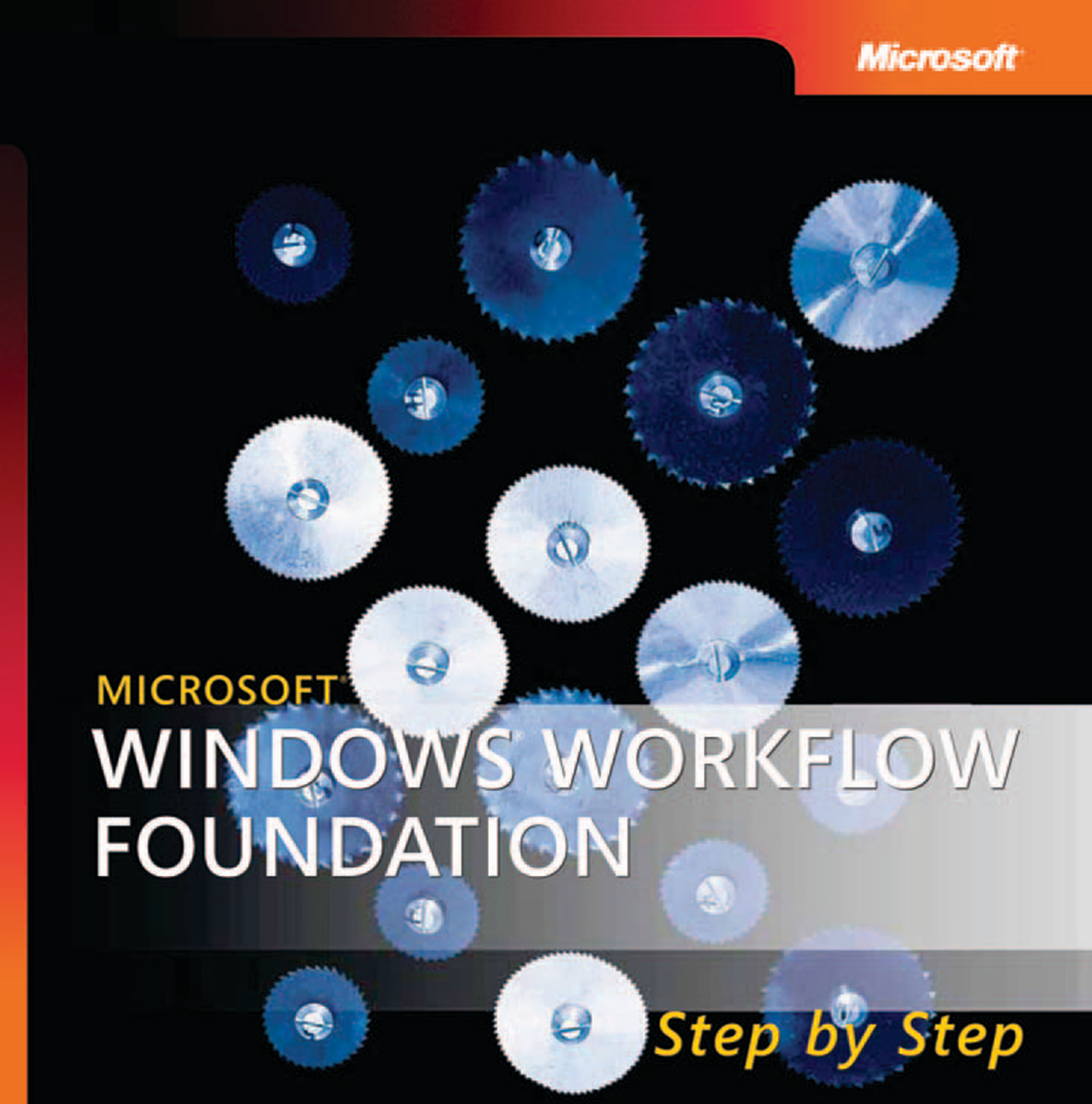
Microsoft



MICROSOFT®

WINDOWS WORKFLOW FOUNDATION

Step by Step

Kenn Scribner

Wintellect®
Know how.

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

Library of Congress Control Number: 2006940677

Printed and bound in the United States of America.

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

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

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

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Acquisitions Editor: Ben Ryan
Project Editor: Lynn Finnel
Copy Editor: Roger LeBlanc
Technical Reviewer: Kurt Meyer
Peer Reviewer: Scott Seely
Editorial and Production Services: Waypoint Press

Body Part No. X12-64035

To my wonderful family, Judi, Aaron, and Katie,
without whose love and support life has little meaning.
Thank you all for always being there.

Table of Contents

Foreword

To me, workflow engines such as Microsoft BizTalk always seemed like a really expensive thing that I didn't need for many projects. I always thought, "How hard can it be to actually code the logic you draw on the screen?" As a result, I spent a lot of time not learning what a workflow engine could do for me and, instead, lived with "just code." Still, I was intrigued and would bug my friends about what they saw as the value of BizTalk. They would usually get me excited enough to try installing the product and learning how it worked. Every time I tried to make the plunge and learn how to use BizTalk, I backed away because it seemed too complex for what I perceived as a flowchart execution engine.

Sometime around 2003, while working at Microsoft on what was to become Windows Communication Foundation, I heard about a general-purpose workflow engine being built within Microsoft. Rumor had it that the workflow engine might even be integrated into the operating system! The reason: Prior to then, groups within Microsoft had created their own workflow engines for their own problem domains. Most of the engines were written in C++ and exhibited limited flexibility outside of their domain area. BizTalk, which had a general-purpose engine, was not then designed to have its engine separated from the BizTalk product. What Microsoft discovered was there was a real need for a single, general-purpose workflow engine so that internal Microsoft teams could stop reinventing workflow. This realization (which occurred prior to me hearing anything) helped create what would eventually become the Windows Workflow Foundation team.

As a part of .NET Frameworks 3.0, Windows Workflow Foundation is a freely distributable .NET component. On Windows Vista and later, it ships as a part of the operating system. What does this mean for developers? It means that they can learn how to use a workflow engine and distribute their applications while being able to rely on a workflow engine just being present on the client machine. Bigger applications will still need tools such as BizTalk Server to manage workflows. But, for smaller applications that could use some of the benefits of workflow (such as state machines and the ability to suspend and resume a workflow), Windows Workflow Foundation is a godsend.

The book you now hold shows all the little things that Windows Workflow Foundation can do. Because it is the introductory book to Windows Workflow Foundation, it takes a tour of the feature set and lets you know what is available. For me, this book has served as a great introduction to the technology. I have discovered that most projects would benefit from a healthy dose of workflow somewhere. The samples in this book proved as much to me.

Kenn, congratulations on finishing up a great book. Thank you for making me a part of the process. I believe many more .NET developers will finally understand what workflow can do for them thanks to your introduction to the topic.

Scott Seely

Acknowledgments

If you ever have the opportunity to write a book, aside from the tremendous amout of work it takes, you'll find it's a lonely business. You'll spend hour after hour after hour, typing, writing, debugging... You can easily be fooled into believing the world consists of only two entities: yourself and your computer.

But the simple truth is you're not an island, and literally dozens of people are behind you, working long hours, night and day, to help you. Some, you know. Others, you may not know, but they're helping just the same. Everyone has a single goal in mind, and that's to help you craft the best book you can possibly write. If you're one of the many people who helped me with this effort and I didn't mention you by name here, rest assured you have my undying thanks and gratitude. If you've remained nameless to me, it's only because we haven't had the good fortune to meet. Forgive me any oversights...your help and support were invaluable to me, and what's more, I have no misunderstandings regarding how hard you worked on my behalf. Thanks to all of you!

Happily, I do happen to know a few people who were instrumental in this process. First and foremost, I'd like to humbly and most appreciatively thank my wife, Judi, who encouraged me to take the plunge yet another time. She knows how much work it is to write one of these and sacrified our time together on many evenings so that I could hurridly work to meet deadlines. I'd like to thank my children Aaron and Katie, who sacrified many backyard baseball and football games with Dad and put up with my evil alter-ego, Mr. Stressed, for the months it took me to complete this manuscript. Their understanding and encouragement kept me going as well.

To my editor, Lynn, words can't express how thankful I am, certainly in part because of your efforts with this book, but also for our friendship. Lynn and I have worked together on many books (me being a technical editor), and they've all been a treat because of your care and guidance. Sometime dinner's on me, unless I find you atop Mt. Ranier (she's an avid climber). Then you're on your own, unless you're good with freeze dried!

Thank you Roger for taking my raw, grammatically incoherent text and turning it into something I can claim I wrote. We both know better, but let that be our secret. I can't tell a dangling participle from a misplaced modifier, but I boldy went forth and wrote them anyway. Thanks for kindly fixing things!

And thank you, Kurt, for your dogged determination to root out every bug I cleverly hid in the text and code. I know well the work you did for me. Sometimes I could almost hear you say "Technical editor turned author, correct thyself!" when you ran into the more obvious of my technical errors. I have a new appreciation for the author's side of the technical editing job, and any errors that remain are mine alone and were probably stealthily injected under the cover of darkness, hidden from your keen eye. Either that or...Roger did it. Yeah...Roger did it! That's the ticket.

Introduction

Before diving into programming with Microsoft Windows Workflow Foundation, or WF, it's important to understand what workflow is and why you'd want to invest the effort to learn how to use it. After all, learning new technology means that you have to, well, take the time to learn the new technology. The learning curve can be painful. There are new tools, new ways to think about your applications, and so forth. Given your time investment to learn WF, what sort of return on your investment can you expect? Is it worth learning, or is it just a passing fad?

Workflow, at least as I believe WF most closely defines it, is simply a term that is applied to software that executes in a more rigorous environment. What software? Essentially, the same software you have already been writing. Have you ever written code to take information from a database, process the information, and then write the processed data to another database or data sink? Or how about taking files and moving them from place to place once a person or process approved or otherwise manipulated them? Actually, the examples I could come up with are nearly infinite, limited only by my ability to invent them. *Anything* you write could be considered workflow at some level.

The environment WF provides you with is of great value, if only because it allows for easy multithreaded processing. Your services or user interfaces don't need to worry about creating worker threads and monitoring their use and ultimate demise.

But WF offers other tantalizing features, such as the ability to stop an executing workflow process and shuttle it to a database for safekeeping while a long-running approval or other external process completes. It can automatically record tracking information to a database. It facilitates the development of neatly compartmentalized and readily reusable code. It has nice transactional support. And it's already a part of Microsoft Windows Vista, so you don't have to install it on the next generation of Windows. Even if you're using Microsoft Windows XP or Microsoft Windows Server 2003, it's part of the latest version of .NET, which is something you'd probably be installing anyway at some point.

Oh, and did I mention it's free? The cost to you is the cost of the download and your time investment to learn to use it.

Speaking personally, the investment in learning and using WF is proving to be worth the effort. I write software, and I'd guess that you do as well or you wouldn't be reading this. I'm finding that customers and clients want this technology. From that standpoint, it's not a hard decision, really. I chose to take the time to learn this technology, and I'm applying it in my software solutions today.

So how best to learn WF? I like to write code and experiment. If you do also, you should find this book useful, because with it you'll be able to write code and experiment as well. This book isn't designed to go into great depth regarding any particular topic. Rather, it's designed to get

You also need to have the following additional software installed on your computer. This software is available on the companion CD supplied with this book. Installation and configuration instructions are provided later in the Introduction—as well as in Chapter 1 along with additional information and locations from which you can download the software from the Internet. The order in which you install the supporting software matters. It should be installed in the order listed here.

1. Microsoft .NET Framework 3.0.

> **Note** If you are using Windows Vista, the .NET Framework 3.0 is automatically installed as part of the operating system. You do not need to install it again.

2. Visual Studio 2005 Extensions for .NET Framework 3.0.
3. Visual Studio 2005 Extensions for Windows Workflow Foundation.
4. SQL Server Management Studio Express Edition.

> **Note** If you are using the full retail version of SQL Server 2005, SQL Server Management Studio is installed for you.

You also need the Microsoft Windows Software Development Kit for Windows Vista and .NET Framework 3.0 Runtime Components. You can download this software from the Microsoft Download Center site at *www.microsoft.com/downloads/details.aspx?FamilyId=C2B1E300-F358-4523-B479-F53D234CDCCF&displaylang=en*. If you'd rather not type in such a lengthy Internet address, I've placed the locations of the software for download in the first chapter's page in the accompanying code manual on the CD.

3. On the Welcome To The Visual Studio 2005 Extensions For .NET Framework 3.0 (WCF WPF) November 2006 CTP Setup Wizard page, click Next.
4. On the License Agreement page, read the license agreement. If you agree with the license terms, click I Accept and then click Next.
5. On the Confirm Installation page, click Next.
6. When the Installation Complete page appears, click Close.
7. Close the Internet Explorer window displaying the release notes.

Installing the Visual Studio 2005 Extensions for Windows Workflow Foundation

The exercises and samples in this book have been tested against the November 2006 RTM version of the Visual Studio 2005 Extensions for Windows Workflow Foundation. Follow these instructions to install this software:

1. Using Windows Explorer, move to the \Software folder on the companion CD.
2. Double-click the file Visual Studio 2005 Extensions for Windows Workflow Foundation (EN).exe. If the Open File – Security Warning dialog appears, click Run.
3. On the Visual Studio 2005 Extensions For Windows Workflow Foundation screen, click Visual Studio 2005 Extensions For Windows Workflow Foundation.
4. On the License Agreement page, read the license agreement. If you agree with the license terms, click I Accept and then click Next.
5. On the Component Installation page, click Next.
6. On the Summary page, click Install.
7. When the Installation Complete page appears, click Finish.

Installing the SQL Server Management Studio Express Edition

Some of the applications in this book require the use of SQL Server or SQL Server Express. If you're using SQL Server Express, you can install the very useful SQL Server Management Studio Express Edition application to make administering your SQL Server Express databases much easier. This application will be necessary later in the book for running database creation scripts that ship with .NET 3.0 as well as with this book. Note the installation package is provided on the book's CD.

1. Using Windows Explorer, move to the \Software folder on the companion CD.
2. Double-click the file SQLServer2005_SSMSEE.msi. If the Open File – Security Warning dialog appears, click Run.

Solution Folder	Description
Chapter6	
WorkflowPersister	Although many workflows could conceivably load, execute, and finish in a relatively short period of time, other workflows might take longer to complete. In those cases, you can, if you want, shuttle your executing workflow out of memory and into a SQL Server database for safekeeping until the conditions that merit its return are met. This application demonstrates this WF capability.
WorkflowIdler	In this solution, you learn how *Delay* activities can be configured to automatically persist your workflow to a SQL Server database, allowing you to remove long-running workflows from your computer's memory while the workflow waits.
Chapter7	
Sequencer	This application demonstrates a simple sequential workflow.
ErrorThrower	What do you do when your workflow encounters a runtime condition it can't handle? Why, use the *Throw* activity, of course! In this application, you see how this is done.
ErrorHandler	This solution demonstrates how workflow-based exceptions thrown using the *Throw* activity are handled by your workflow.
ErrorSuspender	Should you need to do so, you can suspend the execution of your workflow using the *Suspend* activity. This application demonstrates the *Suspend* activity.
ErrorTerminator	As with workflow suspension, you have the capability to completely terminate your workflow. This application demonstrates this capability.
Chapter8	
MVDataChecker	Workflows ultimately work with some form of data. If your host application needs to retrieve data directly from your workflow, this application demonstrates the technique.
WorkflowInvoker	If you've ever wondered whether an executing workflow can invoke another workflow, this application shows you that indeed you can do so.
Chapter9	
IfElse Questioner	This chapter's focus is on workflow logic flow. The application for this chapter is written using three different workflows that accomplish the same task. In this case, the *IfElse* activity directs program flow.
While Questioner	This version of the application uses a *While* activity to direct program flow.
Replicator Questioner	Finally, this application uses the *Replicator* activity to direct program execution flow.

Uninstalling the Code Samples

Follow these steps to remove the code samples from your computer:

1. In Control Panel, open Add Or Remove Programs.
2. From the list of Currently Installed Programs, select Microsoft Windows Workflow Foundation Step By Step.
3. Click Remove.
4. Follow the instructions displayed to remove the code samples.

Online Companion Content

The online companion content page has content and links related to this book, including a link to the Microsoft Press Technology Updates Web page. The online companion content page for this book can be found at

www.microsoft.com/mspress/companion/0-7356-2335-4/

Note Code samples for this book are on the companion CD.

Support for This Book

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

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

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

Part I
Introducing Windows Workflow Foundation (WF)

In this part:

Chapter 1

Introducing Microsoft Windows Workflow Foundation

After completing this chapter, you will be able to:

- Understand workflow concepts and principles
- Be able to compare Windows Workflow Foundation (WF) to BizTalk and Windows Communication Foundation (WCF)
- Have begun programming with WF
- Know how to use Visual Studio workflow support

Workflow. It sounds like some new technological buzzword, and in a sense, it is. But the concept of workflow stems from the need to process data quickly and accurately. If I were to conjure a definition for the term, I would define *workflow* as the basic tasks, procedures, people and organizations, system informational input and output, policies and rules, and tools needed for each step in a business process. That's a very broad definition, but it often takes all these components to make a business process work. Or not work. Our goal, of course, is to automate the interworkings of these business process elements with software to improve the odds of a given process's success.

Workflow Concepts and Principles

The origins of workflow processing come from document processing, where documents need to go from place to place for approval or review. But the notion of executing a specific set of tasks, coupled with decision making (such as approved or not approved), is something we can generalize. In fact, if you've ever written a piece of software that processed information, made decisions based on system inputs, and took into account the rules and practices of the system in which the software executed, I'd argue that you've written workflow software.

But I'd also hazard a guess that this task wasn't easy for you because for most of us it seldom is. The task involves writing a lot of the support and infrastructure ourselves. And because of this, budget and schedule concerns likely prevented us from writing the performance-minded application we wanted to write. Unfortunately, this happens more often than we'd like. This is simply how business is—time truly *is* money, and the faster we implement our application the more quickly it will be used and benefit the organization.

Service-based software, however, is integrated using contracts and policy. It's expected that services will morph and change, and the goal is to have dependent processes recognize changes and adjust accordingly. *Contracts* identify *what* services are available. *Policy* describes *how* the services are to be used. A contract might tell me there is a service available that I can use to order stock for my shelves. The policy associated with that service would tell me that I have to authenticate before I use the service and that the information flowing between the remote server and my local system must be encrypted and digitally signed. When contracts or policies change, ideally service consumers automatically adjust and change as well.

Note Actually, when contracts or policies change, things don't magically fix themselves. In reality, service providers should communicate how many previous versions they'll support and service consumers should update their code to the most up-to-date version as soon as possible.

Communication functionality within WF isn't this grand. If you require strong SOA support, by all means use WCF! I do. But for more simplistic "get this data from that Web Service," or "expose this process as a Web Service" needs, WF is able to help, and toward the end of the book we'll look at how WF supports these needs.

Note This book is designed to get you up and running with WF but not necessarily to explain in great detail *why* you're doing something. We'll have to save that level of detail for another book.

Beginning Programming with WF

With this brief introduction in mind, let's start working with WF. Unless you are using Windows Vista (or later), you will need to download and install the runtime environment for WF. To write software that uses WF, you'll also need to download several additional components to Microsoft Visual Studio 2005. The Introduction contains detailed installation instructions, and you'll find all of the necessary software on the book's CD with the exception of the Windows SDK. It was excluded due to size constraints. The workflow support software on the CD was current at the time this was written, but it's best to check to see if updates are available. As long links you see are easily mistyped, I included them in the section for Chapter 1 in the CD-based manual for your convenience. They were also valid at the time this was written and are also subject to change.

Downloading and installing Windows Workflow Foundation

1. Download the following files from the links provided:
   - .NET Framework 3.0 runtime components (file: dotnetfx3setup.exe):

     http://www.microsoft.com/downloads/details.aspx?familyid=10CC340B-F857-4A14-83F5-25634C3BF043&displaylang=en

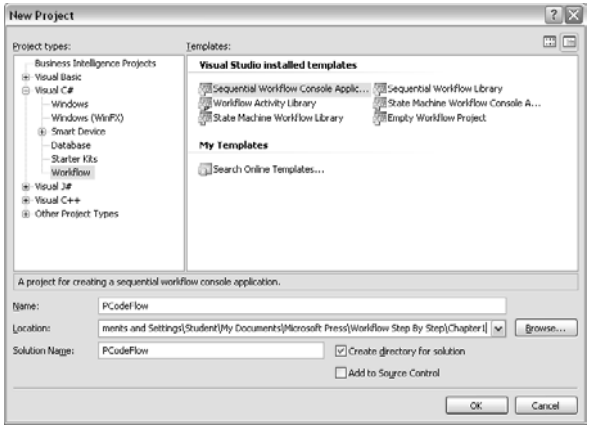
> **Note** If this is the first time you've run Visual Studio 2005, you might see a dialog box that asks you to select certain default development system preferences, such as Microsoft Visual C# as your preferred development language. Visual Studio tailors its user interface to match your preferences. Once configured, the Visual Studio integrated development environment (IDE) will appear.

3. On the File menu, select New and then Project. The New Project dialog box appears. This dialog box contains many project templates, including Windows Forms applications, console applications, class libraries, and so forth. It might even contain templates for different languages if you installed them.

> **Note** The templates installed on your system result from the combination of the version of Visual Studio you are using and any additional software you might have installed, such as for WF. You can even define your own project templates, but that is a topic for another book.

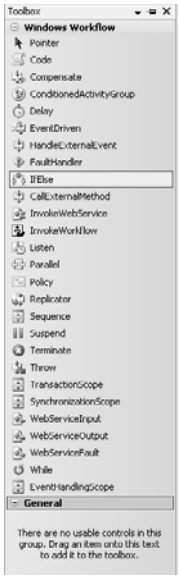
4. In the Project Types pane, click Visual C# to expand the tree node to show the project types available for the C# language.

5. Under the Visual C# node, click the Workflow node to display the workflow-based project templates.



Building a workflow

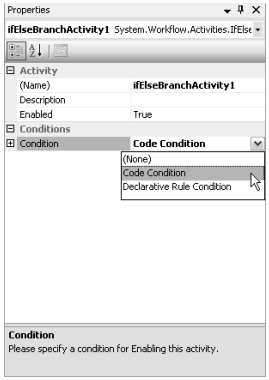
1. With the workflow visual designer showing, move the mouse cursor to the Visual Studio Toolbox and allow it to expand. If the workflow visual designer is not showing, select Workflow1.cs in the Solution Explorer pane and click the Solution Explorer's View Designer toolbar button.



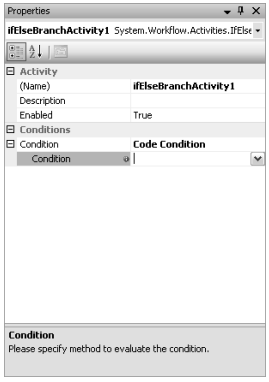
2. Drag the *IfElse* activity component onto the workflow designer's surface. *Activities* are the building blocks of workflow applications, and the Toolbox contains the activities that are currently appropriate for your workflow. As the mouse approaches the Drop Activities To Create A Sequential Workflow area, the designer view changes slightly to indicate that you can drop the *IfElse* activity icon onto the designer surface.

Note Visual Studio 2005 will place only the activities appropriate for your workflow application into the Toolbox. Other activities in fact are available, but Visual Studio won't present them in the ToolBox because they're not designed for use with the workflow type currently being edited. (We'll look at the different workflow types in Chapter 4, "Introduction to Activities and Workflow Types.")

2. We need to add a condition, which is to say a test that forces the workflow to take the actions in the left branch (condition evaluates to *true*) or the right branch (condition evaluates to *false*). To do this, click the *Condition* property to activate the *Condition* type property drop-down list. From that list, you can select a code condition type, a rule condition type, or none. Click the Code Condition option.

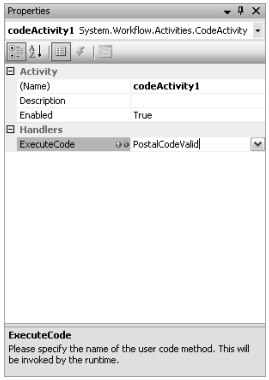


3. The *Condition* type property user interface will now change to include a plus sign (+) that, when clicked, drops a child property, also named *Condition*. This child *Condition* property is where we'll begin adding code. Click the child *Condition* property to again activate the property drop-down list.



4. The *Condition* property is requesting a name for the internal event we want to add. This event will fire when the condition requires evaluation. For this example, type **EvaluatePostalCode** into the Condition property field.

4. Click the *ExecuteCode* property, and activate it by clicking in the edit portion of its drop-down list as you did with the *Condition* property. The workflow runtime will execute the conditional code you specify here in response to an event, and we'll again have the opportunity to name the event. Type **PostalCodeValid** in the *ExecuteCode* property.



To review, at this point Visual Studio 2005 has inserted events that, when fired, will execute code we'll provide in a moment. The first event handler, *EvaluatePostalCode*, executes when the workflow runtime needs to evaluate the test condition. The second event handler, *PostalCodeValid*, executes when the left branch is taken (that is, the test condition evaluated to true).

We could, at this point, add code to the right-hand branch, which would be executed if the test condition evaluated to *false* (that is, the postal code was not valid). To do so, retrace the last set of steps but add the *Code* activity to the right branch and name the event **PostalCodeInvalid**. This adds a third event handler to our workflow, *PostalCodeInvalid*.

If you're familiar with how events are handled in .NET, the next set of steps should be familiar. The event handlers we added will be called by the workflow runtime at the appropriate locations in our application. We'll need to add code to the event handlers Visual Studio added for us to intercept the events and take action. Let's see how that's done.

Adding event handler code to our workflow

1. In the Visual Studio Solution Explorer pane, click Workflow1.cs to select it in the Solution Explorer tree control. Then click Solution Explorer's View Code toolbar button to open the Workflow1.cs C# file for editing.

```
// Modified to accept the input parameter "wfArgs"
WorkflowInstance instance =
    workflowRuntime.CreateWorkflow(typeof(PCodeFlow.Workflow1), wfArgs);
```

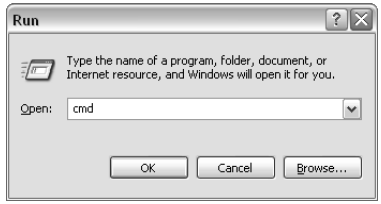
> **Note** The startup code Visual Studio 2005 inserted into the *Main* method would have been perfectly adequate had we not wanted to pass the postal code found on the command line to the workflow. We'll look more closely at passing startup arguments to our workflows in the next chapter.

4. Compile the application by selecting Build PCodeFlow from the Visual Studio Build menu.

That's it! The code we just added allows us to process a postal code that's provided on the command line. Let's try it.

Executing your workflow application

1. In Microsoft Windows, click the Start button, move the cursor to Run, and click to open the Run dialog box.
2. In the Run dialog box Open field, type **cmd** and click OK.



3. This opens a Windows Command Shell. At the command prompt, type **cd \Workflow\Chapter1\PCodeFlow\PCodeFlow\bin\Debug** and press the Enter key. This changes the current directory in the command window to the directory containing our workflow application. If you compiled the application using the Release mode, then be sure to change to the Release directory rather than the Debug directory.

> **Note** Don't forget our convention regarding directory names: "\Workflow" indicates the root directory you selected when you created the PCodeFlow project. You'll need to replace "\Workflow" with the full path name you chose.

4. Type the following command at the prompt, followed by the Enter key: **pcodeflow 12345**. The application should take a moment (to spin up the .NET Framework as well as the workflow runtime) and then spit out "The postal code 12345 was valid."
5. Type the following command at the prompt, followed by the Enter key: **pcodeflow 1234x**. The application should respond with "The postal code 12345 was *invalid*."

Chapter 2

The Workflow Runtime

After completing this chapter, you will be able to:

- Be able to host the workflow runtime in your applications
- Understand the basic capabilities of the *WorkflowRuntime* object
- Know how to start and stop the workflow runtime
- Be able to connect to the various workflow runtime events

When you execute tasks in the Workflow Foundation (WF) environment, *something* needs to oversee that execution and keep things straight. In WF, that something is an object known as *WorkflowRuntime*. *WorkflowRuntime* starts individual workflow tasks. *WorkflowRuntime* fires events for different situations that pop up while your tasks execute. And *WorkflowRuntime* keeps track of and uses pluggable services you can hook in to the execution environment. (We'll look at some of these pluggable services starting in Chapter 5, "Workflow Tracking.")

The overall WF architecture is shown in Figure 2-1.

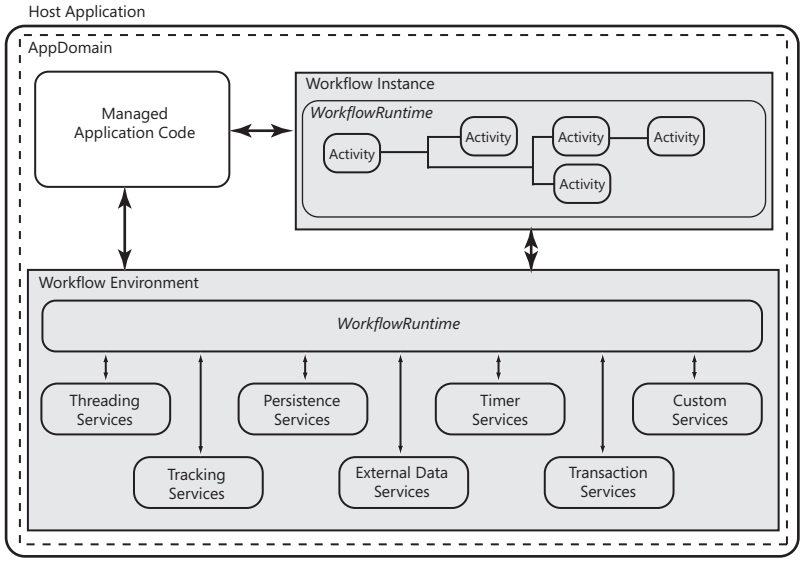


Figure 2-1 WF architecture

WF and your application execute concurrently. In fact, WF requires your application as a host. The host application might be a Windows Forms application, a console application, an ASP.NET Web application, or even a Windows service. The WF runtime and your application execute together in a .NET AppDomain, and there can be only one instance of *WorkflowRuntime* per AppDomain. Attempting to create a second instance of *WorkflowRuntime* in a single AppDomain results in an *InvalidOperationException*.

You build workflow applications—"workflows"—by creating logical groupings of *activities*. These logical groupings work to complete the workflow task you require. When you host the workflow runtime, you essentially hand the workflow your activities and tell it to execute them. This results in a workflow *instance*. The workflow instance is a currently executing workflow task, which is itself composed of logically grouped activities. And, as you recall from the first chapter, activities can execute code you provide as well as make decisions based on input data. We'll cover workflow instances in the next chapter and activities in the chapters to follow.

Hosting WF in Your Applications

In the last chapter, we used the Microsoft Visual Studio workflow project template to build a basic workflow application for us. And in practice you would likely do just that. But if you're like me, just executing wizards and such is fine *only* if you understand the code they're inserting. After all, the code is *yours* to maintain and understand once the code generator's job is complete.

So what does it take to host WF in your application? Well, aside from building the workflow tasks that WF is to run (that's your job), all you really need to do is reference the WF assemblies and provide the necessary code to bring *WorkflowRuntime* into execution, start it, and manage the operational conditions you're interested in managing. In that sense, hosting WF isn't a lot different from using other .NET assemblies. The operational condition management amounts to handling events that the runtime will fire from time to time given specific conditions, such as when the runtime goes idle or an instance sustains an unhandled exception. There is quite a list of available events you can handle, and we'll see some of those a bit later in the chapter, with still others introduced in Chapter 5, "Workflow Tracking," and Chapter 6, "Loading and Unloading Instances."

Note WF can be hosted in a variety of applications, including Microsoft Windows Forms and Windows Presentation Foundation applications, console applications, ASP.NET Web applications, and Windows Services. The basic process remains the same as far as WF is concerned for all of these (very different) host application types.

For now, though, let's build a basic .NET console application and host the workflow runtime ourselves. This will help make the code the Visual Studio workflow project template inserts a little less mysterious.

A Closer Look at the *WorkflowRuntime* Object

Now that we have an instance of *WorkflowRuntime* created in our WorkflowHost application, it's time to take a brief look at how we interact with this object. Like most useful objects, *WorkflowRuntime* exposes a set of methods and properties we use to control the workflow runtime environment. Table 2-1 lists all the *WorkflowRuntime* properties, while Table 2-2 lists the methods we typically use.

Table 2-1 *WorkflowRuntime* Properties

Property	Purpose
IsStarted	Used to determine whether the workflow runtime has been started and is ready to accept workflow instances. *IsStarted* is *false* until the host calls *StartRuntime*. It remains *true* until the host calls *StopRuntime*. Note you cannot add core services to the workflow runtime while it is running. (We'll address starting services in Chapter 5.)
Name	Gets or sets the name associated with the *WorkflowRuntime*. You cannot set *Name* while the workflow runtime is running (that is, when *IsStarted* is *true*). Any attempt to do so will result in an *InvalidOperationException*.

Table 2-2 *WorkflowRuntime* Methods

Method	Purpose
AddService	Adds the specified service to the workflow runtime. There are limitations regarding what services can be added as well as when. We'll look at services in more detail starting in Chapter 5.
CreateWorkflow	Creates a workflow instance, including any specified (but optional) parameters. If the workflow runtime has not been started, the *CreateWorkflow* method calls *StartRuntime*.
GetWorkflow	Retrieves the workflow instance that has the specified workflow instance identifier (which consists of a Guid). If the workflow instance was idled and persisted, it will be reloaded and executed.
StartRuntime	Starts the workflow runtime and the workflow runtime services and then raises the *Started* event.
StopRuntime	Stops the workflow runtime and the runtime services and then raises the *Stopped* event.

There are more methods associated with *WorkflowRuntime*, but the methods shown in Table 2-2 are the ones most commonly used and the ones we'll focus on both here and in the remainder of the book. There are also a number of events *WorkflowRuntime* will raise at various times during workflow execution, but we'll examine those a bit later in the chapter.

After the *using* directive for *System.Text*, add the following line:

```
using System.Workflow.Runtime;
```

4. The using directive introduces the workflow runtime assembly to our source file, but it does little more. We need to add the code to represent the singleton object to the *WorkflowFactory* class. To do that, locate the *WorkflowFactory* class definition:

```
class WorkflowFactory
{
}
```

Not much of a class yet! But we'll fix that. Just after the opening curly brace of the class definition, add these lines of code:

```
// Singleton instance of the workflow runtime.
private static WorkflowRuntime _workflowRuntime = null;|

// Lock (sync) object.
private static object _syncRoot = new object();
```

5. Notice that the field *_workflowRuntime* is initialized to *null*. Our factory will sense this and create a new instance of *WorkflowRuntime*. If *workflowRuntime* is not *null*, our factory won't create a new instance but will hand out the existing instance. To do this, we'll need to add a method designed to create and return our singleton object. Moreover, we'll make the method static so that objects requesting the workflow runtime object don't need to create instances of the factory. To do this, we'll add the following code just after the *_syncRoot* field:

```
// Factory method.
public static WorkflowRuntime GetWorkflowRuntime()
{
    // Lock execution thread in case of multi-threaded
    // (concurrent) access.
    lock (_syncRoot)
    {
        // Check for startup condition.
        if (null == _workflowRuntime)
        {

            // Not started, so create instance.
            _workflowRuntime = new WorkflowRuntime();
        } // if

        // Return singleton instance.
        return _workflowRuntime;
    } // lock
}
```

6. Almost there! When the Visual Studio class template builds a new class, it omits the *public* keyword on the class definition, making it a private class. Because we want other classes to be able to request instances of *WorkflowRuntime*, we'll need to make the factory class public. While we're at it, we'll also mark the class as *static* to prevent direct

```
            if (_workflowRuntime.IsStarted)
            {
                try
                {
                    // Stop the runtime
                    _workflowRuntime.StopRuntime();
                }
                catch (ObjectDisposedException)
                {
                    // Already disposed of, so ignore...
                } // catch
            } // if
        } // if
    }
```

The entire listing for the *WorkflowFactory* object is shown in Listing 2-1. We'll not make any further changes until Chapter 5.

Listing 2-1 The Complete *WorkflowFactory* Object

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Workflow.Runtime;

namespace WorkflowHost
{
    public static class WorkflowFactory
    {
        // Singleton instance of the workflow runtime
        private static WorkflowRuntime _workflowRuntime = null;

        // Lock (sync) object
        private static object _syncRoot = new object();

        // Factory method
        public static WorkflowRuntime GetWorkflowRuntime()
        {
            // Lock execution thread in case of multi-threaded
            // (concurrent) access.
            lock (_syncRoot)
            {
                // Check for startup condition
                if (null == _workflowRuntime)
                {
                    // Provide for shutdown
                    AppDomain.CurrentDomain.ProcessExit += new
                              EventHandler(StopWorkflowRuntime);
                    AppDomain.CurrentDomain.DomainUnload += new
                              EventHandler(StopWorkflowRuntime);

                    // Not started, so create instance
                    _workflowRuntime = new WorkflowRuntime();

                    // Start the runtime
```

Tip As it happens, Visual Studio can add the handler for you. Nice! Here's how it works: After you type the equal sign (=), press the tab key to let IntelliSense add the EventHandler keyword and name. It will leave the name highlighted. Without changing the highlight, type in the name you want to use (workflowIdled in the preceding example). Then just press the Tab key when Visual Studio prompts you, and Visual Studio will insert the handler with the matching name immediately below whatever procedure you are coding. Of course, you can always modify the handler method name after the handler has been inserted into your code if you need to.

```
me.WorkflowIdled +=
                    new EventHandler<WorkflowEventArgs>(workflowRuntime_WorkflowIdled);  (Press TAB to insert)
```

3. Following the code you just added, type in this line of code to add the handler for workflow completion:

```
workflowRuntime.WorkflowCompleted += new
    EventHandler<WorkflowCompletedEventArgs>(workflowCompleted);
```

4. And now add the handler for the *WorkflowTerminated* event:

```
workflowRuntime.WorkflowTerminated += new
    EventHandler<WorkflowTerminatedEventArgs>(workflowTerminated);
```

5. If you compile and run WorkflowHost, the application should compile and execute. But there is no workflow executed because we didn't ask the workflow runtime to start a workflow instance. (We'll add this in the next chapter.) In preparation, though, let's add some code. First, we'll add the automatic reset event we'll need to stop the main thread long enough for the workflow events to fire (so that we can observe them). The *AutoResetEvent* class is perfect for the job. Following the two lines of code you just typed in (in steps 3 and 4), add these lines of code. (We'll define the waitHandle object in the next step.)

