

57

Windows Workflow Foundation 3.0

WHAT'S IN THIS CHAPTER?

- Different types of workflows: Sequential and State Machine
- Built-in activities
- Creating custom activities
- Workflow Services
- Integration with WCF
- Hosting Workflows
- Tips for migrating to Workflow Foundation 4

This chapter presents an overview of the Windows Workflow Foundation 3.0 (known as WF throughout the rest of this chapter), which provides a model to define and execute processes using a set of building blocks called *activities*. WF provides a Designer that, by default, is hosted within Visual Studio, and that allows you to drag and drop activities from the toolbox onto the design surface to create a workflow template.

This template can then be executed by creating a `WorkflowInstance` and then running that instance. The code that executes a workflow is known as the `WorkflowRuntime`, and this object can also host a number of services that the running workflows can access. At any time, there may be several workflow instances executing, and the runtime deals with scheduling these instances and saving and restoring state; it can also record the behavior of each workflow instance as it executes.

A workflow is constructed from a number of activities, and these activities are executed by the runtime. An activity might send an e-mail, update a row in a database, or execute a transaction on a back-end system. There are a number of built-in activities that can be used for general-purpose work, and you can also create your own custom activities and plug these into the workflow as necessary.

Note that with Visual Studio 2010 there is a new version of Windows Workflow which is not backward compatible. This chapter describes the older version of workflow; however, for new projects I recommend using the WF 4 version. Refer to Chapter 44, “Windows Workflow Foundation 4,” for further details. At the end of this chapter, I have also included a set of suggestions to make upgrading easier.

We begin with the canonical example that everyone uses when faced with a new technology — “Hello World” — and also describe what you need to get workflows running on your development machine.

HELLO WORLD

Visual Studio 2010 contains built-in support for creating workflows, and when you open the New Project dialog, you see a list of workflow project types, as shown in Figure 57-1.

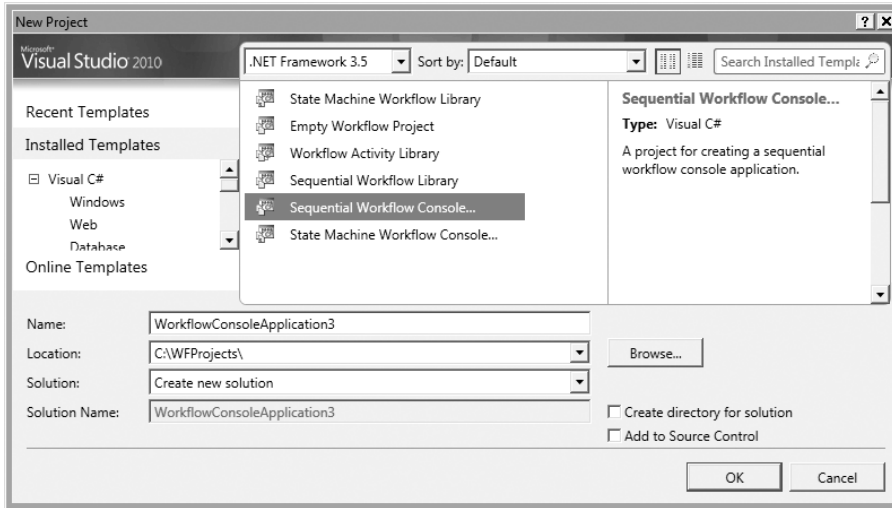


FIGURE 57-1

Select Sequential Workflow Console Application from the available templates (that will create a console application that hosts the workflow runtime) and a default workflow onto which you can then drag and drop activities.

Next, drag a Code activity from the toolbox onto the design surface so that you have a workflow that looks like the one shown in Figure 57-2.

The exclamation mark glyph on the top right of the activity indicates that a mandatory property of that activity has not been defined — in this case, it is the `ExecuteCode` property, which indicates the method that will be called when the activity executes. You learn how to mark your own properties as mandatory in the section on activity validation. If you double-click the code activity, a method will be created for you in the code-behind class, and here you can use `Console.WriteLine` to output the “Hello World” string, as shown in the following code snippet:

```
private void codeActivity1_ExecuteCode(object sender, EventArgs e)
{
    Console.WriteLine("Hello World");
}
```

If you then build and run the program, you will see the output text on the console. When the program executes, an instance of the `WorkflowRuntime` is created, and then an instance of your workflow is constructed and executed. When the code activity executes, it calls the method defined and that outputs the string to the console. The section entitled “The Workflow Runtime” later in the chapter describes in detail how to host the runtime. The code for the preceding example is available in the 01 HelloWorld folder.

