

# SharePoint 2010 Workflows IN ACTION

Phil Wicklund  
**Phil Wicklund**

SAMPLE CHAPTER





***SharePoint 2010  
Workflows in Action***

Phil Wicklund

**Chapter 12**

Copyright 2011 Manning Publications

# *brief contents*

---

## **PART 1 INTRODUCTION TO SHAREPOINT WORKFLOWS .....1**

- 1 ■ SharePoint workflows for your business processes 3
- 2 ■ Your first workflow 35

## **PART 2 NO-CODE SHAREPOINT WORKFLOWS .....63**

- 3 ■ Custom Designer workflows 65
- 4 ■ Task processing in SharePoint Designer workflows 93
- 5 ■ Advanced SharePoint Designer workflows 112
- 6 ■ Custom Visio SharePoint workflows 142
- 7 ■ Custom form fundamentals 157

## **PART 3 CUSTOM-CODED SHAREPOINT WORKFLOWS .....183**

- 8 ■ Custom Visual Studio workflows 185
- 9 ■ Forms in Visual Studio workflows 214
- 10 ■ Workflows and task processes 252
- 11 ■ Custom workflow activities and conditions 271
- 12 ■ A bag of workflow developer tricks 304

# 12

## *A bag of workflow developer tricks*

---

### ***This chapter covers***

- Handling faults and debugging workflows
- Versioning workflows
- Using workflow event receivers
- Receiving external events with pluggable services
- Working with the workflow object model

Everything in the first 11 chapters of this book was fundamental to a SharePoint workflow developer. We discussed out-of-the-box SharePoint workflows, other non-developer techniques with SharePoint Designer and Office Visio, workflows with Visual Studio, and custom forms. With chapter 12, you round out the SharePoint developers skills with a few key techniques.

These techniques include how to debug and handle faults in your workflows, version workflows, set up event receivers, send and receive external events, and work with the SharePoint workflow object model. Debugging and exception handling are an obvious must, but versioning is not well understood. If you don't properly version your workflows, idle workflow instances might break when they resume execution.

Why study event receivers in a workflow book? Event receivers can save you time if they fit the business requirements. Event receivers are typically easier and faster to create for smaller, one-time processing than their workflow counterparts and a few new events for workflows.

Sending and receiving events to and from external sources is often a must for larger business processes. This is handled through a new feature in SharePoint 2010 called pluggable workflow services. With the use of two activities and a new class called `SPWorkflowDataExchangeService`, you can easily communicate with your organization's line of business applications.

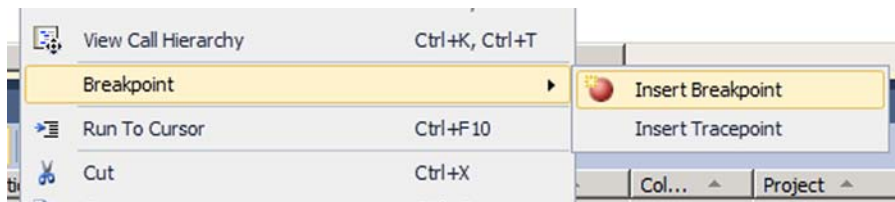
Finally, every developer should take a peek at the workflow object model found in the `Microsoft.SharePoint.Workflow` namespace. You never know when you may need to programmatically start or stop a workflow or perhaps retrieve a workflow's tasks or history.

## 12.1 Fault handling and debugging workflows

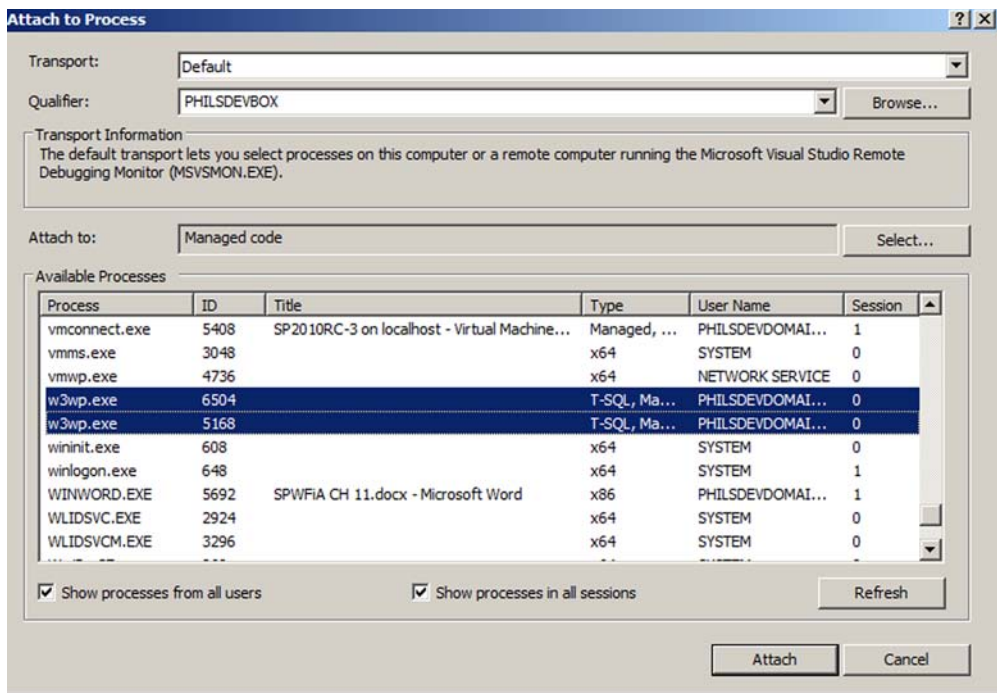
If you don't properly handle exceptions in the workflow and an error occurs, you'll get the dreaded Error Occurred string in the workflow status, without a clue as to what went wrong. You're left with debugging your workflow and, if you don't know how to debug, you're up a creek.

Let's start with what's easy—debugging. You debug your workflows almost the same way as any other .NET application you build. Within your workflow's code view, right-click on the line of code where you want to start debugging and choose **Breakpoint > Insert Breakpoint** (figure 12.1). Also note that you can debug the activities on the templates. To do this, right-click the activity you want to debug and select **Breakpoint > Insert Breakpoint** again.

The next thing to do is attach the Visual Studio debugger to the `w3wp.exe` SharePoint process (figure 12.2). In Visual Studio, click the **Debug** menu dropdown and select **Attach to Process**. Scroll down and select the `w3wp` process and click **Attach**. If there are multiple `w3wp` processes, select all that show up. If you don't see the process, navigate to the SharePoint site to start the process, then go back to the process list and click the **Refresh** button. Also ensure that **Show processes in all sessions** is checked. After you attach, you can start your workflow, and Visual Studio will automatically step into the debugger when the line of code or activity hits.



**Figure 12.1** You debug a workflow in almost the same way you debug any .NET program. Add a breakpoint and attach the debugger to the SharePoint process.



