 501
  EntityName property, 500
  functionality, 496
  IncludeNullOption property, 500
  Items property, 501
  programming, 496–497
  testing, 502–503
CrmReferenceTarget attribute, 203, 217
CrmService Web service
  accessing, 50
  adding directly, 52–53
  additional information, 36
  attributes, 66
  authentication, 56–58
  AuthenticationType property, 57
  availability attribute, 352
  caching considerations, 318
  Create method, 58, 429, 432
  CreateCrmService method, 85
  creating console projects, 59
  creating reports, 329
  Credentials property, 56–58
  CrmAuthenticationTokenValue property, 57
  customizations management, 343
  Delete method, 58

Execute method, 54, 62
ExecuteAsync method, 384
Fetch method, 58, 544
form scripting, 242
functionality, 7, 54–55
GetCrmService method, 549
interacting with workflows, 411–412
MetadataService Web service and, 279
multilingual support, 376–384
offline functionality, 352–354, 372
publishing customizations, 323
Retrieve method, 58, 384, 475
RetrieveMultiple method, 58, 418, 475
retrieving records, 60–62
Update method, 58, 460
Url property, 54–55

CrmServiceUtility class
- functionality, 549
- GetCrmService method, 549
- GetMetadataService method, 549
- GetServerUrlFromRegistry method, 549

CrmWorkflow class, 423, 438
CrmWorkflowActivity attribute, 203, 207
cross-site scripting (XSS), 255–257
CruiseControl.NET, 42
currencies
- programming applications, 403–405
- support considerations, 15–16, 375

custom code, writing, 7
custom controls
- adding keys to web.config, 496
- adding references to assemblies, 495
- adding Web site projects, 495
- benefits of developing, 493
- complex attribute types, 494
- creating project, 494–495
- CrmBooleanControl, 503–506
- CrmDateTimeControl, 507–515
- CrmEntityPicklistControl, 516–522
- CrmGridViewControl, 523–548
- CrmPicklistControl, 496–503

custom reports, deploying, 329–330
custom workflow management tool, 407–408
CustomEntityDefinitionDependency class, 435
customentityname attribute, 414
customizations
- change notifier, 162–168
- deployment considerations, 321–323, 343
- importing, 321–322
- MetadataService Web service, 322
- offline considerations, 16, 347, 361–364
- publishing, 323
- supported during upgrades, 14

customizations utility, 553–555

D

data access, 94–99
data binding, 520
data encryption
- one-way, 100–103
- two-way, 103–107

data manipulation, 7
data migration
- custom code, 34–36
- Data Migration Manager, 34–35
- native import, 34–35
- third-party tools, 34–35

Data Migration Manager, 34
database diagrams, 6
database servers, 335
database structures, 5–7
DateTime attribute type, 494
DebugBar tool, 45
debugging
- form scripting, 244–247
- implementing activity, 207–213
- plug-ins, 149–150

declarative workflows
- adding assemblies, 422–423
- adding rules to workflow definition, 425–427
- adding workflow definition, 421–422
- creating workflow project, 421
- CRM support, 420
- defining local parameters, 424
- deploying, 428–442
- designing, 424–425
- examining native workflow, 442–444
- exporting programmatically, 444–447
- XAML syntax, 419–420

Decrement by operator, 182
default action (workflow step), 180, 186
DeleteEntityRequest, 291, 320
DeleteOptionValue message, 297–298
DeleteRelationshipRequest, 299
dependentattributename attribute, 414
dependententityname attribute, 414
deployment
  assemblies, 326–328
  build step, 313–315, 344
  component inventory, 344
  configuration settings, 330
  configure step, 313, 317–319, 345
  custom assemblies, 193–196
  custom reports, 329–330
  custom Web pages, 325–326
  customization considerations, 321–323, 343
  declarative workflows, 428–442
  deleting metadata, 320
  deleting transactional data, 320
  dialog boxes, 476–478
  example sequence, 343–345
  form scripting considerations, 244–247, 259
  IFrame pages, 361–365, 460–464
  in offline development environment, 349–350
  install step, 313, 315–317, 344
  interface changes, 324–325
  ISV page, 490–491
  missing prerequisites, 341–343
  multilingual IFrames, 391–394
  offline API, 330–332
  online Help files, 328–329
  plug-ins, 113–119, 128–148, 326–328, 331–332, 371
  referenced assemblies, 147–148
  security roles, 323
  templates, 323
  testing strategies, 333–340
  uninstall step, 313, 319–320, 345
  workflow rules, 323
Deployment registration property, 113, 115
description attribute, 409
development environment
  development considerations, 24–25, 46–47
  hybrid environment, 25, 30
  isolated development, 25–28
  multi-tenant functionality, 25, 32
  offline considerations, 349–350
  Outlook customizations, 32–33
  redeployment, 36–40
  setting up, 16, 24
  sharing common organization, 25, 28–30
  support for, 17–18
  testing considerations, 33, 247
development tools
  additional, 40
  continuous integration, 40, 42
  IDE, 40–41
  import/export utility, 13, 34–35, 40, 42–44
  Installer, 40
  source control, 40–42
device CALs, 21
DHTML (dynamic HTML), 233–234, 257
dialog boxes
  deploying, 476–478
  programming, 465–476
  testing, 476–478
Disabled form type, 254
DiscoveryService Web service
  accessing, 50–51
  locating MetadataService URL, 274
div tags, 483
DNE (Do Not Email) list, 343–344
DNE Provisioning Utility, 345
Do Not Email (DNE) list, 343–344
DOM model, 259
DropDownList class, 500–501
dynamic HTML (DHTML), 233–234, 257
dynamic values in workflows
  binding, 187
  overview, 181–185
DynamicEntity class
  assembly references, 51
  Execute method, 65