```
Console.WriteLine("Waiting for workflow completion.");
waitHandle.WaitOne();
Console.WriteLine("Done.");
```

6. We'll need to create the *_waitHandle* object, so add this static class member just prior to the *Main* method:

```
private static AutoResetEvent waitHandle = new AutoResetEvent(false);
```

7. *AutoResetEvent* is exported by *System.Threading*, so add the *using* directive to the list at the top of the Program.cs source file:

```
using System.Threading;
```

8. The three event handlers (created by Visual Studio 2005) both contain "not implemented yet" exceptions. We need to get rid of those and implement some code. Locate the first handler we added, *workflowIdled*, and replace the exception you find there with the following lines of code:

```
Console.WriteLine("Workflow instance idled.");
```

Chapter 3

Workflow Instances

After completing this chapter, you will be able to:

- Initiate a workflow instance, both with and without startup parameters
- Determine the status of your running workflow instances
- Stop workflow instances
- Determine why your workflow instances were idled or terminated

The workflow runtime, when it comes right down to it, is really there for one purpose—supporting your workflow-based tasks. Workflow tasks, called *instances* when they are executing, are the heart of the workflow system. They're why the workflow runtime exists in the first place.

A workflow instance is composed of one or more *activities*. (We'll look at the various activities starting in Chapter 7, "Basic Activity Operations.") The primary activity, or *root activity* is referred to as the *workflow definition*. The workflow definition normally acts as a container for the other activities that will actually do the work.

Note A *workflow definition* is what you ask the workflow runtime to execute, whereas an *instance* is an executing workflow definition. There is a distinct difference. One is executing and the other is not. However, I'll use the terms interchangeably throughout this chapter, and even in the rest of the book, because in the end we're interested in executing software, not just in writing it. Besides, "instance" rolls off the tongue more easily than does "workflow definition."

Where do instances come from? They come from you. You have problems to solve and software to write to solve those problems, and if workflow processing fits the needs of your application requirements, at least part of the software you'll write is the workflow task or tasks that the workflow runtime will execute for you. Microsoft provides the workflow runtime. You provide the rest. After all, it's *your* application.

Windows Workflow Foundation (WF) is there to help. Not only will WF execute the workflow instances you create, but it will also help you create them. WF has a rich graphical designer that's there to help you lay out workflow software in much the same way as you build ASP.NET Web forms, Windows Forms, or Windows Presentation Foundation software. You roll the mouse cursor over the Toolbox, select one of the many activity items you find there, drag that item over to the design surface, and drop it. If the item has configurable properties, you can tailor those to suit your purpose using the Microsoft Visual Studio Properties pane.

Table 3-2 ***WorkflowInstance*** **Methods**

Method	Purpose
Terminate	Synchronously terminates the workflow instance. When the host requests termination of the workflow instance, the workflow runtime kills the instance and tries to persist the instance's final state. Then *WorkflowInstance* sets *SuspendOrTerminateInfoProperty* to the string (*reason*) passed into *Terminate*. Finally, it raises the *WorkflowTerminated* event and passes *reason* in the *Message* property of a *WorkflowTerminatedException* contained in the *WorkflowTerminatedEventArgs*. If another, different, exception is raised during persistence, the workflow runtime passes that exception in *WorkflowTerminatedEventArgs* instead.

There are more methods associated with *WorkflowInstance* than I've shown here. We'll look at those in more detail when we persist workflow instances to a Microsoft SQL Server database in Chapter 6, "Loading and Unloading Instances."

Let's build a workflow task and see how we kick it off.

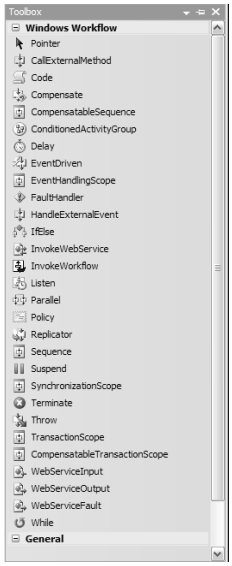
Starting a Workflow Instance

Before we can start a workflow instance, we must have a workflow task for WF to execute. In the first chapter, we asked Visual Studio to create a workflow-based project for us that automatically included a raw workflow task we modified to validate U.S. and Canadian postal codes. We could, if we wanted, go back to that project and physically copy the workflow source code, or we could reference the resulting assembly *PCodeFlow.exe* and try to use the workflow we created directly. And, in practice, you might do that.

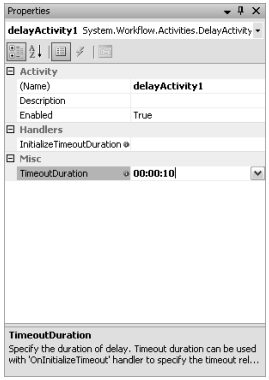
In this case, however, we're attempting to learn to write workflow applications. What fun is swiping existing workflow code when we can build new? Let's simulate a long-running task by using a sequential workflow that contains a delay. We'll execute some code prior to the delay to pop up a message box. After the delay, we'll again pop up a message box to indicate our work has finished. (We'll know our workflow instance finished anyway because WorkflowHost handles the *WorkflowCompleted* event, but this way we get to write a bit more workflow code.) As we progress through the book, our examples will become more detailed and richer, but for now because we're still new to it, we'll keep the examples on the "type in less code" side to concentrate more on the concept than on improving typing skills.

Note Remember, a *sequential workflow* is one that executes activities one after another. This process is in contrast to a *state machine workflow*, which executes activities based on state transitions. If this sounds like so much nonsense right now, don't worry. We'll get into all this in the next chapter.

2. Select Code from the Toolbox, and drag the *Code* activity component onto the workflow designer's surface.

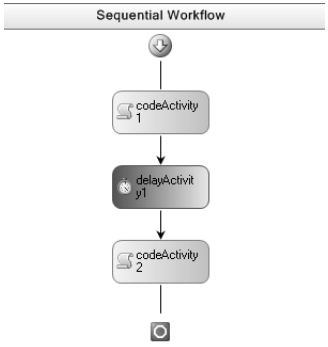


As the mouse approaches the area marked "Drop Activities to Create a Sequential Workflow," the designer view changes slightly to indicate that you can drop the *Code* activity component on the designer surface.



Note With the *Delay* activity's properties showing, note the activity name *delayActivity1*. We could change that if we like, but for now we'll leave it. Later, when we change the delay value dynamically, we'll need to remember this name. Here, we'll just set the time-out duration and continue.

9. To briefly review, we have an initial *Code* activity that we'll use to display a message box prior to a delay. We have the *Delay* activity that will wait 10 seconds and then allow our workflow processing to move on. What we need now is the second *Code* activity to show the second message box. To add that, repeat steps 2 through 6, dropping the new *Code* activity following the *Delay* activity you placed in the preceding two steps. However, when you name the event in the *ExecuteCode* edit control (step 6), type **PostDelayMessage** as the event name. The final workflow as it exists in the workflow visual designer should appear as you see here:



> **Tip** You can combine this step with step 1. After you select the LongRunningWorkflow assembly using the Browse button, simply click the .NET button and search for the system assemblies as described here in step 2.

3. Now let's get down to coding. Click Program.cs for the WorkflowHost project in the Visual Studio Solution Explorer, and then click the Solution Explorer's View Code button.
4. When the source file is displayed for editing, locate the following code in the *Main* method:

```
Console.WriteLine("Waiting for workflow completion.");
```

5. After this line of code, add the following:

```
WorkflowInstance instance =
workflowRuntime.CreateWorkflow(typeof(LongRunningWorkflow.Workflow1));
instance.Start();
```

6. Compile and execute the WorkflowHost application.

> **Tip** If you execute the WorkflowHost application by pressing F5 in Visual Studio, the application will complete and the console window will be destroyed before you could possibly read all of the messages. Setting a breakpoint so that the application stops and breaks into the debugger (so you can access the console window while the application has stopped) or executing the compiled application from the command prompt should allow you to see the screen output.

You should find that you had to dismiss both message boxes. Meanwhile, in the console window, WorkflowHost sent various messages to the console that we can review:

```
Waiting for workflow completion.
Workflow instance idled.
Workflow instance completed.
Done.
```

The golden nugget in all this is rolled into these few lines of code:

```
WorkflowInstance instance =
    workflowRuntime.CreateWorkflow(typeof(LongRunningWorkflow.Workflow1));
instance.Start();
```

Here, we're using the *WorkflowRuntime* object to create a workflow instance by passing into the *CreateWorkflow* method the workflow definition (by its type) we want to execute. When we receive the *WorkflowInstance* object in return, we call its *Start* method to initiate workflow processing. Notice that the workflow instance required no input from us prior to executing. Wouldn't it be nice to be able to pass in a variable delay value? That's the topic of the next section.

4. Next, find this line of code a bit further down:

```
WorkflowInstance instance =
    workflowRuntime.CreateWorkflow(typeof(LongRunningWorkflow.Workflow1));
```

5. Change this line of code to match the following:

```
WorkflowInstance instance =
    workflowRuntime.CreateWorkflow(typeof(LongRunningWorkflow.Workflow1), parms);
```

With that last step, we're done adding code. Compile the application by selecting Build Solution from the Visual Studio Build menu, and execute it by pressing the F5 key. Did it work?

It should have, because we added code to account for the situation where no command line was provided. If you were to execute the application as we did in Chapter 1 (in the "Executing your workflow application" section), by adding different delay values on the command line you would see the delay's effects reflected in the time lag between instances of the message boxes.

Determining Workflow Instance Status

Interestingly, if you look at the methods and properties of both the workflow runtime object and the workflow instance object, you don't find a status property. How do you know if there is a workflow executing, and if there is one, where is that workflow in its process? Is it idled? Is it executing? How do we know?

I'm jumping ahead a little, but this is the most logical place to discuss workflow status determination. As it happens, the workflow definition of a given workflow instance provides you with the execution status. The base class *Activity* exposes an *ExecutionStatus* property that sports a member of the *ActivityExecutionStatus* enumeration. I've listed the *ActivityExecutionStatus* values with their meaning in Table 3-3.

Table 3-3 *ActivityExecutionStatus* Values

Property	Purpose
Canceling	The *Activity* is in the process of canceling.
Closed	The *Activity* is closed.
Compensating	A transaction has failed, causing the compensation action to be initiated. (We'll learn more about this in Chapter 15.)
Executing	The *Activity* is currently running.
Faulting	The *Activity* has sustained an exception.
Initialized	The *Activity* has been initialized but is not yet running.

The enumerated values in Table 3-3 all refer to an activity object, but remember that the workflow definition *is* an activity. That means if we query the workflow definition for its status, we're effectively determining the status of the entire instance. The following process shows how we add the code we need to query the workflow definition.

Chapter 4

Introduction to Activities and Workflow Types

After completing this chapter, you will be able to:

- Explain how activities form workflows
- Describe the differences between a sequential and a state machine workflow
- Create a sequential workflow project
- Create a state machine workflow project

As I write this, I have young children who love Legos. Legos, if you've not seen them, are intricate building blocks and components from which you build larger and more complex systems. (See *www.lego.com*.) There is a rather complete line of *Star Wars* Lego sets, for example. I even have a Lego Yoda I was given to assemble while I recuperated from knee surgery—for injuries incurred the only time I ever was thrown from my horse (don't ask). Yes, you caught me. My children aren't the only Lego fans in the house.

Activities are the Lego blocks of Windows Workflow Foundation (WF) workflow processing. If you divide a business process (or workflow task) into pieces, you typically find it's composed of smaller, more granular tasks. If a high-level business process is designed to route information through some data processing system, the sublevel tasks might include such things as reading data from a database, generating a file using that data, shipping the file to a remote server using FTP or an XML Web service, marking the information as having been processed (through a write to a database and an entry into an audit trail), and so forth. These sublevel tasks are typically focused on a specific job. Read the database. FTP the file. Insert an audit trail entry. In a word, they are *activities*. Actions. Focused tasks.

When you build workflows, you gather the individual activities together and move from one activity to the next. Some activities act as containers for other activities. Some activities perform a single task, as I've described here. One container-based activity is chosen to hold all the rest, and that's the *root activity* I mentioned in the previous chapter. The root activity will either be a *sequential activity* or a *state-machine activity*, and we'll examine those types of activity in this chapter.

Note If you're familiar with ASP.NET programming, this context object serves essentially the same purpose as the *System.Web.HttpContext* object. Those familiar with .NET Framework programming may find similarities between this context object and *System.Threading.Thread.CurrentContext*. The goal of all of these context objects is the same—to provide a place to store and easily recall information specific to a currently executing instance. In this case, it's an instance of an executing activity.

Dependency Properties 101

Within Table 4-2, you'll also see something known as a *DependencyProperty*. If you've looked at the code we've used so far in the preceding chapters, sprinkled into the workflow code that Visual Studio generated for you are events based on dependency properties. What is a *DependencyProperty*?

Normally, if you create a property for a class, you also create a field within the class to store the property value. This type of code is common:

```
class MyClass
{
    protected Int32 _x = 0;
    ...
    public Int32 X
    {
        get { return _x; }
        set { _x = value; }
    }
}
```

The field *_x* is more formally called a *backing store*. Your class, in this example, provides the backing store for the *X* property.

Both WF and Windows Presentation Foundation (WPF), however, quite often need access to your class's properties. WPF will need to determine the sizing and spacing of controls in a container so that things will be optimally rendered. WF needs dependency properties to facilitate *activity binding*, which is a process that allows different activities to reference (and be bound to) the same property. The WF *ActivityBind* class facilitates this for you.

To facilitate the chaining and binding of properties, the backing store for your class properties can be shifted from your class down to the .NET runtime itself. The property data is still stored—it's just that .NET stores it for you. In a sense, it's like having a room full of lockers. You didn't build the building and install the lockers, but you can use them when you need to. In this case, you register your locker request with .NET, and .NET provides you with the locker and the means to access what's inside the locker. These lockers, so to speak, are called *DependencyProperties*.

Sequential workflows are ideal for implementing business processes. If you need to read data from a source, process that data, send notifications, and write results to another data sink, the sequential workflow probably will suit your needs. This does not imply that a sequential workflow is inappropriate for processes that depend on user interaction for approval or disapproval of specific tasks. But such user interaction should not be the focal point of the workflow itself.

If you need a lot of user interaction, the state-machine workflow is likely a better alternative. When your workflow sends notifications to users or other systems (for whatever reason—to notify, to request approval, to select an option, and so on), as the users or other systems respond, their responses form *events*. These events trigger the workflow to move forward from processing state to processing state. I'll discuss this a bit more later in the chapter and again in Chapter 14, "State-Based Workflows."

The final type of workflow we'll look at (in Chapter 12) is the rules-based workflow, where the decisions whether to move forward with the workflow and in what direction are based on business rules. These workflows are usually reserved for more complex scenarios. For example, suppose that a customer orders custom-built formed plastic components, perhaps to install into new automibiles. Your job is to build a workflow task that monitors customer orders and component production.

Let's throw a little complication in, though. Plastics are long polymer chains created using a relatively complex chemical process. In this process, a plasticizing compound is used to facilitate the long polymer chains to form into even longer chains (making the plastic less brittle). The plasticizer easily evaporates, so the process is completed in a low-vapor pressure system.

The customer orders this plastic part, and your system tells you there should be enough plasticizer to complete the order. However, when you check the tank, you find more plasticizer has evaporated than you expected (perhaps because of a leak in the tank) and you cannot fill the order quite yet.

Do you ship a partial order, sending the remainder of the order later at your expense? Or do you have an agreement in place to hold orders until complete? Do you order more plasticizer at great expense (overnight shipment) to meet the customer's need, knowing that several suppliers offer quick shipments of plasticizer for increased cost? Do you offer the customer a way to participate in this process?

What the rules enforce isn't truly the point here. It's only important that there are rules and that they are applied under the given circumstances.

You might believe that all workflow could be created from a rules-driven approach, and indeed many can be. We typically don't always use this approach because other workflow styles, such as the sequential workflow and the state-machine workflow, are easier to build and test. They offer far fewer internal, automated decisions. In the end, it amounts to complexity. Rules-based workflows are often complex and are therefore more costly to create

The *State* Activity

A type of workflow we've not seen so far in this book is one based on the model of a deterministic *state machine*. Crusty old digital microelectronics engineers (and I sadly admit I am one) well understand the design and implementation of state machines. However, if you're not a crusty old digital microelectronics engineer, the concept might be new to you. Chapter 14 is entirely dedicated to working with state-based workflows, but I'll introduce the concept here. We'll build a quick little state-based workflow as well.

People have dedicated their entire lives to the study of *finite state machines*. There is a language specific to finite state machines, custom mathematical notation, and a specific way they're diagrammed. I can't possibly hope to provide you with everything you might need to build state-based workflows, but I can provide enough information so that you can make sense of them as we build one or two.

Tip There are many resources on the Internet that better and more deeply describe finite state machines. One that's worth looking into is *en.wikipedia.org/wiki/Finite_state_machine*.

Breaking the term down, we have three words: finite, state, and machine. *Finite*, in this case, means we have a limited number of states we're willing to transition to. *States* are logical conditions our application transitions to as events occur. And *machine* implies automation. Let me illustrate by using an example.

In engineering school, you might be asked to design any number of digital systems using a finite state machine. The two classic examples are the vending machine and the washing machine. Looking at the vending machine, think about the steps the machine has to take to provide you with its product (soda, candy, snacks...whatever it's designed to dispense). As you insert coins, it counts the money until you've provided at least enough to pay for your selection. As you make your selection, it checks inventory (if it didn't already indicate it was out of a specific choice). If there is inventory, it dispenses your selection. And if you provided too much money, it makes change and dispenses that as well.

We can model the vending machine using a finite state machine. To diagram a finite state machine, we use circles as states and arrows as transitions between states. The transitions are triggered by events. There is a logical starting point and one or more logical stopping points. If we stop somewhere in between, our application is said to be in an indeterminate or invalid state. Our job is to prevent invalid states. We can't give our product away for free, and we shouldn't ask for more money than the product is worth, or worse, "eat" the user's money. Users have been known to get violently angry at vending machines that take their money but don't provide the goods.

In WF, the individual state in a state-based workflow is modeled by the *State* activity. The *State* activity is a composite activity, but it limits the child activities it will contain. You'll learn much more about state-based workflows in Chapter 14.

Note Just as sequential workflows used a specialized version of the *Sequence* activity to contain the entire workflow, so too do state-based workflows have a specialized root activity—the *StateMachineWorkflow* activity, which is a specialization of the *State* activity. The specialization is again necessary so that the root activity can accept initialization parameters when it is initially executed.

Building a State Machine Workflow Application

So how do we build state-based workflows? As it happens, Visual Studio is, as always, there to help. Creating a state-based workflow project is as easy as creating a sequential workflow. Let's build a basic state-based workflow project to see how it's done. We won't add any code to the project quite yet—we need to progress a bit further in the book for that. But when we need it, we'll know how state-based workflows are created.

Creating a state machine workflow application

1. Start Microsoft Visual Studio 2005 as you did for building a sequential workflow.
2. On the File menu, select New and then Project, which activates the New Project dialog box.
3. As you typically do when creating new projects, expand the Visual C# node in the Project Types pane to show the project types available for the C# language.
4. Under the Visual C# node, click the Workflow node to display the workflow-based project templates.
5. In the Templates pane, click State Machine Workflow Console Application or State Machine Workflow Library. As with sequential workflows, the first template creates a full-fledged application designed to work within the Console window. The second template creates a dynamic-link library you can use from other workflow-based applications.

Chapter 5

Workflow Tracking

After completing this chapter, you will be able to:

- Describe workflow-pluggable services
- Create an event tracking database and populate it with tables and scripts
- Activate the event-tracking service
- Create a custom tracking profile
- View your workflow's tracking information

So far we've seen the basic objects that workflows are built from and controlled by. We build workflow tasks from activities, which when executing are managed by a *WorkflowInstance* object. Workflow instances are queued and controlled by the *WorkflowRuntime*. But Windows Workflow Foundation (WF) doesn't just provide us with objects—it also provides us with services that work alongside our objects.

Pluggable Services

Workflow services are additional software functions that your workflows can, and will, use to complete their tasks. Some of the services are optional, like the tracking service we'll work with in this chapter. Other services are required for your workflow to execute at all.

Workflow services are *pluggable*. A pluggable service is a service that can be selected a la carte to perform specific tasks. For example, there are services that manage threading, tracking, transactions, and so forth. You select the service that's right for your workflow. You can even create your own.

So what do these services look like? What do they do for us? Table 5-1 lays out the available base services and gives you a better idea which services are available and what they do.

Table 5-2 Event Tracking Objects

Object	Purpose
SqlTrackingQuery	Provides methods and properties that you can use to access certain kinds of tracking data stored in a SQL database by the *SqlTrackingService*.
SqlTrackingQueryOptions	Contains properties that are used to constrain the set of *SqlTrackingWorkflowInstance* objects returned by the *SqlTrackingQuery.GetWorkflows* method.
SqlTrackingWorkflowInstance	Returned by a call to either *SqlTrackingQuery.TryGetWorkflow* or *SqlTrackingQuery.GetWorkflows* to provide access to the tracking data collected by the *SqlTrackingService* in a SQL database for a specific workflow instance.
TrackingProfile	Filters tracking events, and returns tracking records based on this filtering to a tracking service. There are three kinds of tracking events that can be filtered: activity status events, workflow status events, and user events.
UserTrackingLocation	Specifies a user-defined location that corresponds to a user event in the executing root workflow instance.
UserTrackingRecord	Contains the data sent to a tracking service by the runtime tracking infrastructure when a *UserTrackPoint* is matched.
UserTrackPoint	Defines a point of interest that is associated with a user event.
WorkflowDataTrackingExtract	Specifies a property or a field to be extracted from a workflow and sent to the tracking service together with an associated collection of annotations when a track point is matched.
WorkflowTrackingLocation	Defines a workflow-qualified location that corresponds to a workflow event in the executing root workflow instance.
WorkflowTrackingRecord	Contains the data sent to a tracking service by the runtime tracking infrastructure when a *WorkflowTrackPoint* is matched.
WorkflowTrackPoint	Defines a point of interest that is associated with a workflow event.

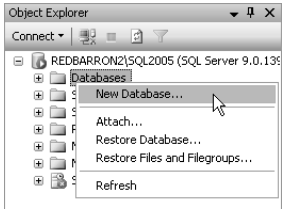
These objects can be thought of as belonging to two main categories: tracking data retrieval and tracking specification. Tracking retrieval objects, such as *SqlTrackingQuery*, help you gather tracking data once it is stored in the database. Tracking specification objects, such as the track points and locations, allow you to dictate what is tracked from your workflow code.

The tracking specification objects, such as the point and location obects, are organized into three main groups: activity events, workflow events, and user events. The activity-related tracking objects, such as *ActivityTrackingPoint* or *ActivityTrackingLocation*, are designed to record activity-related event information to the tracking database. These events include such things as activity cancellation, unhandled exceptions, and execution events. Workflow event-tracking objects work in a similar manner but for workflow-related events, such as the

(If SQL Server Management Studio Express is already running, click the Connect button and then choose Database Engine in the Object Explorer pane.) The Server Type drop-down list should indicate Database Engine (the default value). The Server Name drop-down list should display the server's name and the instance of SQL Server 2005 you want to use on that server. My server is named "Redbarron," and the SQL Server instance I want to use is "SQL2005." If you want to use the default instance, simply provide only the server's name. As for authentication, you should use the authentication methodology you selected when you installed SQL Server (or see your database administrator for any assistance you might require). Click Connect to connect to your database server.

4. SQL Server Management Studio Express's user interface typically consists of two panes. The left pane mimics Windows Explorer and shows the databases and services associated with your database server. The right pane is the work pane, where you'll type in scripts, set up table columns, and so forth. The left pane is known as the Object Explorer, and if it is not visible, you can activate it by selecting Object Explorer from the View menu.

5. Right-click on the Databases node to activate the context menu, and select New Database.



6. The New Database dialog box appears. Type **WorkflowTracking** in the Database Name field, and click OK.

Using the *SqlTrackingService* Service

With the workflow tracking database in place, it's time to actually use it. Let's create a new workflow and see how we track events. We'll start by creating a slightly more complex workflow so that we have a few events to play with. After we have the basic workflow built, we'll add the necessary tracking code.

Create a new workflow for tracking

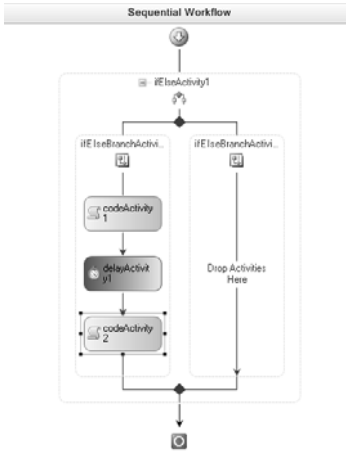
1. To make it easier, I've created two versions of this sample application. (In fact, I'll try to do that for the remainder of the book.) The WorkflowTracker application has two different versions: one incomplete and one complete. The complete version is entirely finished and ready to run, and you will find it in the \Workflow\Chapter5\ WorkflowTracker Completed\ directory. The incomplete version is yours to modify when completing the steps I've outlined here, and you'll find it in the \Workflow\Chapter5\Workflow Tracker\ directory. Feel free to use either one. Whichever version you choose, you can open it for editing by dragging its .sln file onto an executing copy of Visual Studio.
2. After Visual Studio opens the WorkflowTracker solution for editing, create a separate sequential workflow library project as you did in Chapter 3 to house our new workflow. (See the section in Chapter 3 entitled "Adding a sequential workflow project to the WorkflowHost solution".) Name this workflow library **TrackedWorkflow** and save it in the \Workflow\Chapter5\WorkflowTracker directory.

> **Note** This will turn out to be a common theme—creating a workflow to go with a basic application. The good news is you'll become quite adept at creating workflow host applications, which isn't a bad thing, I think. If you prefer, you can create a workflow-based console application directly within Visual Studio, but you should still create a separate workflow library. This arrangement will make it easier to monitor your workflow when we use WorkflowMonitor later in the chapter.

3. After you have completed the steps to add the workflow library project, Visual Studio opens the visual workflow designer for editing. If it doesn't, locate the Workflow1.cs file in Visual Studio's Solution Explorer and click the View Designer toolbar button to activate the designer.
4. For this workflow, let's combine some aspects of previous workflows we've built. This should give us a slightly more complex workflow without pushing us too far away from what we've seen. Let's begin by dragging an *IfElse* activity from the Toolbox onto the designer's surface.

> **Tip** Building this part of the workflow will be a lot like building the workflow from Chapter 1.

another *Code* activity into *ifElseBranchActivity1* and set their properties. The *Delay* activity should delay 10 seconds (00:00:10), as in Chapter 3, and the second *Code* activity should execute a method called **PostDelayMessage**. When completed, the designer should look like the following:



11. With our designer work complete, let's add some code. Click the View Code toolbar button in the Visual Studio Solution Explorer toolbar to bring up the C# code for Workflow1.cs. Begin by adding a reference to *System.Windows.Forms* as well as the corresponding *using* statement at the top of the Workflow1.cs file.

```
using System.Windows.Forms;
```

12. As you scan through the file, you should see the three event handlers Visual Studio added for you as activity properties: *PreDelayMessage*, *PostDelayMessage*, and *QueryDelay*. As you did in Chapter 3, add message boxes to the *Code* activity methods so that the application notifies you when the workflow is executing. To *PreDelayMessage*, add this code:

```
MessageBox.Show("Pre-delay code is being executed.");
To PostDelayMessage, add this code:
MessageBox.Show("Post-delay code is being executed.");
```

13. The slightly more interesting case is the code we'll add to *QueryDelay*:

```
e.Result = false; // assume we'll not delay...
if (MessageBox.Show("Okay to execute delay in workflow processing?",
     "Query Delay",
     MessageBoxButtons.YesNo,
     MessageBoxIcon.Question) == DialogResult.Yes)
{
    // Allow progression
    e.Result = true;

    // Show message
```

4. Click WorkflowFactory.cs for the WorkflowTracker project in the Visual Studio Solution Explorer, and then click the Solution Explorer's View Code button.

5. So that we can use the Visual Studio IntelliSense capability, add the following lines of code to the top of the file, following the last *using* directive you find there:

```
using System.Workflow.Runtime.Tracking;
using System.Configuration;
```

6. While in the WorkflowFactory.cs file, scan down and find the line where we create the *WorkflowRuntime* instance. We'll need to introduce *SqlTrackingService* to the workflow runtime at this point. Add this line of code following the line of code calling *GetWorkflowRuntime*:

```
String conn = ConfigurationManager.
                  ConnectionStrings["TrackingDatabase"].
                  ConnectionString;
_workflowRuntime.AddService(new SqlTrackingService(conn));
```

With that last step, we're done adding code that we'll need to actually perform the tracking. (In a moment, we'll add more code to display the tracking results.) Compile the application by selecting Build Solution from the Build menu, and execute it by pressing the F5 or Ctrl+F5 keys.

Note If the program crashed with an *ArgumentException* telling you that the runtime can't access the given database, you probably need to add the login account you're using as a valid database user (as specified in the connection string we added to the workflow code a moment ago) or otherwise adjust the connection string. If you're unsure how to do this, the online reference material can help (*http://msdn2.microsoft.com/en-us/library/ms254947(VS.80).aspx*), or see your local database administrator for assistance.

Assuming the workflow runs as expected, you should find entries in the WorkflowTracking database's *ActivityInstance* table, as shown in Figure 5-1.

Note To see the table, open SQL Server Management Studio Express (if not already open) and expand the Databases tree control node. Locate the WorkflowTracking database and expand its tree node. Expand the Tables node and right-click the ActivityInstance tree node. From the resulting context menu, select Open Table.

Table - dbo.ActivityInstance | Summary

WorkflowInsta…	ActivityInstanc…	QualifiedName	ContextGuid	ParentContext…
1	1	Workflow1	4d943215-10d0-…	4d943215-10d0-…
1	2	ifElseActivity1	4d943215-10d0-…	4d943215-10d0-…
1	3	ifElseBranchActi…	4d943215-10d0-…	4d943215-10d0-…
1	4	codeActivity1	4d943215-18d0-…	4d943215-18d0-…
1	5	delayActivity1	4d943215-18d0-…	4d943215-18d0-…
1	6	codeActivity2	4d943215-18d0-…	4d943215-18d0-…
NULL	NULL	NULL	NULL	NULL

Figure 5-1 ActivityInstance table

Tracking User Events

SqlTrackingService, because it is a part of WF, is capable of tracking events that are inherently a part of WF. That is, it can track standard events fired from activities and workflow instances. But what about events your workflow generates? How do we track those?

As it happens, the *Activity* activity supports a method named *TrackData*. *TrackData* has two overloaded versions: one that takes an object to be stored in the tracking database and one that accepts a string as a key as well as the object to be stored.

If you execute *TrackData* and pass in data for tracking purposes, usually a string, the information will be stored in the tracking database as user-event data. Try the procedure below with the WorkflowTracker project we created earlier in the chapter.

Retrieve tracking records from your workflow

1. With the WorkflowTracker project open in Visual Studio for editing, open the Workflow1.cs C# file for editing.
2. Scroll down until you find the *PreDelayMessage* and *PostDelayMessage* methods we added when we created the workflow.
3. After the code to display the message box in *PreDelayMessage*, add this code:

```
this.TrackData("Delay commencing");
```

4. Similarly, add this code following the message box code in *PostDelayMessage*:

```
this.TrackData("Delay completed");
```

5. Compile and execute the program using F5 or Ctrl+F5.

Now open the *UserEvent* table in the WorkflowTracking database, following the procedure outlined for the *ActivityInstance* table (Figure 5-1). There should be two rows, one for each time we called *TrackData* in our workflow, as shown in Figure 5-3.

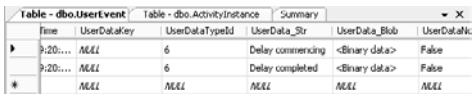


Table - dbo.UserEvent | Table - dbo.ActivityInstance | Summary

	Time	UserDataKey	UserDataTypeId	UserData_Str	UserData_Blob	UserDataNc
▶	9:20:...	NULL	6	Delay commencing	<Binary data>	False
	9:20:...	NULL	6	Delay completed	<Binary data>	False
*		NULL	NULL	NULL	NULL	NULL

Figure 5-3 UserEvent table showing results of calling TrackData

Building Custom Tracking Profiles

I've mentioned tracking profiles a few times already in this chapter, but I intentionally didn't get into the details. I left those for this section.

As you might recall, a tracking profile is a mechanism for limiting the amount of information the WF tracking architecture will store in the tracking database. Ultimately, a tracking profile is no more than an XML document that stipulates what is to be included or excluded from a

```
            SqlCommand cmd = new SqlCommand(storedProc, conn);
            cmd.CommandType = CommandType.StoredProcedure;

            SqlParameter parm = new SqlParameter("@TypeFullName",
                                            SqlDbType.NVarChar, 128);
            parm.Direction = ParameterDirection.Input;
            parm.Value = typeof(TrackedWorkflow.Workflow1).ToString();
            cmd.Parameters.Add(parm);
            parm = new SqlParameter("@AssemblyFullName",
                                            SqlDbType.NVarChar, 256);
            parm.Direction = ParameterDirection.Input;
            parm.Value =
                  typeof(TrackedWorkflow.Workflow1).Assembly.FullName;
            cmd.Parameters.Add(parm);
            parm = new SqlParameter("@Version", SqlDbType.VarChar, 32);
            parm.Direction = ParameterDirection.Input;
            parm.Value = "1.0.0.0";
            cmd.Parameters.Add(parm);
            parm = new SqlParameter("@TrackingProfileXml",
                                                    SqlDbType.NText);
            parm.Direction = ParameterDirection.Input;
            parm.Value = writer.ToString();
            cmd.Parameters.Add(parm);

            conn.Open();
            cmd.ExecuteNonQuery();
        } // if
    } // try
    catch (Exception ex)
    {
        if (ex is SqlException)
        {
            // Check to see if it's a version error.
            if (ex.Message.Substring(0,24) == "A version already exists")
            {
                // Version already exists...
                Console.WriteLine("NOTE: a profile with the same version" +
                                  " already exists in the database");
            } // if
            else
            {
                // Write error message
                Console.WriteLine("Error writing profile to database: {0}",
                                                            ex.ToString());
            } // else
        } // if
        else
        {
            // Write error message
            Console.WriteLine("Error writing profile to database: {0}",
                                                        ex.ToString());
        } // else
    } // catch
    finally
    {
```

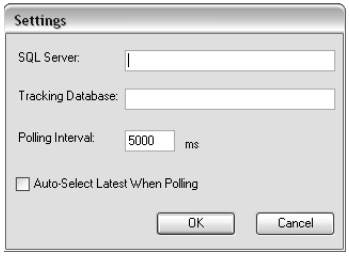
Execute WorkflowMonitor

1. Copy the WorkflowMonitor executable file (WorkflowMonitor.exe) to the **\Workflow\Chapter5\WorkflowTracker\WorkflowTracker\bin\Debug** subdirectory containing the executable and library file for the WorkflowTracker application that we created to demonstrate *SqlTrackingService*. You should find the WorkflowMonitor.exe executable file in the following directory, assuming you built the debuggable version of the application:

   ```
   \Workflow\Chapter5\Applications\WorkflowMonitor\CS\WorkflowMonitor\bin\Debug\
   ```

2. In Windows Explorer, double-click the WorkflowMonitor.exe file to execute the WorkflowMonitor application.

3. WorkflowMonitor stores configuration information in the WorkflowMonitor.config configuration file, found in *Application.LocalUserAppDataPath*. (If you are running SQL Server Express, you may see an error message when WorkflowMonitor tries to connect to SQL Server; simply click OK.) Because this is likely the first time you've run WorkflowMonitor on your system, this configuration file is nonexistent. WorkflowMonitor senses this and immediately displays its Settings dialog box.



4. The settings you can control are the name of the server that hosts the tracking database, the name of the tracking database itself, the polling period (initially set to five seconds), and whether you want the currently executing workflow selected when WorkflowMonitor initializes (defaulted to not selected). For now, all we really need to do is set the server's name and database name. If you've named things according to the steps I outlined when we created the database, the server's name will be **localhost** (or **.\SQLExpress** if you are using SQL Server Express) and the database name will be **WorkflowTracking**. After you've typed these values, click OK.

> **Note** Be sure to type the actual information for your system if it differs from the values I've used here. Note that WorkflowMonitor builds the connection string using the server and database names you provide in the Settings dialog box. It assumes Windows Integrated Security for the connection, so if you are using SQL Server Authentication, you'll need to edit the code that creates the connection string to include this information. You'll find this code in the DatabaseService.cs file.

Chapter 6

Loading and Unloading Instances

After completing this chapter, you will be able to:

- Understand why and when workflow instances are unloaded and then later reloaded
- Understand why and when workflow instances are persisted
- Set up SQL Server 2005 to work with WF and workflow persistence
- Use the *SqlWorkflowPersistenceService*
- Load and unload instances in your workflow code
- Enable the persistence service to automatically load and unload idled workflow instances

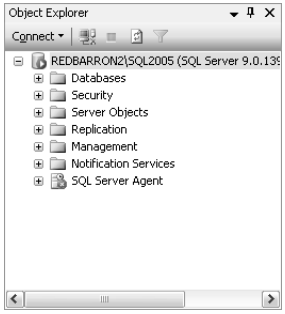
If you take a moment and really consider how you might use Windows Workflow Foundation (WF) and workflow processing in your applications, you'll probably imagine a lot of situations that involve long-running processes. After all, business software essentially simulates and executes business processes, and many of those processes involve people or outside vendors, ordering and shipping, scheduling, and so forth. People aren't capable of responding to automated processes in microseconds, but on loaded business servers, microseconds count. Servers are valuable, busy resources, and having them spin up threads only to have the threads wait minutes, hours, or even days and weeks is unacceptable for many reasons.

So the designers of WF knew they would have to provide a mechanism for taking idle workflows temporarily offline while waiting for some long-running task to complete. They decided to offer Microsoft SQL Server as an optional storage medium because databases are great places to store (and not lose) valuable data. They also created another pluggable service we can easily incorporate into our workflows to support this persistence mechanism. The hows, whys, and whens are what we'll explore in this chapter.

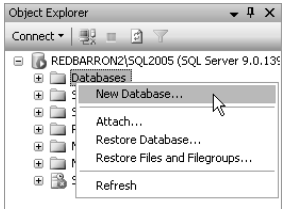
Persisting Workflow Instances

Did you know that at the very core of modern Microsoft Windows operating systems is a very special piece of software that is responsible for allocating time on the computer's processor for the various threads that request it? If a single thread monopolized the processor for an undue period of time, other threads would starve and the system would appear to lock up. So this piece of software, the *task scheduler*, moves threads into and out of the processor's execution stack so that all threads are given execution time.

3. Activate Object Explorer if it is not visible. You can activate it by selecting Object Explorer from the View menu.



4. Right-click on the Databases node to activate the context menu, and select New Database.



5. The New Database dialog box should now appear. Type **WorkflowStore** in the Database Name field, and click OK.

SqlWorkflowPersistenceService if the service is present when the workflow instance is running to perform the save and restore tasks.

On the surface, this all sounds relatively simple. If we need to swap a workflow instance out to the database, we just tell the persistence service to save it for us. But what happens if we're using a single database to persist workflows running in different processes? How do workflow instances actually stop and restart in the middle of their execution?

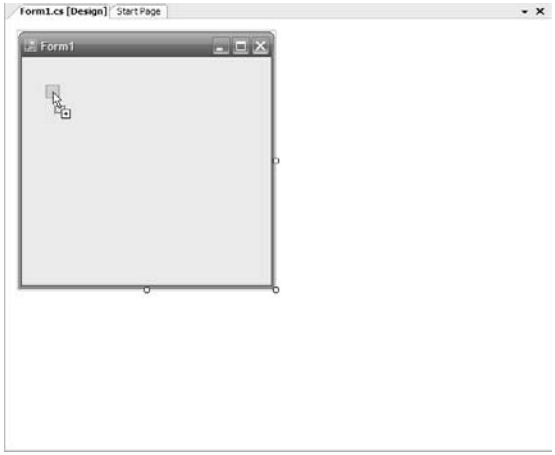
It's not uncommon for there to be a single database used for storing workflow instances. But each instance might have been executing on different machines and possibly within different processes on any given machine. If a workflow instance is saved and later restored, we must have a way to also restore the system state that was in effect at the time the workflow instance was executing. For example, *SqlWorkflowPersistenceService* stores whether or not the instance was blocked (waiting for something), its execution status (executing, idle, and so on), and various and sundry informational items such as serialized instance data and the owner identifier. All this information is necessary to rehydrate the instance at a later time.

We can control this persistence via the *WorkflowInstance* object through three methods, shown in Table 6-1.

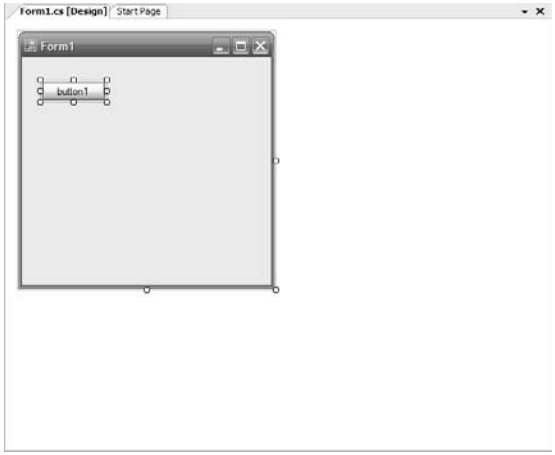
Table 6-1 *WorkflowInstance* Methods, Revisited

Method	Purpose
Load	Loads a previously unloaded (persisted) workflow instance.
TryUnload	Tries to unload (persist) the workflow instance from memory. Unlike calling *Unload*, calling *TryUnload* will not block (hold up execution) if the workflow instance cannot be immediately unloaded.
Unload	Unloads (persists) the workflow instance from memory. Note that this method blocks the currently executing thread that made this unload request until the workflow instance can actually unload. This can be a lengthy operation, depending on the individual workflow task.

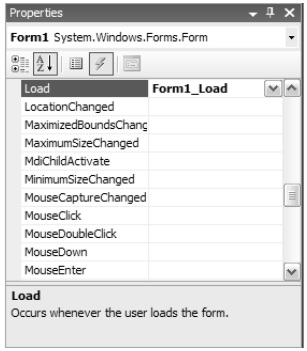
As Table 6-1 indicates, we have two methods available for unloading and persisting a workflow instance. Which method you use depends on what you intend for your code to do. *Unload* waits for the workflow instance to become ready to be persisted. If this takes a long time, the thread executing the *Unload* operation also waits a long time. However, *TryUnload* will return immediately when asked to unload an executing workflow instance. But there is no guarantee the workflow instance actually unloaded and persisted to the database. To check for that, you should examine the return value from *TryUnload*. If the value is *true*, the workflow instance did unload and persist itself. If the value is *false*, the workflow instance didn't unload and persist. The advantage of *TryUnload* is that your thread isn't sitting there waiting. The disadvantage, or course, is that you might have to repeatedly use *TryUnload* to force out the executing workflow instance.



Then drop it to insert the button into your form.



4. We'll want to assign some meaningful text to the button so that we know what we're clicking. (The control text "button1" just isn't descriptive enough!) With the button highlighted (the little squares showing), select the button's *Text* property in the Visual Studio Properties pane and change the text value to **Start Workflow**.



9. Once the load event handler is inserted, Visual Studio will switch you to the code view for the main application form. Because we just added the form's *Load* event handler, we might as well add the initialization code we'll need. Type the following into the *Form1_Load* handler method:

```
_runtime = WorkflowFactory.GetWorkflowRuntime();
_runtime.WorkflowCompleted +=
   new EventHandler<WorkflowCompletedEventArgs>(Runtime_WorkflowCompleted);
_runtime.WorkflowTerminated +=
    new EventHandler<WorkflowTerminatedEventArgs>(Runtime_WorkflowTerminated);
```

We've seen code like this before that creates the workflow runtime and hooks some of the major events we'll be interested in intercepting.

10. Somewhere we need to declare the *_runtime* field, so look for the opening brace for the *Form1* class. After the opening brace, type this:

```
protected WorkflowRuntime _runtime = null;
protected WorkflowInstance _instance = null;
```

11. If you try to compile the application at this point, it won't compile. We'll need to add a reference to the Windows Workflow Foundation assemblies as we've done in previous chapters—that process is the same whether we're building a graphical user interface or a console-based application. So add the workflow assembly references for *System.Workflow.Runtime*, *System.Workflow.ComponentModel*, and *System.Workflow.Activity* and then insert the following *using* declaration at the top of the source file following the other *using* declarations:

```
using System.Workflow.Runtime;
```

12. Although we now have an application that hosts the workflow runtime, it doesn't actually do anything. To make it functional, we'll need to add some code to the button event handlers, starting with *button1_Click*. Scroll through the main application form's source file until you find *button1_Click*, and add this code:

```
button2.Enabled = true;
button1.Enabled = false;
_instance = _runtime.CreateWorkflow(typeof(PersistedWorkflow.Workflow1));
_instance.Start();
```

17. Instead of including the *using* statement for *System.Workflow.Runtime.Tracking*, add the following:

```
using System.Workflow.Runtime.Hosting;
using System.Configuration;
```

18. Finally add the persistence service to the runtime by adding this code following the creation of the workflow runtime object:

```
string conn = ConfigurationManager.
                  ConnectionStrings["StorageDatabase"].
                  ConnectionString;
_workflowRuntime.AddService(new
                              SqlWorkflowPersistenceService(conn));
```

> **Note** Because we inserted code to create a workflow instance from the *PersistedWorkflow.Workflow1* type (in step 12), our host application won't compile and execute. We'll take care of that in the following section.

There you have it! A Windows graphical user interface and host application we can use to house our workflow. Speaking of workflow, shouldn't we create one to execute? In fact, that's next.

Create a new unloadable workflow

1. We're again going to add a new sequential workflow library to our existing project as we've done in previous chapters. With the WorkflowPersister application active in Visual Studio, select Add from the File menu. When the secondary menu pops up, select New Project. Add a sequential workflow library project named **PersistedWorkflow** from the resulting New Project dialog box.
2. After the new project is created and added to the application solution, the workflow visual designer will appear. Drag a *Code* activity from the Toolbox and drop it onto the designer's surface. In the Visual Studio Properties panel, set the *Code* activity's *ExecuteCode* property to **PreUnload** and press the Enter key.
3. Visual Studio will automatically take you to the source code file for your workflow, so while there, add this code to the newly inserted *PreUnload* method:

```
_started = DateTime.Now;
System.Diagnostics.Trace.WriteLine(
    String.Format("*** Workflow {0} started: {1}",
                  WorkflowInstanceId.ToString(),
                  _started.ToString("MM/dd/yyyy hh:mm:ss.fff")));
System.Threading.Thread.Sleep(10000); // 10 seconds
```

Now let's see if all this actually works. We'll run two test workflows—one we'll let run to completion, and one we'll force to unload. Then we'll compare execution times and look inside the SQL Server database to see what was recorded there.

Test the WorkflowPersister application

1. With the WorkflowPersister application open for editing in Visual Studio, press F5 or select Start Debugging from the Debug menu to execute the application. Fix any compilation errors if there are any. Note that for this test we want to write trace output to the Output window, so if you don't already have the Output window available, be sure to activate it by choosing Output from the Visual Studio View menu.
2. Click the Start Workflow button to create and start a workflow instance. The Start Workflow button should become disabled, while the Unload Workflow button will be enabled. Because we told the workflow thread to sleep for 10 seconds, after 10 seconds has elapsed the Unload Workflow button should disable itself and the Start Workflow button should be re-enabled. In this test, the workflow ran to completion, and the duration of time the workflow executed should total 10 seconds.
3. Click the Start Workflow button again. However, this time click the Unload Workflow button within the 10-second sleep period. The button will be disabled for the duration of the 10-second period, after which the Load Workflow button will be enabled. At this point, your workflow is persisted and will remain unloaded until you reload it.
4. But before you reload the workflow instance, open SQL Server Management Studio Express as we've done in the past and open the WorkflowStore database. Expand the Tables node, and right-click the InstanceState table and select Open Table from the context menu to open the table for editing. There should be one row in the table. This row is your persisted workflow instance!

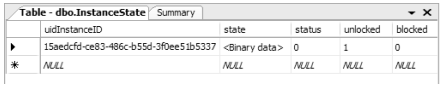


Table - dbo.InstanceState | Summary

	uidInstanceID	state	status	unlocked	blocked
▸	15aedcfd-ce83-486c-b55d-3f0ee51b5337	<Binary data>	0	1	0
*	NULL	NULL	NULL	NULL	NULL

5. Feel free to look at the table; there is no rush. But when you're ready, go back to your executing instance of WorkflowPersister and click the Load Workflow button. The Load Workflow button will then become disabled, while the Start Workflow button will become enabled.
6. Close the WorkflowPersister application by clicking the X in the upper right corner or pressing Alt+F4. The application will shut down.
7. The Visual Studio Output window should now contain information regarding the two workflows we executed (because we wrote trace information to the window as each workflow instance ran). Activate the Output window by clicking on the Output tab at the bottom of the Visual Studio application window.

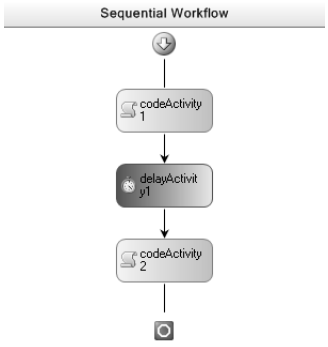
5. Create a separate sequential workflow library project as you did in Chapter 3 to house our new workflow. (See the section in Chapter 3 entitled "Adding a sequential workflow project to the WorkflowHost solution.") Name this workflow library **IdledWorkflow**.

6. Repeat step 2 and then steps 4 through 6 from the previous example, in the section entitled "Create a new unloadable workflow." This places two *Code* Activities in your workflow.

7. Adding the second *Code* Activity in the last step will take you to the Visual Studio code editor. While there, add this code to the *PreUnload* method (you added the *PostUnload* method code in the preceding step):

```
_started = DateTime.Now;
System.Diagnostics.Trace.WriteLine(
    String.Format("*** Workflow {0} started: {1}",
                  WorkflowInstanceId.ToString(),
                  _started.ToString("MM/dd/yyyy hh:mm:ss.fff")));
```

8. Return to the visual workflow designer and drag a *Delay* Activity onto the surface and drop it between the two *Code* Activities.



9. Assign the *Delay* Activity's *TimeoutDuration* property to be 30 seconds. This should be enough time to examine the WorkingStore *InstanceState* database table.

10. With the workflow now complete, add a reference to the workflow from the Workflow-Idler application. Right-click the WorkflowIdler tree control node in Visual Studio's Solution Explorer and select Add Reference. When the Add Reference dialog box appears, click the Projects tab. Select IdledWorkflow from the list and click OK.

11. Open Program.cs in the WorkflowIdler project for editing. Locate this line of code:

```
Console.WriteLine("Waiting for workflow completion.");
```

Part II
Working with Activities

In this part:

Chapter 7

Basic Activity Operations

After completing this chapter, you will be able to:

- Know how to use the *Sequence* activity
- Know how to use the *Code* activity
- Know how exceptions are thrown and handled in workflows
- Know how to suspend and terminate your workflow instances from workflow code

Up to this point, we've seen the basics. We've worked a bit with the workflow runtime, which orchestrates the workflow process. We've looked at the workflow instance, which is an executing workflow. And we've dug into a couple of the pluggable services available to us, such as those used for tracking and persistence. "What's next?" you ask.

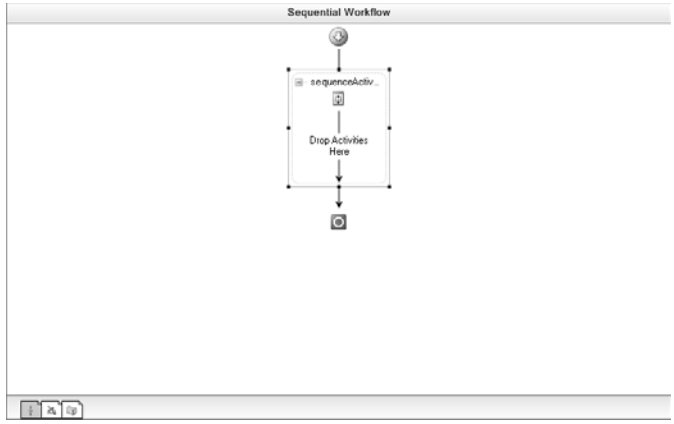
Now it's time to look at the stars of the show, the activities themselves. Windows Workflow Foundation (WF) ships with a large set of activities you can use from the moment you install WF to bring workflow processing to your applications. And given the wide variety of activities, WF can workflow-enable all sorts of applications, not just those designed to interact with people.

In this chapter, we'll go back and formally introduce a couple of activities we've already seen—*Sequence* and *Code*. But I believe proper error handling is critical in well-designed and well-implemented software, so we'll look at how you throw exceptions using workflow activities, catch exceptions, and even suspend and terminate your workflows. Let's start with the *Sequence* activity.

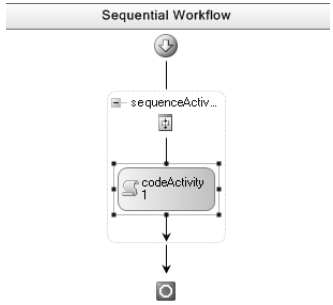
Using the *Sequence* Activity Object

Actually, it's not entirely correct to say we've seen the *Sequence* activity. The workflow applications we've created have actually used the *SequentialWorkflow* activity, but the general idea is the same—this activity contains other activities that are executed in sequence. This is in contrast to parallel execution, which you might do using the parallel activities we'll see in Chapter 11, "Parallel Activities."

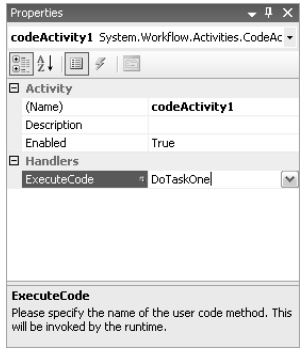
When you execute tasks in a specific order, you're doing things in sequence. This is often necessary. For example, imagine you're making a grilled cheese sandwich for lunch. You find your griddle or frying pan and place it on the stove. You pull a loaf of bread from the pantry and butter one side of two slices. Then you pull the cheese from the refrigerator and place a couple of pieces onto one of the slices of bread, which you've placed butter-side down on the griddle or in the pan. Then you cover the assembly with the second slice of bread,



4. Next, drag a *Code* activity from the Toolbox and drop it onto the *Sequence* activity you just placed.



5. In its *ExecuteCode* property, type **DoTaskOne** and press Enter.



6. Visual Studio automatically brings up the code editor. Locate the *DoTaskOne* method Visual Studio just added, and in that method place this code:

```
Console.WriteLine("Executing Task One...");
```

workflow instance. If we want to deal with exceptions earlier in the termination process, we need to use a combination of the *Throw* and *FaultHandler* activities.

Note The recommended practice is to use the combination of *Throw* and *FaultHandler* rather than *Throw* alone. Using the *Throw* activity by itself is equivalent to using the C# *throw* keyword without an exception handler in traditional application code. In this section, we'll use *Throw* alone to explore what happens. In the next section, we'll combine *Throw* and *FaultHandler* to see how they work together.

Turning our attention to the *Throw* activity, when you drag and drop the *Throw* activity onto the designer, you'll find there are two properties you need to set. The first is the *FaultType* property, where you tell the *Throw* activity what type of exception will be thrown, and the other is the *Fault* property which, if not null at the time the exception is to be thrown, will *be* the actual exception thrown.

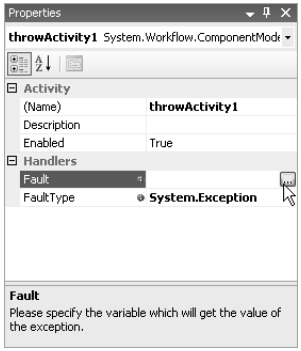
The *FaultType* property probably doesn't need a lot of explanation. It simply tells the workflow instance what exception type will be thrown. But logically the mere existence of this property is telling us that we need to provide a *Throw* activity for the specific types of exceptions we want to throw. Exceptions we don't specifically throw (and later handle, if we want) are handled by the workflow runtime and ignored.

But what's the story behind the *Fault* property? It's simply the actual exception the *Throw* activity uses, if set. If null, the *Throw* activity still throws an exception of type *FaultType*, but it is a new exception with no established *Message* (and remember, it's the *Message* property that provides us some description of the error aside from the exception type itself).

Note Actually, the exception's *Message* property will have a value, but it will be something like the following: "Exception of type 'System.Exception' was thrown." When throwing an exception, the default message (when no message property is set on the exception type) will differ if the exception is constructed using the default constructor. The bottom line is the exception message will depend on the exception and not on anything WF did or didn't do.

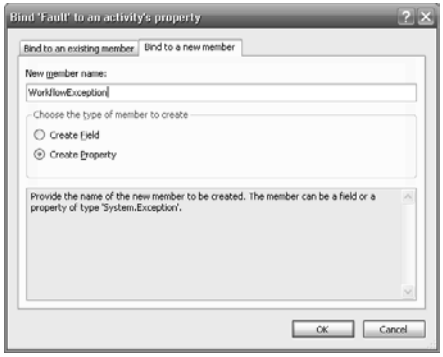
If you want the *Throw* activity to throw an exception you establish, with a meaningful *Message*, you need to create an instance of the exception using the *new* operator and assign it to the same property you bound to the *Throw* activity.

Let me state this in a slightly different way. The *Throw* activity, and more specifically its *Fault* property, are bound to a property of the same exception type in an activity of your choosing in your workflow (including the root activity). That is, if you have a *Throw* activity that throws a *NullReferenceException*, you must provide a property on some activity in your workflow that is of type *NullReferenceException* for *Throw* to use. *Throw* then binds to this activity's property so that it can use the same exception you assigned (or should assign) using the *new* operator.



Note Failing to set the *Fault* property, or even the *FaultType* property, will not result in a compilation error. However, the type will default to *System.Exception* with the message "Property Fault not set."

8. Clicking the *Fault* property's browse button activates the Bind Fault To An Activity's Property dialog box. Because we've added no fault code ourselves, click the Bind To A New Member tab and type **WorkflowException** in the New Member Name edit control. Click OK. This adds the property *WorkflowException* to your root activity.



Note In Chapter 15, "Workflows and Transactions," we'll look at compensatable activities, including compensated transactions. Handling faults is part of the story. You can also "compensate" for them, which means to take action to mitigate the damage the exception might (or might not) cause.

Quick Tour of the Workflow Visual Designer

At this point in the book, if you've built the example workflows I've presented, you're probably comfortable with the idea of dragging and dropping activities onto the workflow visual designer's surface, wiring up their properties, and compiling and executing basic workflow code. There are a couple of things I've not told you yet, and I withheld their description until now only because we were focusing on different aspects of workflow programming at the time—namely, how we write and execute workflow programs.

However, now that you have some experience writing workflows and using Visual Studio as a workflow authoring tool, let's take a minute and see what else Visual Studio offers in terms of workflow authoring assistance. There are two primary areas I'd like to briefly describe: additional visual designer surfaces and debugging.

Additional Visual Designer Surfaces

If you look back at the first six chapters, in each sample application we dragged items from the Toolbox and dropped them onto the visual designer surface that Visual Studio presented us with. But did you happen to notice the three small buttons in the lower-left corner of the designer's window? I've reproduced them here in Figure 7-1 from the graphic from step 3 of the "Creating a workflow using the *Sequence* activity" procedure at the beginning of this chapter.

Figure 7-1 Visual designer surface selection buttons

The left button activates the default workflow visual editor we've been using so far in the book. The center button activates a view that allows you to write workflow code for cancellation (as shown in Figure 7-2), and the right button activates the fault handlers view (as shown in Figure 7-3).

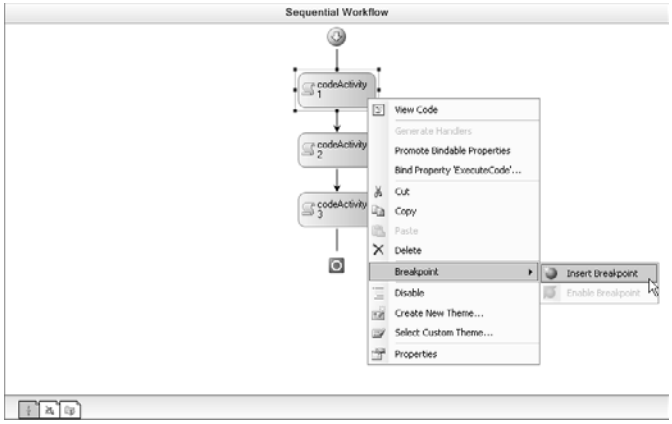


Figure 7-5 Setting a breakpoint using the workflow visual designer

The workflow visual designer then places the familiar red ball within the graphic for the activity, just as it places the red ball next to a line of code you intend to break for debugging. You see this in Figure 7-6. To remove the breakpoint, select Breakpoint and then Delete Breakpoint from the context menu or Toggle Breakpoint (which you can also use to set the breakpoint) from the Debug menu, or even Disable All Breakpoints from the Debug menu.

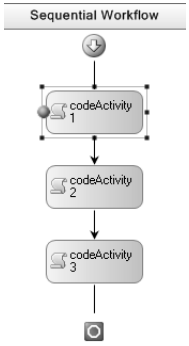
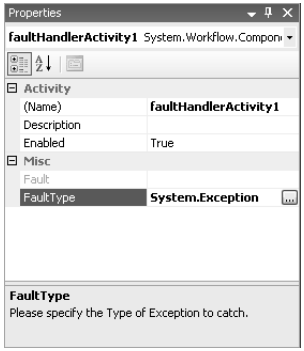


Figure 7-6 An activity in the workflow visual designer with a breakpoint set

Armed with this knowledge, we can now add a *FaultHandler* activity to our workflow.

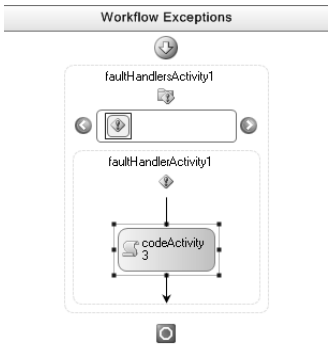
Modifying our workflow to use the *FaultHandler* activity

1. With the ErrorThrower application open for editing in Visual Studio, select the Workflow1.cs file in the ErrorFlow project and click the designer button to activate the workflow visual designer. The general workflow is already established, so we won't need to change anything there. (Even though you're working with ErrorThrower, I created a separate completed solution for this section. If you haven't completed the earlier steps in this chapter, you can jump straight to this point by opening the solution in \Workflow\Chapter7\ErrorHandler, or you can follow along with a completed version in \Workflow\Chapter7\ErrorHandler Completed.)



Note Although you see the *Fault* property in the preceding graphic, it's actually disabled and therefore cannot be set. Ignore it.

6. So far, we've added a *FaultHandler* activity and we've told it what type of exception it will be handling, but we've not actually provided any code to deal with the exception if it's thrown. To do that, drag a *Code* activity from the Toolbox and drop it into the area below where we dropped the *FaultHandler* activity itself. This area, identified by the name *faultHandlerActivity1*, is like a miniature workflow visual designer. So it readily accepts the *Code* activity, and as we've done with other instances of the *Code* activity, assign a value to its *ExecuteCode* property. In this case, type in **OnException** and press Enter.



7. Visual Studio then adds the *OnException* event handler to *Workflow1* and opens the code editor for editing. To the *OnException* event handler, add this code:

```
Console.WriteLine(
    "Exception handled within the workflow! The exception was: '{0}'",
    WorkflowException != null ? WorkflowException.Message :
    "Exception property not set, generic exception thrown");
```

Tip Typing a literal string, as we've done here, is perfectly acceptable. However, you can also bind this to a string-based dependency property that is more easily altered as your workflow executes. Clicking the browse button (the button with the three dots you see in the graphic) activates the binding dialog box we saw in step 7 of the "Creating a workflow using the *Throw* activity" procedure. Simply follow the same basic steps as you did there.

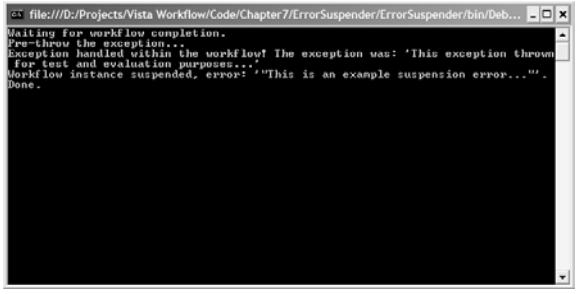
4. Because we don't have a *WorkflowSuspended* event handler in our main application, we need to edit the Program.cs file from the main application and add it. In the *Main* method, locate code to hook the existing event handlers and add the following:

```
workflowRuntime.WorkflowSuspended +=
     new EventHandler<WorkflowSuspendedEventArgs>(workflowSuspended);
```

5. Because we're using an event handler named *workflowSuspended*, we need to code that:

```
static void workflowSuspended(object sender, WorkflowSuspendedEventArgs e)
{
    Console.WriteLine("Workflow instance suspended, error: '{0}'.",
                                                                e.Error);
    waitHandle.Set();
}
```

6. Compile the application by clicking Build, Build Solution from Visual Studio's main menu and then press F5 or Ctrl+F5 to execute the application (after correcting any compilation errors). The program output should be similar to this:



When you run this application, you should see the console output generated by the *WorkflowSuspended* event handler in our main application. But you can do more than simply write text to the console. You can take any other action appropriate for your process flow. Although you could resume the workflow instance processing from here, it's generally not recommended. For one thing, the entire activity that was processing will be skipped, leaving your workflow instance to resume processing at a later stage in its flow, which probably isn't a good thing (what was skipped, and how do you account for it?). At the very least, however, you can cleanly remove the workflow instance from processing and apply any necessary cleanup code.

As if exceptions and suspended workflow instances aren't enough, you can, if you need to do so, terminate your workflow instance. Let's see how.

Chapter 7 Quick Reference

To	Do This
Use the *Sequence* activity	*SequenceActivity* is a composite activity, and as such it acts as a drop site for other activities. Simply drag and drop an instance of the *Sequence* activity onto the workflow visual designer. Then drag and drop other activities onto *SequenceActivity* as necessary to complete your workflow. Those activities will be executed in the order they appear in the designer, from top to bottom.
Use the *Code* activity	With the visual workflow designer visible and active, drag an instance of the *Code* activity onto the surface and drop it into the workflow process as appropriate. Then provide an *ExecuteCode* method name in the Properties window. Populate that method with the code you want to execute.
Use the *Throw* activity	With the visual workflow designer visible and active, drag an instance of the *Throw* activity onto the surface and drop it into the workflow process as appropriate. Then assign the *Fault* and *FaultType* properties to provide the *Throw* activity with the exception to throw as well as the type of exception it should expect.
Use the *FaultHandler* activity	With the visual workflow designer visible and active, and with the Fault Handlers view showing, drag an instance of the *FaultHandler* activity onto the surface and drop it into the workflow process as appropriate. Then assign the *FaultType* property to assign the type of exception it will handle.
Use the *Suspend* activity	With the visual workflow designer visible and active, drag an instance of the *Suspend* activity onto the surface and drop it into the workflow process as appropriate. Then assign the *Error* property to assign the error string value it will report to the workflow runtime via the *WorkflowSuspended* event.
Use the *Terminate* activity	With the visual workflow designer visible and active, drag an instance of the *Terminate* activity onto the surface and drop it into the workflow process as appropriate. Then assign the *Error* property to assign the error string value it will report to the workflow runtime via the *WorkflowTerminated* event.

Chapter 8

Calling External Methods and Workflows

After completing this chapter, you will be able to:

- Build and call local data services that are external to your workflow
- Understand how interfaces are used to communicate between the host process and your workflow
- Use external methods designed to transfer data between you workflow and host application
- Invoke additional workflows from within an executing workflow

As I was writing the preceding chapters, I kept thinking to myself, "I can't wait to get to the part where we return real data to the host application!" Why? Because as interesting as workflow is, there is only so much you can do to demonstrate activities and workflows without returning something realistic to the executing application. Theoretically, there might be an infinite number of interesting workflow examples and demonstrations I could write that only processed initialization data (such as the postal code example you saw in Chapter 1, "Introducing Microsoft Windows Workflow Foundation"). But things become far more interesting, and to be honest, far more realistic when we kick off a workflow that seeks and processes data from external sources and returns that data in some processed form to our application.

So why not just crack open an object and start sending data into an executing workflow, or from an executing workflow to the host application? Actually, you can do this with existing technology outside Windows Workflow Foundation (WF) using some form of marshaled communications, such as .NET Remoting or an XML Web service. *Marshaling*, sometimes also called *serialization*, is a process whereby data is converted from its original form into a form suitable for transmission between different processes and even between different computers.

Why mention marshaling? Because your workflow is executing on a different thread than your host process, and passing data between threads without proper marshaling is a recipe for disaster for reasons beyond the scope of this chapter. In fact, your workflow could be in a persisted state at the time you tried to send it data. It's not on a different thread...it's not even executing.

But wouldn't a .NET Remoting connection or an XML Web service be considered excessive if we just want to pass data between our workflow and the host process that's controlling it? Absolutely! And this is the premise of this chapter—how we can establish *local* communications. We'll be setting up the systems necessary to satisfy the thread data-marshaling

activities designed to send and receive data from the workflow's perspective. From the host application's perspective, receiving data amounts to an event, while sending data is simply a method call on the service object.

Note We'll return to the concept of bidirectional data transfer after looking at a few more activities in later chapters. The workflow activity to receive data from the host application is based on the *HandleExternalEvent* activity, which we'll look at in Chapter 10, "Event Activities." We also need to dig deeper into the concept of correlation, which we'll do in Chapter 17, "Host Communication." For now, we'll simply return complex data to the host once the workflow instance has completed its task.

Although we need to do more than just this, ultimately we need to add the *ExternalDataService* to our workflow runtime. *ExternalDataService* is a pluggable service that facilitates transferring serializable data between workflow instances and the host application. The service code we'll write in the upcoming sections will do that and more. To see what's in store for us, let's look at the overall development process.

Designing and Implementing Workflow Intraprocess Communication

We begin by deciding what data will be transferred. Will it be a *DataSet*? An intrinsic object, such as an integer or string? Or will it be a custom object we design ourselves? Whatever it is, we'll then design an interface that the *ExternalDataService* can bind to. This interface will contain methods we design, and they can be designed to send and receive data from both the workflow instance's and the host's perspective. The data we're transferring will be passed back and forth using the methods in this interface.

We'll then need to write some code—our part of the external data service—which represents the connection or *bridging* code the host and workflow will use to interact with the WF-provided *ExternalDataService*. If we were dealing with an XML Web service, Visual Studio could automatically create proxy code for us. But there is no such tool for workflow, so we need to devise this bridging code ourselves. The "bridge" we'll use here actually consists of two classes: a *connector* class and a *service* class. You can name them anything you like, and my name assignments are not necessarily what WF might call them (I don't believe WF has a name for them!), but I prefer to think of them in this fashion. The connector class manages the data conduit itself (maintains state), while the service class is used directly by the host and workflow to exchange data.

With the interface we created in hand, we'll execute a tool, *wca.exe*, which is typically located in your Program Files\Microsoft SDKs\Windows\v6.0\Bin directory. The tool is called the *Workflow Communications Activity* generator utility and given an interface it will generate two activities you can use to bind the interface to your workflow instance: one for sending data, the *invoker*; and one for receiving data, the *sink*. Once they are created and massaged a bit, you

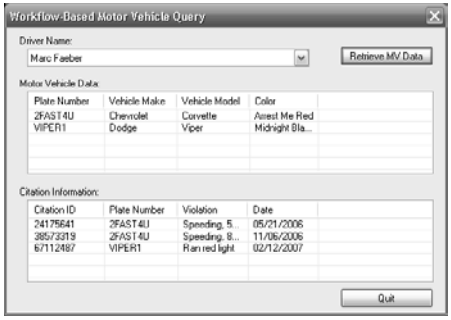


Figure 8-3 The MVDataChecker user interface with retrieved data

At this point in the application's execution, you might decide to retrieve another driver's information or quit the application. If you quit the application while a search is progressing, the actively executing workflow instance is aborted.

With that brief tour of the application complete, let's look at the code we need to write to make all this work, starting with the interface we need to provide to WF so that it can raise that "data available" event I mentioned.

Creating Service Interfaces

The service interface is entirely yours to create, and it should be based on the data you want to communicate between your workflow instance and your host application. For this sample application, imagine you need to design a workflow to retrieve driver information from various sources and that you want the information collated into a single data structure—a *DataSet* with multiple *DataTables*, one table for vehicle identification information and one table for driver traffic violations. In reality, you'd retrieve this data from some source, or from a set of different sources, but we'll simply use imaginary data to keep things more focused on the workflow itself. In the host application, we'll display the (bogus) data in a pair of *ListView* controls.

You'll pass in to the workflow instance the name of the driver, which the workflow instance uses to look up the driver and vehicle information. With the data in hand, the workflow instance notifies the host application that data is ready, and the host application reads and displays the information.

So we really need just a single method in our interface: *MVDataUpdate*. We know we want to send a *DataSet*, so we'll pass a *DataSet* into *MVDataUpdate* as a method parameter.

Using *ExternalDataEventArgs*

Earlier I mentioned that, to the host application, communications from the executing workflow appear as events. The host application can't know beforehand precisely when the workflow instance will have data, and polling for data is terribly inefficient. So WF uses the asynchronous model that .NET itself uses and fires events when data is available. The host application hooks those events and reads the data.

Because we want to send information to the recipient of our event, we need to create a customized event argument class. If you've created a custom event argument class in your previous work, you probably used *System.EventArgs* as the base class.

WF external data events, however, require a different argument base class if only to carry the instance ID of the workflow instance issuing the event. The base class we use for external data events is *ExternalDataEventArgs*, which itself derives from *System.EventArgs*, so we're on familiar ground. In addition, there are two other requirements: we must provide a base constructor that accepts the instance ID (a Guid), which in turn passes the instance ID to the base constructor, and we must mark our argument class as serializable using the *Serializable* attribute. Let's now build the external data event argument class we need.

Creating a workflow data event argument class

1. With the MVDataService project still open in Visual Studio, locate the MVDataAvailableArgs.cs file and open it for editing.
2. You should find only the using directives and the namespace definition in the file, so after the opening brace for the namespace definition, add the following lines of code:

```
[Serializable]
public class MVDataAvailableArgs : ExternalDataEventArgs
{

}
```

3. Finally, we need to add the required constructor to provide the base class with the workflow instance ID:

```
public MVDataAvailableArgs(Guid instanceId)
    : base(instanceId)
{
}
```

The complete event argument class is shown in Listing 8-2.

```
                lock (_syncLock)
                {
                    // Re-verify the service isn't null
                    // now that we're locked...
                    if (value != null)
                    {
                        _service = value;
                    } // if
                    else
                    {
                        throw new InvalidOperationException(
                          "You must provide a service instance.");
                    } // else
                } // lock
            } // if
            else
            {
                throw new InvalidOperationException(
                   "You must provide a service instance.");
            } // else
        }
    }
```

4. Mirroring the accessor property for the service, we need to add a property to access the data. Add this code following the code you added in the preceding step:

```
public DataSet MVData
{
    get { return _dataValue; }
}
```

5. Because the connector class derives from *IMVDataService*, we must implement *MVDataUpdate*:

```
public void MVDataUpdate(DataSet mvData)
{
    // Assign the field for later recall
    _dataValue = mvData;

    // Raise the event to trigger host read
    _service.RaiseMVDataUpdateEvent();
}
```

The workflow uses this method to store the *DataSet* in the data value field. It raises the event to let the host know data is available. The full bridge connector class is shown in Listing 8-3. Note that we're not ready to compile the entire application just yet. We still have a bit more code to add.

Listing 8-3 MVDataconnector.cs completed

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Workflow.Activities;
using System.Workflow.Runtime;
using System.Data;
```

```
        // If we're just starting, plug in ExternalDataExchange service.
        if (_dataExchangeService == null)
        {
            // Data exchange service not registered, so create an
            // instance and register.
            _dataExchangeService = new ExternalDataExchangeService();
            _workflowRuntime.AddService(_dataExchangeService);
        } // if

        // Check to see if we have already added this data
        // exchange service.
        MVDataConnector dataConnector = (MVDataConnector)workflowRuntime.
                                    GetService(typeof(MVDataConnector));
        if (dataConnector == null)
        {
            // First time through, so create the connector and
            // register as a service with the workflow runtime.
            _dataConnector = new MVDataConnector();
            _dataExchangeService.AddService(_dataConnector);
        } // if
        else
        {
            // Use the retrieved data connector.
            _dataConnector = dataConnector;
        } // else

        // Pull the service instance we registered with the
        // connection object.
        WorkflowMVDataService workflowDataService =
                                        MVDataConnector.MVDataService;
        if (workflowDataService == null)
        {
            // First time through, so create the data service and
            // hand it to the connector.
            workflowDataService = new WorkflowMVDataService(instanceID);
            MVDataConnector.MVDataService = workflowDataService;
        } // if
        else
        {
            // The data service is static and already registered with
            // the workflow runtime. The instance ID present when it
            // was registered is invalid for this iteration and must be
            // updated.
            workflowDataService.InstanceID = instanceID;
        } // else

        return workflowDataService;
    } // lock
}
```

```
{
    // Save instance of the workflow runtime.
    _workflowRuntime = workflowRuntime;
} // if

// If we're just starting, plug in ExternalDataExchange
// service.
if (_dataExchangeService == null)
{
    // Data exchange service not registered, so create an
    // instance and register.
    _dataExchangeService =
                    new ExternalDataExchangeService();
    _workflowRuntime.AddService(_dataExchangeService);
} // if

// Check to see if we have already added this data exchange
// service.
MVDataConnector dataConnector =
        (MVDataConnector)workflowRuntime.
          GetService(typeof(MVDataConnector));
if (dataConnector == null)
{
    // First time through, so create the connector and
    // register as a service with the workflow runtime.
    _dataConnector = new MVDataConnector();
    _dataExchangeService.AddService(_dataConnector);
} // if
else
{
    // Use the retrieved data connector.
    _dataConnector = dataConnector;
} // else

// Pull the service instance we registered with the
// connection object.
WorkflowMVDataService workflowDataService =
                            MVDataConnector.MVDataService;
if (workflowDataService == null)
{
    // First time through, so create the data service and
    // hand it to the connector.
    workflowDataService =
                    new WorkflowMVDataService(instanceID);
    MVDataConnector.MVDataService = workflowDataService;
} // if
else
{
    // The data service is static and already registered
    // with the workflow runtime. The instance ID present
    // when  was registered is invalid for this iteration
    // and must be updated.
    workflowDataService.InstanceID = instanceID;
} // else
```

The application we're building in this chapter sends data from the workflow to the host application, which is to say the data transfer is unidirectional. I did this intentionally because there is more to learn before we have all the knowledge required to understand full bidirectional data transfers. The *Workflow Communication Activity* generator utility we'll use is fully capable of creating activities to send and receive host data. We'll just "throw away" part of its output for this particular application because we don't require it. (In fact, the activity it would generate is malformed because our interface did not specify host-to-workflow communications, which we'll save for Chapter 10.)