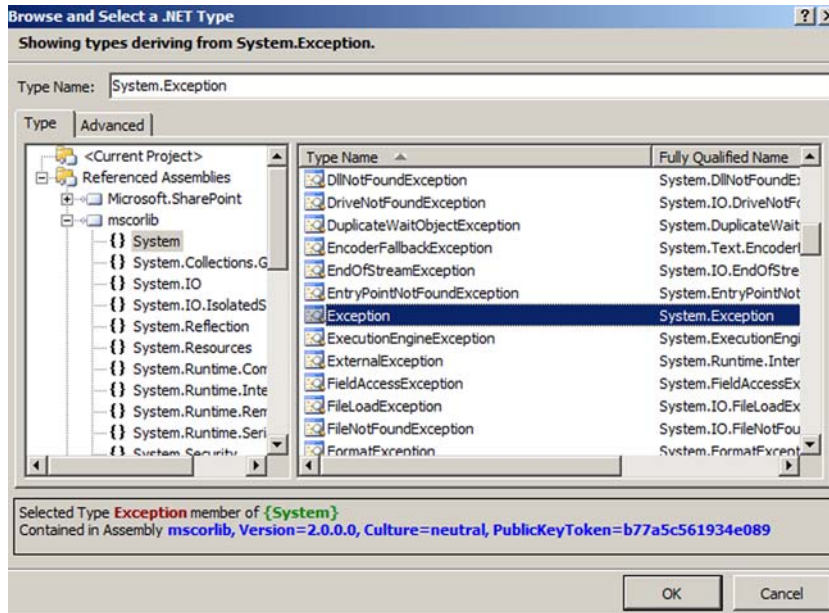
**Figure 12.2** You must attach to the w3wp SharePoint process to debug your SharePoint workflows.

Handling exceptions is a bit different than your standard .NET application. When you're in a workflow template, there's no obvious place to add a Try/Catch block. Many developers never handle exceptions and spend a good deal of time debugging and trying to figure out where the error occurred. The better approach is to use the FaultHandler activity at the root of your workflow (green arrow) as well as all your composite activities (sequence, parallel, IfElse, and so on).

Within the FaultHandler activity, you can insert activities that handle the error in an appropriate way. With SharePoint, it's popular to log the error, location, and stack trace into the workflow's history list. This makes determining the error and location easier.

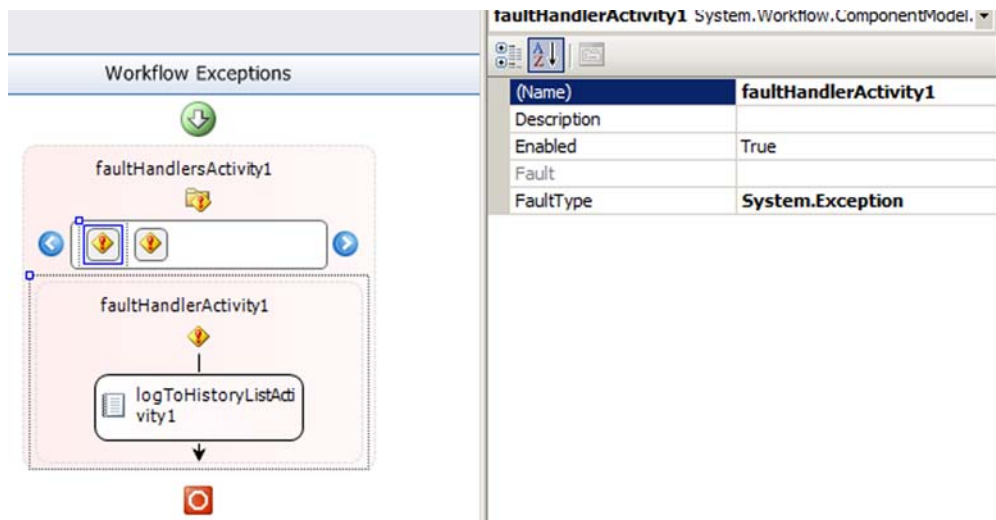
At the least, every workflow should have a Fault Handler set up at the root of the workflow template. To do this, click the dropdown next to the green arrow and select View Fault Handlers.

Within the fault handlers section, you can drag and drop one or more FaultHandler activities. Each activity has a property called FaultType, and you need to set this property to the type of exception you want to handle. To handle all exceptions, set it to System.Exception, which is the most generic exception (figure 12.3). Alternatively, you could set it to a custom exception to handle errors that are specific to your workflow. This is the best approach; it makes debugging easier because you'll know when and why your custom exceptions are being raised.



**Figure 12.3** When you add a fault handler, you need to specify the exception to be handled.

After you have specified the exception to be handled, you should add actions to react to the error appropriately. It's helpful to use the Log to History List activity to log a more descriptive error and the stack trace to help with debugging (figure 12.4).



**Figure 12.4** After you specify the exception to be handled, log an informative error description and, possibly, the stack trace to help with debugging. You can do this by dropping the Log to History List activity within the fault handler.

## 12.2 Versioning workflows

You built this compelling Visual Studio workflow and deployed it into production. But, after a few months, the business requests a small change to the workflow. You go back into the workflow code, add a few activities to fulfill the request, and redeploy the workflow into production. To your shock, all the workflows start breaking! You're frantic because you're certain you adequately unit-tested the changes and can't figure out what might be going wrong. You didn't version your workflow.

Workflow versioning is an important technique. When a workflow goes idle, the state of the workflow is saved into the database. This saving of a workflow's state is called hydration. When the workflow resumes, the state is dehydrated out of the database, and the workflow starts processing again. Versioning is important because, if you change the assembly while the workflow is hydrated (saved in the database), there's no guarantee that, when the workflow is dehydrated, it will match the construct of the new assembly. If it doesn't match the construct upon deserialization, the workflow will break. Changes like adding or removing activities and changing property values may necessitate a new workflow version. The best practice is to create a new version every time rather than deploying the assembly and crossing your fingers.

Think of a new workflow version as a new workflow. The basic technique is to make your assembly increment the version number with each build (rather than leaving it at 1.0.0.0 forever). Then, for each upgrade, you create a new feature for that version of the workflow, pointing to the new assembly. You add the new assembly into the global assembly cache (GAC) alongside the old assembly. Last, you specify that the old version cannot start new instances of the workflow and then you add the new workflow onto the list. This way, the old version of the assembly never changes, so there's no risk of hydrated workflows breaking when they are dehydrated. You deploy another version of the assembly and add the new workflow to the list and disable previous versions. You don't want to remove the previous versions because that will orphan those running instances. For the full set of procedures, follow the steps in table 12.1 to create a new version for an existing workflow.


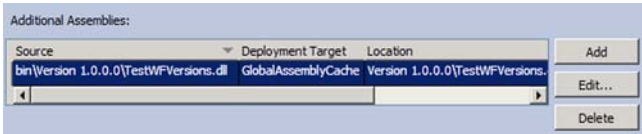
### Old version overwritten

If you don't create a new version and merely upgrade the solution, all running instances of the workflow will be deleted. The old version of the workflow will be removed, and the new version will be added with zero running instances. Don't upgrade without creating a new version unless you're entirely sure you don't need to retain the running instances.