Name property, 65
overview, 65–66
Properties property, 65
DynamicEntityUtility class, 549, 551

E

EditorBrowsable attribute, 501
encryption
one-way, 100–103
two-way, 103–107
entities
additional information, 6
altering database structure, 5
creating, 288–290
deleting, 291
event execution pipeline, 120–121
field-level security, 563
importing, 34
plug-in, 128
remote customization, 288–292
retrieving, 279–281
rolling up attributes, 156–160
updating, 291
workflows as, 409–419
entity images. *See* images
Entity registration property, 114–115
entity relationships
additional information, 6
altering database structure, 5
remote customization, 298–304
entityattributes attribute, 415
EntityId property, 94
EntityImageDependency class, 438
EntityMetadata class, 291, 353
EqualUserId operator, 569
error handling, 201–203, 308
eval method (JavaScript), 261
event execution pipeline, 120–122
event processing pipeline, 127
exception handling, 127
ExpectedException attribute, 155
ExportCompressedTranslationsXml message, 376
exporting
with import/export utility, 13, 34–35, 40, 42–44
workflows programmatically, 444–447
ExportTranslationsXml message, 376
Extensible Application Markup Language. *See* XAML (Extensible Application Markup Language)

F

Fiddler tool, 45, 242
field-level security
defined, 563
hiding tabs/fields on forms, 564–565
hiding/showing fields, 565–566
hiding/showing labels, 565–566
reviewing script, 566–571
script examples, 572–579
fields, hiding, 564–566
filtered views
defined, 7, 96
deployment considerations, 318
programming considerations, 98–99
security considerations, 96–97
Form Assistant, 181–185
form controls, 8
form layouts, 8
form scripting
available events, 238–241
calling Web services, 241
client-side processing, 234–238
configuring events, 9
CRM API SOAP request, 241–244
displaying tooltip information, 262
enhancing display, 266–268
examples, 261–268
field-level security, 563–579
for validation, 257–258
form type considerations, 253–255
global methods, 235
global variables, 235
hiding navigation links, 558–562
ISV.Config buttons, 248–253
loading external JavaScript files, 259–261

offline considerations, 355
setting title at run time, 265
testing and deployment, 244–247
triggering code, 9
XSS, 255–257
form-customization tool, 12–13
FxCop tool, 45

G

GAC (Global Assembly Cache)
deploying plug-ins, 116, 326
deploying referenced assemblies, 147
MSI support, 317
GenerateAuthenticationHeader helper method, 242
GetSecurityObjects function, 570
GetUserRoles function, 569
GetValidManyToManyRequest, 303
GetValidReferencedEntitiesRequest, 303
GetValidReferencingEntitiesRequest, 303
Global Assembly Cache (GAC)
deploying plug-ins, 116, 326
deploying referenced assemblies, 147
MSI support, 317
global methods, 235
global variables, 235
globally unique identifier (GUID), 83
GridView control, 523
GUID (globally unique identifier), 83

H

hardware requirements
additional information, 23
Microsoft Dynamics CRM 4.0 Web Server, 23–24
HTTP_USER_AGENT, 334