With all this in mind, let's execute *wca.exe* and create an activity we can use for sending data to our host application.

Creating the communication activities

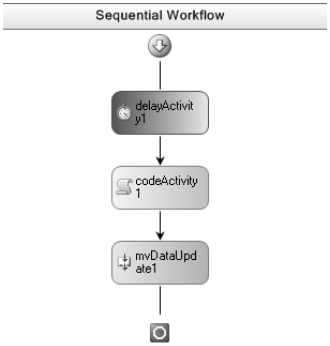
1. So that *wca.exe* can generate the activity code we desire, we need to make sure it has an interface to model the activities against. Therefore, make sure the MVDataService project builds without error and produces the MVDataService assembly (hosted by MVDataService.dll). (Right-click the project in Solution Explorer and select Build. Don't try to build the entire solution.) Correct any build errors before proceeding.
2. Click the Start button and then the Run menu item to activate the Run dialog box.
3. Type **cmd** in the Open combo box control, and click OK. This activates the Windows Command Shell.
4. Change directories so that we can directly access the *MVDataService* assembly in the book's downloaded sample code. Typically, the command to type would be as follows:

   ```
   cd "\Workflow\Chapter8\MVDataChecker\MVDataService\bin\Debug"
   ```

   However, your specific directory might vary. Remember also that the "Workflow" directory represents the actual directory where the code is to be found rather than being an actual directory name in its own right.
5. The *wca.exe* tool was installed by default into the Windows SDK subdirectory under Program Files. (Of course, if you didn't install it into the default directory, you need to use the directory into which you did install the Windows SDK.) To execute the tool, type the following text at the command-line prompt (including the double quotes):

   ```
   "C:\Program Files\Microsoft SDKs\Windows\v6.0\Bin\Wca.exe" MVDataService.dll
   ```

3. Drag the activity to the workflow visual designer's surface, and drop it after the *Code* activity so that it's executed sequentially after the *Code* activity.



4. Our workflow is complete from the workflow visual designer's perspective. It's time now to write a small amount of code to hook things up. So select Workflow1.cs in the MVWorkflow project, and click the View Code toolbar button. Look for the *GenerateMVData* method in the *Workflow1* class. This method is the method *codeActivity1* executes, and here you see the calls to the helper methods *GenerateVehicleTable* and *GenerateViolationTable* to create and fill the *DataSet* to return. (In reality, you would make a call to some external service for driver information, but we're simulating that.) Following the creation of the *DataSet*, we need to add the following lines of code to return the *DataSet* to the host:

```
// Assign the DataSet we just created as the host data
mvDataUpdate1.mvData = ds;
```

With the assignment of the *DataSet* we're returning, we've completed the development of our workflow, as well as the tools we need to ship the *DataSet* to the host application. But what does our host application need to do to receive the data? Let's find out.

Receiving Workflow Data Within the Host Application

Let's now turn to our main application. The basic workflow housekeeping we've seen so far in the book has already been added to the sample simply because we've seen it before and the chapter would expand tenfold if we created the application from scratch. I'd rather concentrate on the workflow aspects we're examining in this chapter. So the workflow factory creates an instance of the workflow runtime, and we even kick off a workflow instance when the Retrieve MV Data button is clicked. What we'll do now is modify the application to use the bridging classes we created in the "Creating External Data Services" section in this chapter.

That's it! The application is now complete. Press F6 to compile, and F5 to execute the application. When you click the Retrieve MV Data button, the selected driver's name is issued to the workflow instance. When the *DataSet* is built, the workflow instance fires the *MVDataUpdate* event. The host application code intercepts that event, retrieves the data, and binds it to the *ListView* controls.

A critical thing to note in the code for that last step is that we called *WorkflowMVDataService*'s static *GetRegisteredWorkflowDataService* method to retrieve the data service containing the *DataSet*. We then used the data service's *Read* method to pull the *DataSet* into our host application's execution environment so that we could perform the data binding.

Invoking External Workflows with *InvokeWorkflow*

Here is a question for you: if you have an executing workflow, can that workflow execute a second workflow?

The answer is yes! There is an activity, *InvokeWorkflow*, that's used to start a secondary workflow. Let's briefly take a look at this activity by way of an example. We'll create a new sample console application that starts a workflow that merely writes a message to the console. After writing this message, the workflow instance starts a second workflow instance that also writes a message, graphically showing us that both workflows executed.

Invoking a secondary workflow

1. Although we could build a fancy demonstration application, we'll revert to using a simple console application as we've done in previous chapters. As you've done throughout the book, decide whether you want to follow along using a solution that's completed or whether you want to create the workflow as you go using a started but incomplete application. The completed version of WorkflowInvoker is in the \Workflow\Chapter8\WorkflowInvoker Completed\ directory, while the incomplete version is in the \Workflow\Chapter8\WorkflowInvoker\ directory. Whichever version you choose, simply drag the .sln file onto an executing copy of Visual Studio and it will load the solution for editing.
2. After Visual Studio has loaded the WorkflowInvoker solution for editing, add a new sequential workflow library project to the WorkflowInvoker solution by clicking File, Add, and then New Project from the Visual Studio menu and then choosing the Workflow project type and Sequential Workflow Library template in the Add New Project dialog box. Name it **Workflow1**. Visual Studio adds the new library project and opens the workflow visual designer for editing. Be sure to save the new workflow project in the \Workflow\Chapter8\WorkflowInvoker directory.
3. Next, drag a *Code* activity from the Toolbox and drop it onto the workflow design surface. In its *ExecuteCode* property, type **SayHello** and press Enter.

13. Next add the code to create and start the instance. Locate this line of code in Program.cs:

```
Console.WriteLine("Waiting for workflow completion.");
```

14. Add this code following the line of code you just located:

```
// Create the workflow instance.
WorkflowInstance instance =
   workflowRuntime.CreateWorkflow(typeof(Workflow1.Workflow1));

// Start the workflow instance.
instance.Start();
```

15. We'll now add a small amount of code to the host application simply to tell us when each workflow completes. Insert the following code in the event handler for *WorkflowCompleted*:

```
if (e.WorkflowDefinition is Workflow1.Workflow1)
    Console.WriteLine("Workflow 1 completed.");
else
    Console.WriteLine("Workflow 2 completed.");
waitHandle.Set();
```

The first workflow to complete sets the *AutoResetEvent* we're using to force the application to wait for workflow completion. We could add code to force the application to wait for both workflows, but for demonstration purposes this should suffice. If you compile and execute the WorkflowInvoker application, you'll see console output similar to what you see in Figure 8-4. If the output messages appear in a slightly different order, don't be alarmed. This is the nature of multithreaded programming...

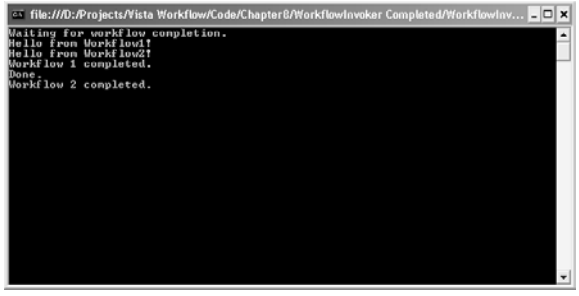


Figure 8-4 The WorkflowInvoker application console output

If you want to continue to the next chapter, keep Visual Studio 2005 running and turn to Chapter 9, "Logic Flow Activities." It's one thing to crunch numbers, but we also need tools to make decisions, and that's the next topic.

If you want to stop, exit Visual Studio 2005 now, save your spot in the book, and watch your favorite movie on DVD. Be sure to fast-forward through any boring parts.

Chapter 9

Logic Flow Activities

After completing this chapter, you will be able to:

- Explain how to execute conditional expressions using the *IfElse* activity
- Show how the *While* activity can be used to execute loops
- Understand how the *Replicator* activity simulates a *for* loop, as well as how it's used

We're starting to piece together some of the critical components we'll need to build real-world workflows. We've seen how to execute code, both within and outside our workflow instances, and we know how to handle exceptions, suspend processing, and even terminate our workflow if things get out of hand. But certainly a major component for any computational system is the ability to make decisions based on runtime conditions. In this chapter, we begin to address workflow activities that require us to tackle if/else scenarios as well as basic looping.

Conditions and Condition Processing

By now, it probably won't surprise you to find that Windows Workflow Foundation (WF) provides activities for logical process control flow based on runtime conditions. After all, if WF provides activities to both raise and catch exceptions, why not have activities to ask questions regarding executing workflow conditions and make decisions based on those findings?

The activities we'll examine in this chapter include the *IfElse* activity, the *While* activity, and the *Replicator* activity. The *IfElse* activity is designed to test a condition and execute a different workflow path depending on the result of the test. (We actually used this activity in Chapter 1, "Introducing Microsoft Windows Workflow Foundation," when we asked whether or not a given postal code was valid when tested against a regular expression.) The *While* activity, perhaps not too surprisingly, is used to perform a *while* loop. A *for* loop, however, is accomplished using something known as the *Replicator* activity. Let's start by looking at this chapter's sample application.

Note The conditional processing you'll do in this chapter is based on the *CodeCondition*, which means you'll write C# code to process the conditional expression. In Chapter 12, "Policy And Rules," you'll use the *RuleCondition* which uses WF rules-based processing for conditional expression evaluation. Both are equally valid. I simply chose to include *RuleCondition*, in the same chapter I discuss rules-based processing in general.

The Questioner Application

This chapter's sample application is a Windows Forms application that asks you three questions, the text for which you can modify. (The question text is stored in the application's settings property bag.) You can also indicate whether the questions are dependent or independent. You'll pass the questions and dependency status into the workflow as it begins execution.

Dependent questions will continue to be asked only if the previous questions were answered in the affirmative. For example, if you're asked, "Have you seen the document in question?" and you have not, it makes little sense to ask, "Do you approve this document?" If the questions are dependent, the first negative response returns negative for the given response as well as for all remaining question responses.

Independent questions will always be asked regardless of preceding responses. The question, "Do you like ice cream?" is unrelated to "Is it raining outside at this time?" Whether you do or do not like ice cream, the answer to that question is independent of the weather outside. Independent questions continue to be asked whether you provide a negative response to an earlier question or not.

The user interface appears as you see in Figure 9-1. If you modify the text for any of the three questions, the new question text will automatically be stored in your application settings property bag. (The same is true of the question type.) The questions are intended to generate yes/no responses so that the workflow can pass the responses back to the host application as an array of Boolean values.

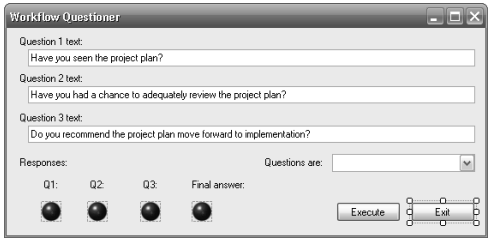
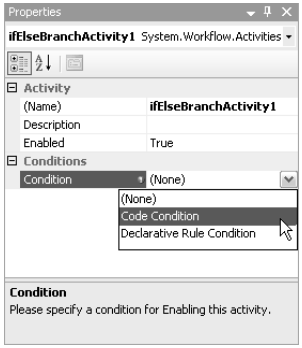


Figure 9-1 The Questioner primary user interface

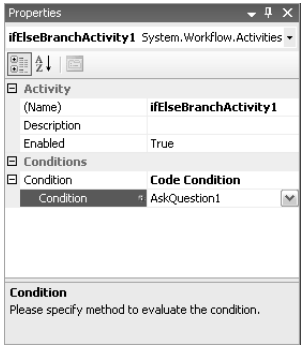
When you click the Execute button, the questions appear in order as message boxes with Yes and No buttons. Once the workflow has processed all the questions, it returns a Boolean array to the host application. The host application will examine the array for user-interface display purposes.

While the workflow is executing, the responses appear as blue balls (as you see in Figure 9-1). When the workflow task has completed, affirmative responses are shown as green balls and negative responses are shown as red balls. If all responses were affirmative, the "final answer" image appears as a green ball. However, if any of the three questions resulted in a negative response, the final answer appears as an "8 ball." You can see the application in action in Figure 9-2.



Note You actually have two choices for conditional expressions: code and rules-based. We'll use the code-based conditional expression here, saving the rules-based technique for Chapter 12, "Policy Activities."

5. Expand the resulting *Condition* property, type in the value **AskQuestion1**, and press Enter. Visual Studio inserts the *AskQuestion1* event handler for you and switches to code view. For now, return to the workflow visual designer so that you can drag more activities into your workflow.



6. With the Visual Studio workflow visual designer active, drag a *CodeActivity* onto the designer's surface and drop it into the right-hand branch of *IfElseActivity1*.

13. Now scan down the code file until you find the *AskQuestion1* event handler Visual Studio added for you. To this event handler, add the following lines of code:

```
// Ask the question!
DialogResult result = MessageBox.Show(Questions[0], "Questioner:",
                MessageBoxButtons.YesNo, MessageBoxIcon.Question);
e.Result = (result == DialogResult.Yes);
```

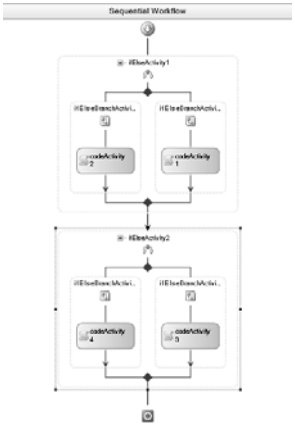
14. To the *NegateQ1* event handler, add this code:

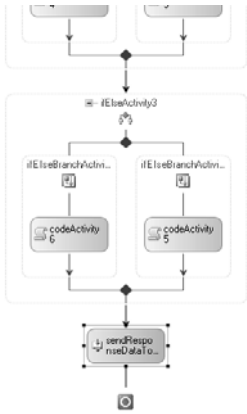
```
// Negate answer.
_response[0] = false;

if (Dependent)
{
    // Negate remaining answers.
    _response[1] = false;
    _response[2] = false;
}
```

15. Next, locate the *AffirmQ1* event handler and add this code:

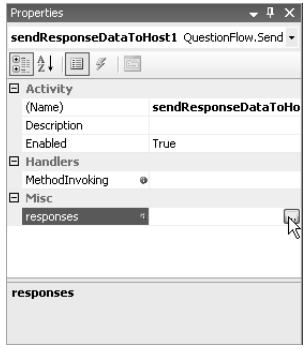
```
// Affirm answer.
_response[0] = true;
```

16. You have just added the workflow components designed to ask the first question. However, two more questions remain. For the second question, repeat steps 3 through 8 to add the *IfElse* activity to the workflow, substituting references to question 1 with references to question 2. Doing this inserts the event handlers *AskQuestion2*, *NegateQ2*, and *AffirmQ2*. The workflow visual designer will appear as follows:

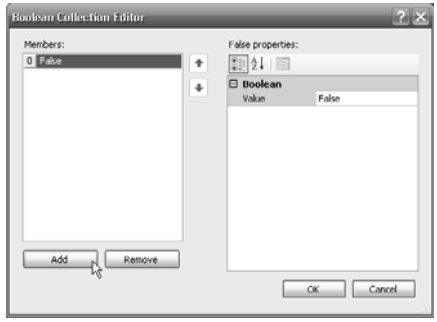




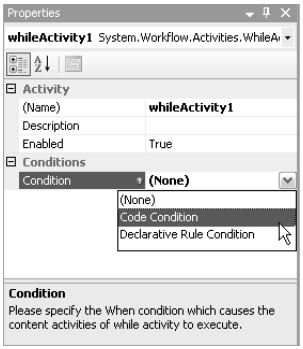
26. Because the data to be returned is simply an array of value types (Boolean values), the process is slightly different than in the previous chapter. Instead of adding a dependency property to contain the Boolean array, the *SendResponseDataToHost* activity contains the data as a field. The user interface to create the field differs from the user interface you saw in Chapter 7 (in step 8 of the "Creating a workflow using the *Throw* activity" procedure). Select the *responses* property in the Visual Studio Properties pane, and click the browse (...) button.



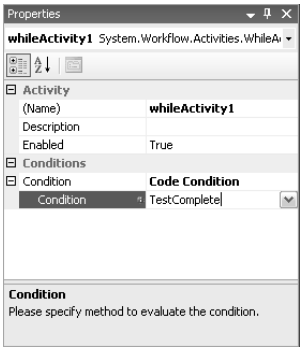
This activates the Boolean Collection Editor dialog box.



4. In a manner similar to the *IfElse* activity, select the *Condition* property for the *whileActivity1* activity to activate its drop-down list. From the drop-down list, select Code Condition.



5. Expand the *Condition* property, type **TestComplete**, and press Enter to add the *TestComplete* event handler to your workflow code. After Visual Studio inserts the event handler and switches the user interface to the code editor, return to the workflow visual designer.



6. With the workflow visual designer active, drag an instance of *CodeActivity* and drop it in the center of *whileActivity1*. Assign the value **AskQuestion** to the *ExecuteCode* property, and return to the workflow visual editor when the *AskQuestion* event handler has been added.

If there is a workflow-equivalent *while* loop, could there also be a workflow-equivalent *for* loop? In fact, there is. It's the *Replicator* activity, and it happens to be the next topic of discussion.

Using the *Replicator* Activity

It would be incorrect to say that the *Replicator* activity is equivalent to a *for* loop in C# processing terms. The C# Language Specification 1.2 tells us the *for* loop in C# looks like the following:

```
for ( for-initializer ; for-condition ; for-iterator ) embedded-statement
```

embedded-statement is executed until the *for-condition* evaluates to *true* (if omitted, it's assumed to be *true*), beginning with *for-initializer* and executing *for-iterator* for every iteration. There is *nothing* mentioned regarding replication in any of the C# *for* statement components. With replication, we envision a cookie-cutter software factory that stamps out exact replicas of the original code. C# *for* loops don't operate in this fashion.

In fact, the cookie-cutter concept isn't terribly far off the mark when looking at WF's equivalent *for* loop activity. If you're familiar with ASP.NET, you might have used the *Repeater* control (a favorite of mine). The ASP.NET *Repeater* control accepts an item template (and alternatively, an alternating item template) and replicates it as many times as required, depending on the number of items in the data object to which it is bound.

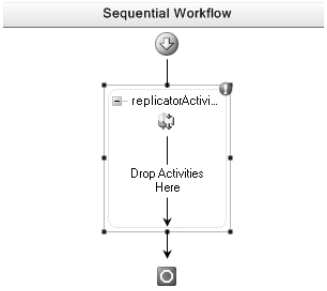
The *Replicator* activity is similar to the ASP.NET *Repeater* control in that it binds to an *IList*-based data source and replicates its embedded (single) child activity, with one child activity instance per element in the *IList*-based data source. Yet the *Replicator* activity is similar to a C# *for* statement in some respects because it allows a loop initialization event (similar to *for-initializer*), a loop completion event (such as when *for-iterator* is compared with *for-condition*), and a loop continuation event (similar to *for-condition*). It provides events to indicate the creation of a replicated (*embedded-statement*) child activity, so that you can individualize the data binding, and it fires an event for child activity completion so that you can perform any cleanup or housekeeping tasks on a per–child activity instance basis.

In a nutshell, the *Replicator* activity accepts—and requires—a single child activity, which can be a container activity (such as the *Sequence* activity), and it fires an initialization event to kick things off. During the initialization event, you can bind an *IList*-based collection to the *Replicator* activity's *InitialChildData* property.

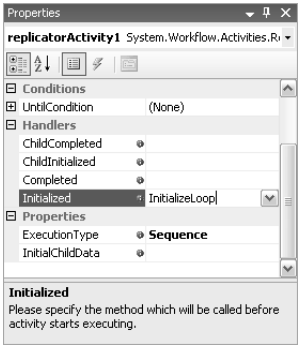
The *Replicator* activity then replicates the child activity you provided to match the number of items in your *IList*-based collection. These child activity instances can then be executed sequentially or in parallel (by setting the *ExecutionType* property). The *UntilCondition* event fires before each child activity is executed, and you tell the *Replicator* activity to continue

2. As with both preceding sections, the application is once again essentially complete so that you can concentrate on the workflow aspects. Select the Workflow1.cs file in Solution Explorer's tree control, and click the View Designer toolbar button to load it into the Visual Studio workflow visual designer.

3. When *Workflow1* is ready for editing in the workflow visual designer, drag an instance of *Replicator* from the Toolbox to the designer's surface and drop it. This, of course, inserts an instance of the *Replicator* activity into your workflow.

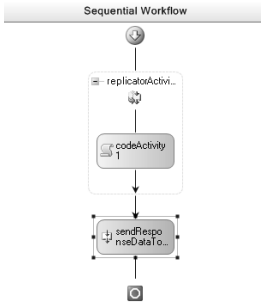


4. In the Visual Studio Properties pane, select the *Initialized* property and enter **InitializeLoop**. Visual Studio inserts the event handler in your code and shifts you to the code editor. Return to the workflow visual designer so that you can continue setting properties.



5. For the *Completed* property, enter **LoopCompleted** to add the *LoopCompleted* event handler to your workflow code. Again return to the workflow visual designer.

11. For the final task in the workflow visual designer, drag an instance of *SendResponseDataToHost* onto the designer's surface and drop it below *replicatorActivity1*. Follow steps 24 through 30 in the "Creating the QuestionFlow workflow using the *IfElse* activity" procedure shown earlier to properly configure this activity. (As before, you might need to compile the application for the *SendResponseDataToHost* activity to show up in the Toolbox.)



12. At this point the code file for *Workflow1* should be open for editing. If not, select it in Solution Explorer and click the View Code toolbar button.

13. Because we modified the various *replicatorActivity1* properties, Visual Studio added several event handlers. It's time to complete those as well as provide the other supporting code the workflow will require. To start, follow steps 9 through 12 in the "Creating the QuestionFlow workflow using the *IfElse* Activity" procedure to add the basic properties the workflow will require to begin processing questions.

14. The *Replicator* activity requires an *IList*-based collection of items on which to base the replication of its child activity. We have an array of questions we can use because the basic array type is based on *IList*. However, if we simply hand the replicated activity the question text, how will we return the result? There is no direct tie between the question text and the question number. Without that, we can't assign the Boolean return value within the returned array. Therefore, we'll change the rules slightly and create a new array—an array of integers that represent element offsets—into the question text array. It's this integer array we'll hand to the *Replicator* activity. The replicated child activity, then, will have access to the question number it is to ask, giving it an index into both the question text array and the Boolean response array. To do this, add the following code after the declaration of the *_response* array:

```
private Int32[] _qNums = null;
```

```
                // Continue processing.
                e.Result = false;
            } // else
        } // else
    } // if
```

22. Somewhere along the way, you might expect the workflow would finally ask a question, and the time is now for that code to be inserted. Scan the code file to find the *AskQuestion* method Visual Studio added for you, and add this code:

```
// Ask the question!
DialogResult result = MessageBox.Show(Questions[_currentQuestion],
  "Questioner:", MessageBoxButtons.YesNo, MessageBoxIcon.Question);
_currentQuestionResponse = (result == DialogResult.Yes);
```

23. Compile the entire solution by pressing F6, and correct any errors that crop up.

If you press F5 to run the application and compare its behavior to the previous iterations, you should find that it functions exactly the same as the two earlier versions, at least at the user-interface level.

If you want to continue to the next chapter, keep Visual Studio 2005 running and turn to Chapter 10, "Event Activities." Pat yourself on the back...you're just about halfway through the book!

If you want to stop, exit Visual Studio 2005 now, save your spot in the book, and close it. The next chapter should be a good one...we'll deal with events and event handling.

Chapter 9 Quick Reference

To	Do This
Process if-then-else conditional branching scenarios	Drop an instance of the *IfElse* activity into your workflow process, and assign a condition event handler as well as "true" and "false" branch activities.
Process workflow activities while a given condition is true	Consider the *While* activity, and if it's appropriate, drop it into your workflow. Be sure to assign a conditional handler. Remember that assigning the *ConditionalEventArgs Result* property a *false* value exits the loop.
Replicate activities to be processed in a for-next scenario	Insert an instance of the *Replicator* activity into your workflow, keeping in mind that its single child activity can be a container activity (allowing you to process more than one activity in the looping construct). Note that you must provide an *IList*-based collection of items for the *Replicator* activity to use to replicate the child activity. There will be one child activity instance for each item in the *IList*-based collection.

Chapter 10

Event Activities

After completing this chapter, you will be able to:

- Create specific event handlers using the *HandleExternalEvent* activity
- Add delays to your workflow using the *Delay* activity
- Incorporate event-driven activities into your workflow using the *EventDriven* activity
- Use the *Listen* activity to gather event handlers
- Understand how the *EventHandlingScope* activity allows for concurrent activity execution while listening for events

With Chapter 8, "Calling External Methods and Workflows," you saw how a workflow communicates with its host application using the *CallExternalMethod* activity. When the workflow calls an external method, using a local communication service you provide, the host application receives an event. The host then processes the data and takes any appropriate actions.

The converse process involves the host application raising events to be handled by the workflow (although workflow event handling can be used for a far wider array of tasks than just host communication). In Chapter 8, I mentioned we'd revisit host/workflow communication after describing the activities that workflows used to handle events, and in this chapter we'll do just that.

Unlike other chapters so far, where I describe an individual workflow activity and then provide a small application designed to show that activity in action, this chapter will describe multiple activities and then present a single sample application. Why? Because the activities I describe here are all related and depend on one another. I can't show one and not show the others. The *Listen* activity is a container in which you find *EventDriven* activities. Inside an *EventDriven* activity, you'd expect to find a single *HandleExternalEvent* activity. And so forth. So I'll describe the activities themselves but build a single application toward the end of the chapter. The "Host to Workflow" section should tie it all together. Let's start with the workhorse *HandleExternalEvent* activity.

Using the *HandleExternalEvent* Activity

No matter where in your workflow you handle an event, and no matter in what composite activity your workflow execution finds itself when it's active and executing, when an event comes your workflow's way the *HandleExternalEvent* activity is the workflow activity that ultimately deals with the event. To me, of all the powerful features .NET itself brings to the

Table 10-2 Often-Used *HandleExternalEvent* Activity Method

Method	Purpose
OnInvoked	This protected method is useful for binding values found in the event arguments to fields or dependency properties within your workflow. Overriding this method (or handling the event that it fires) is your primary mechanism for retrieving the data from the event argument as the data comes in from the host. Generally, you create a custom event argument with the data embedded in the argument object itself.

In practice, although you can use the *HandleExternalEvent* activity directly from the Microsoft Visual Studio Toolbox, it's more common to use the *wca.exe* tool you saw in Chapter 8 to create classes derived from *HandleExternalEvent* that are customized for the communications interface you are using. For example, if in your interface you defined an event named *SendDataToHost*, *wca.exe* would create a new activity called *SendDataToHost* (derived from *HandleExternalEvent*) and assign for you the *EventName* and *InterfaceType*, as well as assign data bindings as specified by the event arguments you create to use with your *SendDataToHost* event. I'll provide an example of this later in the chapter.

Using *HandleExternalEvent* is easy to do. Simply place an instance of this activity in your workflow, assign the interface and event name, provide an event handler for the *Invoked* event if you'd like, and then execute your workflow. If you use *wca.exe*, you are provided with activities derived from *HandleExternalEvent* that you can drop directly into your workflow, adding bindings in the Properties window to bind the data in the event arguments with locally defined fields or dependency properties.

With a *HandleExternalEvent* activity in your workflow, all processing through the sequential flow stops while waiting for the event. In a sense, placing this activity in your workflow acts like an *AutoResetEvent* in .NET Framework programming terms. Unlike *AutoResetEvent*, the processing thread doesn't suspend (workflow queue processing is what is suspended), but it controls the flow of your workflow in much the same way *AutoResetEvent* holds processing until the event is set. It's like a door or gate that allows workflow processing to continue through its sequential path only if the event is triggered.

Using the *Delay* Activity

We've seen and used the *Delay* activity a few times so far in the book, but I've saved a more formal description until this point. Why? As it happens, the *Delay* activity implements the *IEventActivity* interface. Because of this, it's also classified as a Windows Workflow Foundation (WF) event-based activity.

provide that is exposed by a local communications service you write. The service then takes the information destined for the host and fires an event. The event signals the availability of data, and the host can take measures to read the data from the service (which cached the data after receiving it from the workflow).

The reverse process, where the host sends data to an already-executing workflow, also involves the local communication service as well as events and the handlers responsible for dealing with those events. When you design the interface to be used for communication between host and workflow (as shown in the "Creating Service Interfaces" section in Chapter 8), the methods you add to the interface are for the workflow to use to send data to the host. Adding events to your interface allows the host to send data to the workflow *after* it has begun executing.

The sample application for this chapter will use each of the activities I've described. An *EventHandlingScope* activity will handle a "stop processing" event. While waiting for that event, a *Sequence* activity will contain a workflow process that simulates updating stock-market values. As quotes are updated, the new values are passed to the host for inclusion in the user interface (shown in Figure 10-1). The application, eBroker, doesn't actually check the current stock value for each stock "ticker symbol," which is the three- to four-character nickname representing the company that issued the stock. It calculates new values using a simple *Monte Carlo simulation*. A Monte Carlo simulation is a simulation using random numbers, similar to rolling dice to decide an outcome. The intention is to see how workflows and hosts communicate. Although it's not a minor detail, actually checking current stock-market values is a detail I'll omit for this sample.

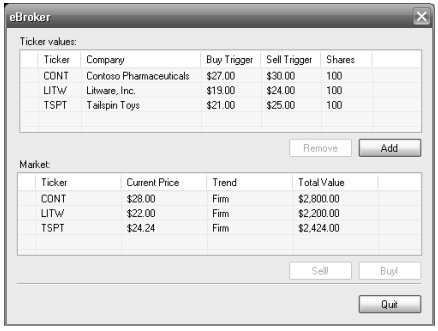


Figure 10-1 The eBroker primary user interface

The eBroker application is able to let the workflow know that new stocks are to be checked or that existing stocks should be removed from consideration. In this case, you can add or remove stock values from the simulation using the Add and Remove buttons. Clicking Add pops up the dialog box you see in Figure 10-2. When you complete the dialog box and click OK, the new watched stock is added to the watched stock list.

Project/Solution. Using the resulting Open Project dialog box, browse your computer's file system until you find this chapter's sample and open its solution file.

> **Note** As with the most sample applications in this book, the eBroker sample application comes in two forms: incomplete and complete. You can follow along and add code to the incomplete version, or you can open the complete version and verify that the code I mention here is in place.

2. You will find that three projects have been added to the solution. In Visual Studio Solution Explorer, expand the eBrokerService project and open the IWFBroker.cs file for editing.

3. Locate the namespace definition. After the opening brace for the *eBrokerService* namespace, add this code and then save the file:

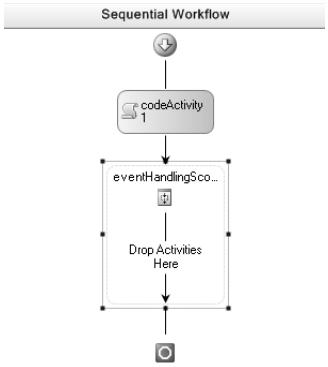
```
[ExternalDataExchange]
public interface IWFBroker
{
    void MarketUpdate(string xmlMarketValues);

    event EventHandler<TickerActionEventArgs> AddTicker;
    event EventHandler<TickerActionEventArgs> RemoveTicker;
    event EventHandler<SharesActionEventArgs> BuyStock;
    event EventHandler<SharesActionEventArgs> SellStock;
    event EventHandler<StopActionEventArgs> Stop;
}
```

4. Compile the project by pressing Shift+F6 or by selecting Build eBrokerService from the main Visual Studio Build menu. Correct compilation errors, if any.

Don't forget the *ExternalDataExchange* attribute. Without it you cannot successfully transfer information between workflow and host using the data transfer mechanism I describe here.

Before you create the communication activities (using *wca.exe*), take a moment to open and glance through the event arguments you see in the eBrokerService project. *MarketUpdateEventArgs* is really no more than a strongly typed version of *System.Workflow.ExternalDataEventArgs*, as is *StopActionEventArgs*. These event argument classes convey no data. However, *TickerActionEventArgs* and *SharesActionEventArgs* both convey information to the workflow. *TickerActionEventArgs* carries XML representing the stock to add or remove, while *SharesActionEventArgs* carries the ticker symbol as a primary key, as well as the number of shares to buy or sell.

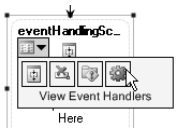
> **Tip** Designing the event arguments is important because the event arguments carry data from the host to the workflow. Moreover, *wca.exe* examines the event arguments and builds bindings into the derived classes that allow you to access the data from the event arguments as if the data were intrinsic to the derived activity. Put another way, if the event argument has a property named *OrderNumber*, the class that *wca.exe* builds would have a property named *OrderNumber*. Its value would come from the underlying event's event argument and would be assigned automatically for you.



4. Remember that you need to provide an event handler as well as a child activity for *EventHandlingScope* to execute while it waits for the event. Let's set up the event handler first. To access the event handlers, move the mouse pointer to the tiny rectangular icon below the first *e* in *eventHandlingScope1*. (The rectangle is known as a "Smart Tag.")

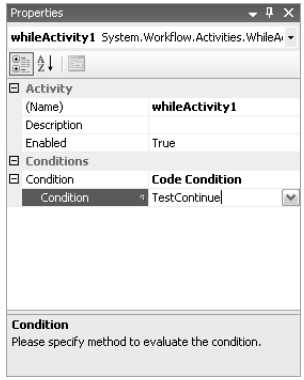
The Smart Tag is transformed into a larger, darker rectangle with a down arrow.

Click the down arrow to activate a small window with four icons: View EventHandling-Scope, View Cancel Handler, View Fault Handlers, and View Event Handlers.

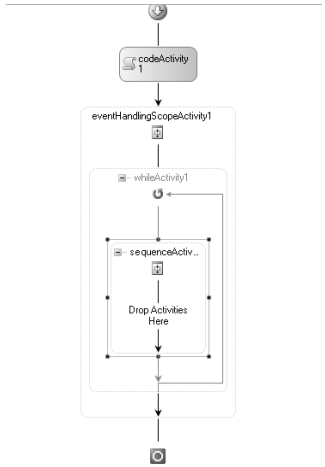


Click the rightmost icon to activate the Event Handlers view. The user interface you see is a lot like the user interface associated with fault handlers that you saw in Chapter 7, "Basic Activity Operations."

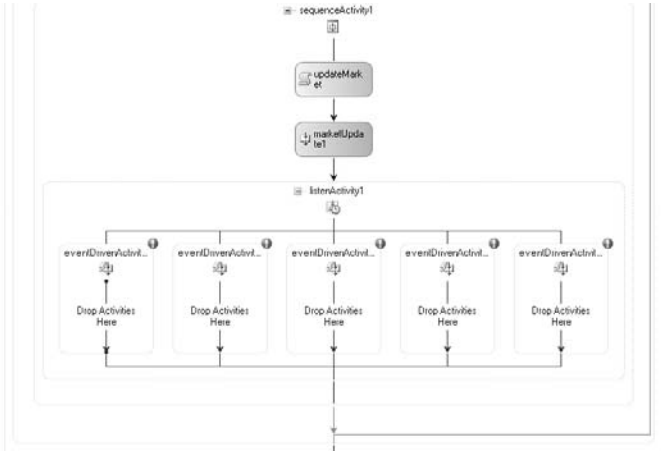
adds the *TestContinue* event handler, return to the visual workflow designer to add more activities.



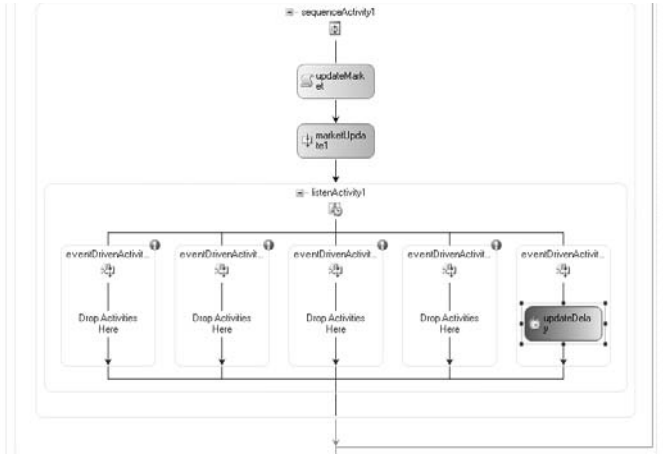
9. The *While* activity accepts only a single child activity, so drop an instance of the *Sequence* activity into the *While* activity you just placed in your workflow.



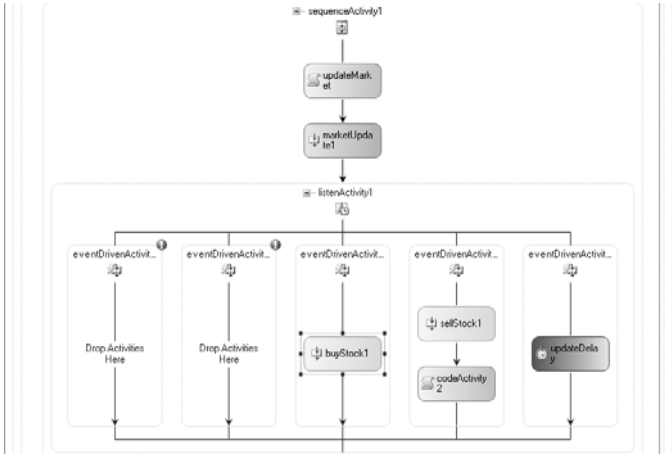
10. You need a *Code* activity at this point to perform the Monte Carlo stock-value simulation, so drag an instance of *Code* onto the designer's surface and drop it into the *Sequence* activity you added in the preceding step. Use the Properties window to rename it **updateMarket.**



16. Into the rightmost *EventDriven* activity, drag and drop a *Delay* activity. Name it **updateDelay** using the Properties pane.