**Table 12.1** Creating a new version for an existing workflow

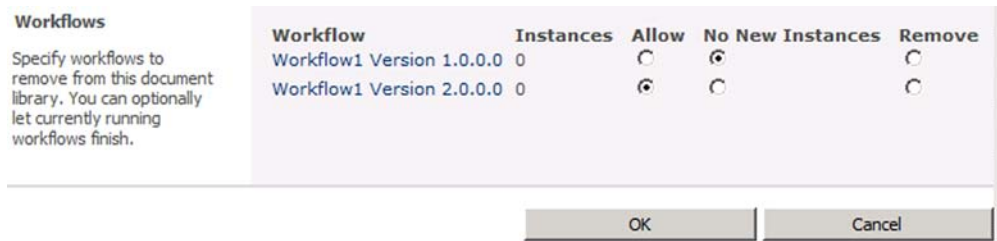
Action	Steps	Result
Create version 1.0.0.0 in your workflow's elements file.	<ol style="list-style-type: none"> <li>In the Elements.xml file of your workflow, replace \$assemblyname\$ in CodeBesideAssembly with the following:  <code>[assembly name], Version=1.0.0.0, Culture=neutral, PublicKeyToken=[token]</code> </li> </ol>	Your workflow's feature is now specifically referencing the 1.0.0.0 version of your assembly.

**Table 12.1** Creating a new version for an existing workflow (*continued*)

Action	Steps	Result
	<p>2 Replace [assembly name] with your assembly name.</p> <p>3 Replace [token] with your public key token. You can do this by finding your assembly in the GAC (c:\windows\assembly) and right-clicking it, choosing Properties, and copying the token.</p> <p>4 Change the name of the workflow in the Elements file to reference the new version.</p> <p><b>NOTE:</b> This ensures that the user working with the workflow knows what version it is. It has no technical implications. For example:</p> <p><b>Workflow</b></p> <p>Select a workflow to add to this document library. If the workflow template you want does not appear, contact your administrator to get it added to your site collection or workspace.</p>  <p>5 Add a copy of version 1.0.0.0 to solution package by double-clicking on the Package and, under the Advanced tab, add an existing assembly and browse to your 1.0.0.0 assembly version:</p>  <p><b>NOTE:</b> Notice how the Location and Source of the assembly are in a path under Version 1.0.0.0. When the package is created, the 1.0.0.0 version is put in its own path. It cannot be in the same path as the current version because they both have the same name.</p>	
With version 1.0.0.0 established, you can now simulate the need to create version 2.0.0.0. Change the version of the assembly to 2.0.0.0.	<p>1 Under the Properties folder in the solution, open the Assembly-Info.cs file.</p> <p>2 Scroll to the bottom of the file and change the two versions to 2.0.0.0.</p>	The current version of the workflow's assembly is now 2.0.0.0.
Update the workflow's Elements.xml file to reference both the version 1.0.0.0 workflow and now the new 2.0.0.0 version.	<p>1 Under the workflow, open the Elements.xml file.</p> <p>2 Copy the Workflow element in its entirety and paste it directly after the &lt;/workflow&gt; tag.</p> <p>3 Change the name and the version (in the CodeBesideAssembly) of the second workflow to reference version 2.0.0.0.</p> <p>4 Change the ID in the 2.0.0.0 version to a new GUID. You can create a new GUID by using the Create GUID tool under the tools menu.</p> <p>5 Build and deploy the solution.</p>	The workflow's feature now enables two workflows. The main difference is that one is referencing the 1.0.0.0 assembly, and the other is referencing the 2.0.0.0 assembly.

**Table 12.1** Creating a new version for an existing workflow (*continued*)

Action	Steps	Result
Activate version 2.0.0.0.	<ol style="list-style-type: none"> <li>1 Within the root site in the site collection, click Site Actions and, then, click Site Settings.</li> <li>2 Click Site Collection Features.</li> <li>3 Deactivate and reactivate your workflow's feature.</li> </ol>	The 2.0.0.0 workflow is able to associate with lists or sites.
With version 2.0.0.0 deployed and activated, you now need to prevent any new instances of version 1.0.0.0 from being created.	<ol style="list-style-type: none"> <li>1 Navigate to the workflow settings page of the list or site you're working on.</li> <li>2 Click Remove a workflow.</li> <li>3 Change version 1.0.0.0 to not allow new instances and click OK (figure 12.5).</li> </ol>	The 1.0.0.0 version can no longer be started, but running instances dehydrate without error.

**Figure 12.5** By deploying a 2.0.0.0 version of the workflow, all running instances of 1.0.0.0 will dehydrate without error when they resume execution. Users can no longer start new instances of version 1.0.0.0.

## 12.3 Building workflow event receivers

You can't have a SharePoint workflow book without a discussion of event receivers. You might immediately think you need a workflow when, in fact, an event receiver will do. The main difference between the two is that a workflow is typically long running whereas an event receiver is immediate. Why would you want an event receiver? What if all you want to do is execute a piece of code when a document is deleted. For example, you want with code to archive that document when a user deletes it. This example shows how useful an event receiver can be because a deleting event can trigger your custom code. You could do this with a Visual Studio workflow that has one only activity in it, but that's a lot of overhead for something that an event receiver does with much less effort. Table 12.2 shows more comparisons between workflows and event receivers.

**Table 12.2** Comparing event receivers and workflows

Event receivers	Workflows
Immediate execution	Long running
Lives and dies (no state)	Maintains state

**Table 12.2** Comparing event receivers and workflows (*continued*)

Event receivers	Workflows
.NET code only	.NET or SharePoint Designer
No human interaction	Typically involves human interaction
Before or after events	Only after events
Executes on sites, features, lists, and list items	Executes on sites, items, and content types.

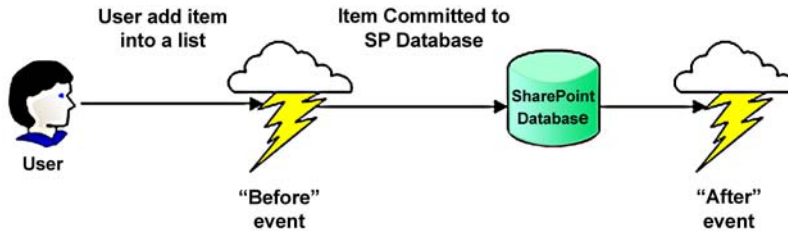
You can't say one is better than the other. It depends entirely on your business requirements. In addition, when documents are deleted, there are many other events you can respond to. The events fall into six categories, as shown in table 12.3. Each category has a few of the more common events shown in the second column, but note that there are many more events available.

**Table 12.3** Event receiver categories

Event category	Common events
List events	Adding or added a new list, field Updating or updated a field
List item events	Adding or added a new list item or document Document checking or checked in or out Adding or added an attachment Deleting or deleted an item or document
List email events	A list received an email
Web events	Deleting or deleted a site collection or site Creating or created a new site collection or site
Feature events	Feature activating or activated, deactivating or deactivated
List workflow events	A workflow is starting or started, postponed, or completed

You'll notice two things in this table. First, there are many events that you can have custom code respond to, and second most events have a before and after (adding or added) event associated with it. As shown in figure 12.6, *before events* happened *before* the change is committed to the SharePoint content database. This is helpful when you want to cancel a change before it is saved. The event receiver for when a site is being deleted is a good example. Before the site is deleted, you could do some additional processing such as backing it up.