I

ICrmService interface, 204, 224, 335
Id attribute, 237
IDCRL (Identity Client Run-Time Library), 71
IDE (Integrated Development Environment), 40–41
Identity Client Run-Time Library (IDRTL), 71
IE Dev Toolbar tool, 46
IFD (Internet-facing deployment) option
connecting to, 69
CrmImpersonator class, 325
MetadataService Web service, 273
testing authentication, 335
IfElseActivity class, 186
IfElseBranchActivity class, 186
IFrame pages
deploying, 361–365, 460–464
developing for offline, 356–366
field-level security, 563
form scripting, 257
functionality, 8, 12
multilingual applications, 385–394
potential uses, 450–468
programming, 356–361, 451–460
query string parameters, 450–451
testing, 361–365
testing offline, 365–366
updating with resource files, 396–401
IIS (Internet Information Services), 343
ILMerge tool, 45
images
defined, 138
in custom workflow activities, 206–213
programmatic registration, 139–142
specifying for plug-ins, 138–139
Images registration property, 114, 116–119
IMetadataService interface, 126, 204, 274–275
impersonation
Credentials property, 83
defined, 82
during execution, 127
during registration, 126
for plug-ins, 83–85
for Web applications, 82–83
Import Organization Wizard, 37, 39
ImportAllXml message, 322
ImportCompressedAllXml message, 322
ImportCompressedXml message, 321

import/export utility, 13, 34–35, 40, 42–44
importing
  custom security entities, 573–574
  customizations, 321–322
  entities, 34
  organization, 37, 39, 323
ImportTranslationsXmlWithProgress message, 376, 384
ImportXml message, 322
Increment by operator, 182
InsertOptionValue message, 297
installer programs, 40, 42–44, 316
Integrated Development Environment (IDE), 40–41
Internet Explorer, 244
Internet Information Services (IIS), 343
InvalidPluginExecutionException, 127
IPlugin interface, 122
IPluginExecutionContext interface
  BusinessUnitId property, 122
  CallerOrigin property, 122, 370
  CorrelationId property, 122
  CorrelationTokenValue property, 122
  CorrelationUpdatedTime property, 122
  CreateCrmService method, 122, 126
  CreateMetadataService method, 126, 155, 275
  Depth property, 122
  functionality, 122–126
  InitiatingUserId property, 123
  InputParameters property, 123, 154
  InvocationSource property, 123
  IsExecutingOfflineMode property, 123
  MessageName property, 124
  Mode property, 124
  OrganizationId property, 124
  OrganizationName property, 124
  OutputParameters property, 124
  ParentContext property, 124
  PostEntityImages property, 125, 138
  PreEntityImages property, 125, 138
  PrimaryEntityName property, 125
  SecondaryEntityName property, 125
  SharedVariables property, 125
  Stage property, 125
  unit testing, 151
  UserId property, 126
iscrmuiworkflow attribute, 410
IsOnline global method, 235, 359
IsOutlookClient global method, 235
IsOutlookLaptopClient global method, 235
IsOutlookWorkstationClient global method, 235
ISV pages
  deploying, 490–491
  development overview, 478
  programming, 479–489
  testing, 490–491
ISV.Config file
  Client attribute, 350–351
  form scripting, 248–253
  interface changes, 324–325
  offline navigation, 350
ITemplate interface, 537
IWorkflowContext interface
  ActivationId property, 205
  AsyncOperationId property, 205
  CreateCrmService method, 204
  CreateMetadataService method, 204
  EntityDependencies property, 205
  EvaluateCondition method, 205
  EvaluateExpression method, 205
  InputParameters property, 205
  MessageName property, 205
  OrganizationId property, 205
  OutputParameters property, 205
  PluginTypeId property, 205
  PopulateEntitiesFrom method, 205
  PrimaryEntityId property, 206
  PrimaryEntityImage property, 206
  PrimaryEntityName property, 206
  PrimaryEntityPostImage property, 206
  PrimaryEntityPreImage property, 206
  RetrieveActivityCount method, 205
  SharedVariables property, 206
  StageName property, 206
  UserId property, 206

J

JavaScript
  debugger statement, 244–247
  loading external files, 259–261
JS extension, 13

K

keys. *See* entity relationships

L

Label controls, 387, 472
labels, hiding/showing, 565–566
languages
  CrmService support, 376–384
  deployment prerequisites, 342
  LCID support, 287
  MetadataService support, 385–394
  packs supported, 286
  resource assemblies, 395–402
  support considerations, 15–16, 375
  testing deployment, 336
laptops, 350
LCIDs (Locale IDs)
  MetadataService Web service, 322
  multilingual strings, 285, 287
  online Help files, 328
leadsourcecode attribute, 502
licensing types, 21–22
Locale IDs. *See* LCIDs (Locale IDs)
LocalParameterDependency class, 437
LocLabel class, 285, 291, 390
logging, 149–150
logical database diagrams, 6
logical names, 279
LogonManager class, 71
Lookup attribute, 494