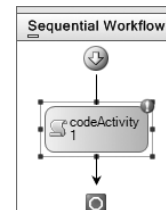


FIGURE 57-2

ACTIVITIES

Everything in a workflow is an activity, including the workflow itself. The workflow is a specific type of activity that typically allows other activities to be defined within it — this is known as a composite activity, and you see other composite activities later in this chapter. An activity is just a class that ultimately derives from the `Activity` class.

The `Activity` class defines a number of overridable methods, and arguably the most important of these is the `Execute` method, shown in the following snippet:

```
protected override ActivityExecutionStatus Execute
( ActivityExecutionContext executionContext )
{
    return ActivityExecutionStatus.Closed;
}
```

When the runtime schedules an activity for execution, the `Execute` method is ultimately called, and that is where you have the opportunity to write custom code to provide the behavior of the activity. In the simple example in the previous section, when the workflow runtime calls `Execute` on the `CodeActivity`, the implementation of this method on the code activity will execute the method defined in the code-behind class, and that displays the message on the console.

The `Execute` method is passed a context parameter of type `ActivityExecutionContext`. You will see more about this as the chapter progresses. The method has a return value of type `ActivityExecutionStatus`, and this is used by the runtime to determine whether the activity has completed successfully, is still processing, or is in one of several other potential states that can describe to the workflow runtime what state the activity is in. Returning `ActivityExecutionStatus.Closed` from this method indicates that the activity has completed its work and can be disposed of.

Numerous standard activities are provided with WF, and the following sections provide examples of some of these together with scenarios in which you might use these activities. The naming convention for activities is to append `Activity` to the name; so for example, the code activity previously shown in Figure 57-2 is defined by the `CodeActivity` class.

All of the standard activities are defined within the `System.Workflow.Activities` namespace, which in turn forms part of the `System.Workflow.Activities.dll` assembly. There are two other assemblies that make up WF — these are `System.Workflow.ComponentModel.dll` and `System.Workflow.Runtime.dll`.

IfElseActivity

As its name implies, this activity acts like an If-Else statement in C#.

When you drop an `IfElseActivity` onto the design surface, you will see an activity as displayed in Figure 57-3. The `IfElseActivity` is a composite activity in that it constructs two branches (which themselves are types of activity, in this case `IfElseBranchActivity`). Each branch is also a composite activity that derives from `SequenceActivity` — this class executes each activity in turn from top to bottom. The Designer adds the “Drop Activities Here” text to indicate where child activities can be added.

The first branch, as shown in Figure 57-3, includes a glyph indicating that the `Condition` property needs to be defined. A condition derives from `ActivityCondition` and is used to determine whether that branch should be executed.

When the `IfElseActivity` is executed, it evaluates the condition of the first branch, and if the condition evaluates to `true` the branch is executed. If the condition evaluates to `false`, the `IfElseActivity`

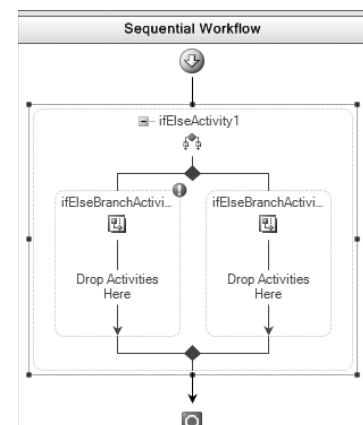


FIGURE 57-3

then tries the next branch, and so on until the last branch in the activity. It is worth noting that the `IfElseActivity` can have any number of branches, each with its own condition. The last branch needs no condition because it is in effect the `else` part of the `If-Else` statement. To add a new branch, you can display the context menu for the activity and select `Add Branch` from that menu — this is also available from the `Workflow` menu within Visual Studio. As you add branches, each will have a mandatory condition except for the last one.