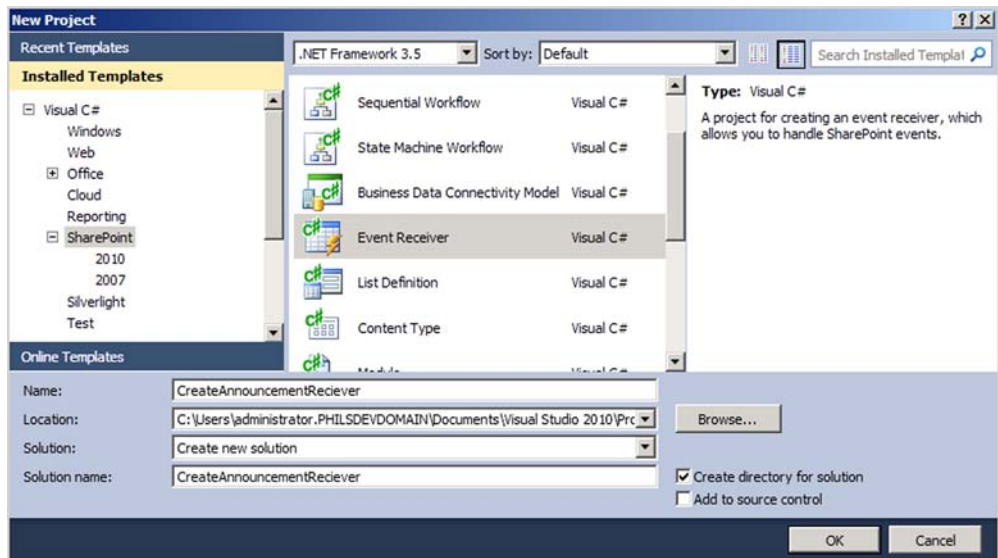
To demonstrate how to build an event receiver, you're going to use the workflow events as an example. (After all, this is a book about workflows.) Table 12.3 shows that there are four events under the list workflow events category. You can respond to when a workflow is starting, started, postponed, and completed. To keep the example simple, let's write an event receiver that creates an announcement when a new calendar event is created.



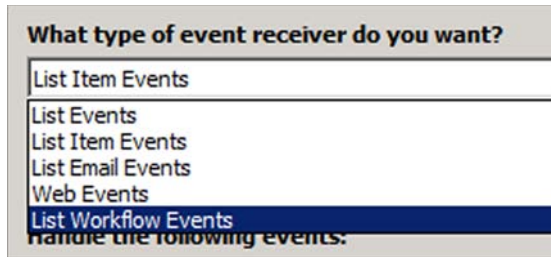
**Figure 12.6** Events typically have a before and an after event corresponding to when the event happens in relation to when the source is committed to the database.

Start by creating a new project in Visual Studio 2010. You'll notice under the SharePoint tab that there's a new project template called Event Receiver (figure 12.7). You can use this template to create any of the previously mentioned event receivers.

After you create the project, you'll get a dialog menu asking you to specify the URL of the site where you want to deploy and unit-test your event receiver and if you want to choose a full-trust (farm) or sandboxed solution. Specify the URL and, then, the farm solution because our code example requires full trust. Click Next and you'll be prompted to specify the type of event receiver you want to create (figure 12.8). The dropdown will contain the event types for each of the six categories except the feature events category. Feature events are created by right-clicking the feature in the project after it's created. For the announcement example, select the List Workflow Events category.



**Figure 12.7** Visual Studio 2010 now has a new project template you can use to easily create new event receivers.



**Figure 12.8** There are six main event categories. Five are shown.

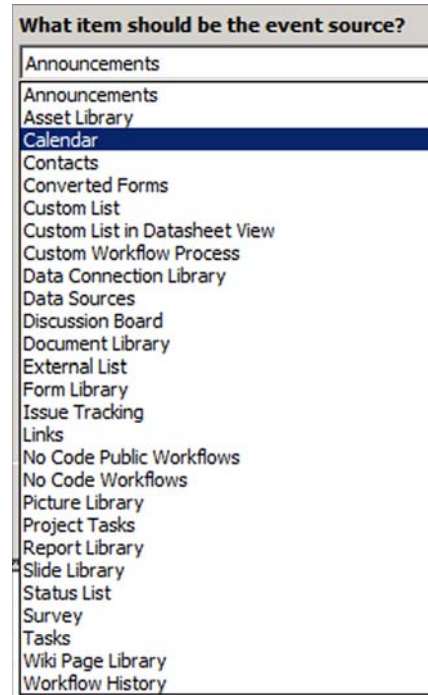
After you select the event category, specify the event source. This tells the feature that you're going to create events you want to respond to. Notice in figure 12.9 that you can handle events from announcements, document libraries, and many other list and library options. Select the Calendar source.

Next, you specify which workflow event you want to handle. Specify the A workflow has started event. Now, your code will execute each time a workflow is started on a calendar (figure 12.10).

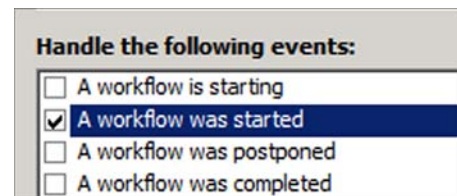
After you create the project, you'll be sent to a method named `Workflow-Started`. This is where you can add your code to create the announcement. Enter the code in listing 12.1 to create the announcement.

First, this code block looks at the activation properties to determine which calendar the event came from **1**. Remember that your source was Calendar, which means any calendar on the site will raise this event when a workflow is started. If you have more than one calendar on the site, you'll want to determine which calendar the event came from.

Next, you elevate the running user's privileges to the service account **2**. You're not sure if the running user has contribute rights on the announcements list, so you elevate his permissions to be



**Figure 12.9** You'll need to specify the event source that will raise the event and call your event receiver.



**Figure 12.10** After you choose the event source, you need to specify the event you want to respond to. In this case you want to execute your code when a workflow has started.

safe. Next, you create the announcement, assign a title value, and commit the announcement to the database **3**.