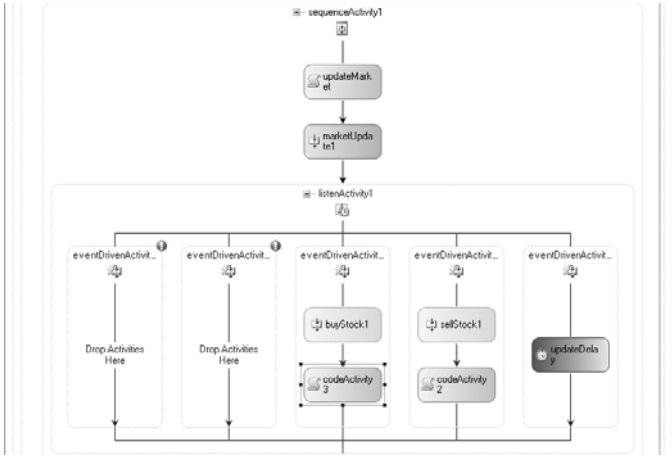


17. Next drag an instance of *SellStock*, from eBrokerFlow, onto the designer's surface, and drop it into the next rightmost *EventDriven* activity.



22. Bind the *BuyStock* activity's *NumberOfShares* property to a new field, *_sharesToBuy*, using the method outlined in step 18. Bind its *Symbol* property to a new field, *_tickerToBuy*, as you did in step 19.

23. Just as you needed a *Code* activity to sell stock, so will you need a *Code* activity to buy stock. Repeat step 20 and add a new *Code* activity, naming its *ExecuteCode* method **BuyStock**.



24. Repeat steps 17 through 20 two more times, adding the *RemoveTicker* and *AddTicker* events to the *Listen* activity. The *RemoveTicker* activity should have its *TickerXML* property bound to a new **_tickerToRemove** field, while the *Code* activity for the *RemoveTicker* event should have its *ExecuteCode* property assigned to be **RemoveTicker**. Similarly, *AddTicker* should have its *TickerXML* property bound to **_tickerToAdd**, with the

```
{
    // Assign a price...
    decimal delta = (item.SellTrigger - item.BuyTrigger) / 2.0m;

    // The last price must be a positive value, so add
    // the delta to the smaller value.
    if (delta >= 0.0m)
    {
        // Add delta to buy trigger value
        item.LastPrice = item.BuyTrigger + delta;
    } // if
    else
    {
        // Reverse it and add to the sell trigger
        // value
        item.LastPrice = item.SellTrigger + delta;
    } // else
} // if

// Set up the simulation
decimal newPrice = item.LastPrice;
decimal onePercent = item.LastPrice * 0.1m;
Int32 multiplier = 0; // no change

// We'll now roll some dice. First roll: does the
// market value change? 0-79, no. 80-99, yes.
if (rand.Next(0, 99) >= 80)
{
    // Yes, update the price. Next roll: will the
    // value increase or decrease? 0-49, increase.
    // 50-99, decrease
    multiplier = 1;
    if (rand.Next(0, 99) >= 50)
    {
        // Decrease the price.
        multiplier = -1;
    } // if

    // Next roll, by how much? We'll calculate it
    // as a percentage of the current share value.
    // 0-74, .1% change. 75-89, .2% change. 90-97,
    // .3% change. And 98-99, .4% change.
    Int32 roll = rand.Next(0, 99);
    if (roll < 75)
    {
        // 1% change
        newPrice = item.LastPrice + (onePercent * multiplier * 0.1m);
    } // if
    else if (roll < 90)
    {
        // 2% change
        newPrice = item.LastPrice + (onePercent * multiplier * 0.2m);
    } // else if
    else if (roll < 98)
    {
```

```
            _items.Remove(ticker.Symbol);
        } // if
    } // try
    catch
    {
        // Do nothing...we just won't have removed it.
    } // catch
```

34. Finally, modify *AddTicker* by inserting this code:

```
try
{
    // Deserialize
    eBrokerService.Ticker ticker = null;
    using (StringReader rdr = new StringReader(_tickerToAdd))
    {
        XmlSerializer serializer =
           new XmlSerializer(typeof(eBrokerService.Ticker));
        ticker = (eBrokerService.Ticker)serializer.Deserialize(rdr);
    } // using

    // Add the item if not already existing.
    if (!_items.ContainsKey(ticker.Symbol))
    {
        // Add it.
        _items.Add(ticker.Symbol, ticker);
    } // if
} // try
catch
{
    // Do nothing...we just won't have added it.
} // catch
```

35. If you press Shift+F6, the workflow project should compile without error.

With the workflow complete, we now need to turn our attention to the local communication service and host integration. Because we covered both of these topics in some detail in Chapter 8, I won't revisit them in their entirety here. If you open the associated files for this example, you will see code similar to what you saw in Chapter 8.

Note I mentioned the following in Chapter 8, but it's an important issue and your awareness of this issue should be reinforced: If you share objects or collections of objects between workflow and host application, you run the risk of introducing multithreaded data access problems since the workflow and host application will share references to the same objects. If this is an issue for your application, consider cloning the objects as they're moved between workflow and host (by implementing *ICloneable* within your data class), or use serialization techniques. For this application, I chose XML serialization.

However, I would like to touch on some of the code in the connector class, *BrokerDataConnector*. The *IWFBroker* interface is different from the sample interface we saw in Chapter 8 because of the events *IWFBroker* contains. Because the connector class must implement the interface (*BrokerDataConnector* implements *IWFBroker*, in this case), the connector must also deal with the

```
        public void RaiseSellStock(Guid instanceID,
                                   string symbol,
                                   Int32 numShares)
        {
            if (SellStock != null)
            {
                // Fire event
                SellStock(null,
                   new SharesActionEventArgs(instanceID,
                                             symbol,
                                             numShares));
            } // if
        }

        public void RaiseStop(Guid instanceID)
        {
            if (Stop != null)
            {
                // Fire event
                Stop(null, new StopActionEventArgs(instanceID));
            } // if
        }
    }
}
```

The workflow executes the connector's *MarketUpdate* method, while the host executes the "raise" methods to fire the various events based on user inputs. Chapter 8 describes the mechanism the workflow uses to invoke the *MarketUpdate* method. To see the host invoke an event designed to ripple down to the workflow—which might or might not carry data in the event arguments—look at this code snippet. This code is used by the Quit button to exit the application.

```
private void cmdQuit_Click(object sender, EventArgs e)
{
    // Stop the processing
    // Remove from workflow
    eBrokerService.BrokerDataConnector dataConnector =
        (eBrokerService.BrokerDataConnector)_workflowRuntime.GetService(
        typeof(eBrokerService.BrokerDataConnector));
    dataConnector.RaiseStop(_workflowInstance.InstanceId);

    // Just quit...
    Application.Exit();
}
```

To fire the events that carry data to the workflow, you first retrieve the connector using the workflow runtime's *GetService* method. Note the service is cast to its appropriate type so that the "raise" methods are known and available. Once the service is retrieved, you simply call the appropriate "raise" method, sending in the information necessary to create the appropriate event arguments.

Chapter 11

Parallel Activities

After completing this chapter, you will be able to:

- Understand how *Parallel* activities execute in the workflow environment, and know how they're used
- Synchronize data access and critical code sections within parallel execution paths
- Use the *ConditionedActivityGroup* activity to execute activities in parallel based on conditional expressions that are evaluated before each parallel execution

Up until this point in the book, we've dealt exclusively with sequential processes. Activity A executes and transfers execution context to Activity B, and so forth. We've not looked at parallel execution paths and the complexities that typically come with that. In this chapter, we'll look at parallel activity processing and see how to synchronize access to shared information across parallel execution paths.

Using the *Parallel* Activity

Whenever I go to the grocery store for something I've run out of, it seems like there is usually only one checkout line operating. All the customers have to pass through this single line to pay for their goods. Of course, there are 14 other cash registers in this store, but nobody is there to staff them. On those rare occasions when two or more checkout lines are open, however, people and groceries move through more quickly because they're processed in parallel.

In a sense, you can do the same thing with workflow activities. There are times when you cannot perform specific activities out of order, or worse, in random order. In those cases, you must select a *Sequence* activity to house your workflow. But at other times, you might be able to lay out workflow processes that can execute at the same time (or at nearly the same time, as we'll see). For those cases, the *Parallel* activity is the choice.

The *Parallel* activity is a composite activity, but it can support *Sequence* activities only as children. (Of course, feel free to place whatever activities into the *Sequence* activities you want.) A minimum of two *Sequence* activities is required.

The child *Sequence* activities do not execute using separate threads, so the *Parallel* activity isn't a multithreaded activity. Instead, the child *Sequence* activities execute on a single thread. Windows Workflow Foundation (WF) processes an individual activity executing in one *Parallel* activity execution path until it completes before switching execution to an activity in the next parallel execution path. That is, as one activity in one branch finishes, another

activity in another branch is scheduled for execution. What is not guaranteed is the order in which parallel activities are actually executed.

The effect of this is that parallel execution paths do not execute concurrently, so they're not truly executing in parallel in the multithreaded sense. (This is called *cooperative multithreading* and was last seen by Windows software developers in Windows 3.11, although it is also popular in many control systems in use today.) However, they are executed *as if* they were operating concurrently, and you should think of them as such. Execution can, and does, switch back and forth between the activities within the parallel paths. Viewing the *Parallel* activity as truly parallel is the wisest course—treat parallel activities as you would treat any multithreaded process.

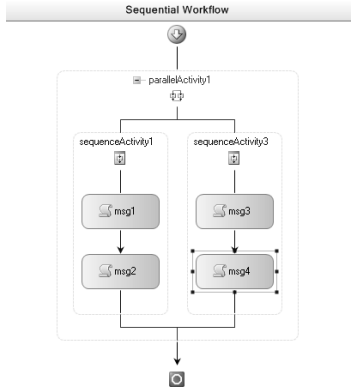
Note If you need to impose order between the parallel execution paths, consider using the *SynchronizationScope* activity. I'll show you how later in this chapter.

A good question to ask at this point is "What about *Delay* activities?" As you know, *Delay* activities stop execution in a sequential workflow for the duration of their *TimeoutDuration*. Does this stop the *Parallel* activity from processing? No. The delay does cause that particular sequential workflow path to be stopped, but the other parallel paths continue processing normally.

Given all the multithreading warnings I've issued, you might think using the *Parallel* activity is challenging. In fact, though, it's very easy to use. It appears a lot like the *Listen* activity (discussed in Chapter 10, "Event Activities") in the visual workflow designer. Instead of *EventDriven* activities, you will find *Sequence* activities; otherwise, the visual representation is similar. Let's create a simple example to show the *Parallel* activity in action.

Creating a new workflow application with parallel execution

1. To quickly demonstrate the *Parallel* activity, this example uses a Windows console-based application. I took the liberty of creating for you two versions of a sample application to experiment with the *Parallel* activity: a completed version and an incomplete version. Both are found in \Workflow\Chapter11\. The ParallelHelloWorld application is incomplete, but it requires only the workflow definition. The ParallelHelloWorld completed version is ready to run. If you'd like to follow the steps I've outlined here, open the incomplete version. If you'd rather follow along but not type in code or drag activities, open the completed version. To open either version, drag the .sln file onto Visual Studio and it will open it for you.
2. After Visual Studio has opened the ParallelHelloWorld solution, look for and open the *Workflow1* workflow for editing in the visual workflow designer. Select Workflow1.cs in Solution Explorer, and click the View Designer button. The visual workflow designer appears, and you can begin adding activities.



11. In the resulting *Message4* event handler, place this code:

```
Console.WriteLine(" jumps over the lazy dog.");
```

12. With the workflow now complete, add a reference to the workflow from the ParallelHelloWorld application. Right-click the ParallelHelloWorld tree control node in Visual Studio's Solution Explorer, and select Add Reference. When the Add Reference dialog box appears, click the Projects tab. Select ParallelFlow from the list and click OK.

13. Open Program.cs in the ParallelHelloWorld project for editing and then look for this line of code:

```
Console.WriteLine("Waiting for workflow completion.");
```

14. To create a workflow instance, add this code following the line of code you just located:

```
// Create the workflow instance.
WorkflowInstance instance =
   workflowRuntime.CreateWorkflow(typeof(ParallelFlow.Workflow1));

// Start the workflow instance.
instance.Start();
```

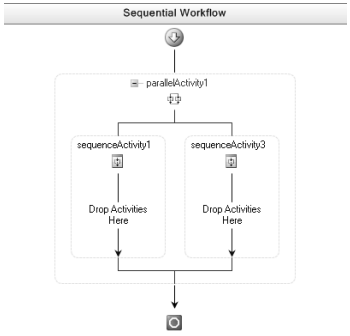
15. Compile the solution by pressing F6. Correct any compilation errors that might be present.

16. Execute the application by pressing F5 (or Ctrl+F5). Note you might have to set a breakpoint in Program.cs (the *Main* method) to see the output.

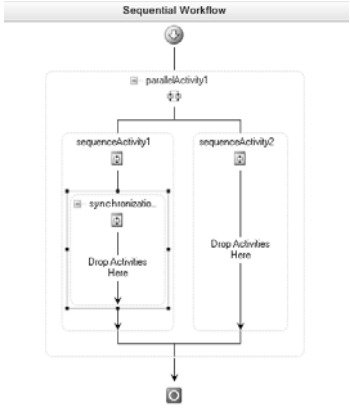
force critical sections of code to run to completion before the execution context switch, but I'll also introduce volatile memory just to show that working as well.

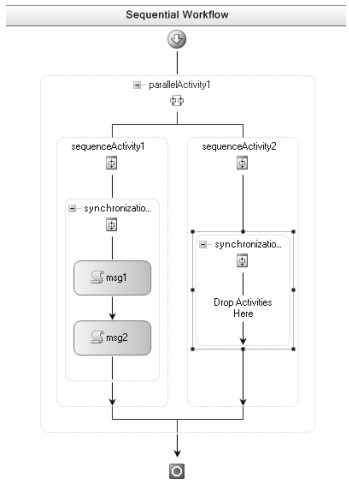
Creating a new workflow application with synchronized parallel execution

1. In this example, you'll again use a Windows console-based application, one that is very similar to the preceding example. As before, I've created both a completed and an incomplete version of the sample, SynchronizedHelloWorld. You'll find both versions in the \Workflow\Chapter11\ directory. If you want to follow along with the book but not edit code, open the completed version, SynchronizedHelloWorld Completed. Or, if you'd rather, open the incomplete version, SynchronizedHelloWorld, and follow the steps as I have them here. To open either solution, drag its respective .sln file onto an executing copy of Visual Studio.
2. After Visual Studio has added the SynchronizedFlow project and opened the *Workflow1* workflow for editing in the visual workflow designer, drag an instance of the *Parallel* activity onto the designer's surface and drop it.



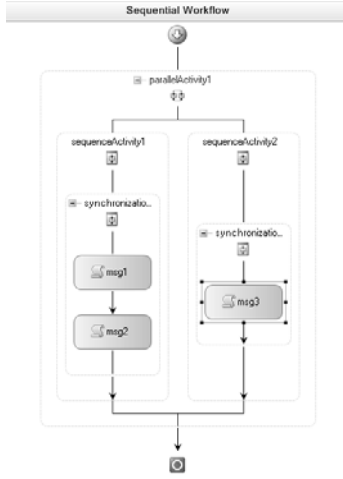
3. Now drag an instance of *SynchronizationScope* onto the designer's surface and drop it into the left *Sequence* activity.





11. So that this *SynchronizationScope* activity synchronizes with the one you inserted in step 4, type **SyncLock** into this *SynchronizationScope* activity's *SynchronizationHandles* property.

12. Now drag a *Code* activity and drop it into the *SynchronizationScope* activity you just inserted. Name it **msg3**, and type **Message3** into *ExecuteCode*.



13. Type this code in the *Message3* event handler:

```
_msg = "The quick brown fox";
PrintMessage();
```

14. To top it off, drag a fourth instance of the *Code* activity and drop it in the *Synchronization-Scope* activity in the right *Sequence* activity . Name it **msg4**, and type **Message4** into the text control for its *ExecuteCode* property.

child activities into this rectangle, their designer images appear in the window below. In Figure 11-1, you see an activity named *setLevel1*, which comes from the sample application you'll build shortly.

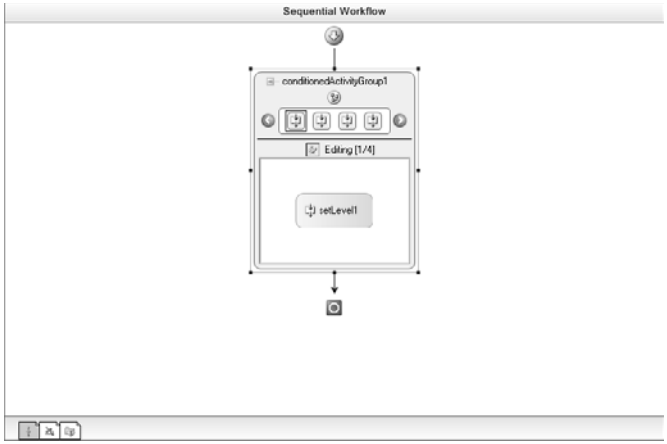


Figure 11-1 The *ConditionedActivityGroup* activity designer user interface

The child activities, when dropped into the CAG, are either in a *preview* mode or in an *editing* mode. Figure 11-1 shows the editing mode. In editing mode, you can set the properties for the child activity, for example to add a *when* condition. When in preview mode, you can view only the child activity's designer image. The properties that appear will be for the CAG itself. To toggle betwen editing and preview mode, click the small square button to the left of the word *Editing* in Figure 11-1, or double-click the child activity in the main CAG window.

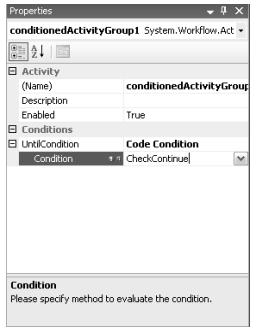
Only individual activities can be placed into the CAG as child activities. If one of the child activities in your workflow process model needs to execute more than one function, such as when handling an event and then executing a *Code* activity in response, you should wrap those in a *Sequence* activity and drop the *Sequence* activity into the CAG as a direct child activity.

When might an activity such as the CAG be used? I think it's one of those activities that you will rarely use, but when it fits your process model, it makes things a lot easier for you.

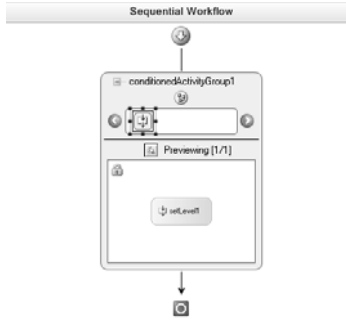
For example, imagine a workflow process that monitors levels of some chemical or material. If when filling the tank you fill it too full, the workflow would automatically release some of the material to an overflow tank. As the tank empties, the workflow monitors the level and sends an alert if the level falls below a specific value. Otherwise, the workflow continues to monitor the tank level but takes no overt action.

Translating this to the CAG, the CAG itself runs until you decide monitoring isn't necessary. One child activity would issue an alert if the tank became too empty, while another would activate the overflow tank if the material level exceeded the specified maximum value. You could do the same thing with nested *IfElse* activities housed in a *While* activity, but the CAG is a

5. Type **CheckContinue** into the *Condition* property, after first clicking the plus sign (+) to expand the *UntilCondition* property. Once Visual Studio adds the event handler and switches you to the code editor, return to the visual workflow designer.



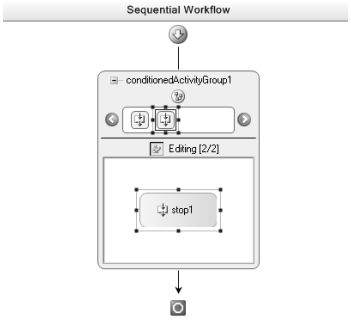
6. Build the solution (press F6) so that the custom activities in the project will appear in the Toolbox. Now drag your first child activity and drop it into the CAG. From the Visual Studio Toolbox, drag a copy of the custom *SetLevel* activity onto the visual workflow designer surface and drop it into the CAG activity, to the right of the < button in the rectangular area. The *setLevel1* activity appears in the main CAG window.



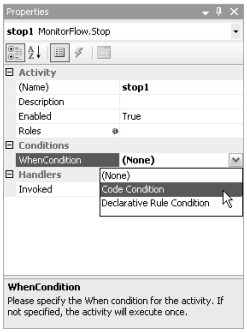
Note The lock icon in the main CAG window indicates that the child activity is in preview mode. (This is also indicated by the text above the main CAG window.) Although you can edit *setLevel1*'s properties by entering edit mode, you can also work with *setLevel1*'s properties when in preview mode by clicking the *setLevel1*'s activity icon in the rectangular window to the right of the < button.

7. Let's enter the CAG's edit mode. Click the tiny square button next to the word *Preview*. This button is a toggle button, so selecting it again directs the CAG to enter preview mode.

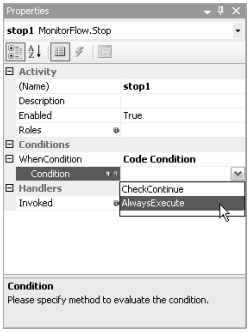
8. With the CAG in edit mode, you can work with a child activity's properties by selecting the activity in the main CAG window (the familiar rounded-corner shaded rectangle



14. The properties for *stop1* are now available for editing in the Properties window. Select its *WhenCondition* to activate the down arrow, and select Code Condition from the selection list.

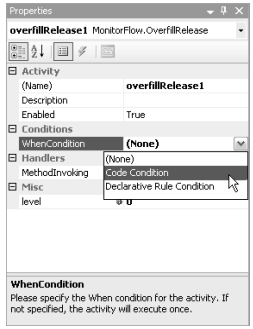


15. Click the + next to *WhenCondition* to show the *Condition* property. In this case, you can share conditional evaluation with the *setLevel1* activity, so select the *Condition* property to enable its down arrow and select *AlwaysExecute* from the list of available conditional evaluation methods.



16. Next select *stop1*'s *Invoked* property, and enter **OnStop** into its text input box. Of course, this adds the *OnStop* method. Simply return to the visual workflow editor once the method is added.

22. Once again, you need to apply a *WhenCondition*, so click *overfillRelease1*'s *WhenCondition* property and select Code Condition from the list.



23. Expand the + next to *WhenCondition* to show the *Condition* property. Into the *Condition* property, enter **CheckOverfill**. This adds the *CheckOverfill* method and switches you to the code editor, so return to the visual workflow designer.

24. Now bind *overfillRelease1*'s *level* property to the *TankLevel* property as you did for *underfillAlert* in step 20. Click the *level* property to activate the browse (...) button. Click the browse button to activate the Bind 'level' To An Activity's Property dialog box. Select *TankLevel* from the list of available properties, and click OK.

25. The workflow is complete from a visual editing perspective, so switch to the code editor for Workflow1.cs so that you can now add code. To do that, select Workflow1.cs in the Solution Explorer window and click the View Code button on the toolbar.

Chapter 11 Quick Reference

To	Do This
Perform workflow activities in parallel	Drop an instance of the *Parallel* activity into your workflow, and place activities in the parallel branches (within the provided *Sequence* activities).
Synchronize workflow activities, either to lock access to volatile memory or to make sure specific activities complete before your workflow sustains an execution context switch	Wrap the activities you want to synchronize within *SynchronizationScope* activities. Provide identical *SynchronizationHandle* values for all *SynchronizationScope* activities that are to be synchronized.
To execute activities in parallel based on conditional evaluations	Consider using a *ConditionedActivityGroup* activity, or CAG. Each activity is evaluated for execution through its *WhenCondition* before each time the parallel paths are executed. The CAG as a whole can also be conditionally executed using its *UntilCondition*.

Chapter 12

Policy and Rules

After completing this chapter, you will be able to:

- Know how policy and rules are handled in workflow processing
- Understand forward chaining and how this affects rules-based workflow processing
- Build rules for workflow processing
- Use rules in conjunction with the *Policy* activity

Most of us, I'm sure, are very comfortable writing *imperative* code. Imperative code is C# code that implements business processes through programmatic constructs—for example, reading a database table, adding the values from some columns in that table, and then writing the total into another database table.

In this chapter, however, we'll dig into *rules*, which are mechanisms for controlling workflow execution but are considered *declarative*. Declarative code, in general, is code you create that isn't compiled into assemblies but rather interpreted as the application executes. Many of the new freatures in ASP.NET 2.0 are declarative, including data binding and improved templated controls. They allow you to write ASP.NET applications without writing C# code to perform data-binding or other complex control rendering tasks.

Windows Workflow Foundation (WF) has a declarative capability as well, though for binding rules and policy instead of data. You don't declare your rules using HTML or ASP.NET constructs, of course, but the concepts involved are similar.

But what is a rule? What is policy?

Policy and Rules

When I write a program that involves data or a process, I'm taking my understanding of the data or process and putting it into code for a computer to execute. For example, consider this logic to handle a checking account: "If the value in the *AvailableBalance* column is less than some requested value, throw a new *OverdraftException*." This seems simple enough...here is some pseudocode that models this behavior:

```
IF (requestedValue > AvailableBalance) THEN
   throw new OverdraftException("Insufficient funds.")
```

But what if the banking customer has overdraft protection by accessing a secondary account if the primary account has insufficient funds? What if the customer doesn't have overdraft

evaluates the condition and then directs workflow execution based on the result of the conditional processing. In a sense, rules are analogous to scripted code, with the rules engine serving as the script execution environment. The advantage to using rules over imperative code is that rules can be easily changed, allowing parts of your business process to more easily adapt to changing conditions.

Policies in WF terms are collections of rules contained in a *RuleSet*. This facilitates something known as *forward chaining*, which is a fancy term for reevaluating rules later based on state changes caused by a rule being processed currently.

Implementing Rules

Rules are based on XML, with that XML being compiled as a resource when your workflow is built in Microsoft Visual Studio. Many WF-based classes understand specific aspects of working with rules, all based in *System.Workflow.Activities.Rules*. These classes work together with the XML to execute the scripted rules, ultimately resulting in a true or false conditional statement your workflow logic uses to direct process flow.

Working with rules in Visual Studio is performed through two main user interfaces. For simple rules editing, such as for conditional evaluation in the flow-based activities (which is discussed in Chapters 9 and 11), you edit the rule using a user interface that allows you to build your rule as text. Within your rule, you combine scripted relational operators (shown in Table 12-1), arithmetic operators (shown in Table 12-2), logical operators (shown in Table 12-3), keywords (shown in Table 12-4), and fields, properties, and methods in your workflow to evaluate the conditional expression for the flow-based activity.

To reference fields or properties in your workflow, you type *this* followed by a dot into the editor. After the dot is typed, a list appears that provides you with fields and properties from your workflow that you can to select to work with. (Of course, you can always type in the field or property name directly.) If the field or property represents a class, you can nest calls into that class by using more dots, as in *this.Customer.Name*.

You can call methods as well, including static methods. To call a static method, type the class name followed by the method name just as you would in imperative code.

Table 12-1 Rule Relational Operators

Operator	Purpose
== *or* =	Tests for equality
> *or* >=	Tests for greater (>) or for greater than or equal to (>=)
< *or* <=	Tests for less than (<) or for less than or equal to (<=)

using special rules-based workflow attributes, which are listed in Table 12-5. The following code shows one of these attributes in action:

```
[RuleWrite("HandlingCost")]
public void DiscountShipping(decimal percentageDiscount)
{
   ...
   // Code here to update handling cost
   ...
}
```

These attributes come into play when dealing with forward chaining.

Table 12-5 Rules-Based Attributes

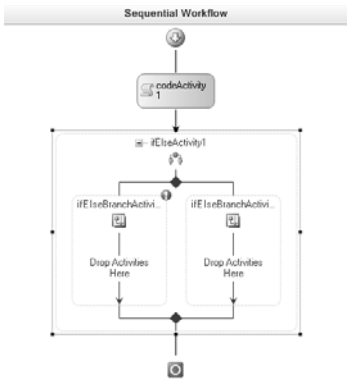
Attribute	Purpose
RuleRead	The default. This attribute tells the rules engine that the method reads workflow instance properties and fields but does not update their values.
RuleWrite	This attribute tells the rules engine that the workflow method updates the value of a potentially dependent field or property.
RuleInvoke	This attribute tells the rules engine that the method this attribute decorates calls one or more other methods that might also update potentially dependent fields or properties.

The *Update* Statement

Table 12-4 listed the rules-based keywords you have at your disposal. They're relatively self-explanatory, with the exception of *Update*. As with the rules-based attributes, we'll talk more about *Update* when we get to forward chaining, but the idea is to inform the rules engine that your rule is explicitly updating a field or property so that other dependent rules are made aware of the change. *Update* doesn't actually modify the field or property—it informs the rules engine that the field or property changed.

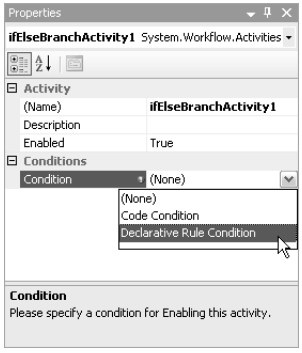
Update takes a single string value, which represents the name of the field or property, and it uses that to notify the rules engine that dependent rules might require reevaluation. Although best practices dictate that use of the rules-based attributes is preferred, there are times when *Update* is appropriate. A good example of this is when you're modifying a property on a workflow assembly you didn't write (one that doesn't have rules-based attributes, and one for which you can't update the source code to include the necessary attributes).

Probably the best way to begin to understand how rules can be used in workflow processing is to write some code and try them out. We'll start with rule conditions, which are in contrast to the code conditions we used in Chapter 9.

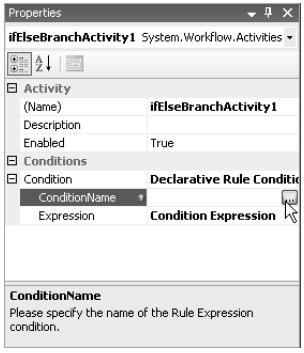


9. In the visual workflow designer, select the left branch, *ifElseBranchActivity1*. This activates its properties in Visual Studio's Properties pane.

10. Select the *Condition* property, and click the down arrow to display the selection list of available conditional processing options. Choose the Declarative Rule Condition option.



11. Expand the *Condition* property by clicking the plus sign (+) next to the property name. Once the property expands, click the *ConditionName* property to activate the Browse (...) button. Click it.



19. With the workflow now complete, add a reference to the workflow from the RuleQuestioner application. Right-click the RuleQuestioner tree control node in Visual Studio's Solution Explorer, and select Add Reference. When the Add Reference dialog box appears, click the Projects tab. Select RuleFlow from the list and click OK.

20. Open Program.cs in the RuleQuestioner project for editing and then look for this line of code:

```
// Print banner.
Console.WriteLine("Waiting for workflow completion.");
```

21. To create a workflow instance, add this code following the line of code you just located:

```
// Create the workflow instance.
WorkflowInstance instance =
   workflowRuntime.CreateWorkflow(typeof(RuleFlow.Workflow1));

// Start the workflow instance.
instance.Start();
```

22. Compile the solution by pressing F6 or by selecting Build Solution from the main Visual Studio Build menu. Correct any compilation errors that might be present.

23. Execute the application by pressing F5 (or Ctrl+F5).

If you look closely at step 13, the rule we added has nothing to do with whether the user told the workflow that today is or is not Tuesday. The rule checked the actual day of the week. It *could* have taken the user's input into account. (I would have added *this._bAnswer* to the rule to access the Boolean value.)

You also might wonder why this is better than using a code condition. Actually, it's not that one is better than the other (code condition over rule condition). The effect is the same. What changed is the decision that was made was made using stored rule material, which at run time could be replaced with different rule material. It's a powerful concept. It becomes even more powerful when more than one rule is involved, which is the case with *policy*. But before we get to policy, we need to look at *forward chaining*.

Forward Chaining

If you've ever watched cars being assembled, you can't help but be amazed. They're actually quite complex, and the assembly process is necessarily even more complex. Wrapped up in the assembly process is the concept of an option. Cars have optional components. Maybe some have satellite radio, or others come with Global Positioning System receivers so that the driver never becomes lost. Not all cars on the assembly line have every option.

So when a car comes down the line that does have more options than others, the assembly process often changes. Some options require different wiring harnesses very early in their assembly. Or they require stronger batteries or different engine components.

This might sound odd, but it has value. If you write your own workflows, you should use the rules-based attributes. However, as workflow-based software grows in popularity and people begin using third-party workflows, they might find the rules-based attributes haven't been applied to the various workflow methods. In that case, they should use the *Update* statement to maintain the correct workflow state and keep the rules engine in sync. The rules-based attributes state changes declaratively, while the *Update* statement is imperative. You need an imperative solution when working with precompiled third party software.

Returning to the preceding example, assume the *SetDiscount* method did not have the *RuleWrite* attribute applied. The two rules would then look like this:

```
IF this.OrderQuantity > 500 THEN this.SetDiscount(0.1)
 Update(this.Discount)
```

And

```
IF this.Discount > 0 && this.Customer == "Contoso"
THEN this.ShippingCost = 0
```

Armed with this information, the rules engine is aware that the *Discount* property has been updated and will reevaluate the application of the rules accordingly.

Controlling Forward Chaining

You might think that once you initiate rules-based workflow execution you give up control and allow the rules engine to make all the decisions. Although in most cases this is precisely what you want, you do have some control over how rule dependencies and forward chaining are handled.

Table 12-6 contains the three types of forward chaining control you have.

Table 12-6 Forward Chaining Control Actions

Action	Purpose
Full Chaining	The default. This action allows the rules engine to process and reevaluate rules as it deems necessary.
Explicit Chaining	When applied, this control action limits forward chaining behavior to rules that include the *Update* statement.
Sequential	This effectively turns forward chaining off. No dependencies are evaluated, and rules are applied in order, once per rule.

Full chaining allows the rules engine to process rules as it was designed to do, including implicit and attributed reevaluations as required.

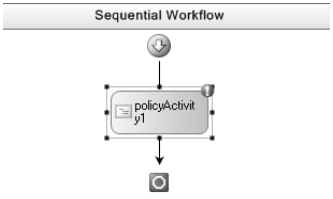
Explicit chaining deactivates implicit and attributed forward chaining, and it places the burden of notifying the rules engine of dependencies squarely on your shoulders, using explicit forward chaining. Where the *Update* statement is used, you have total control over

shipment? It's this scenario I'm interested in demonstrating because it shows the rules evaluation process in action.

Imagine we're the plastics manufacturer and we have two major customers, Tailspin Toys and Wingtip Toys. Tailspin Toys has told us they accept partial shipments, but Wingtip requires the full order to be delivered. Our workflow will use a *Policy* activity to apply the rules I outlined to these customers, their orders, and the amount of raw material we have on hand, which might or might not be enough to complete their order. Let's see this activity in action.

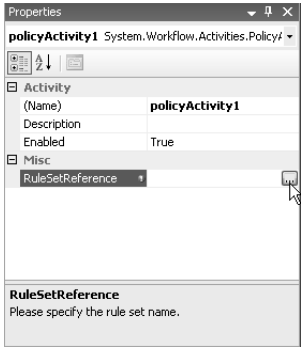
Create a new workflow application with the *Policy* activity

1. The PlasticPolicy application is again provided to you in two varieties: completed and incomplete. You can use the completed version, so simply follow along, and you'll find it in the \Workflow\Chapter12\PlasticPolicy Completed\directory. The incomplete version will require you to work through the steps I've outlined here, and you can find it in the \Workflow\Chapter12\PlasticPolicy\ folder. To open either solution, just drag its .sln file onto an executing copy of Visual Studio.
2. Once Visual Studio has loaded the PlasticPolicy solution and made it available for editing, create a separate sequential workflow library project as you did in Chapter 3, in the "Adding a sequential workflow project to the WorkflowHost solution" procedure. Name this workflow library **PlasticFlow** and save it in the \Workflow\Chapter12\ PlasticPolicy directory.
3. After Visual Studio has added the PlasticFlow project, Visual Studio opens the *Workflow1* workflow for editing in the visual workflow designer. Open the Toolbox, and drag an instance of the *Policy* activity onto the designer's surface and drop it.

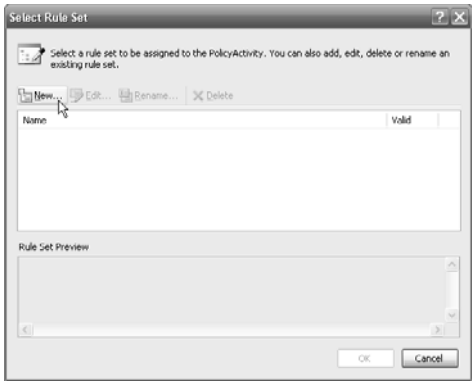


4. Before you actually create the rules to go with the *Policy* activity you just inserted into your workflow, you need to add some initialization code and helper methods. To begin, open Workflow1.cs in the code editor by selecting it in the Solution Explorer tree control and clicking the View Code toolbar button. Prior to the constructor, type in this code:

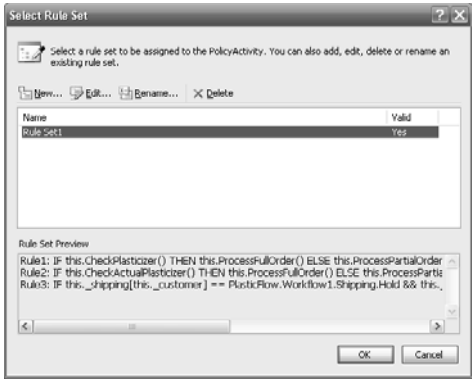
```
private enum Shipping { Hold, Partial };
private decimal _plasticizer = 14592.7m;
private decimal _plasticizerActual = 12879.2m;
private decimal _plasticizerRatio = 27.4m; // plasticizer for one item
private Dictionary<string, Shipping> _shipping = null;

// Results storage
private bool _shipPartial = false;
```



11. Click the browse button to activate the Select Rule Set dialog box. Once the Select Rule Set dialog box is active, click its New button.



12. Clicking the New button activates the Rule Set Editor dialog box. Click Add Rule to add a new rule and activate the dialog box controls.



17. Your workflow is now complete. Although it might seem odd to have an entire workflow reside in a single activity, in reality you've told your workflow what to do by the rules you provided. In any case, add a reference to the workflow from the PlasticPolicy application. Right-click the PlasticPolicy tree control node in Visual Studio's Solution Explorer, and select Add Reference. When the Add Reference dialog box appears, click the Projects tab and select PlasticFlow from the list. Click OK.