M

manual installation, 316
manual workflows, 179
ManyToManyRelationships
  creating entities, 288
  modifying, 302
  retrieving entities, 279
ManyToOneRelationships
  relationship considerations, 283
  retrieving entities, 279
math building blocks, 214–215
MD5 encryption, 103
MD5CryptoServiceProvider class, 103, 105
MessageInvocationSource class, 123
MessageProcessingMode class, 124
MessageProcessingStage class, 125
Messages registration property, 114–115
metadata API, 5
metadata service API, 6
MetadataService Web service
  accessing, 50–51
  actions supported, 271
  additional information, 67
  application settings, 274
  caching considerations, 318
  creating entities, 288–290, 322
  CrmImpersonator class, 276
  custom controls, 501
  customizations management, 343
  customizing attributes, 292–298
  customizing relationships, 298–304
  deleting entities, 291, 320
  deploying customizations, 322
  Discovery service option, 274
  error handling, 308
  form scripting, 242
  functionality, 7, 66–67
  GetMetadataService method, 549
  locating endpoints, 273–277
  multilingual applications, 385–394
  multilingual strings, 285–287
  names used, 279
  offline functionality, 352–353, 372
  offline messages, 276
  Picklist attribute, 296–298
  plug-ins, 275
  publishing metadata, 304
  referencing, 273
  retrieving attributes and relationships,
    282–285

retrieving entities, 279–281
security considerations, 277
updating entities, 291
MetadataUtility class, 549, 552
method calls, 57–58
Microsoft Active Directory. *See* Active Directory
Microsoft ASP.NET Cassini, 348
Microsoft BizTalk Server, 186
Microsoft Dynamics CRM
additional information, 19
components supported, 20–21
editions supported, 20
Microsoft Dynamics CRM 4.0 Connector for SQL Server Reporting Services, 20
Microsoft Dynamics CRM 4.0 Enterprise Server, 20
Microsoft Dynamics CRM 4.0 Language Pack, 21
Microsoft Dynamics CRM 4.0 Professional Server, 20
Microsoft Dynamics CRM 4.0 SDK, 49–50, 85–94
Microsoft Dynamics CRM 4.0 Web Server, 23–24, 317
Microsoft Dynamics CRM 4.0 Workgroup Server, 20
Microsoft Dynamics CRM Data Migration Manager, 272, 303
Microsoft Dynamics CRM E-mail Router, 21
Microsoft Dynamics CRM for Microsoft Office Outlook, 21, 347
Microsoft Dynamics CRM for Outlook with Offline Access
deploying IFrame pages, 365
installing, 348
offline development environment, 349
offline navigation, 350
Microsoft Dynamics CRM Virtual Path Provider, 482
Microsoft Dynamics CRM Web Server, 20, 364
Microsoft Dynamics CRM Workflow, 9–10
Microsoft Office SharePoint Server, 186
Microsoft Outlook
CRM support, 16
developing customizations from, 32–33
Microsoft SQL Server
accessing data, 94–99
accessing filtered views, 97
deployment considerations, 318
description, 20
Microsoft SQL Server Management Studio, 97
Microsoft SQL Server Management Studio Express, 348
Microsoft SQL Server Reporting Services, 20
Microsoft Unit Testing Framework (MSTest), 151
Microsoft Virtual Server 2005, 24
Microsoft Visual Studio. *See* Visual Studio
Microsoft.Crm.Outlook.Sdk assembly, 51, 354–355
Microsoft.Crm.Sdk assembly, 50–51, 273
Microsoft.Crm.SdkTypeProxy assembly, 50–51, 273
mock objects, 150–151, 154–156
MockRepository class, 154
Mode registration property, 113–114
modifiedby attribute
workflow entity, 410
workflowdependency entity, 415
modifiedon attribute
workflow entity, 410
workflowdependency entity, 415
MSBuild tool, 42
MSDN Code Gallery, 348
multi-currency support. *See* currencies
multilingual support. *See* languages
Multiply by operator, 182
multi-tenants
development environment, 25, 32
testing deployment, 335

N

name attribute
crmForm, 237
workflow entity, 410, 427
named user CALs, 21
naming conventions, 279
NAnt tool, 42

navigation
  hiding links, 558–562
  offline, 350–351
.NET Reflector tool, 45
notification bar, adding, 267–268
NSIS system, 43–44
NUnit tool, 46