### Listing 12.1 Event receiver that creates an announcement

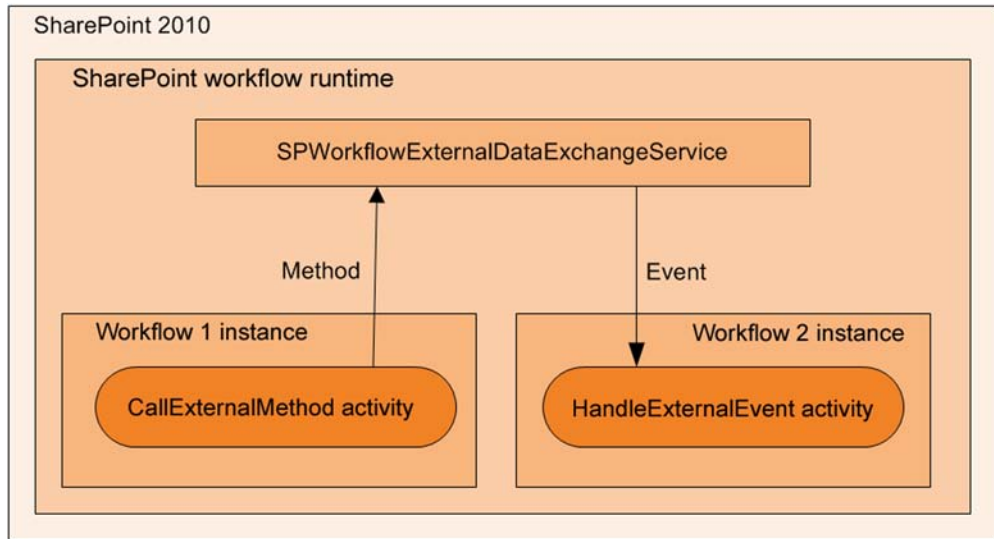
```
if (properties.ActivationProperties.List.Title == "Main Calendar")
{
    string siteurl = properties.ActivationProperties.Site.Url;
    SPSecurity.RunWithElevatedPrivileges(delegate()
    {
        using (SPSite site = new SPSite(siteurl))
        {
            using (SPWeb web = site.RootWeb)
            {
                SPListItem item = web.Lists["Announcements"].Items.Add();
                item["Title"] = "The workflow has started!";
                item.Update();
            }
        }
    });
}
```

With this code in place, you deploy the solution. Right-click the project name in Visual Studio, and click Deploy. This will deploy the feature and assembly on your *before*. Next, create a new calendar entry and start a workflow on that event. As a result, a new announcement will show up in the announcements list on that site.

## 12.4 Pluggable workflow services

Pluggable workflow services is one of the most highly anticipated new workflow features for SharePoint 2010. This is because SharePoint 2007 workflows lacked the ability to communicate with the outside world. The most basic scenario is a workflow that needs to go idle and wait for a message from a separate system, like a line of business applications such as customer relationship management (CRM). Another desired technique was for interworkflow communication, when one workflow needs to send a message to another workflow. A third scenario would be a long running process. If you had a calculation or a service call that took thirty minutes to execute, there would be no sense in keeping the workflow instance in memory. It would be better to hydrate the instance and dehydrate it when the process is finished.

All three of those examples were not easily accomplished in SharePoint 2007. Now, Windows Workflow Foundation on the .NET 3.5 Framework has the ability to meet these needs through Workflow Communication Services. Since SharePoint 2007 is on the 3.5 Framework, you'd think it wouldn't have been a problem. Because SharePoint was the hosting provider, there was no class that was provided to easily get at workflow instances and raise events into those instances that the workflow was listening for. This changes in SharePoint 2010, with the introduction of a new class, `SPWorkflowExternalDataExchangeService`.



**Figure 12.11** Windows Communication Services has become friendly with SharePoint workflows. Using the SharePoint workflow external data exchange service, your workflows can send and receive messages from other workflows or .NET applications.

Just as in Workflow Communication Services, in SharePoint 2010 workflows, you can create a local service that your workflows can use to communicate with each other. Using the `CallExternalMethod` activity and the `HandleExternalEvent` activity, SharePoint workflows and .NET applications can send and receive messages to each other (figure 12.11).

Before we get into how to set up a local service that uses the SharePoint external data exchange services, let's talk briefly about the example. The example you're going to build will be a glorified Hello World! example. A workflow is going to say Hello Event Handler! to an event receiver on an announcements list by creating a new announcement. Then, an event receiver is going to say Hello Workflow! back to the workflow.

To accomplish this, the workflow will call into a local service. The local service then creates an announcement in an announcements list. Then, an event receiver responds to the new announcement by raising an event through the local service that the workflow is listening for. This example will demonstrate how a SharePoint workflow can communicate with a .NET application. Follow the steps in table 12.4 to build the Hello World! pluggable workflow service.

**Table 12.4** Creating a pluggable workflow service

Action	Steps	Result
Create a new Visual Studio 2010 project.	<ol style="list-style-type: none"> <li>1 Open Visual Studio 2010 and create a new sequential workflow project titled <code>PluggableWorkflowServices</code> and click OK.</li> <li>2 Type the URL of the site you'll use to debug, click Next, select a Site Workflow, and click Finish.</li> </ol>	You have a new Visual Studio project with a Site workflow.

A local service is composed of two components, a service interface and a service class. The service interface lets the sending and receiving parties know what type of data to send to each other. This is done by declaring a method the sender calls and an event the receiver listens for. To tap into the external data exchange services, the interface must be declared with an `ExternalDataExchange` attribute.

**Table 12.4** Creating a pluggable workflow service (*continued*)

Action	Steps	Result
Create a new class to use for our local service and add our interface.	<ol style="list-style-type: none"> <li>1 Create a new class titled <code>HelloWorldService.cs</code>.</li> <li>2 Add the following using statements to the class file:  <pre>using System.Workflow.Activities; using Microsoft.SharePoint; using Microsoft.SharePoint.Workflow; using System.Workflow.Runtime;</pre> </li> <li>3 Above the <code>HelloWorldService</code> class, add the following interface:  <pre>[ExternalDataExchange] public interface IHelloWorldService {     event EventHandler&lt;HelloWorldEventArgs&gt;         HelloWorkflow;     void HelloHost(string message); }</pre> <p>The <code>HelloHost</code> method is called by the workflow via the <code>CallExternalMethod</code> activity, in the example, which creates the announcement. The event receiver then executes and invokes the <code>HelloWorkflow</code> event that the workflow is listening for through the <code>HandleExternalEvent</code> activity.</p> </li> </ol>	A new file named <code>HelloWorldService.cs</code> that contains your local service interface is created.
Extend the <code>HelloWorldService</code> class, and add the <code>HelloHost</code> method and the event defined in the interface.	<ol style="list-style-type: none"> <li>1 Make the <code>HelloWorldService</code> class extend <code>Microsoft.SharePoint.Workflow.SPWorkflowExternalDataExchangeService</code>.</li> <li>2 Make the <code>HelloWorldService</code> class implement the <code>IHelloWorldService</code> interface.</li> <li>3 Add listing 12.2 into the <code>HelloWorldService</code> class.</li> </ol>	The <code>HelloWorldService</code> class is extended and implements the interface you built in the previous step.

**Listing 12.2** HelloHost local service method

```
public event EventHandler<HelloWorldEventArgs> HelloWorkflow;
public void HelloHost(string message)
{
    SPWeb web = this.CurrentWorkflow.ParentWeb;
    SPList list = web.Lists["Announcements"];
    SPListItem item = list.Items.Add();
    item["Title"] = message;
    item["Instance"] = WorkflowEnvironment.WorkflowInstanceId.ToString();
    item.Update();
}
```

1 Declares the event

2 Defines the method

3 Adds a new announcement

In the `HelloHost` method you want to create two things, the event handler **1** and the method **2** that is defined in the interface. Within the method, you're creating the new announcement **3** and passing the workflow's instance ID into the `Instance` column within the announcement. This is how the event receiver will know to which workflow to send a message.