18. Open Program.cs in the PlasticPolicy project for editing, and then look for the *Main* method. Following the opening brace for *Main*, add this code:

```
// Parse the command line arguments
string company = String.Empty;
Int32 quantity = -1;
try
{
    // Try to parse the command line args...
    GetArgs(ref company, ref quantity, args);
} // try
catch
{
    // Just exit...
    return;
} // catch
```

19. Then find this line of code a bit further down in *Main*:

```
// Print banner.
Console.WriteLine("Waiting for workflow completion.");
```

20. Add this code following the line of code you just located:

```
// Create the argument.
Dictionary<string, object> parms = new Dictionary<string, object>();
parms.Add("Customer", company);
parms.Add("OrderQuantity", quantity);

// Create the workflow instance.
WorkflowInstance instance =
    workflowRuntime.CreateWorkflow(typeof(PlasticFlow.Workflow1), parms);

// Start the workflow instance.
instance.Start();
```

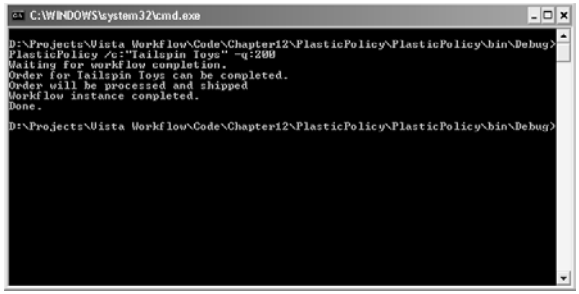


Figure 12-3 Tailspin full and complete shipment

Tailspin is registered as accepting partial shipments. Wingtip Toys, however, wants orders held until its entire order can be filled. Does the workflow handle Wingtip as well? Moreover, what if Wingtip's order fell into that range where we thought we had enough plasticizer but in reality didn't? To find out, try this command: **PlasticPolicy.exe /c:"Wingtip Toys" /q:500**. As Figure 12-4 shows, we find out we can only partially complete Wingtip's order. On top of that, when we accessed our customer preference records, we elected to withhold Wingtip's order for the moment.

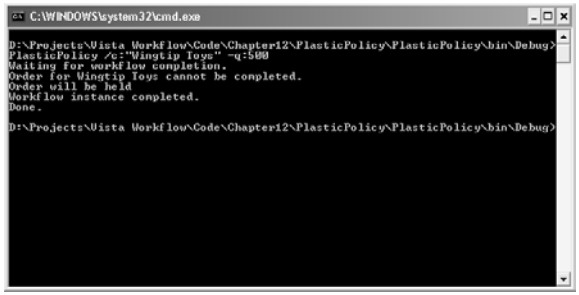


Figure 12-4 Wingtip Toys partial shipment

To test a final scenario, one where we can meet Wingtip's needs regardless of the actual level of plasticizer, at the command prompt type the following command: **PlasticPolicy.exe /c:"Wingtip Toys" /q:200**. Wingtip Toys now has ordered 200 items, and indeed, as Figure 12-5 indicates we can completely fill Wingtip's order.

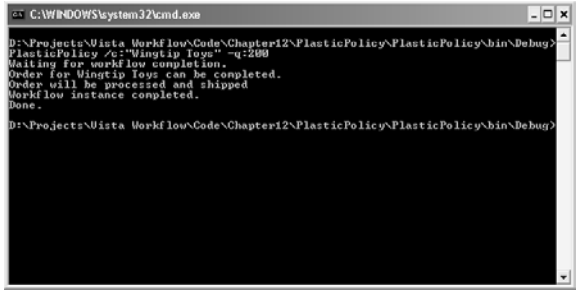


Figure 12-5 Wingtip Toys full and complete shipment

Chapter 13

Crafting Custom Activities

After completing this chapter, you will be able to:

- Understand what components are necessary to create a fully functional custom workflow activity
- Create a basic custom workflow activity
- Apply validation rules to a basic custom workflow activity
- Integrate a basic custom workflow activity into the Microsoft Visual Studio visual workflow designer and Toolbox

As deep and functional as Windows Workflow Foundation (WF) is, it can't possibly encompass everything you might want to achieve with your workflows. Even though WF is still very new to the development community, many freely distributed custom activities are already available, and you can be sure commercial-grade activities eventually will follow.

In this chapter, you'll get a look inside WF by creating a new workflow activity, one that retrieves a file from a remote File Transfer Protocol (FTP) server. You'll see what pieces are necessary, as well as what parts are nice to have when building your own activity. You'll also dig a little into how activities interact with the workflow runtime.

Note It won't be possible to explore every nuance of custom activity development in a single chapter. There are simply too many details. However, the good news is it's easy to get a fully functional activity working without knowing every detail. Where there is more detail, I'll provide links to more information.

More About Activities

In Chapter 4, "Introduction to Activities and Workflow Types," we took an initial look at activities and discussed topics such as the *ActivityExecutionContext*, which is used to contain information about executing activities the workflow runtime needs to access from time to time. We'll dig into WF activities a little deeper here.

Activity Virtual Methods

The first thing to know when creating custom activities is what the base class provides you by way of virtual methods and properties. Table 13-1 shows the commonly overridden methods for *Activity*. (There are no virtual properties.)

Table 13-1 Commonly Overridden *Activity* Virtual Methods

Method	Purpose
Cancel	Invoked when the workflow is canceled.
Compensate	This method isn't actually implemented by the *Activity* base class but rather required by the *ICompensatableActivity* interface from which many activities derive. Therefore, for all intents and purposes it is an *Activity* method. You'll implement this method to compensate for failed transactions.
Execute	The main activity worker method, *Execute*, is used to perform the work that the activity was designed to perform.
HandleFault	Called when internal activity code throws an unhandled exception. Note there is no way to restart the activity once this method is invoked.
Initialize	Called when the activity is initialized.
OnActivityExecutionContextLoad	Called when the activity is handed an *ActivityExecutionContext* for processing.
OnActivityExecutionContextUnload	Called when the activity has finished its workflow process. The current execution context is being shifted to another activity.
Uninitialize	Called when the activity is to be uninitialized.

If you need to handle some specific processing once your activity has been loaded into the workflow runtime but before it is executing, a great place to do that is in the *Initialize* method. You would perform similar out-processing in the *Uninitialize* method.

The *OnActivityExecutionContextLoad* and *OnActivityExecutionContextUnload* methods signify the activity loading into the workflow runtime and the activity's removal from it, respectively. Before *OnActivityExecutionContextLoad* is called, and after *OnActivityExecutionContextUnload* is called, the activity is in an unloaded state from a WF perspective. It might be serialized into a queue, stored in a database, or even on disk waiting to be loaded. But it does not exist in the workflow runtime before or after these methods are called.

Cancel, *HandleFault*, and *Compensate* are all called when the obvious conditions arise (canceling, faulting, and compensating). Their primary purpose is to perform any additional work you want to perform (logging, for example), although *Compensate* is where you truly implement your transaction compensation. (See Chapter 15, "Workflows and Transactions.") Keep in mind that at the point these methods are called, it's too late. You can't revive a transaction

Note Activities are generally created from a deserialization process rather than from the workflow runtime calling their constructors directly. Therefore, if you need to allocate resources when the activity is created, *OnActivityContextLoad* is the best place to do so, rather than from within a constructor.

Although *OnActivityExecutionContextLoad* and *OnActivityExecutionContextUnload* denote the activity's creation from a memory perspective, *Initialize* and *Uninitialize* identify the activity's execution lifetime within the workflow runtime. When the workflow runtime calls the *Initialize* method, your activity is ready to go. When *Uninitialize* is executed, your activity has finished from a workflow runtime point of view and is ready to be shifted out of memory. *Dispose*, the archetypical .NET object destruction method, is useful for deallocating static resources.

Of course, the workflow can't always control the execution of some of the methods. *Compensate*, for example, is called only when a compensatable transaction fails. These remaining methods will nondeterministically be called while *Execute* is in effect.

Creating an FTP Activity

To demonstrate some of what I've described so far in the chapter, I decided to create an activity many of us writing business process software will (hopefully) find useful—an FTP activity. This activity, *FtpGetFileActivity*, retrieves files from a remote FTP server using the built-in .NET Web-based FTP classes. It is possible to use those same classes for writing files to remote FTP resources, but I leave that activity for you to create as an exercise.

Note I'll work under the assumption that you have a known (and correctly configured) FTP site to work with. For the purposes of discussion here, I'll use the well-known Internet Protocol (IP) address 127.0.0.1 as the server's IP address. (Of course, this represents *localhost*.) Feel free to replace this IP address with any valid FTP server address or host name you prefer. It is beyond the scope of this chapter to address FTP security issues and server configuration. If you are using Internet Information Server (IIS) and need more information regarding FTP configuration, see *http://msdn2.microsoft.com/en-us/library/6ws081sa.aspx* for assistance.

To host the FTP activity, I created a sample application I called FileGrabber. (Its user interface is shown in Figure 13-1.) With it, you can provide an FTP user account and password as well as the FTP resource you want to retrieve. The resource I'll be downloading is an image file of the Saturn V rocket moving into position for launch, and I've provided the image on the book's CD for you to place on your FTP server as well. Assuming your FTP server was your local machine, the URL for the image is *ftp://127.0.0.1/SaturnV.jpg*. If you don't use my image file, you'll need to modify the file in the URL to match whatever file you have available on your local server, or use any valid URL from which you can download files.

5. Now we can add the *using* statements we'll need. Following the list of *using* statements Visual Studio inserted for you when the source file was created, add these:

```
using System.IO;
using System.Net;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Workflow.ComponentModel;
using System.Workflow.ComponentModel.Compiler;
using System.Workflow.ComponentModel.Design;
using System.Workflow.Activities;
using System.Drawing;
```

6. Because we're building an activity, we need to derive *FtpGetFileActivity* from the appropriate base class. Change the current class definition to the following:

```
public sealed class FtpGetFileActivity :
                        System.Workflow.ComponentModel.Activity
```

> **Note** Because you're creating a basic activity, the FTP activity derives from *System.Workflow.ComponentModel.Activity*. However, if you were creating a composite activity, it would derive from *System.Workflow.ComponentModel.CompositeActivity*.

7. For this example, the *FtpGetFileActivity* will expose three properties: *FtpUrl*, *FtpUser*, and *FtpPassword*. Activity properties are nearly always dependency properties, so we'll add three dependency properties, starting with the *FtpUrl*. Type this code into the *FtpGetFileActivity* class following the class's opening brace (at this point the class contains no other code):

```
public static DependencyProperty FtpUrlProperty =
        DependencyProperty.Register("FtpUrl", typeof(System.String),
            typeof(FtpGetFileActivity));

[Description ("Please provide the full URL for the file to download.")]
[DesignerSerializationVisibility(DesignerSerializationVisibility.Visible)]
[ValidationOption(ValidationOption.Required)]
[Browsable(true)]
[Category("FTP Parameters")]
public string FtpUrl
{
    get
    {
        return ((string)
            (base.GetValue(FtpGetFileActivity.FtpUrlProperty)));
    }
    set
    {
        Uri tempUri = null;
        if (Uri.TryCreate(value, UriKind.Absolute, out tempUri))
        {
            if (tempUri.Scheme == Uri.UriSchemeFtp)
            {
                base.SetValue(FtpGetFileActivity.FtpUrlProperty,
```

12. *Execute* invokes the *GetFile* method, so add that following *Execute*:

```
private void GetFile()
{
    // Create the URI. We check the validity again
    // even though we checked it in the property
    // setter since binding may have taken place.
    // Binding shoots the new value directly to the
    // dependency property, skipping our local
    // getter/setter logic. Note that if the URL
    // is very malformed, the URI constructor will
    // throw an exception.
    Uri requestUri = new Uri(FtpUrl);
    if (requestUri.Scheme != Uri.UriSchemeFtp)
    {
        // Not a valid FTP URI...
        throw new ArgumentException("The value assigned to the" +
          "FtpUrl property is not a valid FTP URI.");
    } // if

    string fileName =
        Path.GetFileName(requestUri.AbsolutePath);

    if (String.IsNullOrEmpty(fileName))
    {
        // No file to retrieve.
        return;
    } // if

    Stream bitStream = null;
    FileStream fileStream = null;
    StreamReader reader = null;
    try
    {
        // Open the connection
        FtpWebRequest request =
            (FtpWebRequest)WebRequest.Create(requestUri);

        // Establish the authentication credentials.
        if (!String.IsNullOrEmpty(FtpUser))
        {
            request.Credentials =
                new NetworkCredential(FtpUser, FtpPassword);
        } // if
        else
        {
            request.Credentials =
                new NetworkCredential(AnonymousUser,
                !String.IsNullOrEmpty(FtpPassword) ?
                FtpPassword : AnonymousPassword);
        } // else
```

```
        }
    }

    public static DependencyProperty FtpUserProperty =
          DependencyProperty.Register("FtpUser", typeof(System.String),
          typeof(FtpGetFileActivity));

    [Description("Please provide the FTP user account name.")]
    [DesignerSerializationVisibility(
                              DesignerSerializationVisibility.Visible)]
    [ValidationOption(ValidationOption.Optional)]
    [Browsable(true)]
    [Category("FTP Parameters")]
    public string FtpUser
    {
        get
        {
            return ((string)(
                  base.GetValue(FtpGetFileActivity.FtpUserProperty)));
        }
        set
        {
            base.SetValue(FtpGetFileActivity.FtpUserProperty, value);
        }
    }

    public static DependencyProperty FtpPasswordProperty =
      DependencyProperty.Register("FtpPassword", typeof(System.String),
      typeof(FtpGetFileActivity));

    [Description("Please provide the FTP user account password.")]
    [DesignerSerializationVisibility(
                              DesignerSerializationVisibility.Visible)]
    [ValidationOption(ValidationOption.Optional)]
    [Browsable(true)]
    [Category("FTP Parameters")]
    public string FtpPassword
    {
        get
        {
            return ((string)(
               base.GetValue(FtpGetFileActivity.FtpPasswordProperty)));
        }
        set
        {
            base.SetValue(FtpGetFileActivity.FtpPasswordProperty,
                                                                value);
        }
    }

    private const string AnonymousUser = "anonymous";
    private const string AnonymousPassword = "someone@example.com";

    protected override ActivityExecutionStatus Execute(
        ActivityExecutionContext executionContext)
```

properties when the state of the designer changes (that is, when new activities are added or properties change) and when the workflow is compiled.

The validator can choose to ignore property configurations, or it can mark them as warnings or outright errors. The FTP activity has three properties, one of which is critical (the URL). The other two can be left untouched, which will cause authentication with the default (anonymous) user. As we complete our validator, we'll mark the lack of a URL (or lack of a binding to a URL property in the main workflow activity) as an error. If the user name or password is omitted we'll generate warnings stating that the anonymous login will be used.

Creating a validator for the *FtpGetFileActivity* workflow activity

1. Activity validators in WF are just classes, so we'll begin by adding a new class to the FtpActivity project. Right-click the FtpActivity project in Visual Studio's Solution Explorer window, select Add, and then select Class. When the Add New Item dialog box appears, type **FtpGetFileActivityValidator.cs** in the Name field and click the dialog box's Add button.

2. Add the following *using* statement to the list of preexisting *using* statements:

```
using System.Workflow.ComponentModel.Compiler;
```

3. When the new *FtpGetFileActivityValidator* class is created, it's created as a private class. Moreover, WF activity validators must use *ActivityValidator* as a base class. Visual Studio opens the source file for editing, so change the class definition to the following by adding the public keyword as well as the *ActivityValidator* as the base:

```
public class FtpGetFileActivityValidator : ActivityValidator
```

4. To actually perform validation yourself, you must override the *Validate* method. Here, you'll examine the properties, and if they're lacking, you'll add an error to an errors collection the designer maintains. Here is the completed *Validate* override you need to add to the *FtpGetFileActivityValidator* class:

```
public override ValidationErrorCollection
    Validate(ValidationManager manager, object obj)
{
    FtpGetFileActivity fget = obj as FtpGetFileActivity;

    if (null == fget)
        throw new InvalidOperationException();

    ValidationErrorCollection errors = base.Validate(manager, obj);

    if (null != fget.Parent)
    {
        // Now actually validate the activity...
        if (String.IsNullOrEmpty(fget.FtpUrl) &&
            fget.GetBinding(FtpGetFileActivity.FtpUrlProperty) == null)
        {
            ValidationError err =
                new ValidationError("Note you must specify a URL " +
```

```
                                                                    null)
                {
                    ValidationError err =
                        new ValidationError("The 'anonymous' user " +
                            "account will be used for logging in to the " +
                            "FTP server.", 200, true);
                    errors.Add(err);
                } // if
                if (String.IsNullOrEmpty(fget.FtpPassword) &&
                    fget.GetBinding(FtpGetFileActivity.FtpPasswordProperty)
                                                                == null)
                {
                    ValidationError err =
                        new ValidationError("The default anonymous " +
                           "password 'someone@example.com' will be used " +
                           "for logging in to the FTP server.", 300, true);
                    errors.Add(err);
                } // if
            }
            return errors;
        }
    }
}
```

Providing a Toolbox Bitmap

The next thing we'll do to our activity is give it a Toolbox bitmap. This isn't truly a WF task. This capability is built into.NET, to be primarily used for Visual Studio designer support. It also isn't hard to do.

Assigning a Toolbox bitmap to the *FtpGetFileActivity* workflow activity

1. To assign a bitmap, you must first have a bitmap. In the Chapter13 directory of the book's sample code, you will find a bitmap file named FtpImage. I find the easiest thing to do is to drag the FtpImage from a Windows Explorer window and drop it onto the FtpActivity's project tree control node in Visual Studio's Solution Explorer window. This both copies the file into your project directory and adds it to the project.
2. With the bitmap included in the project, you now must compile it into your assembly as a resource. Select the FtpImage file in the FtpActivity project in Solution Explorer to activate its properties. Change the Build Action property from Compile to Embedded Resource.
3. As with the validator, it isn't enough just having a bitmap compiled into your activity's assembly. You must also tell Visual Studio the activity has an associated Toolbox bitmap. And as before, you tell Visual Studio using an attribute. Add this attribute to the *FtpGetFileActivity* class definition (just before the *ActivityValidator* you added in the preceding section):

```
[ToolboxBitmap(typeof(FtpGetFileActivity), "FtpImage")]
```

The complete *FtpGetFileActivityDesigner* file is shown in Listing 13-3. Although we could have done more in the designer class itself if there were designer behavior we required, in this case the designer class exists merely to inject the theme. The activity will be rendered in the visual workflow designer using a horizontal gradient coloration (silver to light blue), contained by a solid black border.

Listing 13-3 FtpGetFileActivityDesigner.cs completed

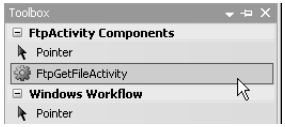
```
using System;
using System.Collections.Generic;
using System.Text;
using System.ComponentModel;
using System.ComponentModel.Design;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Workflow.Activities;
using System.Workflow.ComponentModel.Design;

namespace FtpActivity
{
    [ActivityDesignerThemeAttribute(typeof(FtpGetFileActivityDesignerTheme))]
    public class FtpGetFileActivityDesigner : ActivityDesigner
    {
    }

    internal sealed class FtpGetFileActivityDesignerTheme :
                                                    ActivityDesignerTheme
    {
        public FtpGetFileActivityDesignerTheme(WorkflowTheme theme)
            : base(theme)
        {
            this.BorderColor = Color.Black;
            this.BorderStyle = DashStyle.Solid;
            this.BackColorStart = Color.Silver;
            this.BackColorEnd = Color.LightBlue;
            this.BackgroundStyle = LinearGradientMode.Horizontal;
        }
    }
}
```

There is one detail remaining—the name the *FtpGetFileActivity* icon will display when it's loaded into the Toolbox.

Integrating Custom Activities into the Toolbox

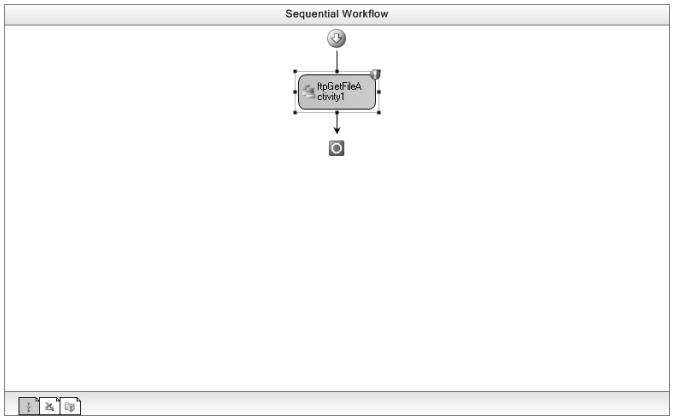
As you know, *ToolboxBitmapAttribute* displays an icon that is associated with your activity when your activity is installed in the Visual Studio Toolbox. But as it happens, there is more you can do than just show a bitmap.

Composite activities, for example, often create child activities that are necessary for the proper operation of the parent composite activity. A good example is the *IfElse* activity. When you

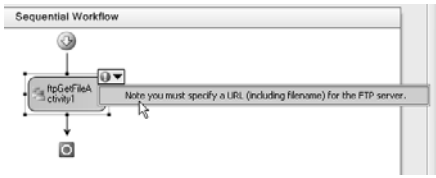


Note You might be wondering where the nice little FTP bitmap went (to be replaced by the blue gear icon), as well as why the display text you added in the *FtpGetFileActivityToolboxItem* class didn't appear in the Toolbox. This is because the *FtpGetFileActivity* is supported by an assembly in the current solution. I'll describe the solution to this after you complete the workflow.

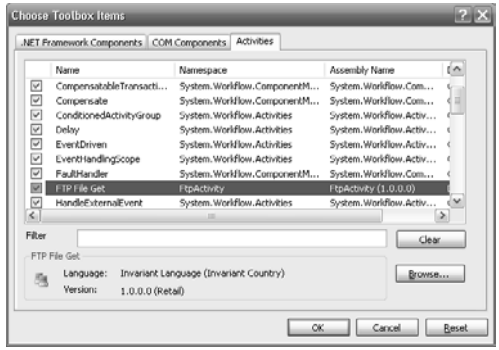
4. Drag an instance of *FtpGetFileActivity* onto the designer's surface, and drop it in the center.



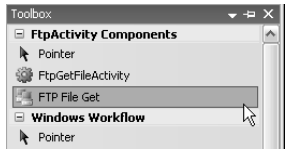
5. The red dot with the exclamation mark is an indication that there are validation errors present. And, in fact, if you place the mouse over the down arrow and click once, you'll see the particular validation failure. Does it look familiar? It should...it's the validation error you inserted when you created the activity validation class in an earlier section.



6. The main FileGrabber application is capable of passing in to your workflow the user name, the password, and the file's URL. Therefore, you need to provide properties in your main workflow for each of these values. Here is a great way to do just that—have Visual Studio add them for you. If you select the FTP activity and then look at its properties in the Properties pane, you'll see the three properties you added to the activity: *FtpUrl*, *FtpUser*, and *FtpPassword*. So that you clear the error condition first, select the *FtpUrl* property to activate the browse (...) button. Click the browse button.



4. This loads the *FtpGetFileActivity* into the Toolbox, and you should see the custom icon and display text you added previously.



Our last task is to add the code we need to kick off the workflow from the main application.

Executing the *FtpGetFileActivity* workflow

1. Open the file Form1.cs in the code editor. To do so, click the Form1.cs file in the FileGrabber project and click the View Code toolbar button.
2. All the code you'll need has been added for you with the exception of the code required to actually start a workflow instance. That code will be placed in the *Get_Click* event handler. Scroll down through the source file until you find the *Get_Click* event handler. Following the code you find there (to disable the user interface controls), add this code:

```
// Process the request, starting by creating the parameters.
Dictionary<string, object> parms = new Dictionary<string, object>();
parms.Add("FtpUrl", tbFtpUrl.Text);
parms.Add("FtpUser", tbUsername.Text);
parms.Add("FtpPassword", tbPassword.Text);

// Create instance.
_workflowInstance = _workflowRuntime.CreateWorkflow(typeof(GrabberFlow.Workflow1),
parms);

// Start instance.
_workflowInstance.Start();
```

3. Because you're using the workflow from the *GrabberFlow* namespace, you need to reference the workflow assembly. Simply right-click the FileGrabber project in Solution Explorer and select Add Reference from the context menu. When the Add Reference dialog box appears, click the Projects tab and select the GrabberFlow project from the list. Click OK.

Part III
Workflow Processing

In this part:

Chapter 14

State-Based Workflows

After completing this chapter, you will be able to:

- Understand the notional concept of a state machine and how it is modeled in workflow processing
- Create state-based workflows
- Apply initial and terminal state conditions
- Incorporate code to transition from state to state

In Chapter 4, "Introduction to Activities and Workflow Types," where I described the types of workflows you can create using Windows Workflow Foundation (WF), I mentioned the state-based workflow. State-based workflows model something known as the *finite state machine*. State-based workflows shine when the workflow requires much interaction with outside events. As events fire and are handled by the workflow, the workflow can transition from state to state as required.

WF provides a rich development experience for creating state-based workflows, and much of what you've seen in the book so far applies to state-based workflows. For example, when a state is transitioned into, you can, if you want, execute a few sequential activities, make conditional decisions (using rules or code), or iterate through some data points using an iterative activity structure. The only real difference is how the activities are queued for execution. In a sequential or parallel workflow, they're queued as they come up. But in a state-based workflow, activities are queued as states are transitioned into and out of. Events generally drive those transitions, but this is not a universal rule. Let's take another look at the conceptual state machine and relate those concepts to WF activities you can use to model your workflows.

The State Machine Concept

State machines are meant to model discrete points within your processing logic, the transitions to which are controlled by events. For example, you load your washing machine, close the door, and push the start button. Pushing the start button initiates a state machine that runs your laundry through the various cleaning cycles until the cycles are complete.

State machines have known starting points and known termination or end points. The states in between should be controlled by events expected to occur while the machine is at a specific state. Sometimes events throw state machines into invalid states, which are conditions not unlike sustaining unhandled exceptions in your applications. The entire process either comes

StateInitialization derives from the *Sequence* activity, so nearly any activity you want to place in this composite activity is available to you and will be executed in sequential order. Certain activities are not allowed, such as any activity based on *IEventActivity*. To handle events in your state, you must use the *EventDriven* activity.

StateInitialization is also executed in a nonblocking manner. This approach is necessary because your state needs to be able to listen for events. If *StateInitialization* were a blocking activity, the thread executing the initialization code would be tied up and unable to listen for events. Note, however, that the event, although it has been received, will not be acted upon until *StateInitialization* completes. After all, critical initialization code might need to be executed prior to actually handling the event.

Using *StateInitialization*, and indeed any of the child activities in a given *State* activity, requires you to interact with the visual workflow designer in a slightly different way than you have so far in this book. If you drag and drop an instance of *StateInitialization* into a *State* activity, you'll find you can't then drop child activities directly into *StateInitialization*. (This is true for the *EventDriven* and *StateFinalization* activities as well.) To drop child activities into *StateInitialization*, you must first double-click the instance of *StateInitialization* you just dropped to activate the sequential workflow editor you've used in previous chapters. To return to the state-based workflow editor, you'll find hyperlink-style buttons in the upper-left corner of the workflow designer that will return you to a view of the particular state you're editing or the entire workflow.

Using the *StateFinalization* Activity

The *StateFinalization* activity is a mirror image of the *StateInitialization* activity and is used in a similar way. Where the *StateInitialization* activity is executed when the state itself begins execution, *StateFinalization* executes just prior to transitioning out of the state. Like *StateInitialization*, the *StateFinalization* activity is based on the *Sequence* activity. *StateFinalization* also limits the types of activities it can contain—*IEventActivity* activities are disallowed, as are *State* and *SetState*.

Creating a State-Based Workflow Application

If you recall the sample state machine I presented in Chapter 4, Figure 14-1 will look familiar. Yes, it's the (simplified) vending machine state diagram. I thought it might be interesting to build this state diagram into a true WF state-based workflow and then drive it using a user interface that, given my feeble artistic abilities, models a crude soft drink ("soda") vending machine.

Figure 14-5 The SodaMachine user interface once a selection has been made

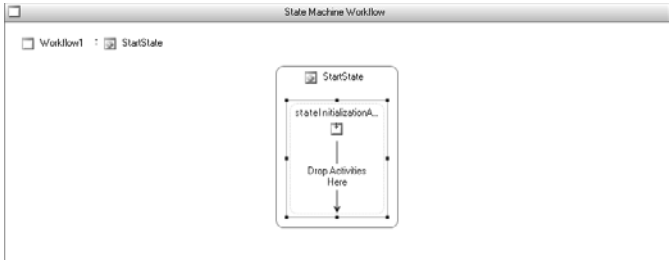
A great deal of the application has been created for you. If you wade through the sample SodaMachine code, you'll find I used the *CallExternalMethod* activity (from Chapter 8, "Workflow Data Transfer") and the *HandleExternalEvent* activity (from Chapter 10, "Event Activities"). These are great tools for interacting with your workflow from your application. What's left to create is the workflow itself, and here's how.

Building a state-based workflow

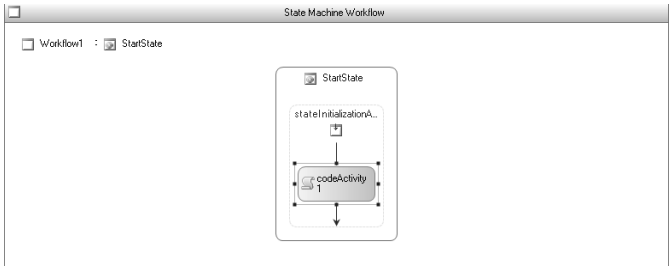
1. The SodaMachine application in its initial form can be found in the \Workflow\Chapter14\SodaMachine directory. As I've done in the past for the more complex sample applications, I placed two different versions in the Chapter14 directory—an incomplete version and a completed version. If you're interested in following along but don't want to perform the steps outlined here, open the solution file for the completed version. (It will be in the SodaMachine Completed directory.) The steps you'll follow here take you through building the state-based workflow. If you're interested in working through the steps, open the incomplete version instead. To open either solution, drag the .sln file onto a copy of Visual Studio to open the solution for editing and compilation. (If you decide to compile and execute the completed version directly, compile it twice before executing it. Internal project-level dependencies must be resolved after the first successful compilation.)
2. With the SodaMachine solution open in Visual Studio, press F6 or select Build Solution from Visual Studio's Build menu. The projects have various dependencies, and compiling the solution generates assemblies that dependent projects can reference.
3. Find the Workflow1.cs file in the SodaFlow project, within Visual Studio's Solution Explorer window. (You might need to expand tree control nodes to find it.) When the file is in view in the tree control, select it with a single mouse click and then click the View Designer toolbar button. This brings the workflow into the visual workflow designer for editing.

10. With the state activities in place, let's now add details. Starting with *StartState*, drag an instance of the *StateInitialization* activity from the Toolbox and drop it into *StartState*.

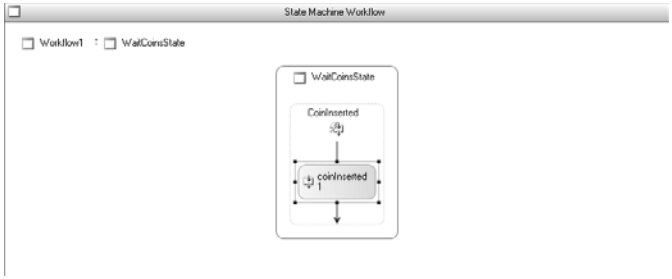
11. Double-click the activity you just inserted, *stateInitialization1*, to enter the sequential workflow editor.



12. Drag a copy of the *Code* activity from the Toolbox, and drop it into the state initialization activity. Assign its *ExecuteCode* method to be **ResetTotal**. Visual Studio then adds the *ResetTotal* method for you and switches you to the code editor. Rather than add code at this point, return to the visual workflow designer.

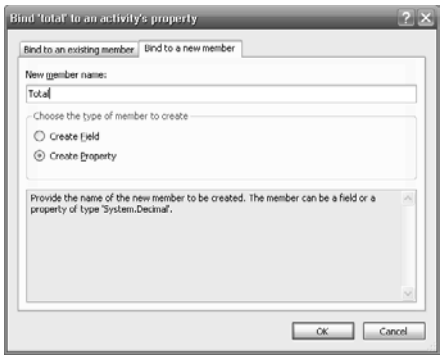


13. Next drag an instance of *SetState* onto the designer's surface, and drop it just below the *Code* activity you just inserted.

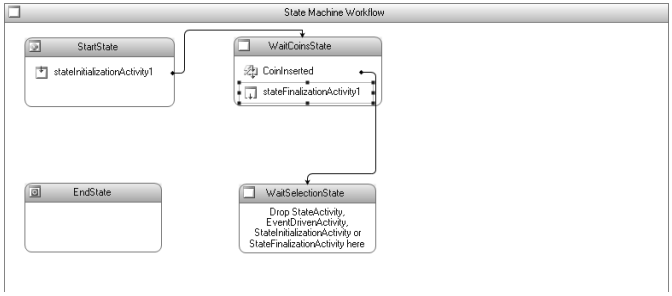


19. With the *ExternalEventHandler coinInserted1* activity selected in the visual workflow designer, click the *CoinValue* property in the Properties pane to activate the browse (...) button, and then click the browse button. This brings up the Bind 'CoinValue' To An Activity's Property dialog box. Click the Bind To A New Member tab, and type **LastCoinDropped** in the New Member Name field. The Create Property option should be selected, but if it isn't, select it so that you create a new dependency property. Click OK.

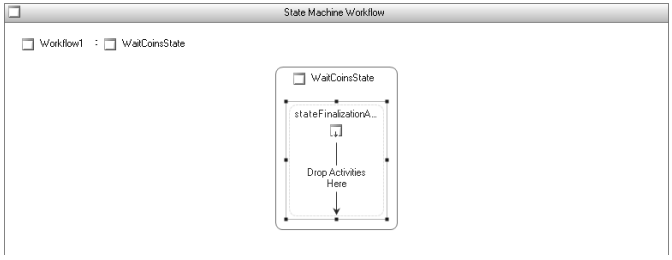
20. Now we need to make a decision—did the user just drop enough money to enable soda selection? To do this, drag an instance of the *IfElse* activity onto the visual workflow designer's surface and drop it into the *CoinInserted EventDriven* activity, following the *coinInserted1* event handler.



25. Click the Workflow1 hyperlink-style button in the upper-left corner to return to the state designer view. Drag an instance of *StateFinalization* onto the visual workflow designer's surface, and drop it into *WaitCoinsState*.



26. Double-click the *stateFinalizationActivity1* activity you just inserted to reactivate the sequential designer view.



27. From the Toolbox, drag an instance of *ReadyToDispense* and drop it into *stateFinalizationActivity1*. *ReadyToDispense* is also a customized *CallExternalMethod* activity.

34. The workflow is complete from a visual workflow designer's point of view, but we still have some code to write. Select Workflow1.cs in Visual Studio's Solution Explorer, and click the View Code toolbar button to open the file for editing in the code editor.

35. Scan the Workflow1.cs source file, and locate the *ResetTotal* method you added in step 12. Insert the following code in the *ResetTotal* method:

```
// Start with no total.
Total = 0.0m;
```

36. Finally, locate the *TestTotal* method you added in step 21. To that method, add this code:

```
// Add the last coin dropped to the total and check
// to see if the total exceeds 1.25.
Total += LastCoinDropped;
e.Result = Total >= 1.25m;
```

37. Compile the entire solution by pressing F6 or by selecting Build Solution from Visual Studio's Build menu. Correct any compilation errors.

Now you can run the application by pressing F5 or Ctrl+F5. Click a coin button. Does the total update in the LCD display? When you insert enough money, can you select a soda?

Note If the application crashes with an *InvalidOperationException,* it's most likely due to the references not being fully updated by the first complete solution compilation. Simply recompile the entire application (repeat step 37) and run the application again. It should run cleanly.

If you want to continue to the next chapter, keep Visual Studio 2005 running and turn to Chapter 15, "Workflows and Transactions." In Chapter 15, you'll take your first steps into the fascinating world of workflow transactional processing.

If you want to stop, exit Visual Studio 2005 now, save your spot in the book, and close it. Who needs transactions anyway? Actually, we all do, but we'll wait for you.

Chapter 15

Workflows and Transactions

After completing this chapter, you will be able to:

- Understand the classical transaction model and where that model does and does not fit
- Know where classical transactions do not fit and when compensated transactions are appropriate
- See how transactions are rolled back or compensated
- See how to modify the default order of compensation

If you write software, sooner or later you'll need to understand transactional processing. *Transactional processing* in this sense means writing software that records information to a *durable resource*, such as a database, Microsoft Message Queue (which uses a database under the covers), Windows Vista with transacted file system and Registry access, or even some other software system that supports transactional processing. Durable resources retain the written information no matter what happens to them once the data has been recorded.

Transactions are critical to any business process because, by using transactions, you can be sure the data contained within your application is consistent. If the business process sustains an error yet still persists any data, the erroneous data most likely will propagate throughout the system, leaving you to question which data is good and which data is bad. Imagine ordering this book from an online merchant, only to find the merchant "had a little accident" with your credit card transaction and charged you 100 times the face value of the book instead of their discounted price. Transactional processing isn't a laughable or avoidable subject when errors such as this can happen.

Understanding Transactions

Transactional processing, at its very core, is all about managing your application's state. By *state*, I really mean the condition of all the application's data. An application is in a determinate state when all of its data is consistent. If you insert a new customer record into your database and that update requires two insertions (one to add a normalized row to tie the address to your customer and one to record the actual address information), adding the normalized row but failing to insert the address itself places your application in an indeterminate state. What will happen later when someone tries to retrieve that address? The system says the address should be there, but the actual address record is missing. Your application data is now inconsistent.

To be sure both updates are successful, a *transaction* comes into play. A transaction itself is a single unit of work that either completely succeeds or completely fails. That's not to say you can't update two different database tables. It just means that both table updates are considered a single unit of work, and both must be updated or else neither one is. If either or both updates fail, ideally you want the system to return to its state just prior to your attempt to update the tables. Your application should move forward with no evidence that there had been an incomplete attempt to modify the tables, and more important, you don't want to have data from the unsuccessful update in one table but not in the other.

Note Entire volumes have been written about transactions and transactional processing. Although I'll describe the concepts in sufficient depth to explain how Microsoft Windows Workflow Foundation (WF) supports transactions, I cannot possibly cover transactional processing in great depth in this book. If you haven't reviewed general transactional support in .NET 2.0, you should do so. WF transactions model .NET 2.0 transactional support very closely, and you might find the information in the following article helpful to understanding WF transactional support: *msdn2.microsoft.com/en-us/library/ms973865.aspx*.

Traditionally, transactions have come in a single form—that of the *XA*, or *two-phase commit*, style of transaction. However, with the advent of Internet-based communication and the need to commit long-running transactions, a newer style of transaction was introduced known as the *compensated* transaction. WF supports both styles. We'll first discuss the classical transaction, and then after noting the conditions that make this type of transaction a poor architectural choice, we'll discuss the compensated transaction.

Classic (XA) Transactions

The first system known to have implemented transactional processing was an airline reservation system. Reservations that required multiple flights could not progress if any of the individual flights could not be booked. The architects of that system knew this and designed a transactional approach that today we know as the. X/Open Distributed Transaction Processing Model, known as *XA*. (See *en.wikipedia.org/wiki/X/Open_XA*.)