Two standard condition types are defined in WF — the `CodeCondition` and the `RuleConditionReference`. The `CodeCondition` class executes a method on your code-behind class, which can return `true` or `false` as appropriate. To create a `CodeCondition`, display the property grid for the `IfElseActivity` and set the condition to `Code Condition`, then type in a name for the code to be executed, as shown in Figure 57-4.

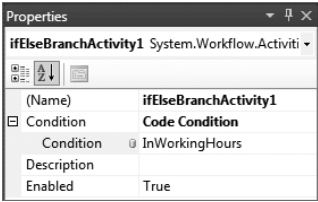


FIGURE 57-4

When you have typed the method name into the property grid, the Designer will construct a method on your code-behind class, as shown in the following snippet:

```
private void InWorkingHours(object sender, ConditionalEventArgs e)
{
    int hour = DateTime.Now.Hour;

    e.Result = ((hour >= 9) && (hour <= 17));
}
```

This code sets the `Result` property of the passed `ConditionalEventArgs` to `true` if the current hour is between 9 AM and 5 PM. Conditions can be defined in code as shown here, but another option is to define a condition based on a rule that is evaluated in a similar manner. The `Workflow Designer` contains a rule editor, which can be used to declare conditions and statements (much like the `If-Else` statement shown previously). These rules are evaluated at runtime based on the current state of the workflow.

ParallelActivity

This activity permits you to define a set of activities that execute in parallel — or rather in a pseudo-parallel manner. When the workflow runtime schedules an activity, it does so on a single thread. This thread executes the first activity, then the second, and so on until all activities have completed (or until an activity is waiting on some form of input). When the `ParallelActivity` executes, it iterates through each branch and schedules execution of each branch in turn. The workflow runtime maintains a queue of scheduled activities for each workflow instance, and typically executes these in a `FIFO` (first in, first out) manner.

Assuming that you have a `ParallelActivity`, as shown in Figure 57-5, this will schedule execution of `sequenceActivity1` and then `sequenceActivity2`. The `SequenceActivity` type works by scheduling execution of its first activity with the runtime, and when this activity completes, it then schedules the second activity. This `schedule/wait-for-completion` method is used to traverse through all child activities of the sequence, until all child activities have executed, at which time the sequence activity can complete.

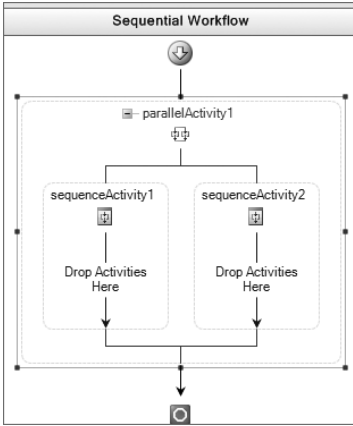


FIGURE 57-5

Given that the `SequenceActivity` schedules execution of one activity at a time, it means that the queue maintained by the `WorkflowRuntime` is continually updated with schedulable activities. Assuming that we have a parallel activity `P1`, which contains two sequences, `S1` and `S2`, each with two code activities, `C1` and `C2`, this would produce entries in the scheduler queue, as shown in the following table.

WORKFLOW QUEUE	INITIALLY THERE ARE NO ACTIVITIES IN THE QUEUE
P1	Parallel is executed when the workflow runs
S1, S2	Added to the queue when P1 executes
S2, S1.C1	S1 executes and adds S1.C1 to the queue
S1.C1, S2.C1	S2 executes and adds S2.C1 to the queue
S2.C1, S1.C2	S1.C1 completes, so S1.C2 is queued
S1.C2, S2.C2	S2.C1 completes, so S2.C2 is queued
S2.C2	The last entry in the queue

Here, the queue processes the first entry (the parallel activity P1), and this adds the sequence activities S1 and S2 to the workflow queue. As the sequence activity S1 executes, it pushes its first child activity (S1.C1) to the end of the queue, and when this activity is scheduled and completes, it then adds the second child activity to the queue.

As can be seen from the preceding example, execution of the `ParallelActivity` is not truly parallel — it effectively interleaves execution between the two sequential branches. From this, you could infer that it's best that an activity execute in a minimal amount of time because, given that there is only one thread servicing the scheduler queue for each workflow, a long-running activity could hamper the execution of other activities in the queue. That said, often, an activity needs to execute for an arbitrary amount of time, so there must be some way to mark an activity as “long-running” so that other activities get a chance to execute. You can do this by returning `ActivityExecutionStatus.Executing` from the `Execute` method, which lets the runtime know that you will call it back later when the activity has finished. An example of this type of activity is the `DelayActivity`.