After you add listing 12.1 and the code found in the second and third actions, you may notice that the compiler cannot find the class for `HelloWorldEventArgs`. This class you have yet to define, but it will allow your event receiver to send a custom message to your workflow.

**Table 12.4** Creating a pluggable workflow service (continued)

Action	Steps	Result
Add the code that will allow passing a custom set of event arguments.	Add the following code below the <code>HelloWorldService</code> class: <pre>[Serializable] public class HelloWorldEventArgs :     ExternalDataEventArgs {     public HelloWorldEventArgs(Guid id) :     base(id) { }     public string Answer; }</pre>	Your local service will now be set up to pass a custom set of event arguments.

Notice that your custom arguments take two values, a GUID that will store the workflow instance ID and the `Answer`, which is the message the event receiver will pass to the workflow. This message will eventually be logged into the workflow's History List.

There's one more thing you must do before your local service is complete and you can build the workflow and the event receiver. You need to add three more methods to satisfy interface requirements in `SPExternalDataExchangeService`.

**Table 12.4** Creating a pluggable workflow service (continued)

Action	Steps	Result
Add listing 12.3 within the <code>HelloWorldService</code> class.		The <code>HelloWorldService</code> class now satisfies all interface requirements.

### Listing 12.3 `SPExternalExchangeService` interface methods

```
public override void CallEventHandler(Type type, string eventName,
    object[] parameters, SPWorkflow workflow, string identity,
    System.Workflow.Runtime.IPendingWork handler, object item)
{
    switch (eventName)
    {
        case "HelloWorkflow":
            var args = new HelloWorldEventArgs(workflow.InstanceId);
```

← **1** Switches event type

**2** Creates args with instance ID

←

```

        args.Answer = parameters[0].ToString();
        this.HelloWorkflow(null, args);
        break;
    }

    public override void CreateSubscription(
        MessageEventSubscription subscription)
    { throw new NotImplementedException(); }

    public override void DeleteSubscription(Guid subscriptionId)
    { throw new NotImplementedException(); }

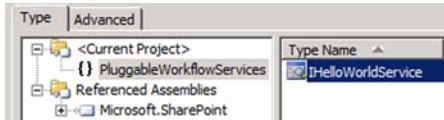
```

4 Invokes event      3 Pulls answer out of parameters

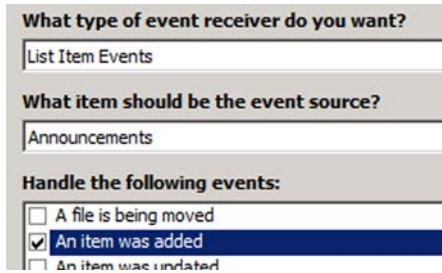
The `CallEventHandler` method is called each time an event is requested in the local service. First, you check to see which event is being requested 1. If it's your `HelloWorkflow` event, you create a new `HelloWorldEventArgs` instance and pass in the workflow's instance ID 2. This will let the event know which workflow to invoke the event with. Next, you pass in the message string 3 from the event receiver and, last, invoke the event 4.

With the local service now complete, you can start building the workflow and the event receiver that interfaces with this service. Continue the steps in table 12.4 to build the workflow and the event receiver.

**Table 12.4** Creating a pluggable workflow service (*continued*)

Action	Steps	Result
Configure the <code>CallExternalMethod</code> activity.	<ol style="list-style-type: none"> <li>Within <code>Workflow1</code>, add the <code>CallExternalMethod</code> activity.</li> <li>In the properties of the activity, click the ellipsis next to the <code>InterfaceType</code> property, specify the <code>IHelloworldService</code> interface, and click OK:</li> </ol>  <ol style="list-style-type: none"> <li>Change the <code>MethodName</code> property to <code>HelloHost</code>.</li> <li>Change the message property to <code>Hello Event Handler!</code></li> </ol>	The <code>CallExternalMethod</code> activity is configured to call the <code>HelloHost</code> method through the local service.
Configure the <code>HandleExternalEvent</code> activity.	<ol style="list-style-type: none"> <li>Add the <code>HandleExternalMethod</code> activity below the <code>CallExternalEvent</code> activity.</li> <li>Within the properties of the activity, click the ellipsis next to the <code>InterfaceType</code> property, specify the <code>IHelloworldService</code> interface, and click OK.</li> <li>Change the <code>EventName</code> property to be <code>HelloWorkflow</code>.</li> </ol> <p><b>NOTE:</b> This is the only event the workflow will listen for. The event receiver must invoke this event to communicate with the workflow.</p> <ol style="list-style-type: none"> <li>Bind the <code>e</code> property to a new field <code>handleArgs</code> by clicking the ellipses and choosing <code>Field</code> in the <code>Bind to New Member</code> tab and clicking OK.</li> </ol>	The <code>HandleExternalEvent</code> activity is now set up to listen and wait for the <code>HelloWorkflow</code> event.

**Table 12.4** Creating a pluggable workflow service (continued)

Action	Steps	Result
Configure a Log to History List activity.	<p>5 Go to the code view of the workflow and take the = new ... off the end of the handleArgs property so it looks like this:</p> <pre>1 public HelloWorldEventArgs handleArgs;</pre> <p>1 Below the HandleExternalEvent activity, add a LogToHistoryList activity.</p> <p>2 Right-click the LogToHistoryList activity and choose Generate Handlers.</p> <p>3 Within the activity's MethodInvoking method, add the following line of code to write the event receiver's message to the history:</p> <pre>logToHistoryListActivity1. HistoryDescription =     handleArgs.Answer;</pre>	After the HelloWorld event is raised, the workflow is configured to log the message sent from the event receiver to the workflow history list.
Add an Instance column to an announcements list.	<p>1 Find or create the announcements list on the site on which you're unit-testing and, under List Settings, click Create Column.</p> <p>2 Type a name of Instance, choose a Single Line of Text column type, and click OK.</p>	The HelloHost method saves a GUID into the Instance column in the announcements list, which is configured with that column.
Add a new Event Receiver to the project.	<p>1 Right-click the project, choose Add &gt; New Item, and select the Event Receiver item.</p> <p>2 Give the receiver the name of AnnouncementsReceiver and click Add.</p> <p>3 Choose List Item Events, Announcements, An item was added and click Finish:</p>  <p>4 Add listing 12.4 in the ItemAdded method of the receiver.</p>	The event receiver is configured to fire when an announcement is created and to invoke the HelloWorld event on the workflow instance found in the Instance column.

**Listing 12.4** ItemAdded event receiver method


```

if (properties.ListTitle == "Announcements")
{
    Guid instance = new Guid(properties.ListItem["Instance"].ToString());
    string answer = "Hello Workflow!";

```

Grabs workflow instance ID **1**

```
SPWorkflowExternalDataExchangeService.RaiseEvent(
    properties.Web, instance, typeof(HelloWorldService),
    "HelloWorkflow", new object[] { answer });
}
```


**Invokes  
HelloWorkflow  
event**

This listing first grabs the workflow's instance ID out of the Instance column and sets it to a GUID ❶. This GUID is passed as a parameter into the RaiseEvent method ❷, which is how the RaiseEvent method knows to which workflow to send the message. Other parameters of interest are the SharePoint site where the workflow is running, the event to invoke (HelloWorld event), and your message to the workflow, Hello Workflow! That last parameter is an object array so you can load that up with any serializable object you think the workflow needs.