O

object-based security model, 80
offline solutions
  APIs, 330–332
  CrmService Web service, 352–353
  customizations, 16, 347
  deploying Web applications, 330–331
  developing IFrames for offline, 356–366
  developing offline plug-ins, 367–371
  development considerations, 349–350, 372
  messages supported, 276
  MetadataService Web service, 353
  Microsoft.Crm.Outlook.Sdk assembly, 354–355
  navigation, 350–351
  scripting, 355
OfflineOrigin class, 122
onChange form event
  conditionally enabling attributes, 557
  functionality, 9, 238
  loading external JavaScript files, 259
  tooltip information example, 262
ondemand attribute, 410
OneToManyRelationships
  modifying, 299–301
  relationship considerations, 283
  retrieving entities, 279
online Help files, 328–329
onLoad form event
  conditionally enabling attributes, 557
  functionality, 9, 238
  hiding navigation links, 561
  loading external JavaScript files, 259, 261
  scripting, 451, 574–576
  tooltip information example, 262
onreadystatechange event, 260
onSave form event
  deploying IFrame pages, 463
  functionality, 9, 238
operating systems
  deployment prerequisites, 342
  testing deployment, 334
OrderOptionValue message, 297–298
ORG_LANGUAGE_CODE global variable, 235
ORG_UNIQUE_NAME global variable, 235
Outlook. *See* Microsoft Outlook
OutlookSyncType enumeration, 354
ownerid attribute, 93, 410
owningbusinessunit attribute
  workflow entity, 410
  workflowdependency entity, 415
owninguser attribute, 415

P

parallel wait branch (workflow step), 180, 186
parametername attribute, 415
parametertype attribute, 415
parent entities, 156–160
parent pipelines, 121
parentworkflow attribute, 410
permissions, role-based, 15, 80
Picklist attribute type, 296–298, 494, 496
Platform server role, 22
Plug-in Message-Entity Table.xls, 120
PluginRegistration tool
  accessing, 113
  connecting to servers, 116
  deployment considerations, 345
  registering assemblies, 117–118, 194–195
  registering plug-in steps, 118–119
plug-ins
  building registration tool, 113
  creating, 110–111
  customization change notifier, 162–168
  debugging, 149–150
  deploying, 113–119, 128–148, 326–328, 331–332, 371
  event execution pipeline, 120–122
  exception handling, 127
  impersonation, 83–85, 126–127

implementing, 111–112
IPluginExecutionContext interface, 122–126
MetadataService Web service, 275
offline considerations, 367–372
programmatic registration, 128–138
registering, 113–119
rolling up child entity attributes, 156–160
sample, 156–168
system view hider, 161–162
testing, 371
unit testing, 151–156
plugintypeid attribute, 410, 413
PolicyActivity class, 187
PopulateActivityTable method, 490
PopulateLeadTable method, 489
PopulateOpportunity method, 489
Preview command, 247
primaryentity attribute, 410, 416
PrimaryEntityImageDependency class, 438–439
PrimaryEntityPostImageDependency class, 439
PrimaryEntityPreImageDependency class, 439
PrincipalAccess class, 91
privilege depth, 80
privileges
adding, 85–87
retrieving, 88–89
role-based security model, 15, 80
sharing, 90–92
Process.Start method, 343
programming workflow. *See* workflow programming
Publish message, 163, 304
PublishAll message, 163, 304
publishing
customizations, 323
metadata, 304
workflows, 412–413
PublishXml message, 323

Q

Quick create form type, 254

R

Rank registration property, 114, 116
Read-only form type, 253
records
assigning, 93–94
sharing privileges, 90–92
redeployment, 36–40
referenced assemblies
adding, 495–497
deploying, 147–148
regardingojbectid attribute, 81–82, 427
registering
assemblies, 117–118, 193–195
plug-ins, 113–119
registry keys
creating CrmService instance, 353
deploying plug-ins, 371–372
ServerUrl value, 549
allow lists, 331–332
relatedattributename attribute, 415
RelatedEntityImageDependency class, 441
relatedentityname attribute, 415
relationships. *See* entity relationships
remote debugging, 149
Request class, 63–64
resource files, 395–402
Response class, 63–64
RetrieveAllEntitiesRequest
functionality, 281
MetadataItems property, 281, 305
offline support, 276
RetrieveAsIfPublished property, 281–282
Timestamp property, 304
RetrieveAllEntitiesResponse, 276, 304
RetrieveAttributeRequest, 276, 282
RetrieveAttributeResponse, 276
RetrieveAvailableLanguagesRequest, 287, 336, 376
RetrieveDeprovisionedLanguages message, 376
RetrieveEntityRequest
EntityItems property, 279–280
functionality, 279–280