CallExternalMethodActivity

A workflow will typically need to call methods outside of the workflow, and this activity allows you to define an interface and a method to call on that interface. The `WorkflowRuntime` maintains a list of services (keyed on a `System.Type` value) that can be accessed by using the `ActivityExecutionContext` parameter passed to the `Execute` method.

You can define your own services to add to this collection and then access these services from within your own activities. You could, for example, construct a data access layer exposed as a service interface and then provide different implementations of this service for SQL Server and Oracle. Because the activities simply call interface methods, the swap from SQL Server to Oracle would be opaque to the activities.

When you add a `CallExternalMethodActivity` to your workflow, you then define the two mandatory properties of `InterfaceType` and `MethodName`. The interface type defines which runtime service will be used when the activity executes, and the method name defines which method of that interface will be called.

When this activity executes, it looks up the service with the defined interface by querying the execution context for that service type, and it then calls the appropriate method on that interface. You can also pass parameters to the method from within the workflow — this is discussed later in the section titled “Binding Parameters to Activities.”

DelayActivity

Business processes often need to wait for a period of time before completing. Consider using a workflow for expense approval. Your workflow might send an e-mail to your immediate manager asking him or her to approve your expense claim. The workflow then enters a waiting state, where it either waits for approval (or, horror of horrors, rejection), but it would also be nice to define a timeout so that if no response is returned within, say, one day, the expense claim is then routed to the next manager up the chain of command.

The `DelayActivity` can form part of this scenario (the other part is the `ListenActivity`, defined later). Its job is to wait for a predefined time before continuing execution of the workflow. There are two ways to define the duration of the delay — you can either set the `TimeoutDuration` property of the delay to a string, such as “1.00:00:00” (1 day, no hours, minutes, or seconds), or you can provide a method that is called when the activity is executed that sets the duration to a value from code. To do this, you need to define a value for the `InitializeTimeoutDuration` property of the delay activity. This creates a method in the code-behind, as shown in the following snippet:

```
private void DefineTimeout(object sender, EventArgs e)
{
    DelayActivity delay = sender as DelayActivity;

    if (null != delay)
    {
        delay.TimeoutDuration = new TimeSpan(1, 0, 0, 0);
    }
}
```

Here, the `DefineTimeout` method casts the sender to a `DelayActivity` and then sets the `TimeoutDuration` property in code to a `TimeSpan`. Even though the value is hard-coded here, it is more likely that you would construct this from some other data — maybe a parameter passed into the workflow or a value read from the configuration file. Workflow parameters are discussed in the section “Workflows” later in the chapter.

ListenActivity

A common programming construct is to wait for one of a set of possible events — one example of this is the `WaitAny` method of the `System.Threading.WaitHandle` class. The `ListenActivity` is the way to do this in a workflow, because it can define any number of branches, each with an event-based activity as that branch’s first activity.

An event activity is one that implements the `IEventActivity` interface defined in the `System.Workflow.Activities` namespace. There are currently three such activities defined as standard in WF — `DelayActivity`, `HandleExternalEventActivity`, and the `WebServiceInputActivity`. Figure 57-6 shows a workflow that is waiting for either external input or a delay — this is an example of the expense approval workflow discussed earlier.

In the example, the `CallExternalMethodActivity` is used as the first activity in the workflow. This calls a method defined on a service interface that would prompt the manager for approval or rejection. Because this is an external service, this prompt could be an e-mail, an IM message, or any other manner of notifying your manager that an expense claim needs to be processed. The workflow then executes the `ListenActivity`, which awaits input from this external service (either an approval or a rejection), and also waits on a delay.

When the listen executes, it effectively queues a wait on the first activity in each branch, and when one event is triggered, this cancels all other waiting events and then processes the rest of the branch where the event was raised. So, in the instance where the expense report is approved, the `Approved` event is raised and the `PayMe` activity is then scheduled. If, however, your manager rejects the claim, the `Rejected` event is raised, and in the example you then `Panic`.