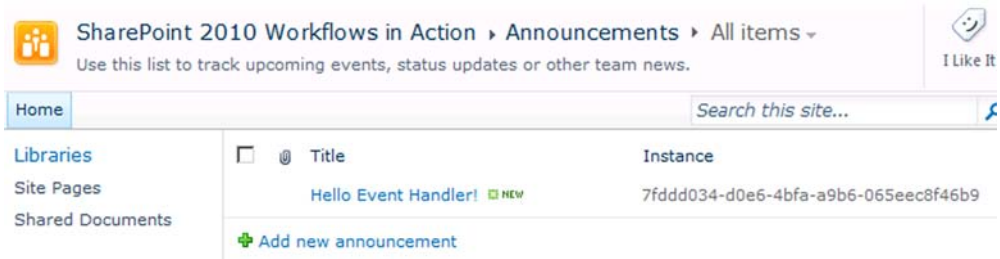
Before you test, register your local service with the SharePoint workflow runtime. You do that by adding an entry into the web.config. Follow this last step to register your service.

**Table 12.4** Creating a pluggable workflow service (continued)

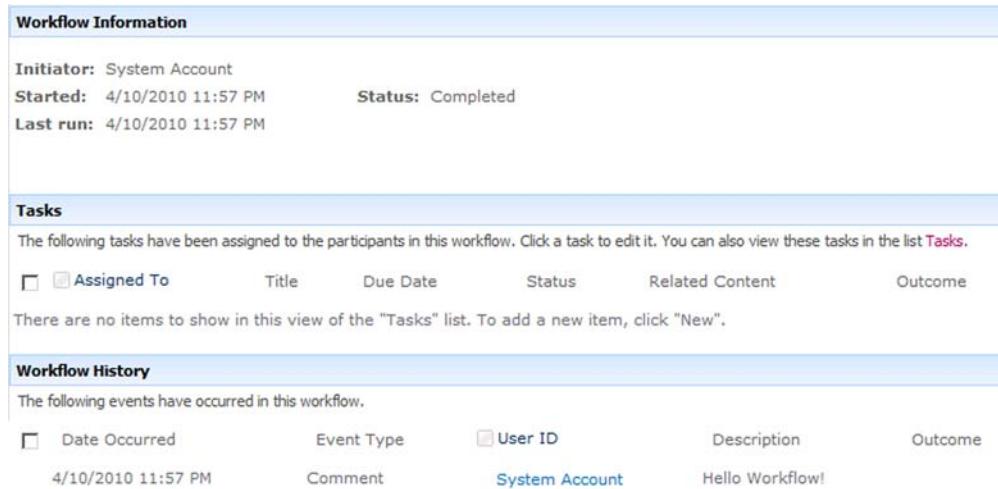
Action	Steps	Result
Register the HelloWorldService with the SharePoint workflow runtime.	<ol style="list-style-type: none"> <li>1 Open your web application's web.config under c:\inetpub\wwwroot\wss\virtual directories\ (plus the appropriate web application's unique folder name)</li> <li>2 Find the &lt;WorkflowServices&gt; element.</li> <li>3 Add the following WorkflowService in the WorkflowServices element (all on one line and note that you'll need to change the PublicKeyToken to your assembly's token): <pre>&lt;WorkflowService Assembly="PluggableWorkflowServices, Version=1.0.0.0, Culture=neutral, PublicKeyToken=c1c16502a94a0846" Class="PluggableWorkflowServices. HelloWorldService"&gt; &lt;/WorkflowService&gt;</pre> <p><b>NOTE:</b> If you find yourself getting an error "The workflow failed to start due to an internal error" when you try to start a workflow, you probably didn't perform the last correctly or the DLL isn't found in the GAC.</p> </li> </ol>	The local service is registered with the SharePoint workflow runtime.

You're *finally* ready to test. Build and deploy your project. Navigate to your SharePoint site and, under View Site Content, click Site Workflows. Start your pluggable workflow. (It should be named PluggableWorkflowServices-Workflow1.) Navigate to the Announcements list and you should see a new announcement titled Hello Event Handler!, as shown in figure 12.12.

Go back to Site Workflow and click the Completed status of the workflow titled PluggableWorkflowServices-Workflow1. You should see the event receiver's response Hello Workflow! (figure 12.13).



**Figure 12.12** The workflow wrote to an announcements list, and an event receiver on that will respond by calling back into the hydrated workflow instance.



**Figure 12.13** The event receiver has called back into the hydrated workflow instance, and the workflow logged the string message received from the event receiver.

## 12.5 SharePoint workflow object model

The SharePoint workflow object model falls within the `Microsoft.SharePoint.Workflow` namespace. You can leverage this object model to programmatically work with your workflows. You can start and stop a workflow, check a workflow's status or history, or retrieve a list of tasks associated with a workflow. This section will provide an introduction into the workflow object model and some common uses. Reference the complete SDK on [www.msdn.microsoft.com](http://www.msdn.microsoft.com). The namespace has many classes, but the following are the top two:

- *SPWorkflow*—This class represents a workflow instance on an item or site. It can be used to see who started the workflow (Author property) and get the state of the workflow (InternalState property).

- *SPWorkflowManager*—This is the class with many helper methods that you can use with workflows. The most useful methods include the following:
  - *GetItemActiveWorkflows*
  - *GetItemWorkflows*
  - *GetWorkflowTasks*
  - *RemoveWorkflowFromListItem*
  - *StartWorkflow*

Although these classes are most useful, they work in concert with a host of other classes in the same namespace. Table 12.5 shows a list of classes and their SKD descriptions.

**Table 12.5** Microsoft.SharePoint.Workflow main classes and SDK definitions

Main classes	SDK Definitions
SPWorkflow	A workflow instance that has run or is currently running on an item or site.
SPWorkflowActivationProperties	Represents the initial properties of the workflow instance as it starts, such as the user who added the workflow and the list and item to which the workflow was added.
SPWorkflowAssociation	Represents the association of a workflow template with a specific list, content type, or site that contains members that return custom information about that workflow's association with the specific list or content type.
SPWorkflowAssociationCollection	Represents the workflow associations on a SharePoint list or site.
SPWorkflowCollection	A collection of the workflow instances that have run or are currently running on a list item or site.
SPWorkflowFilter	Represents the filter criteria to apply to a workflow or workflow task collections, such as to whom the workflow is assigned and the workflow state.
SPWorkflowManager	Contains members that enable you to centrally control the workflow templates and instances across a site collection.
SPWorkflowModification	Represents a workflow modification.
SPWorkflowModificationCollection	Represents the collection of workflow modifications that are currently in scope for the workflow instance.
SPWorkflowTask	Represents a single workflow task for a given workflow instance.
SPWorkflowTaskCollection	Represents a collection of the workflow tasks for a workflow instance.
SPWorkflowTaskProperties	Represents the properties of a workflow task.
SPWorkflowTemplate	Represents a workflow template currently deployed on the SharePoint site and contains members you can use to get or set information about the template, such as the instantiation data and the history and task lists for the template.
SPWorkflowTemplateCollection	The collection of workflow templates currently deployed on a site.