LogicalName property, 279–280
offline support, 276
RetrieveAsIfPublished property, 279–280, 282
RetrieveEntityResponse, 276
RetrieveInstalledLanguagePacks message, 376
RetrieveLocLabels message, 376
RetrieveProvisionedLanguages message, 376
RetrieveRelationshipRequest, 276, 283
RetrieveRelationshipResponse, 276
RetrieveRequest class, 459
RetrieveTimestampRequest, 276, 304
RetrieveTimestampResponse, 276
RetrieveUserPrivilegesRequest class, 89
RhinoMocks tool, 46
role-based security model, 15, 80
RolePrivilege class, 89
Rules attribute, 410

S

schema names, 279
Scope attribute, 410
Scribe third-party tools, 35
SdkAssociationDependency class, 436
sdkmessageid attribute, 415, 429
security
access rights and, 81–82
field-level, 563–579
impersonation, 82–85
MetadataService Web service, 277
XSS considerations, 257
security principals, 81
security roles
assigning programmatically, 87–88
creating programmatically, 85–87
deploying, 323
overview, 15, 80
retrieving, 88–89
security tokens, 57
SecurityPrincipal class, 91, 94
SecurityPrincipalType enumeration, 91, 94
send e-mail (workflow step), 180, 186
SendEmailActivity class, 186
SendEmailFromTemplateActivity class, 186
server controls
adding keys to web.config, 496
adding references to assemblies, 495
adding Web site projects, 495
benefits of developing, 493
complex attribute types, 494
creating project, 494–495
CrmBooleanControl, 503–506
CrmDateTimeControl, 507–515
CrmEntityPicklistControl, 516–522
CrmGridViewControl, 523–548
CrmPicklistControl, 496–503
server environments
hybrid environment, 25, 30
isolated development, 25–28
multi-tenant functionality, 25, 32
sharing common organization, 25, 28–30
testing deployment, 334
server licenses, 21
server roles, 22
server-side events, 9–10
SERVER_URL global variable, 235
Set to operator, 182
SetFieldSecurity function, 567
SetLocLabels message, 376
SetPageSecurity function, 566
SetStateActivity class, 186
SetStateWorkflowRequest, 412
SetTabSecurity function, 567
Share privilege, 90–92
shared hosting, 335
site maps
Client attribute, 350–351
interface changes, 324–325
offline navigation, 350
SOAP headers, 57
SOAP messages
error handling, 308
form scripting, 241–244
tooltip information example, 262
SoapException, 308
software licenses, 21–22
source control, 40–42
SourceForge.NET, 42–43

SQL queries, 202
stage (workflow step), 180, 186
Stage registration property, 113–114
StageActivity class, 186
start child workflow (workflow step), 181, 186
Statecode attribute, 410
Status attribute type, 494, 496
Statuscode attribute, 410
stop workflow (workflow step), 181, 186
StopWorkflowActivity class, 186
String attribute type, 543
style attribute, 564
Subprocess attribute, 410
Subversion (SVN), 41–42
system view hider, 161–162

T

TableCell class, 489–490
TableRow class, 489–490
tables. *See* entities
tabs, hiding, 564–566
templates
  creating for CrmGridViewControl, 537–543
  deploying, 323
TestClass attribute, 152
testing. *See also* unit testing
  accessibility deployment, 336–340
  authentication deployment, 335
  CrmBooleanControl, 506
  CrmDateTimeControl, 515
  CrmEntityPicklistControl, 521–522
  CrmGridViewControl, 545–548
  CrmPicklistControl, 502–503
  custom activities, 197–201
  database server deployment, 335
  dialog boxes, 476–478
  form scripting, 244–247
  IFrame pages, 361–365
  in offline databases, 348
  in offline development environment, 349–350
  ISV page, 490–491
  multilingual deployment, 336
  multilingual IFrames, 391–394, 402
  multi-tenant deployment, 335
  operating system deployment, 334
  plug-ins, 371
  server topologies, 333
  Web browser deployment, 334
  workflows, 176–177
testing environment, 33, 247
TestMethod attribute, 152
tooltips, displaying information, 262
TortoiseSVN browser, 41–42
transactioncurrencyid attribute, 403
Type attribute, 411, 415
TypeName attribute, 237

U

uidata attribute, 411, 413
Undefined form type, 253
unit testing
  mock objects, 150–151, 154–156
  plug-ins, 151–156
  sample tests, 151–156
  test frameworks, 151
UnregisterSolution message, 320
Update form type, 253
update record (workflow step), 180, 186
UpdateActivity class, 186
UpdateOptionValue message, 297
UpdateRelationshipRequest, 299
upgrades, supported customizations, 14
USER_LANGUAGE_CODE global variable, 235
UserLocLabel helper property, 286
UTF8Encoding class, 103
utility classes, 549–552