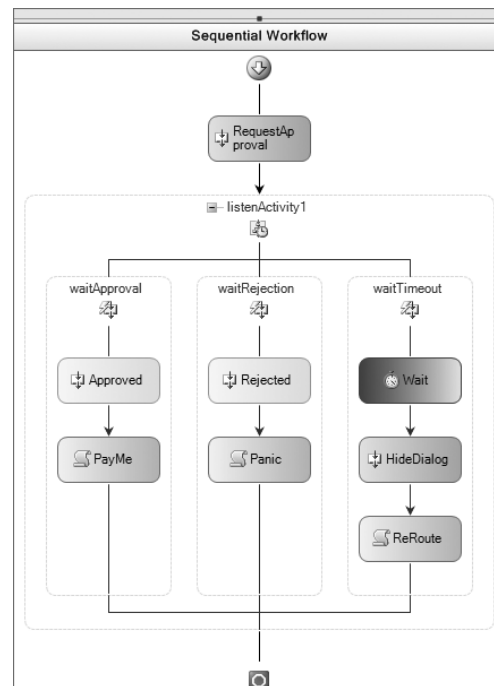


FIGURE 57-6

Last, if neither the `Approved` nor `Rejected` event is raised, the `DelayActivity` ultimately completes after its delay expires, and the expense report could then be routed to another manager — potentially looking up that person in Active Directory. In the example, a dialog is displayed to the user when the `RequestApproval` activity is executed, so if the delay executes, you also need to close the dialog, which is the purpose of the activity named `HideDialog` in Figure 57-6.

The code for this example is available in the `02 Listen` directory. Some concepts used in that example have not been covered yet — such as how a workflow instance is identified and how events are raised back into the workflow runtime and ultimately delivered to the right workflow instance. These concepts are covered in the section titled “Workflows.”

Activity Execution Model

So far, this chapter has discussed the execution of an activity only by the runtime calling the `Execute` method. However, an activity may go through a number of states while it executes — these are presented in Figure 57-7.

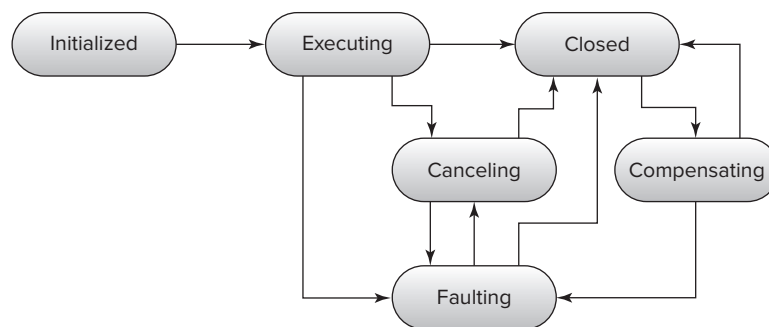


FIGURE 57-7

An activity is first initialized by the `WorkflowRuntime` when the runtime calls the activity’s `Initialize` method. This method is passed an `IServiceProvider` instance, which maps to the services available within the runtime. These services are discussed in the “Workflow Services” section later in the chapter. Most activities do nothing in this method, but the method is there for you to do any setup necessary.

The runtime then calls the `Execute` method, and the activity can return any one of the values from the `ActivityExecutionStatus` enum. Typically, you will return `Closed` from your `Execute` method, which indicates that your activity has finished processing; however, if you return one of the other status values, the runtime will use this to determine what state your activity is in.

You can return `Executing` from this method to indicate to the runtime that you have extra work to do — a typical example of this is when you have a composite activity that needs to execute its children. In this case, your activity can schedule each child for execution and then wait for all children to complete before notifying the runtime that your activity has completed.

CUSTOM ACTIVITIES

So far, you have used activities that are defined within the `System.Workflow.Activities` namespace. In this section, you learn how to create custom activities and extend these activities to provide a good user experience at both design time and runtime.

To begin, you create a `WriteLineActivity` that can be used to output a line of text to the console. Although this is a trivial example, it will be expanded to show the full gamut of options available for custom

activities using this example. When creating custom activities, you can simply construct a class within a workflow project; however, it is preferable to construct your custom activities inside a separate assembly, because the Visual Studio design time environment (and specifically workflow projects) will load activities from your assemblies and can lock the assembly that you are trying to update. For this reason, you should create a simple class library project to construct your custom activities within.