Now that you have a high level understanding of classes, let's take a look at a few common uses of the workflow object model. The following code snippets are five common examples.

The first snippet shows starting a workflow programmatically:

```
foreach (SPWorkflowAssociation association in
    splistitem.ParentList.WorkflowAssociations)
{
    if (association.AllowManual)
    {
        splistitem.Web.Site.WorkflowManager.StartWorkflow(
            splistitem, association, association.AssociationData, true);
    }
}
```

The code in this snippet first gets all the workflows associated with the list. This could easily be a content type or a site for site workflows. The `SPWorkflowAssociation` object contains properties such as the workflow's start options. Next, the statement checks if manual starts are allowed through the UI. If so, it starts the workflow through the workflow manager's `StartWorkflow` method.

The second snippet shows how to stop a workflow programmatically:

```
SPWorkflow workflow = splistitem.Workflows[1];
web.Site.WorkflowManager.RemoveWorkflowFromListItem(workflow);
```

Stopping a workflow is simple. Use the workflow manager and call the `RemoveWorkflowFromListItem` method and pass the workflow you want to terminate. The workflow manager is again useful to retrieve the list of active workflows on an item, as seen in the following snippet:

```
SPWorkflowCollection runningWFs =
    web.Site.WorkflowManager.GetItemActiveWorkflows(splistitem);

Console.WriteLine("Names of Running Workflows:");

foreach (SPWorkflow workflow in runningWFs)
{
    Console.WriteLine(workflow.ParentAssociation.Name);
}
```

The workflow manager's `GetItemActiveWorkflows` method retrieves a collection of workflows representing all the running workflows on that item. The workflow collection on the item would contain all the workflows, regardless of whether they are currently running or not. Some may have completed or faulted. You can optionally use `GetItemWorkflows` and pass in an `SPWorkflowFilter` parameter that specifies an `SPWorkflowState` object. By using the filter and the state, you could retrieve only orphaned workflows, for example.

With an active workflow, you may want to get the workflow's tasks. The following snippet shows how to do this:

```
SPWorkflow workflow = splistitem.Workflows[1];

Console.WriteLine("Titles of Workflow's Tasks:");
foreach (SPWorkflowTask task in workflow.Tasks)
```

```
{
    Console.WriteLine(task["Title"].ToString());
}
```

Every workflow's `Tasks` property is an `SPWorkflowTaskCollection` object. You can iterate through each workflow task to retrieve all the tasks that the workflow has created. This can optionally be done through the workflow manager's `GetWorkflowTasks` method, and you can also pass in an `SPWorkflowFilter` parameter again to filter the tasks. The counterpart of tasks is the workflow's history.

The following snippet shows how to retrieve a workflow's history programmatically:

```
SPWorkflow workflow = splistitem.Workflows[1];
SPList historyList = workflow.HistoryList;
SPQuery query = new SPQuery();
query.Query =
    "<OrderBy><FieldRef Name=\"ID\"/></OrderBy>" +
    "<Where><Eq><FieldRef Name=\"WorkflowInstance\"/>" +
    "<Value Type=\"Text\">{\"+ workflow.InstanceId.ToString() +\"}</Value>" +
    "</Eq></Where>";

SPListItemCollection historyItems = historyList.GetItems(query);
foreach (SPListItem historyItem in historyItems)
{
    Console.WriteLine(historyItem["Description"].ToString());
}
```

Every workflow has a `HistoryList` property that points to the `SPList` object in which the workflow's history is stored. That list can be queried to get the history items. The querying is done using the `SPQuery` object (as shown previously) or LINQ to SharePoint. Enter the CAML query and pass the workflow's instance ID to get only that workflow's history items. The history is stored in the `Description` column of each `SPListItem` that is returned.

## 12.6 Summary

Any programmer will tell you that exception handling and debugging code is important. This is no exception with Visual Studio SharePoint workflows. Fortunately, your workflows can use the `HandleFault` activity and can be debugged like any other .NET application, so the learning curve should be minimal. Versioning workflows is sometimes new to developers. When a workflow is persisted to disk (hydrated), subsequent changes to the workflow's assembly can cause problems. If you change the assembly while the workflow is hydrated (saved in the database), there's no guarantee that, when the workflow dehydrated, it will match the construct of the new assembly. If it doesn't, the workflow will break. Instead, a new version of the workflow must be published.

SharePoint 2010 offers new event capabilities to developers. They include new workflow event receivers and an easier ability to send and receive events with applications external to the workflow. External communication is handled through the new pluggable workflow services feature.

Under all these excellent SharePoint workflow features resides a workflow object model that a developer may sometimes want to use. Using the `SPWorkflowManager` object, for example, you can programmatically start and stop a workflow. You can also do other tasks such as retrieve the running workflows on a document or, perhaps, look up the tasks and history associated with a workflow instance.

This chapter isn't the end of what you can read on workflows. As mentioned in chapter 1, SharePoint workflows are built on Windows Workflow Foundation. This book discusses the most critical areas for SharePoint workflow developers, but there's a whole world underneath the SharePoint surface. In that sense, this book is only the beginning.

# SharePoint 2010 Workflows IN ACTION

Phil Wicklund



**Y**ou can use SharePoint 2010 workflows to transform a set of business processes into working SharePoint applications. For that task, a power user gets prepackaged workflows, wizards, and design tools, and a programmer benefits from Visual Studio to handle advanced workflow requirements.

**SharePoint 2010 Workflows in Action** is a hands-on guide for workflow application development in SharePoint. Power users are introduced to the simplicity of building and integrating workflows using SharePoint Designer, Visio, InfoPath, and Office. Developers will learn to build custom processes and use external data sources. They will learn about state machine workflows, ASP.NET forms, event handlers, and much more. This book requires no previous experience with workflow app development.

## What's Inside

- Out-of-the-box and custom workflows
- How to integrate external data
- Advanced forms with InfoPath and ASP.NET
- External events with pluggable workflow services
- Custom workflow actions and conditions
- Model your business process in Visio

As a SharePoint consultant and trainer for RBA Consulting, **Phil Wicklund** has implemented countless workflows. He is a frequent speaker at SharePoint conferences and he blogs at [www.philwicklund.com](http://www.philwicklund.com).

For online access to the author and a free ebook for owners of this book, go to [manning.com/SharePoint2010WorkflowsinAction](http://manning.com/SharePoint2010WorkflowsinAction)

“Covers all aspects of SharePoint workflows.”  
—Wayne Ewington, Microsoft

“Great for learning or reference.”  
—Raymond Mitchell  
Inetium, Inc

“A must-have.”  
—Justin Kobel  
KiZAN Technologies

“The go-to resource.”  
—Andrew Grothe  
Triware Technologies Inc.

“Every SharePoint dev needs this book!”  
—Nikander & Margriet  
Bruggeman  
Lois & Clark IT Services