An XA transaction involves the XA protocol, which is the two-phase commit I mentioned earlier, and three entities: the application, resource, and transactional manager. The application is, well, your application. The resource is a software system that is designed to join in XA-style transactions, which is to say it *enlists* (joins) in the transaction and understands how to participate in the two phases of committing data as well as provides for durability (discussed shortly). The transactional manager oversees the entire transactional process.

So what is a two-phase commit? In the end, imagine your application needs to write data to, say, a database. If that write is performed under the guise of a transaction, the database holds the data to be written until the transactional manager issues a *prepare* instruction. At that point, the database responds with a *vote*. If the vote is to go ahead and commit (write) the data into a table, the transaction manager proceeds to the next participating resource, if any.

compensation function for your part of the transaction. If you debited an online consumer's credit card and were later told to compensate, you would immediately credit the customer's account with the same amount of money you originally debited. In an XA-style transaction, the account would never have been debited in the first place. With the compensated transaction, you initiate two actions—one to debit the account and one to later credit it.

Note Make no mistake, it would be a rare system that could successfully perform XA-style transactions over the Internet. (I would argue that no system can, but I would be doing just that—starting an argument—so I accept the fact that some systems will try and even succeed in some cases.) Compensation is generally called for. But craft your compensation functions very carefully. Pay attention to details. If you don't, you could be making a bad situation worse by injecting error upon error. It is often not easy to write accurate compensation functions.

Initiating Transactions in Your Workflows

In general, initiating transactions in WF is as simple as dropping a transaction-based activity into your workflow. If you're using transactional activities, however, there is a little more you should know.

Workflow Runtime and Transactional Services

When you use a transaction-based activity in your workflow, two workflow-pluggable services are required. First, because the two out-of-the-box transaction-based WF activities are both decorated with the *PersistOnClose* attribute (mentioned in Chapter 6, "Loading and Unloading Instances"), you must also start the *SqlWorkflowPersistenceService*. If you do not, WF won't crash, but neither will your transactions commit.

Perhaps more interesting for this chapter is the *DefaultWorkflowTransactionService* that WF starts on your behalf when the workflow runtime is started. This service is responsible for both starting and committing your transactional operations. Without such a service, transactions within the workflow runtime are not possible.

Note Although it's beyond the scope of this chapter, you can create your own transactional services. All WF transactional services derive from *WorkflowTransactionService*, so creating your own service is a matter of overriding the base functionality you want to change. In fact, WF ships with a customized transactional service for shared Microsoft SQL Server connections, *SharedConnectionWorkflowTransactionService*. You can find more information at *msdn2.microsoft.com/en-us/library/ms734716.aspx*.

transaction. That is, you initiate the two-phase commit protocol and the database permanently records or removes the data.

However, this is not necessary with the *TransactionScope* activity. If the transaction is successful (no errors while inserting, updating, or deleting the data), the transaction is automatically committed for you when the workflow execution leaves the transactional scope.

Rolling Back Transactions

How about rolling back failed transactions? Well, just as transactions are committed for you, so too will the data be rolled back if the transaction fails. What is interesting about this is the rollback is silent, at least as far as WF is concerned. If you need to check the success or failure of your transaction, you need to incorporate logic for doing so yourself. *TransactionScope* doesn't automatically throw an exception if the transaction fails. It merely rolls back the data and moves on.

Using the *CompensatableTransactionScope* Activity

If an XA-style transaction won't do, you can instead drop the *CompensatableTransactionScope* activity into your workflow and provide for compensated transactional processing. The *CompensatableTransactionScope* activity, like *TransactionScope*, is a composite activity. However, *CompensatableTransactionScope* also implements the *ICompensatableActivity* interface, which gives it the ability to compensate for failed transactions by implementing the *Compensate* method.

Also like *TransactionScope*, the *CompensatableTransactionScope* activity creates an ambient transaction. Activities contained within *CompensatableTransactionScope* share this transaction. If their operations succeed, the data is committed. However, should any of them fail, you generally initiate the compensation by executing a *Throw* activity.

Tip Compensated transactions can enlist traditional resources, such as databases, and when the transaction commits, the data is committed just as if it were an XA-style transaction. However, a nice feature of compensated transactions is that you do not have to enlist an XA-style resource to store data. Sending data to a remote site using a Web service is the classic example for a nonenlistable transactional resource. If you send data to the remote site but later must compensate, you need to somehow communicate with the remote site that the data is no longer valid. (How you accomplish this depends on the individual remote site.)

Throw causes the transaction to fail and calls into execution your compensation handler for your *CompensatableTransactionScope* activity. You access the compensation handler through the Smart Tag associated with the *CompensatableTransactionScope* activity in much the same way you would add a *FaultHandler*.

Creating a Transacted Workflow

I've created an application that simulates an automated teller machine (ATM), one where you provide your personal identification number, or PIN as it's called, and make deposits to or withdrawals from your bank account. Deposits will be embedded in an XA-style transaction, while withdrawals will be compensated if the action fails. To really exercise the transactional nature of the application, I placed a "force transactional error" check box in the application. Simply select the check box and the next database-related operation will fail.

The workflow for this application is a state-based one, and it is more complex than the application you saw in the previous chapter (Chapter 14, "State-Based Workflows"). I've shown the state machine I based the workflow on in Figure 15-1. Most of the application has already been written for you. You'll add the transactional components in the exercises to follow.

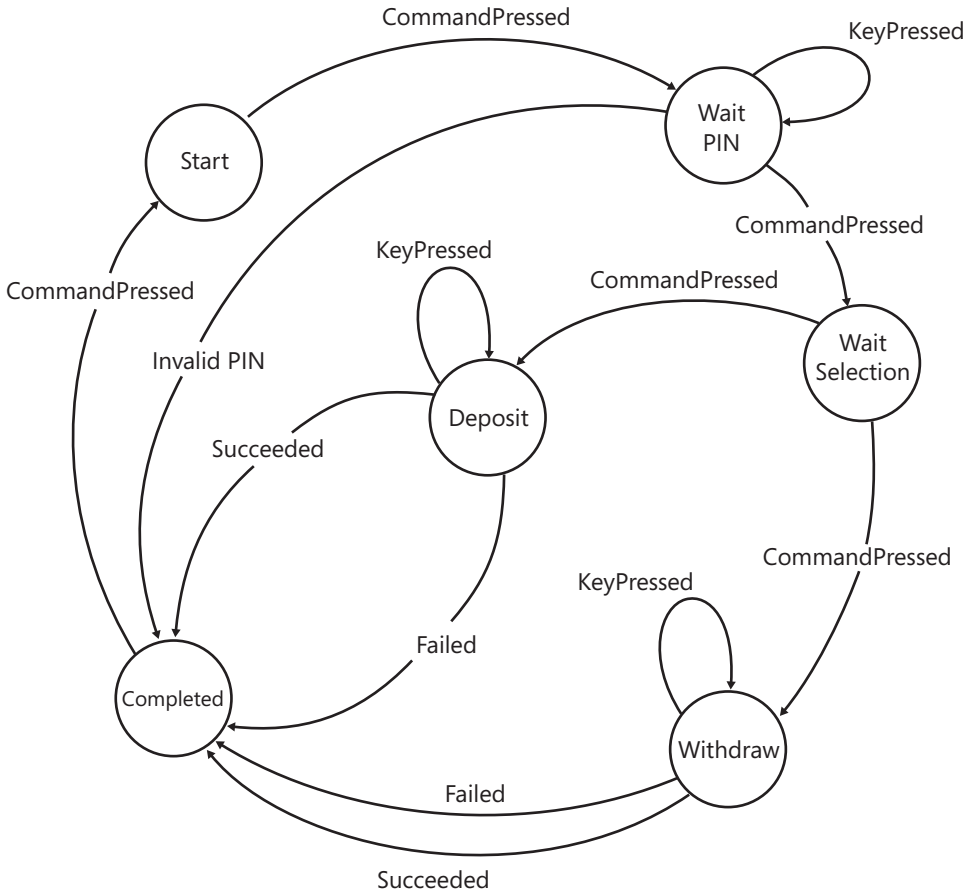


Figure 15-1 The WorkflowATM state diagram

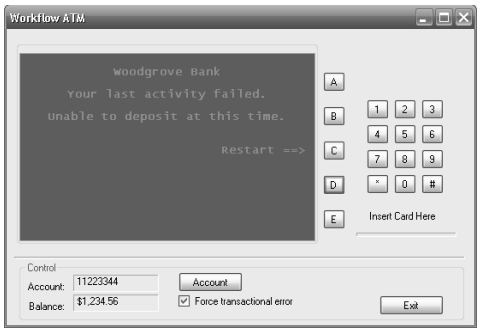


Figure 15-7 The WorkflowATM transaction failed user interface

The application requires a database to fully test WF's transactional capabilities. Therefore, I created a simple database used to store both user accounts with PINs and account balances. Several stored procedures are also available to help with the database interactions. All the stored procedures that involve a database update are required to execute within a transaction—I check *@@trancount*, and if it is zero, I return an error from each stored procedure. What this should prove is that the ambient transaction is being used if I fail to provide any ADO.NET code to initiate my own SQL Server transaction. What this also means is you need to create an instance of the database, but that's easily accomplished because you've learned how to execute queries in SQL Server Management Studio Express in previous chapters. In fact, let's start with that task because we'll soon need the database for application development and testing.

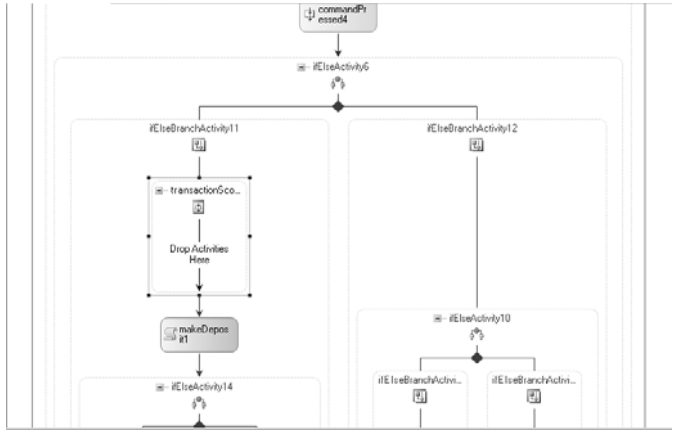
Note Before I forget to mention it, the database creation script creates a single account, 11223344, with the PIN 1234. The application allows you to change accounts and provide any PIN value you like, but unless you use this account (11223344) and this PIN (1234), or create your own account record, you will not be authorized to make deposits or withdrawals.

Creating the Woodgrove ATM databases

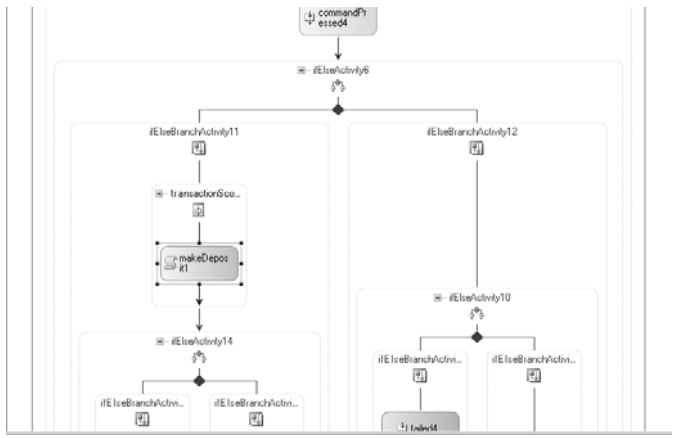
1. You should find the *Create Woodgrove Database.sql* database creation script in the \Workflow\Chapter15 directory. First find it and then start SQL Server Management Studio Express.

Note Keep in mind that the full version of SQL Server will work here as well.

2. When SQL Server Management Studio Express is up and running, drag the Create Woodgrove Database.sql file from Windows Explorer and drop it onto SQL Server Management Studio Express. This opens the script file for editing and execution.



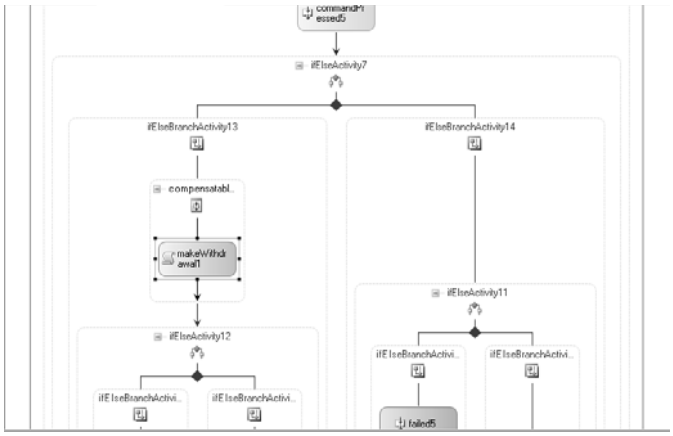
6. Drag *makeDeposit1* from below the transaction scope activity you just inserted, and drop it inside so that the *makeDeposit1 Code* activity will execute within the transactional scope.



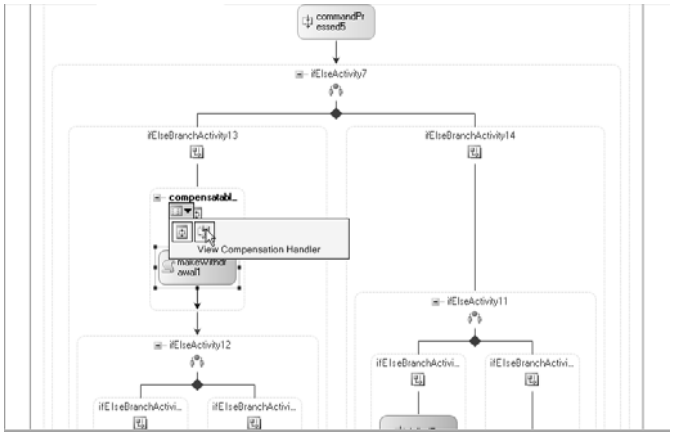
Note Feel free to examine the code contained in the *MakeDeposit* method, which is bound to the *makeDeposit1* activity. The code you find there is typical ADO.NET database access code. An interesting thing to see is that no SQL Server transaction is initiated in the code. Instead, the ambient transaction will be used when the code is executed.

7. Compile the entire solution by pressing F6 or by selecting Build Solution from the Visual Studio Build menu.

8. To test the application, press F5 or select Start Debugging from Visual Studio's Debug menu. The account should already be set. Click the B key to access the PIN verification screen, and then type **1234** (the PIN). Click the C key to verify the PIN and proceed to the activity selection screen.



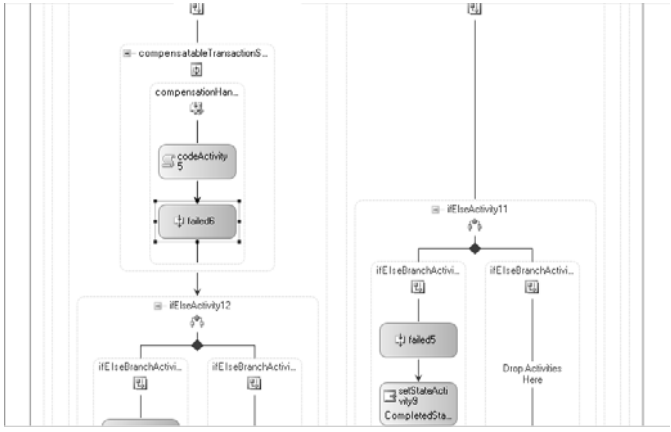
4. However, unlike the deposit functionality, you must provide the compensation logic. The transaction isn't rolled back in the traditional sense. Instead, you need to access the *compensatableTransactionScope1* compensation handler and add the compensating function yourself. To do that, move the mouse over the Smart Tag beneath the *compensatableTransactionScope1* title in the visual workflow designer and click it once to drop the view menu associated with this activity.



5. Click the right icon, View Compensation Handler, to activate the compensation handler view.

```
            // Initiate the SQL transaction
            trans = conn.BeginTransaction();
            cmd.Transaction = trans;

            // Execute the command
            cmd.ExecuteNonQuery();

            // Commit the SQL transaction
            trans.Commit();

            // Pull the output parameter and examine
            CurrentBalance = (decimal)outParm.Value;
        } // try
        catch
        {
            // Rollback... Note we could issue a workflow exception here
            // or continue trying to compensate (by writing a transactional
            // service). It would be wise to notify someone...
            if (trans != null) trans.Rollback();
        } // catch
        finally
        {
            // Close the connection
            if (conn != null) conn.Close();
        } // finally
    } // if
```

9. With the compensation code added to your workflow, return to the visual workflow designer and drop an instance of the custom *Failed* activity into the compensation handler, following the *Code* activity you just entered. Note Visual Studio might reformat and return you to the top-level state activity layout as you return to the visual workflow designer. If this happens, simply double-click the *CmdPressed5* activity in *WithdrawState* once again to access the *compensatableTransactionScope1* activity, and once again select the compensation handler view from its Smart Tag.



10. For the *Failed* activity's error property, type **Unable to withdraw funds**.

Chapter 16

Declarative Workflows

After completing this chapter, you will be able to:

- Understand the primary differences between imperative and declarative workflow models
- Create declarative workflows
- Use the XAML XML vocabulary to build workflows
- Call XAML-based workflows into execution

Many developers probably don't realize that Microsoft Windows Workflow Foundation (WF) is able to execute workflows based on both imperative definitions (using the visual workflow designer) and declarative definitions (workflows defined using XML).

There are advantages to each style. When you create a workflow application using the techniques we've used throughout this book, the workflow model is actually compiled into an executable assembly. The advantage there is the execution is fast, as is the loading of the workflow itself.

But this style is also inflexible. Although there are dynamic capabilities built into WF (which are not covered in this book), in general your workflows remain as you compiled them. If your business process changes, unless you're using declarative rules for making workflow decisions (as discussed in Chapter 12, "Policy and Rules"), you will have to edit your workflow definition, recompile, and redeploy, as well as perform all the associated testing and validation that typically goes with that.

However, the workflow runtime is capable of accepting nearly any form of workflow definition. You just have to write some code to translate the definition you provide into a model the workflow runtime can execute. This is, in fact, precisely what WF does with XML-based workflow definitions.

As you might expect, recording your workflow in an XML format allows for very easy modification and redeployment. Instead of recompiling your workflow in Microsoft Visual Studio, you simply edit the XML-based workflow definition using any XML editor (even Notepad in Windows) and provide that to the workflow runtime as it creates the workflow model. You can even have the best of both worlds by compiling your XML workflow definition using the WF workflow compiler. We'll explore these topics in this chapter.

The CLR namespace that XAML will use is created using two keywords: *clr-namespace* and *assembly=*. For example, if you wanted to use *Console.WriteLine* from within your XAML-based workflow, you would need to create a namespace for the .NET namespace *System*:

```
xmlns:sys="clr-namespace:System;assembly=System"
```

The XML namespace is then a concatenation of the .NET namespace, followed by a semicolon, followed again by the assembly name that hosts the .NET namespace. Keep in mind this holds true for both .NET assemblies and assemblies you create.

But when you create custom assemblies you want to use in XAML-based workflows, it isn't enough to merely create the assembly, provide it with a namespace, and include it with the workflow. The workflow runtime will load the assembly you specify, but it requires the use of an additional attribute—the *XmlnsDefinition* attribute—to actually call objects into execution.

For example, consider this activity you want to use within your XAML-based workflow:

```
public class MyActivity : Activity
{
. . .
}
```

Assuming the class is contained within the *MyNamespace* namespace in the *MyAssembly* assembly, the code to introduce it to the XAML-based workflow is shown in Listing 16-2.

Listing 16-2 Example use of XmlnsDefinition attribute

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Workflow.ComponentModel;
using System.Workflow.Activities;

[assembly:XmlnsDefinition("urn:MyXmlNamespace", "MyNamespace")]
namespace MyNamespace
{
    public class MyActivity : Activity
    {
        // Not a very active activity...!
    }
}
```

The XML to reference this class would then be as follows:

```
xmlns:ns0="urn:MyXmlNamespace"
```

Perhaps surprisingly, the string the namespace uses isn't important. What is important is that the string defining the namespace in *XmlnsDefinition* is the same as the namespace string used in the XML file when that namespace is declared, and the namespace must be unique

6. Open the Program.cs file for editing. Add the following *using* statement to the end of the list of existing *using* statements:

```
using System.Xml;
```

7. Scan through the code and find this line of code:

```
Console.WriteLine("Waiting for workflow completion.");
```

8. So that you actually invoke the XAML-based workflow you just created, add the following lines of code after the line you found in the preceding step:

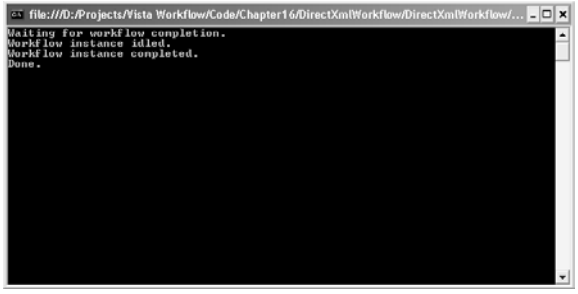
```
// Load the XAML-based workflow
XmlTextReader rdr = new XmlTextReader("Workflow1.xml");
```

9. To create a workflow instance, add this code following the line of code you just inserted:

```
// Create the workflow instance.
WorkflowInstance instance =
   workflowRuntime.CreateWorkflow(rdr);

// Start the workflow instance.
instance.Start();
```

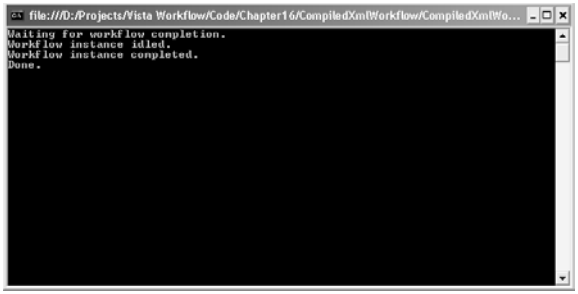
10. Compile the solution by pressing F6, correcting any compilation errors.

11. Execute the application by pressing Ctrl+F5, or F5 if debugging is desired. If you don't run the application from within an open command window, you might have to set a breakpoint in *Main* to see the output.



You can create some rather complex workflows in this manner. However, it is limited in the following ways. First, although there is a mechanism for executing C# code from within the workflow when invoked in this way, it's far preferable to create custom assemblies that, if nothing else, house the code you want to execute. And second, you can't pass parameters into the workflow without first creating a custom root activity to accept them. At some point in the future that might change, but for now this is how WF operates. We'll create a custom assembly for this purpose later in the chapter.

However, an intermediate step is to compile your XAML-based workflow into an assembly you can reference for execution. To compile the XAML-based XML, we'll use the workflow compiler, *wfc.exe*.



Although for this sample we didn't use one, you could also create a *code-beside* file and include code to be compiled into your workflow assembly. By convention, if the XML file uses the .xoml file extension, the code-beside file uses .xoml.cs. The *wfc.exe* tool accepts both a list of .xoml files and an associated list of .xoml.cs files, and it will compile everything together into a single workflow assembly, assuming there are no compilation errors.

Although it's seemingly a minor detail, this detail will prevent you from compiling your workflow. The only difference between the workflow in the first sample and the second sample is the addition of the *x:Class* attribute to the markup. XAML-based workflows lacking the *x:Class* attribute are candidates for direct execution only. Merely adding the *x:Class* attribute means you must compile the XAML-based workflow using *wfc.exe*. If you try to directly execute your XML-based workflow and generate a *WorkflowValidationFailedException* as a result, this is the most likely problem.

The instance in which compiling your XAML-based workflow makes the most sense occurs when you need to pass initialization parameters into your workflow. Directly executed XAML-based workflows can't accept initialization parameters. To demonstrate this, let's return all the way to Chapter 1, "Introducing Microsoft Windows Workflow Foundation," and re-create the postal code validation sample application. However, we'll host the workflow in XML rather than directly in C# code. We'll want to use a code-beside file because we'll have conditions to evaluate and code to execute.

Note You could place the code directly into the XML markup using the *x:Code* element. If this is interesting to you, see *msdn2.microsoft.com/en-gb/library/ms750494.aspx*.

Creating a new workflow application with compiled XML that accepts initialization parameters

1. Again, I've provided two versions of the sample application: a completed version and an incomplete version, both found in \Workflow\Chapter16\. The PCodeXaml application is incomplete, but it requires only the workflow definition. The PCodeXaml Completed version is ready to run. If you'd like to run through the steps here, open the incomplete version. If you'd rather follow along but not deal with coding the solution, then open the completed version. To open either version, drag the .sln file onto an executing copy of Visual Studio and it will open the solution for editing.

11. Turning to the main application, select the Program.cs file for editing. Select it, and scan through the code to find this line:

```
Console.WriteLine("Waiting for workflow completion.");
```

12. Add this code following the line of code you just found:

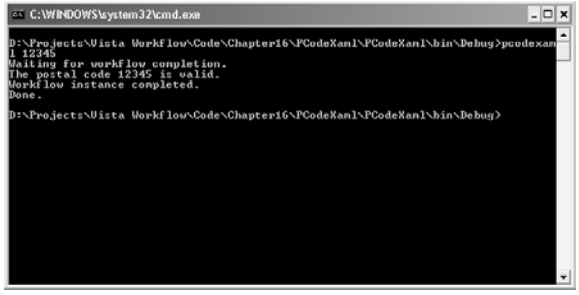
```
// Create the execution parameters
Dictionary<string, object> parms = new Dictionary<string, object>();
parms.Add("PostalCode", args.Length > 0 ? args[0] : "");

// Create the workflow instance.
WorkflowInstance instance =
   workflowRuntime.CreateWorkflow(typeof(Workflow1), parms);

// Start the workflow instance.
instance.Start();
```

13. There is a slight problem with the project as it stands. You referenced the workflow1.dll assembly, but yet there is code to create the *Workflow1* class resident in the code-beside file. This sets up a conflict as there are two instances of a class named Workflow1. Because you want the class defined in the compiled assembly, right-click the Workflow1.xoml and Workflow.xoml.cs files in Solution Explorer and select Exclude From Project to remove them from the compilation process.

14. Compile the solution by pressing F6, correcting any compilation errors.

15. To test the application, we'll use the command prompt window that should still be open from the workflow compilation. Change directories to the Debug directory (or Release if you compiled the application in Release mode). To do so, from the command prompt type **cd bin\Debug** and press Enter (substitute *Release* for *Debug* if you compiled in Release mode).

16. At the command prompt, type **PCodeXaml 12345** and press Enter. You should see the following application output:



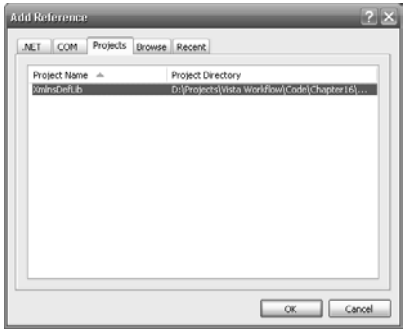
9. And now the secret ingredient—adding the *XmlnsDefinition* attribute. Just before the namespace declaration, insert this code:

```
[assembly: XmlnsDefinition("urn:PrintMessage", "XmlnsDefLib")]
```

> **Note** If you're developing the activity for use in a large application or for distribution to customers or other external users, use a namespace URI that includes your company name, product group, and project or another typically unique value to disambiguate namespaces. This is considered a general best practice when working with XML.

10. Compile the XmlnsDefLib project so that you create a DLL that can be referenced by the workflow compiler. Note whether you compiled the assembly using Debug or Release settings because you'll need to reference the DLL later and the directory in which the DLL resides depends on the compilation setting.

11. Even though we'll be executing a XAML-based workflow, the workflow runtime still needs to access the *PrintMessage* activity we just created. Therefore, add a project reference for XmlnsDefLib to XmlnsDefFlow by right-clicking the XmlnsDefFlow project name in Solution Explorer and selecting the Add Reference option. When the Add Reference dialog box comes up, click the Project tab and select XmlnsDefLib from the list. Click OK.



12. Open a command window as with the previous sample applications.

13. Type **cd \Workflow\Chapter16\XmlnsDefFlow\XmlnsDefFlow** at the command prompt, and press Enter to change directories. The Workflow1.xoml file is now directly accessible in this directory.

14. Type **"C:\Program Files\Microsoft SDKs\Windows\v6.0\Bin\Wfc.exe" workflow1.xoml /r:..\XmlnsDefLib\bin\Debug\XmlnsDefLib.dll** at the command prompt, and press Enter. As before, if you installed the Windows SDK elsewhere, use that directory instead in the command to execute *wfc.exe*. Also substitute *Release* for *Debug* if you compiled the XmlnsDefLib project with Release settings.

15. The workflow compiler should execute without error. Assuming this is so, it produces a dynamic-link library *workflow1.dll* located in the same directory as Workflow1.xoml.

Chapter 17

Correlation and Local Host Communication

After completing this chapter, you will be able to:

- Understand workflow correlation and—where it's necessary—why it's important
- Use the workflow correlation parameters
- Build and use a correlated local communication service

The applications you've seen throughout this book have had a single architecture in common, aside from performing work in workflow instances supported by the Microsoft Windows Workflow Foundation (WF). There has always been a one-to-one correspondence between the application and its workflow instance. If you communicate with a workflow instance, you do so with the assurance that any data that passes between the application and workflow couldn't be confused in any way. One application, one workflow.

Another situation is possible, however, at least when applications and workflows execute in the same AppDomain. Your sole application could invoke multiple copies of the same workflow. What happens to data shipped back and forth then?

Clearly, somebody needs to keep track of which workflow is working with what piece of data. Often we can't mix and match. Once a workflow instance is created and queued for execution, if it is bound to a given data identifier, using that workflow to process information for a different data identifier could present problems with data integrity.

And, in fact, WF provides us with some internal bookkeeping to help prevent data-integrity issues. In WF terms, it's called *correlation*, and WF has very strong correlation support that is also easy to use.

Host and Workflow Local Communication

Before we get into correlation, let's briefly review the entire host and workflow communication process. Chapter 8, "Calling External Methods and Workflows," introduced the *CallExternalEvent* activity and used a local communication service to send data from the workflow to the host application. Chapter 10, "Event Activities," used the *HandleExternalEvent* activity for the reverse process—the host could send data to the workflow.

No matter which way data is passed, we first create an interface. Methods in the interface are destined to be *CallExternalEvent* activities, while events in the interface become

The *CorrelationParameter* Attribute

If you think about situations where a single host application might orchestrate multiple workflow instances, you will probably find that methods and events that pass data also pass some sort of unique identifier. An order processing system might pass a customer ID, or a packaging system might pass a lot number. This type of unique identifier is a perfect candidate for identifying unique instances of data, and in fact, that's precisely what happens.

When you design the methods and events in your communication interface, you design into their signatures a data correlation ID. The data correlation ID doesn't have to be unique for all space and time, like a *Guid*. However, if it isn't a *Guid*, it must be used uniquely for the duration of the workflow instance's execution.

Note Perhaps surprisingly, it isn't an error if you create two correlated workflow instances that run simultaneously using the same correlation parameter value (akin to creating two workflows working with the same customer ID). Correlation merely associates a single workflow instance with a single correlation parameter value. Calling methods or events to exchange data with a workflow created with one correlation parameter value using a different correlation value is where the error lies, and this is where WF helps you keep things straight.

You tell WF what method parameter carries this data correlation ID value by including the *CorrelationParameter* attribute in your interface definition (placed there alongside the *ExternalDataExchange* attribute). WF can then examine the contents of the parameter as the data is moved about the system. If your logic attempts to mix customers or lot numbers, for example, WF will throw the *System.Workflow.Activity.EventDeliveryFailedException*.

This exception is your friend, because it indicates processing logic on your part that could conceivably cross-match data. One customer could be charged for another's purchase, for instance. Obviously, this result is not desirable. If you receive this exception, you need to check your application logic for incorrect logical operation.

The *CorrelationParameter* attribute accepts a string in its constructor. This string represents the name of the parameter used throughout your interface to contain the unique ID. If you elect to rename the parameter for a given method, you can rename it for a selected event or method using the *CorrelationAlias* parameter. You'll read more about this parameter later in the chapter.

The *CorrelationInitializer* Attribute

WF also needs to initialize the correlation token when data communications commence. To facilitate this, you place the *CorrelationIntializer* attribute on the method or event that kicks off the data communication, and there might be more than one. Any attempt to send correlated

Each truck you see is supported by its own workflow instance. (Because it might be difficult to see the trucks in a two-color book, I circled them.) The heart of the workflow asynchronously updates the truck's geographic location. When updates are made, the workflow communicates the new coordinates to the host application, which then visually updates the truck's position in the user interface. Of course, we're simulating the GPS reception, and the simulation moves the trucks at a speed far greater than real vehicles could sustain. (It would be silly to run the sample for four days just to see whether a truck actually made it to New Jersey from California.) The true point of the application is to use correlated workflow instances when communicating data with the host application.

The trucks follow specific routes to their destinations, driving though other cities on the map. You select the truck's route when you click Add Truck, the supporting dialog box for which is shown in Figure 17-2. The routes themselves are stored in an XML file that is read as the application loads. The trip from Sacramento to Trenton, for example, has the truck pass through waypoints Phoenix, Santa Fe, Austin, and Tallahassee.

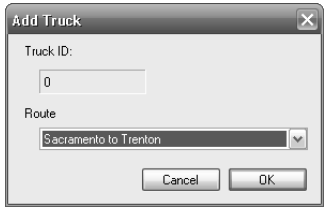


Figure 17-2 The Add Truck dialog box

The application's main program has been completed for you. What remains is completing the service and creating the workflow. We'll begin by creating the service interface.

Adding a correlated communications interface to your application

1. You should find the TruckTracker application in the \Workflow\Chapter17\TruckTracker directory. As usual, I placed two different versions of the application in the Chapter17 directory—an incomplete version for you to work with, and a completed version that you can execute right now. If you want to follow along but don't want to actually perform the steps, open the solution file for the completed version. (It will be in the TruckTracker Completed directory.) If you do want to work through the steps, open the TruckTracker version. To open either solution, drag the .sln file onto an executing copy of Microsoft Visual Studio to open the solution for editing and compilation.
2. The solution contains two projects: TruckTracker (the main application) and TruckService. Looking at the TruckService project in Visual Studio's Solution Explorer window, open the ITruckService.cs file for editing by double-clicking the filename or selecting it and clicking the View Code button on the Solution Explorer toolbar. Note you might have to expand the TruckService project's tree control node to see the project files.

```
    return serviceInstance;
}

public static void
   RegisterDataService(WorkflowTruckTrackingDataService dataService)
{
    string key = String.Format(KeyFormat,
                               dataService.InstanceID.ToString(),
                               dataService.TruckID);
    lock (_syncLock)
    {
        _dataServices.Add(key, dataService);
    } // lock
}
```

4. Once a data service is registered, which occurs in the main application when a new workflow instance is started (one data service per workflow instance), it stores correlated data in the data connector. We need to have a way to retrieve the data. Previously, we used a property, but that won't work for us now because we have to pass in both a workflow instance ID and the correlation value (a truck identifier in this case). To retrieve data, then, add this method following the static registration methods:

```
public string RetrieveTruckInfo(Guid instanceID, Int32 truckID)
{
    string payload = String.Empty;

    string key = String.Format(KeyFormat, instanceID, truckID);
    if (_dataValues.ContainsKey(key))
    {
        payload = _dataValues[key];
    } // if

    return payload;
}
```

5. With that last method, the housekeeping code is complete. Now let's add the methods from the *ITruckService* interface. These follow the data-retrieval method from the preceding step:

```
// Workflow-to-host communication methods
public void ReadyTruck(Int32 truckID, Int32 startingX, Int32 startingY)
{
    // Pull correlated service.
    WorkflowTruckTrackingDataService service =
                                  GetRegisteredWorkflowDataService(
                                  WorkflowEnvironment.WorkflowInstanceId,
                                  truckID);

    // Place data in correlated store.
    UpdateTruckData(service.InstanceID, truckID, startingX, startingY);
```

```
            // Serialize values.
            StringBuilder sb = new StringBuilder();
            using (StringWriter wtr = new StringWriter(sb))
            {
                XmlSerializer serializer = new XmlSerializer(typeof(Truck));
                serializer.Serialize(wtr, truck);
            } // using

            // Ship the data back...
            _dataValues[key] = sb.ToString();

            return truck;
        }
```

8. Save the file.

The entire *TruckServiceDataConnector* class is shown in Listing 17-2. Again, keep in mind that the purpose of this class is to store correlated data coming from the various workflow instances. The data is stored in a *Dictionary* object, the key for which is an amalgam of the workflow instance identifier and the truck identifier. Therefore, the data is keyed in a correlated fashion. The data connector class is a singleton service in the workflow runtime, so we have registration methods the individual workflow instance data services will use to identify themselves and establish their presence as far as data communication is concerned.

Note You might wonder why the data being transferred is in XML rather than the data objects themselves. WF doesn't serialize objects when passed between workflow and host, or the reverse. As a result, copies of the objects are not created (undoubtedly to boost performance). Exchanged objects are passed by reference, so both workflow and host continue to work on the *same* object. If you don't want this behavior, as I did not for this sample application, you can serialize the objects as I have or implement *ICloneable* and pass copies. If this behavior doesn't affect your design, you don't need to do anything but pass your objects back and forth by reference. Keep in mind, though, that your objects will be shared by code executing on two different threads.

Listing 17-2 TruckServiceDataConnector.cs completed

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;
using System.Workflow.Activities;
using System.Workflow.Runtime;
using System.Xml;
using System.Xml.Serialization;
using System.IO;

namespace TruckService
{
   public class TruckServiceDataConnector : ITruckService
   {
```

```
            // Raise the event to trigger host activity.
            if (service != null)
            {
               service.RaiseTruckArrivedEvent(truckID);
            } // if
         }

         // Host-to-workflow events
         public event EventHandler<CancelTruckEventArgs> CancelTruck;

         public void RaiseCancelTruck(Guid instanceID, Int32 truckID)
         {
            if (CancelTruck != null)
            {
               // Fire event.
               CancelTruck(null,
                           new CancelTruckEventArgs(instanceID, truckID));
            } // if
         }

         public event EventHandler<AddTruckEventArgs> AddTruck;

         public void RaiseAddTruck(Guid instanceID,
                                   Int32 truckID,
                                   Int32 routeID)
         {
            if (AddTruck != null)
            {
               // Fire event.
               AddTruck(null,
                        new AddTruckEventArgs(instanceID, truckID, routeID));
            } // if
         }

         protected Truck UpdateTruckData(Guid instanceID,
                                         Int32 truckID,
                                         Int32 X,
                                         Int32 Y)
         {
            string key = String.Format(KeyFormat, instanceID, truckID);
            Truck truck = null;
            if (!_dataValues.ContainsKey(key))
            {
               // Create new truck.
               truck = new Truck();
               truck.ID = truckID;
            } // if
            else
            {
               // Pull existing truck.
               string serializedTruck = _dataValues[key];
               StringReader rdr = new StringReader(serializedTruck);
               XmlSerializer serializer = new XmlSerializer(typeof(Truck));
               truck = (Truck)serializer.Deserialize(rdr);
            } // else
```

```
            _dataConnector = dataConnector;
        } // else

        // Pull the service instance we registered with the connection
        // object.
        return WorkflowTruckTrackingDataService.
                    GetRegisteredWorkflowDataService(instanceID, truckID);
    } // lock
}

public static WorkflowTruckTrackingDataService
   GetRegisteredWorkflowDataService(Guid instanceID,
                                    Int32 truckID)
{
    lock (_syncRoot)
    {
        WorkflowTruckTrackingDataService workflowDataService =
              TruckServiceDataConnector.GetRegisteredWorkflowDataService(
                                                        instanceID, truckID);
        if (workflowDataService == null)
        {
            workflowDataService =
                 new WorkflowTruckTrackingDataService(instanceID, truckID);
            TruckServiceDataConnector.RegisterDataService(
                                                         workflowDataService);
        } // if

        return workflowDataService;
    } // lock
}
```

6. Now add a constructor and destructor:

```
private WorkflowTruckTrackingDataService(Guid instanceID, Int32 truckID)
{
    this._instanceID = instanceID;
    this._truckID = truckID;
}

~WorkflowTruckTrackingDataService()
{
    // Clean up.
    _workflowRuntime = null;
    _dataExchangeService = null;
    _dataConnector = null;
}
```

Note As you might recall from Chapter 8, the destructor is required to break circular links between the service and connector classes. Implementing *IDisposable* won't work for this because the *Dispose* method isn't called when the service is removed from the workflow runtime.

```
            // If we're just starting, save a copy of the workflow
            // runtime reference.
            if (_workflowRuntime == null)
            {
                _workflowRuntime = workflowRuntime;
            } // if

            // If we're just starting, plug in ExternalDataExchange
            // service.
            if (_dataExchangeService == null)
            {
                _dataExchangeService =
                   new ExternalDataExchangeService();
                _workflowRuntime.AddService(_dataExchangeService);
            } // if

            // Check to see if we have already added this data
            // exchange service.
            TruckServiceDataConnector dataConnector =
              (TruckServiceDataConnector)workflowRuntime.
                 GetService(typeof(TruckServiceDataConnector));
            if (dataConnector == null)
            {
                _dataConnector = new TruckServiceDataConnector();
                _dataExchangeService.AddService(_dataConnector);
            } // if
            else
            {
                _dataConnector = dataConnector;
            } // else

            // Pull the service instance we registered with the
            // connection object.
            return WorkflowTruckTrackingDataService.
               GetRegisteredWorkflowDataService(instanceID,
                                                truckID);
        } // lock
    }

    public static WorkflowTruckTrackingDataService
       GetRegisteredWorkflowDataService(Guid instanceID,
                                        Int32 truckID)
    {
        lock (_syncRoot)
        {
            WorkflowTruckTrackingDataService workflowDataService =
               TruckServiceDataConnector.
                 GetRegisteredWorkflowDataService(instanceID, truckID);
            if (workflowDataService == null)
            {
                workflowDataService =
                    new WorkflowTruckTrackingDataService(
```

With the workflow project created, we can now use the *wca.exe* tool to generate the custom activities we'll need to communicate between workflow and host application, and vice versa. We're going to follow the same recipe we used in the Chapter 8 "Creating the communication activities" procedure.