V

validation, form scripting, 257–258
Virtual PCs, 350
virtualization, 24
Visual Studio
  accessing APIs, 50–54
  adding Web site projects, 495
  creating projects, 451–452

overview, 41
PluginRegistration tool, 113
Web Deployment Projects add-in, 315
workflow support, 186

W

wait condition (workflow step), 180, 186
WaitLoopActivity class, 186
Web applications
impersonation, 82–83
offline deployment, 330–331
Web browsers
caching external files, 259
testing deployment, 334
Web pages
deploying custom, 325–326
embedding, 9
offline considerations, 356–366
Web.config file, 326
web.config file, 496
WebServiceApiOrigin class, 122
WhoAmIRequest message, 489
Windows High-Contrast mode, 336–340
Windows Live ID, 71, 273
Windows Online Authentication, 56
Windows Workflow Foundation
activities, 186–187
dynamic value binding, 187
functionality, 185–186
workflow assemblies. *See* assemblies
workflow designer, 170–185, 213–214, 416–417
workflow entity, 409–411
workflow programming. *See also* declarative workflows
calculating related aggregate, 221–231
custom workflow management tool, 407–408
deploying custom assemblies, 193–196
entity images, 206–213
implementing custom activities, 188–193
investigating activity errors, 201–203
math building blocks, 214–215
retrieving dependencies, 418–419
retrieving most available user, 216–221
testing custom activities, 197–201
Windows Workflow Foundation, 185–187
workflow attributes, 203
workflow context, 204–206
workflow designer, 170–185, 416–417
workflow designer limitations, 213–214
workflows as entities, 409–419
workflow steps, 180–181
Workflow Web interface, 10–11
WorkflowConfiguration class
Dependencies property, 429, 433
ImportWorkflow method, 432
Load method, 430
Save method, 446
ToEntity method, 432–433
XmlArrayItem, 430
XmlAttribute, 430
WorkflowDependency class, 446
workflowdependency entity, 414–415
workflowdependencyid attribute, 415
WorkflowDependencyType class
AttributeDefinition field, 414, 417
CustomEntityDefinition field, 414
LocalParameter field, 414
PrimaryEntityImage field, 414
PrimaryEntityPostImage field, 414
PrimaryEntityPreImage field, 414
RelatedEntityImage field, 414
SdkAssociation field, 414, 417
WorkflowConfiguration class and, 433
workflowid attribute
workflow entity, 411
workflowdependency entity, 415, 418
WorkflowLogic class
ExportWorkflow method, 444–446
ImportWorkflow method, 429
RetrieveWorkflowDependencies method, 418
WorkflowStatusDraft constant, 412
WorkflowStatusPublished constant, 412
workflows. *See also* declarative workflows
as entities, 409–419
automatic, 178–179

child, 179
creating native, 170–175
CRM attributes, 203
CrmService Web service, 411–412
custom activities, 188–193
default values, 185
dynamic values, 181–185
Dynamic Values box, 184
Look for options, 183–184
manual, 179
offline considerations, 372
operator options, 182
publishing, 412–413
steps in process, 180–181
testing, 176–177
WorkflowState class, 412
workstations, 349

X

XAML (Extensible Application Markup Language)
additional information, 420
CRM support, 413, 420
exporting workflows programmatically, 42–44
native workflow, 442–444
syntax, 419–420
XmlDocument class, 490
XmlHttp class, 241, 261
XmlNodeList class, 490
XmlSerializer class, 430, 446
XOML file extension, 421
XPath class, 490
XXS (cross-site scripting), 255–257

About the Authors

Jim Steger

Jim Steger is cofounder and principal of Sonoma Partners, a Chicago-based consulting firm that specializes in Microsoft Dynamics CRM implementations. Sonoma Partners won the Global Microsoft CRM Partner of the Year award in both 2003 and 2005 and was a finalist in 2008. He is a Microsoft Certified Professional and has architected multiple award-winning Microsoft Dynamics CRM deployments, including complex enterprise integration projects. He has been developing solutions for Microsoft Dynamics CRM since the version 1.0 beta.

Before starting Sonoma Partners, Jim designed and led various global software development projects at Motorola and ACCO Office Products. Jim earned his bachelor's degree in engineering from Northwestern University. He currently lives in Naperville, Illinois, with his wife and two children.

Mike Snyder

Mike Snyder is cofounder and principal of Sonoma Partners. Recognized as one of the industry's leading Microsoft Dynamics CRM experts, Mike is a member of the Microsoft Dynamics Partner Advisory Council, and he writes a popular blog about Microsoft Dynamics CRM.