Creating the custom data exchange activities

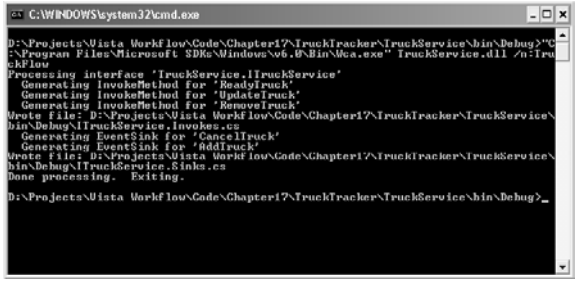
1. Before you begin, make sure you didn't skip step 9 of the earlier "Completing the correlated data service" procedure. The *wca.exe* tool will need a compiled assembly when it executes.
2. Click the Start button and then the Run menu item to activate the Run dialog box.
3. Type **cmd** in the Open combo box control, and click OK to activate the Windows Command Shell.
4. Change directories so that you can directly access the *TruckService* assembly you previously created. Typically, the command to type is as follows:

   ```
   cd "\Workflow\Chapter17\TruckTracker\TruckService\bin\Debug"
   ```

   However, your specific directory might vary.
5. Next execute the *wca.exe* tool by typing the following text at the command-line prompt (including the double quotes):

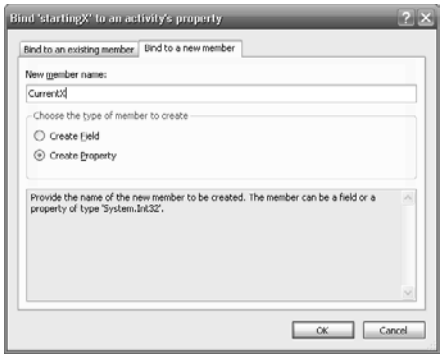
   ```
   "C:\Program Files\Microsoft SDKs\Windows\v6.0\Bin\Wca.exe" TruckService.dll /
   n:TruckFlow
   ```

   Press the Enter key. The tool's output should be similar to the following:

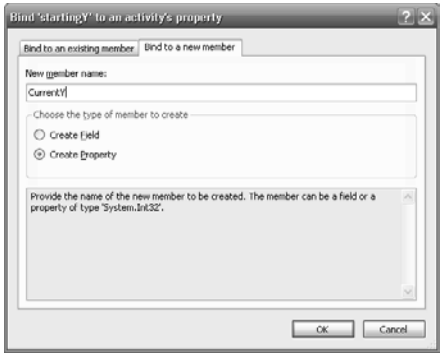


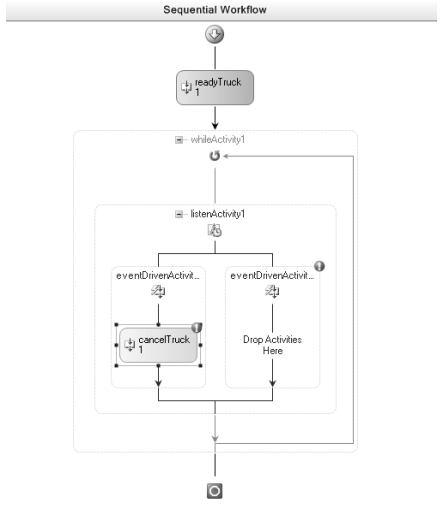
6. The *wca.exe* tool created two files for you, each of which you'll rename and move to the workflow directory. (Renaming isn't required but makes for easier source code tracking, I think.) Type **ren ITruckService.Invokes.cs ExternalEventActivities.cs** at the command prompt, and press Enter to rename the file. This file contains the generated *CallExternalEvent* activities.
7. Because the file we just renamed is a workflow activity, we need to move it from the current directory into the TruckFlow directory for compilation and use. At the command prompt, type **move ExternalEventActivities.cs ..\..\..\TruckFlow** and press Enter.

7. You need to bind the data properties, starting with the *startingX* property. Select the *startingX* property in the Properties pane, and click the browse (...) button to activate the Bind 'startingX' To An Activity's Property dialog box. Select the Bind To A New Member tab, and type **CurrentX** into the New Member Name field. Click OK.



8. Do the same for the *startingY* property. Click the *startingY* property and then the browse (...) button to activate the Bind 'startingY' To An Activity's Property dialog box. Select the Bind To A New Member tab, and type **CurrentY** into the New Member Name field. Click OK.

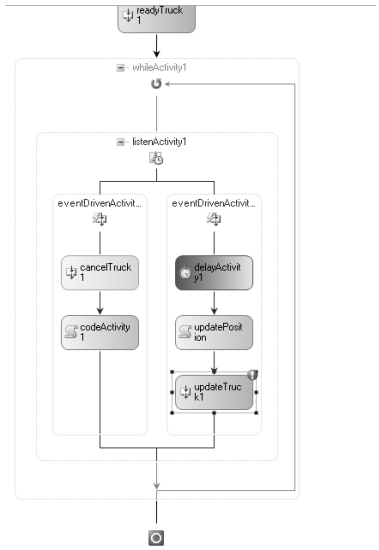




14. You need to establish the correlation token for *cancelTruck1*. To do so, simply drop the arrow associated with *cancelTruck1*'s *CorrelationToken* property and select the **TruckIDCorrelation** option. If the arrow isn't visible, select the *CorrelationToken* property to activate it.

15. *cancelTruck1* needs to have the truck identifier established, so click the browse (...) button in the *truckID* property. If the browse (...) button isn't present, click the property once to activate it as you might have just done for the correlation token property. Once the Bind 'truckID' To An Activity's Property dialog box is showing, select *TruckID* from the list of existing properties and click OK.

21. Return to the visual workflow designer. The *updatePosition Code* activity performs the truck location determination simulation, the result of which needs to be issued to the host application for visual processing. To issue the result to the host application, drag a copy of *UpdateTruck* activity onto the designer's surface and drop it below the *updatePosition* activity.



22. Assign *updateTruck1*'s *CorrelationToken* property to be **TruckIDCorrelation** by selecting it from the selection list, after clicking the down arrow. You might have to select the *CorrelationToken* property with a single mouse click to activate the down arrow.

23. Click the *truckID* property once to activate the browse (...) button so that you can assign the correlated truck identifier to the *updateTruck1* activity. Click the button, select *TruckID* from the list of existing properties in the Bind 'truckID' To An Activity's Property dialog box, and click OK.

30. Scroll down through the source for *Workflow1* and locate the constructor. Following the constructor, add these fields:

```
private bool _cancelTruck = false;
private TruckService.RouteInfo _routes = null;
private TruckService.Truck _myTruck = null;
private TruckService.Route _myRoute = null;
private TruckService.Destination _currentOrigin = null;
private TruckService.Destination _currentDestination = null;
```

31. Following the fields you just added, you need to add two properties used for workflow initialization:

```
public string Routes
{
    set
    {
        // Deserialize route information.
        using (StringReader rdr = new StringReader(value))
        {
            XmlSerializer serializer =
               new XmlSerializer(typeof(TruckService.RouteInfo));
            _routes = (TruckService.RouteInfo)serializer.Deserialize(rdr);
        } // using
    }
}

public string TrackedTruck
{
    set
    {
        // Deserialize truck information.
        using (StringReader rdr = new StringReader(value))
        {
            XmlSerializer serializer =
               new XmlSerializer(typeof(TruckService.Truck));
            _myTruck = (TruckService.Truck)serializer.Deserialize(rdr);
        } // using

        // Assign the truck ID.
        TruckID = _myTruck.ID;

        // Pull the route so we can retrieve the starting coordinates.
        foreach (TruckService.Route route in _routes.Routes)
        {
            // Check this route to see if it's ours.
            if (route.ID == _myTruck.RouteID)
            {
                // Ours, so save...
                _myRoute = route;
                break;
            } // if
        } // foreach

        // Pull origin.
        _currentOrigin = FindDestination(_myRoute.Start);
```

```
double b = (double)_currentDestination.Y -
              (m * (double)_currentDestination.X);

// With slope and intercept, we increment x to find the new y.
Int32 multiplier = (_currentDestination.X - _currentOrigin.X) < 0 ? -1 : 1;
CurrentX += (multiplier * 2);
CurrentY = (Int32)((m * (double)CurrentX) + b);
```

36. Save all open files.

37. The workflow is now complete, but one last task remains for you to execute the application. The main application, TruckTracker, requires a reference to the workflow. So right-click the TruckTracker project, and select Add Reference. From the Projects tab, select TruckFlow and click OK.

38. Now you can compile the entire solution by pressing F6 or by selecting Build Solution from Visual Studio's Build menu option. Correct any compilation errors you encounter.

39. To execute the application, press Shift+F5, or just press F5 to run the application in debug mode. Add a truck by clicking Add Truck, selecting a route, and clicking OK. Add as many trucks as you like. To remove a truck, select it in the listview control and click Cancel Truck.

The application's code isn't very different from what you saw in Chapters 8 and 10. The one difference is when accessing the data for a given truck, you need to pass in the truck identifier for that vehicle. In some cases, that's even provided for you in the event arguments.

If you want to continue to the next chapter, keep Visual Studio 2005 running and turn to Chapter 18, "Invoking Web Services from Within Your Workflows." We've been working with single-computer data long enough...it's time to branch out and hit the Web.

If you want to stop, exit Visual Studio 2005 now, save your spot in the book, and close it. I'm sure you're as excited about using Web services from your workflow as I am, but man doesn't live by work alone. Time for a break!

Chapter 17 Quick Reference

To	Do This
Add correlation to your external workflow data processing	Add the workflow correlation parameters to your communication interface. WF then automatically inspects workflows and data exchanges and throws exceptions if there is a data-correlation problem.
Build a correlated local communication service	However you choose to architect your local communication service, you must first make sure the method or event that is declared as the initializer is called before other correlated data paths are used (or you'll receive an exception). Also, you must return data in a correlated fashion, so saving data for use by using the workflow instance ID and the correlation parameter makes sense.

Chapter 18

Invoking Web Services from Within Your Workflows

After completing this chapter, you will be able to:

- Invoke a Web service from your workflow
- Add and configure Web service proxies
- Manage sessions in your workflow

Speaking personally, something about sending and receiving data over networks struck a chord with me, and I've enjoyed writing communications-based code for years because of it. When I saw that Microsoft Windows Workflow Foundation (WF) had built-in capabilities to both connect to and act as a Web service, well, that was something I just had to dig into a bit deeper.

WF ships with several XML Web service–based activities, the client side of which we'll examine in this chapter. (We'll work with the server-side activities in the final chapter, "Workflow as Web Services.") Before actually working with the *InvokeWebService* activity, I thought I'd describe how Web services work, as we'll need to understand the terminology as we progress through this and the final chapter.

Web Services Architecture

One could argue that any Internet-based server that performs work at your request is a Web service. However, I'll use the term in both this and the next chapter as measured against a more strict interpretation. Here, I'll be referring to an *XML Web service*, which is a service based on the *SOAP protocol*.

Note Originally, "SOAP" was an acronym that stood for the *Simple Object Access Protocol*, which is an XML-based communications framework. However, with the release of the current version of SOAP, the protocol is simply known as the SOAP protocol. You can find the latest SOAP specification here: *http://www.w3.org/TR/soap/*.

The basic idea that drove the creation of the SOAP protocol was the notion that intrinsic data types and even complex data structures could be converted from their native binary format into XML, and that the XML could then be sent over the Internet in much the same way Web

The crucial properties you need to work with when using the *InvokeWebService* activity are shown in Table 18-1.

Table 18-1 Critical *InvokeWebService* Activity Properties

Property	Purpose
MethodName	Gets or sets the method to be invoked when the activity is executed. This property represents the method name of the remote method you want to call.
ProxyClass	Gets or sets the type name for the proxy class. Either you can provide this yourself or WF will help you by creating a proxy class for you when you drop the activity into your workflow.
SessionId	Gets or sets the session to be used. You use this mechanism to tie different XML Web service invocations together. I discuss this property further in the "Working with Sessions" section.
URL	Gets or sets the URL to be used for communicating with the XML Web service. The URL itself is stored in your workflow project's Settings property bag, although you can easily change it using the Properties pane for your *InvokeWebService* activity instance.

In addition to these properties, you might also from time to time need to provide event handlers for the *Invoking* and *Invoked* events. You'll see why you might need these when you configure your proxy in this chapter's "Configuring the Proxy" section.

Using the *InvokeWebService* activity is simply a matter of dropping it into your workflow. When you do, Microsoft Visual Studio helps you configure the activity by requesting the server information so that it can retrieve the WSDL and create the proxy. If you want to create the proxy yourself, simply cancel the dialog box Visual Studio pops up. If you want Visual Studio to create the proxy, you'll recognize the dialog box as the same dialog box Visual Studio uses for selecting a Web reference for a typical Visual Studio project, as you see in Figure 18-1.

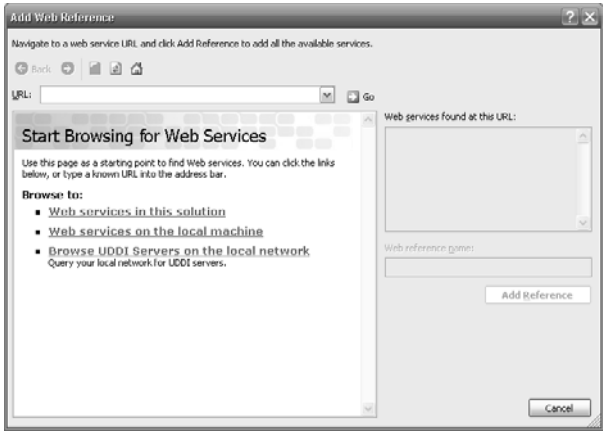


Figure 18-1 Visual Studio's Add Web Reference dialog box user interface

Working with Sessions

Sessions, in Web parlance, are sets of connected request-response pairs. That is, if you make a request of a Web resource in one invocation, you expect the next invocation to keep track of the previous invocation. This is definitely not how the HyperText Transfer Protocol (HTTP) works under the covers. HTTP was developed to support a single request-response pair. Any knowledge of prior dealings with the server are completely forgotten.

However, as Web users, we demand that Web request-response pairs be remembered. The most obvious example is the Web-based shopping cart. When you place orders for goods over the Internet and those goods are collected into a virtual shopping cart, you expect the things you selected will be available for checkout at a later time. How do we reconcile the disparity between what HTTP expects to support and what we, as users, demand?

The answer is Web applications maintain *session state*. When you begin using a Web application, you initiate a session. That session is tracked until you cease using the Web application, either by logging out of the application (if it's secured) or by simply closing your browser.

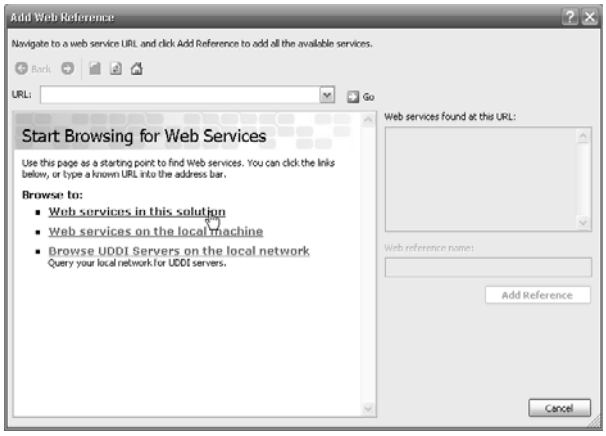
A common way to track session state is through the use of a *cookie*. Cookies are auxiliary data containers that ride along with your HTTP request-response in the HTTP packet, along with the actual payload (SOAP-based XML, HTML, or whatever). On the server, they're typically extracted into memory and are accessible by your Web application for whatever reason you're tracking session information. On the client, they're commonly stored as files. When you access the given Web resource, any cookies destined for that resource are retrieved from the cookie folder and shipped back to the server. Although this process is not the only way to maintain session information, it is a common scenario.

XML Web services are a little different, however. For one thing, they're not invoked using a browser, at least not when using WF. (Other technologies most certainly do invoke XML Web services from a browser.) Because of this, the cookies associated with an XML Web service are not file-based but merely exist in memory or on the wire when transferred back and forth to the server. They're not stored in files on the client.

Another way they differ is that they might not always be present, so you can't depend on their existence. If the XML Web service isn't configured to send session-based cookies, session continuance isn't possible.

Tip If you're writing your XML Web service using the .NET Framework and you want sessions to be in effect, be sure to enable session management using the *EnableSession* key in the *WebMethod* attribute identifying your Web-based methods. By default, the session management subsystem is deactivated in .NET XML Web service operations.

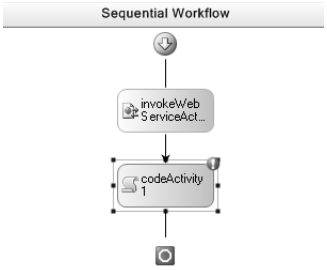
4. Although you might expect to see the behavior you've seen throughout this book—a rounded rectangle appearing on the designer's surface—what you are presented with instead is the Visual Studio Add Web Reference dialog box. Because you've not identified a previous Web reference suitable for use in this project, let's complete the addition of the Web reference and the initial *InvokeWebService* activity property settings. Click the Web Services In This Solution link to proceed.



5. The only XML Web service present in the solution is for the QuoteService. Click the QuoteService link to proceed.

Note QuoteService prepopulates a *Dictionary* object with the three stock symbols used in Chapter 10 along with their initial values. Then it places the *Dictionary* object in the ASP.NET cache for use when the XML Web service is invoked. (See Global.asax for the code that initializes the cache.)

10. The *InvokeWebService* activity is now fully configured for this workflow, so drag a copy of the *Code* activity onto the visual workflow designer's surface and drop it below *invokeWebService1*. Assign its *ExecuteCode* property to be **DisplayQuoteValue**.

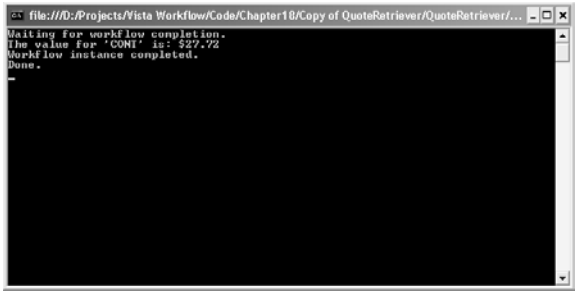


11. Visual Studio adds the *DisplayQuoteValue* method and switches you to the code editor. Add this code to the *DisplayQuoteValue* method Visual Studio just added:

```
if (StockValue >= 0)
{
    // Found the stock value.
    Console.WriteLine("The value for '{0}' is: {1}",
                          Symbol, StockValue.ToString("C"));
} // if
else
{
    // Unknown stock.
    Console.WriteLine("Stock symbol '{0}' is unknown...please try" +
                      " again using a valid stock symbol.", Symbol);
} // else
```

12. Compile the entire solution by pressing F6, or by selecting Build Solution from Visual Studio's Build menu. Correct any compilation errors that might appear.

13. The main application expects a stock ticker symbol as the only command-line argument. The XML Web service has information for only three ticker symbols: CONT, LITW, and TSPT. Therefore, these are the only three values you can pass into the application. If you execute the application and pass in the CONT stock symbol, the program's output appears something like the following:



Chapter 19

Workflows as Web Services

After completing this chapter, you will be able to:

- Understand how the various workflow activities designed to expose your workflow as an XML Web service are used
- Understand what is required to host workflow in ASP.NET
- See how faults are handled in XML Web service–based workfow
- Configure your XML Web service–based workflow for various conditions

In the previous chapter, "Invoking Web Services from Within Your Workflows," you saw how to call XML Web services from your client-side workflow using the *InvokeWebService* activity that Windows Workflow Foundation (WF) provides for this purpose. The XML Web service in that chapter's sample application, however, was a typical ASP.NET XML Web service—nothing special.

In this final chapter, you'll learn how to take workflow processes and automatically expose the workflow as an XML Web service for clients to consume. It's not as simple as creating a workflow assembly library and referencing that from a Web service project, but then again, it's not difficult to do once you understand some essential concepts and see it accomplished in a sample application.

Note This chapter focuses on integrating WF into ASP.NET for use as an XML Web service. But there are many critical issues you should be aware of when exposing XML Web services, not the least of which is security. A full discussion of security is well beyond what I can present here, but this link should get you started: *http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetsec/html/THCMCh12.asp*. If you're going to expose your workflow as an XML Web service, I strongly encourage you to review ASP.NET security best practices, especially practices that are centered around XML Web services.

Exposing a Workflow as an XML Web Service

Part of the reason you can't directly execute the workflows you've seen throughout the book in an ASP.NET environment is that the workflow runtime, by default, executes workflow instances asynchronously. In fact, that's a particularly valuable feature when working with workflow in non-Web applications.

But in a Web-based scenario, this presents a problem. If an ASP.NET request comes in, whether it's for an XML Web service or even an ASP.NET Web page, the workflow instance

begins execution and the runtime returns control to ASP.NET. Your XML Web service or ASP.NET page immediately continues preparing output and likely finishes before the workflow instance completes. Because the workflow instance is asynchronous, executing in parallel with your ASP.NET application, your ASP.NET code could easily finish and return a response to the caller without ever completing the workflow processing.

Tip Properly executing workflow instances within ASP.NET Web pages really calls for ASP.NET asynchronous Web pages, the discussion of which is beyond the scope of this book. However, this link gives you some details: *http://msdn.microsoft.com/msdnmag/issues/05/10/WickedCode/*.

This behavior presents at least two challenges for us. First, we need to disable, or at least work around, the asynchronous execution of our workflows. We need for them to execute synchronously such that they use the same thread our page or XML Web service is using so that the Web application doesn't return a response to the caller until we're finished. Of course, this leaves open the question of long-running workflows, which is the second challenge we'll need to address.

The challenge long-running workflows present is centered around the nature of Web-based applications themselves. From the last chapter, you know that Web applications are by nature stateless. Requests made milliseconds apart are completely unaware of each other unless we build a framework to provide that awareness. Web applications also execute on Web servers, which are normally very expensive systems designed to serve a great many clients. If a workflow takes a long time to complete, it ties up the Web server and reduces the application's scalability (the ability to serve a greater number of client requests).

The solution to both state management and long-running workflows is persistence. If your workflow process completes over the span of more than one Web-based invocation (ASP.NET page request or XML Web service), you must persist the workflow instance and reload it during the next execution cycle. This is also the reason why I mentioned regenerating the session-state cookie in the previous chapter's "Long-Running XML Web Services" section, because the client must also be aware of this possibility and make allowances for more than a single request-response.

Internet Information Services (IIS) in particular, however, is quite adept at conserving system resources. In a typical client application, and in fact in every application you've seen in the book so far, the workflow runtime is started when the application begins execution and runs throughout the lifetime of the application. IIS, though, recycles applications to reclaim server resources. This means two things to us as ASP.NET programmers.

First, we have to somehow decide where and how to start the workflow runtime. Different requests for the same ASP.NET application are processed on different threads, but they

Note To learn more about *HttpModules* and how they enhance the ASP.NET HTTP request-response pipeline, see *http://msdn2.microsoft.com/en-us/library/ms178468.aspx*.

Listing 19-1 provides you with a basic Web.config file, common to many WF-based applications hosted in ASP.NET environments. Note it contains information specific to hosting WF in your ASP.NET application—you still need to add any ASP.NET configuration that's appropriate for your application (such as to the *<system.web />* section).

Note Listing 19-1 is valid for the release of WF current at the time of this writing. However, if new releases become public, the version numbers and potentially the public key token values in your configuration files will require updating. You might also need to update the connection string to match your SQL Server installation.

Listing 19-1 Web.config with workflow extensions

```
<?xml version="1.0"?>
<configuration xmlns="http://schemas.microsoft.com/.NetConfiguration/v2.0">
  <configSections>
    <section name="WorkflowRuntime" type=
      "System.Workflow.Runtime.Configuration.WorkflowRuntimeSection,
      System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,
      PublicKeyToken=31bf3856ad364e35"/>
  </configSections>
  <WorkflowRuntime Name="WorkflowServiceContainer">
    <Services>
      <add type=
        "System.Workflow.Runtime.Hosting.ManualWorkflowSchedulerService,
        System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,
        PublicKeyToken=31bf3856ad364e35"/>
       <add type=
        "System.Workflow.Runtime.Hosting.SqlWorkflowPersistenceService,
        System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,
        PublicKeyToken=31bf3856ad364e35"/>
      <add type=
        "System.Workflow.Runtime.Hosting.DefaultWorkflowTransactionService,
        System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,
        PublicKeyToken=31bf3856ad364e35"/>
    </Services>
  </WorkflowRuntime>
  <appSettings/>
  <connectionStrings>
    <add name="MyPersistenceDB"
       connectionString=
"server=(local)\SQLEXPRESS;database=WorkflowStore;Integrated Security=true"
       />
  </connectionStrings>
  <system.web>
    . . .
    <httpModules>
```

a method to act as the *MethodName*, the method's parameters automatically appear in the activity's Properties pane for you to bind using the property binding dialog box you've used throughout the book. You'll see this when working with the sample application for this chapter.

Using the *WebServiceOutput* Activity

The *WebServiceOutput* activity completes the XML Web service's processing by returning the value identified by the *MethodName* in the *WebServiceInput* activity. For this reason, you must assign the *WebServiceOuput*'s *InputActivityName* property to be one of the *WebServiceInput* activities present in your workflow. Failing to assign the input activity name results in both validation and compilation errors.

WebServiceOutput has a single method parameter to bind—that being the return result if any is indicated by the method in the interface bound to the *WebServiceInput* activity this output activity is tied to. If the method returns *void*, no return value binding is possible. The *WebServiceOutput* activity then is reduced to signaling the end of the workflow processing for this Web method invocation.

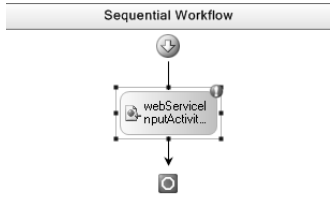
Interestingly, you can have multiple outputs for a single input, as might happen if the output activities are placed within separate *Parallel* activity execution paths, or in different branches of an *IfElse* activity. Different paths through your workflow might result in different outputs. What you cannot do is have multiple output activities in the same execution path. (The same holds true for the *WebServiceFault* activity as well.) If WF determines that more than one *WebServiceOutput*, or *WebServiceFault* combined with a *WebServiceOutput*, is in the same execution path, WF invalidates the activities and you need to correct the execution logic by moving or removing the offending activity or activities.

Using the *WebServiceFault* Activity

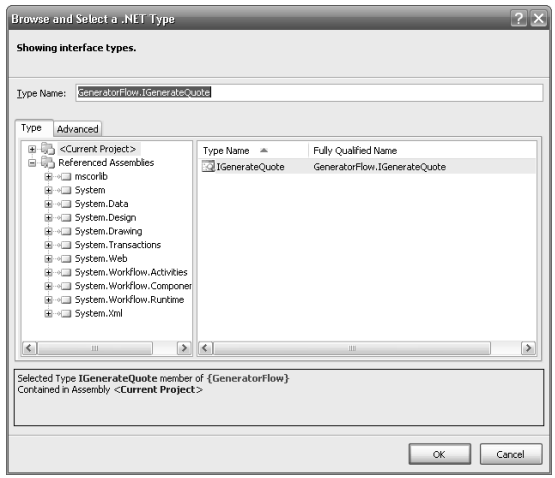
The *WebServiceFault* activity is closely related to the *WebServiceOutput* activity. Both indicate the termination of workflow processing for the particular invocation they happen to be supporting. And, as it turns out, *WebServiceFault* is used just as *WebServiceOutput* is used.

WebServiceFault is also tied to a single *WebServiceInput* activity. Like its cousin the *WebServiceOutput* activity, *WebServiceFault* has a single output property you must bind, in this case *Fault*. *Fault* is bound to a field or property based on *System.Exception* that represents the exception to report back to the client. Ultimately, *Fault* is translated into a *SoapException* and sent over the wire to the client. But any exception you care to bind is acceptable. ASP.NET translates the exception you provide into *SoapException* automatically.

7. Drag an instance of *WebServiceInput* from the Visual Studio Toolbox, and drop it into your workflow definition.



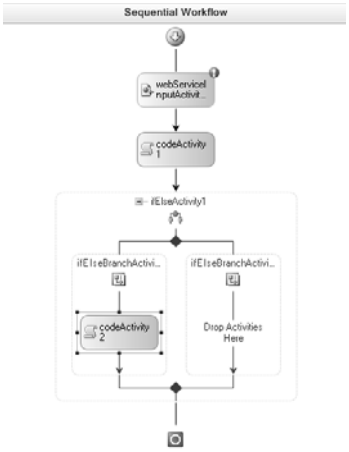
8. Now let's set *webServiceInputActivity1*'s properties. To begin, click the *InterfaceType* property to activate the browse (...) button. Click the browse button, which activates the Browse And Select A .NET Type dialog box. *GeneratorFlow.IGenerateQuote* should already be inserted into the Type Name field because it belongs to the current project and is the only interface available. Click OK.



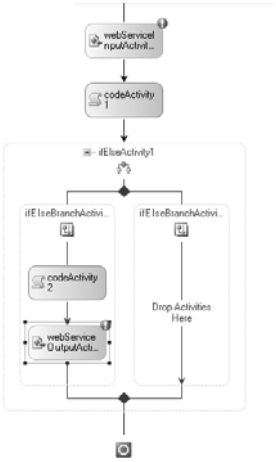
9. Click the *IsActivating* property to activate the down arrow. Click the down arrow, and select *True* from the list of available options.

property and press Enter. Once Visual Studio again adds the event handler for you, return to the visual workflow designer.

16. Drag a copy of the *Code* activity onto the designer's surface, and drop it into the left *ifElseActivity1* branch. Enter **RecordStockValue** as its *ExecuteCode* property, and return to the visual workflow designer once Visual Studio adds the event handler.



17. Now drag an instance of the *WebServiceOutput* activity, and drop it below the *Code* activity you just inserted.



18. Assign *webServiceOutputActivity1*'s *InputActivityName* to be *webServiceInputActivity1* by clicking the *InputActivityName* property once to activate the down arrow, and then by clicking the down arrow to select the *webServiceInputActivity1* activity. Note *(ReturnValue)* is added to the list of *webServiceOutputActivity1* properties.

23. The workflow is now complete from a visual design perspective, so open the Workflow1.cs file for code editing. Select Workflow1.cs in Solution Explorer, and click the View Code toolbar button.

24. The first bit of code to add will be a pair of *using* statements that allow us to access some ASP.NET structures. Add these to the end of the existing list of *using* statements:

```
using System.Web;
using System.Web.Caching;
```

25. Next let's complete the *CreateStocks* event handler. Scroll down until you find *CreateStocks*, and add the following lines of code to that method:

```
System.Collections.Generic.Dictionary<string, decimal> stockVals =
    HttpContext.Current.Cache["StockVals"] as
    System.Collections.Generic.Dictionary<string, decimal>;
if (stockVals == null)
{
    // Create and cache the known stock values.
    stockVals =
       new System.Collections.Generic.Dictionary<string, decimal>();
    stockVals.Add("CONT", 28.0m);
    stockVals.Add("LITW", 22.0m);
    stockVals.Add("TSPT", 24.0m);

    // Add to the cache.
    HttpContext.Current.Cache.Add("StockVals", stockVals, null,
                                  Cache.NoAbsoluteExpiration,
                                  Cache.NoSlidingExpiration,
                                  CacheItemPriority.Normal, null);
} // if
```

26. Scroll down further and locate the *UpdateMarketValues* event handler. To the *UpdateMarketValues* handler, add the market update simulation code:

```
// Iterate over each item in the dictionary and decide
// what its current value should be. Normally we'd call
// some external service with each of our watch values,
// but for demo purposes we'll just use random values.
//
```

29. The workflow itself is now complete. Save all open files, and compile the workflow by pressing Shift+F6 or by selecting Build GeneratorFlow from the Visual Studio Build menu. Correct any compilation errors you encounter before moving on.

The workflow is now ready to be placed into an ASP.NET setting so that external clients can connect to the service and request stock values. Although you could create a new ASP.NET application and perform all the configuration magic yourself, the WF team kindly provided you with a superb feature that builds the ASP.NET project for you, error free. It could hardly be easier to use, too.

Creating the ASP.NET Web application

1. This is so easy they should give the WF developer who came up with the idea a pay raise. Simply right-click the GeneratorFlow project name in Solution Explorer, and select Publish As Web Service. Visual Studio will grind for the briefest of moments and then display a confirmation dialog box. Click OK to dismiss this confirmation dialog box.

2. Because that was entirely too easy, let's rename the .asmx file. (The name Visual Studio selects is a bit overwhelming.) Right-click the newly created GeneratorFlow.Workflow1_WebService.asmx file's name in Solution Explorer, and select Rename. Change its name to **QuoteService.asmx**.

3. Just to be sure, you can press Shift+F6 or select Build Web Site from the main Build menu. This step isn't required, but it will precompile the Web application so that when you reference it to retrieve its WSDL you'll save a bit of time at that point. If there were any errors, you could correct them, but you should find none. If you receive schema warnings for the values Visual Studio placed for you into the Web.config file, ignore those.

The final task is to return to the original program file, add a Web Reference to its project, and test the XML Web service. That's next.

Creating the XML Web Service client application

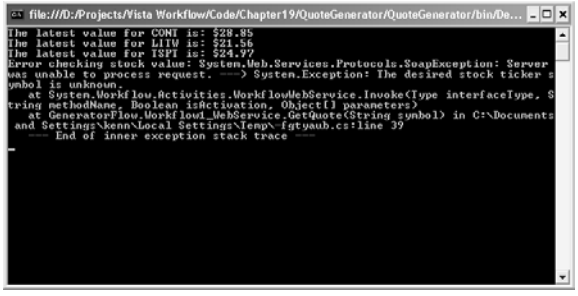
1. Right-click the QuoteGenerator project name in Solution Explorer, and select Add Web Reference.

6. Look for the *Main* method. To the *Main* method, add this code:

```
// Create the proxy.
QuoteGenerator.localhost.Workflow1_WebService ws =
   new QuoteGenerator.localhost.Workflow1_WebService();

try
{
    // Call the service a few times to
    // test its logic.
    decimal val = ws.GetQuote("CONT");
    Console.WriteLine("The latest value for CONT is: {0}",
                                              val.ToString("C"));

    val = ws.GetQuote("LITW");
    Console.WriteLine("The latest value for LITW is: {0}",
                                              val.ToString("C"));

    val = ws.GetQuote("TSPT");
    Console.WriteLine("The latest value for TSPT is: {0}",
                                              val.ToString("C"));

    // Error test
    val = ws.GetQuote("ABC");
    Console.WriteLine("The latest value for ABC is: {0}",
                                              val.ToString("C"));
} // try
catch (Exception ex)
{
    Console.WriteLine("Error checking stock value: {0}", ex.Message);
} // catch
```

7. Compile the solution by pressing F6 or by selecting Build Solution from the Visual Studio Build menu. Correct any compilation errors.

8. Execute the application by pressing Shift+F5, or simply F5 for debugging mode. The output should appear as you see here. If the output scrolls by too quickly and the console window disappears before you have a chance to see the output, set a breakpoint in the *Main* method and step through until you see all the output.



With this application, your quick tour of WF is complete.

Kenn Scribner

Kenn is a software developer who happens to write books as an excuse to learn new technologies. There is nothing quite like a deadline to get those creative neurons firing. He's been working with computers since he built a breadboarded Z-80 in college a few too many years ago. (Happily the Z-80 survived the encounter.) While he fondly remembers writing 6502 assembly language on his Apple][, he's quite happy writing complex C# applications today, both for the Web and for the desktop. This is his fifth computer book—his second solo effort—and he truly hopes you'll enjoy reading it and learning Windows Workflow Foundation. His personal Web site is *www.endurasoft.com*.