Before starting Sonoma Partners, Mike led multiple product development teams at Motorola and Fortune Brands. Mike graduated with honors from Northwestern's Kellogg Graduate School of Management with a Master of Business Administration degree, majoring in marketing and entrepreneurship. He has a bachelor's degree in engineering from the University of Notre Dame. Mike lives in Naperville, Illinois, with his wife and three children. He enjoys ice hockey and playing with his kids in his free time.

Brad Bosak

Brad Bosak is a lead architect at Sonoma Partners, and he has been developing client solutions on Microsoft Dynamics CRM since version 1.2. Brad works on the most complex CRM projects at Sonoma Partners, using his deep product experience to meet various types of customer requirements.

Before starting at Sonoma Partners, Brad worked for several years as a .NET application developer and consultant. Brad earned his bachelor's degree in computer technology from Purdue University. He currently lives in Chicago, Illinois. In his free time, he enjoys taekwondo and playing guitar.

Corey O'Brien

Corey O'Brien is a lead architect at Sonoma Partners and certified in Microsoft Dynamics CRM. Corey has designed numerous Microsoft Dynamics CRM solutions for clients in a wide range of industries.

Corey has more than 10 years of experience designing and developing software solutions using Microsoft technologies. Corey holds patents for software concepts in both the Industrial Automation and Education industries. Corey earned his bachelor's degree in computer science from Hope College in Holland, Michigan. He currently lives in Hanover Park, Illinois with his wife and child. He enjoys volleyball and basking in the warm glow of his various electronic devices.

Philip Richardson

Philip Richardson has worked at Microsoft since 2000 and currently works as a Senior Program Manager in the Cloud Services team. Prior to his current role he was a lead on the Dynamics CRM team for the 4.0 release and for the first milestone of the next release (codename: CRM5). He is passionate about sales and marketing business systems and the positive impact they can have on a organization and its end customers

Philip is a native of Sydney, Australia, and currently resides (with his wife, son, and dog) near Microsoft's global headquarters in Redmond, Washington. In 2007 his blog (*http://www.philiprichardson.org/blog*) took the #4 position on *InsideCRM*'s Top 20 CRM bloggers.

About Sonoma Partners

This book's authors, Jim Steger, Mike Snyder, Brad Bosak, and Corey O'Brien, are executives at the Chicago-based consulting firm Sonoma Partners. Sonoma Partners is a Microsoft Gold Certified Partner that sells, customizes, and implements Microsoft Dynamics CRM for enterprise and midsize companies throughout the United States. Sonoma Partners has worked exclusively with Microsoft Dynamics CRM since the version 1.0 pre-release beta software. Founded in 2001, Sonoma Partners possesses extensive experience in several industries, including financial services, professional services, health care, and real estate.

Sonoma Partners is unique for the following reasons:

- We are focused 100 percent on the Microsoft Dynamics CRM software product. We do not spread our resources over any other products or services.
- We have successfully implemented more than 150 Microsoft Dynamics CRM deployments.
- Microsoft awarded Sonoma Partners as the Global Microsoft Dynamics CRM Partner of the Year in 2005 and 2003. Microsoft recognized Sonoma Partners as one of three finalists for the 2008 Microsoft Dynamics CRM Partner of the Year award.
- More than half of our staff includes application and database developers so that we can perform very complex Microsoft Dynamics CRM customizations and integrations.
- We were named one of 101 Best and Brightest Companies to Work for in Chicago in 2007 and 2008.
- We are a member of Microsoft Dynamics Partner Advisory Council.

In addition to the books we've written for Microsoft Press, we share our Microsoft Dynamics CRM product knowledge through our e-mail newsletter and online blog. If you're interested in receiving this information, you can find out more on our Web site at *http://www.sonomapartners.com*.

Even though our headquarters is in Chicago, Illinois, we work with customers throughout the United States. If you're interested in discussing your Microsoft Dynamics CRM system with us, please don't hesitate to contact us! In addition to working with customers who want to deploy Microsoft Dynamics CRM for themselves, we also act as a technology provider for independent software vendors (ISVs) looking to develop solutions for the Microsoft Dynamics CRM platform.

Sometimes people ask us where we got our name. The name *Sonoma Partners* was inspired by Sonoma County in the wine-producing region of northern California. The wineries in Sonoma County are smaller than their more well-known competitors in Napa Valley, but they have a reputation for producing some of the highest quality wines in the world. We think that their smaller size allows the Sonoma winemakers to be more intimately involved with creating the wine. By using this hands-on approach, the Sonoma County wineries can deliver a superior product to their customers—and that’s what we strive to do as well.