A simple activity such as the `WriteLineActivity` will be derived directly from the `Activity` base class. The following code shows a constructed activity class and defines a `Message` property that is displayed when the `Execute` method is called:

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

namespace SimpleActivity
{
    /// <summary>
    /// A simple activity that displays a message to the console when it executes
    /// </summary>
    public class WriteLineActivity: Activity
    {
        /// <summary>
        /// Execute the activity – display the message on screen
        /// </summary>
        /// <param name="executionContext"></param>
        /// <returns></returns>
        protected override ActivityExecutionStatus Execute
            (ActivityExecutionContext executionContext)
        {
            Console.WriteLine(Message);

            return ActivityExecutionStatus.Closed;
        }

        /// <summary>
        /// Get/Set the message displayed to the user
        /// </summary>
        [Description("The message to display")]
        [Category("Parameters")]
        public string Message
        {
            get { return _message; }
            set { _message = value; }
        }

        /// <summary>
        /// Store the message displayed to the user
        /// </summary>
        private string _message;
    }
}
```

Within the `Execute` method, you can write the message to the console and then return a status of `Closed` to notify the runtime that the activity has completed.

You can also define attributes on the `Message` property so that a description and category are defined for that property. This is used in the property grid within Visual Studio, as shown in Figure 57-8.

The code for the activities created in this section is in the 03 `CustomActivities` solution. If you compile that solution, you can then add the custom activities to the toolbox within Visual Studio by choosing the `Choose Items` menu item from the context menu on the toolbox and navigating to the folder where the

assembly containing the activities resides. All activities within the assembly will be added to the toolbox.

As it stands, the activity is perfectly usable; however, there are several areas that should be addressed to make this more user-friendly. As you saw with the `CodeActivity` earlier in the chapter, it has some mandatory properties that, when not defined, produce an error glyph on the design surface. To get the same behavior from your activity, you need to construct a class that derives from `ActivityValidator` and associate this class with your activity.

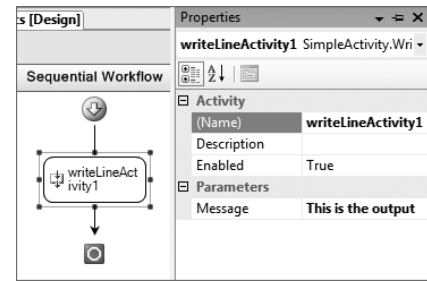


FIGURE 57-8

Activity Validation

When an activity is placed onto the design surface, the Workflow Designer looks for an attribute on that activity that defines a class that performs validation on that activity. To validate your activity, you need to check if the `Message` property has been set.

A custom validator is passed the activity instance, and from this you can then determine which mandatory properties (if any) have not been defined and add an error to the `ValidationErrorsCollection` used by the Designer. This collection is then read by the Workflow Designer, and any errors found in the collection will cause a glyph to be added to the activity and optionally link each error to the property that needs attention.

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

namespace SimpleActivity
{
    public class WriteLineValidator: ActivityValidator
    {
        public override ValidationErrorsCollection Validate
            (ValidationManager manager, object obj)
        {
            if (null == manager)
                throw new ArgumentNullException("manager");
            if (null == obj)
                throw new ArgumentNullException("obj");

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

            // Coerce to a WriteLineActivity
            WriteLineActivity act = obj as WriteLineActivity;

            if (null != act)
            {
                if (null != act.Parent)
                {
                    // Check the Message property
                    if (string.IsNullOrEmpty(act.Message))
                        errors.Add(ValidationError.GetNotSetValidationError("Message"));
                }
            }

            return errors;
        }
    }
}
```

The `Validate` method is called by the Designer when any part of the activity is updated and also when the activity is dropped onto the design surface. The Designer calls the `Validate` method and passes through the activity as the untyped `obj` parameter.

In this method, first validate the arguments passed in, and then call the base class `Validate` method to obtain a `ValidationErrorsCollection`. Although this is not strictly necessary here, if you are deriving from an activity that has a number of properties that also need to be validated, calling the base class method will ensure that these are also checked.

The code then coerces the passed `obj` parameter into a `WriteLineActivity` instance, and also checks if the activity has a parent. This test is necessary because the `Validate` function is called during compilation of the activity (if the activity is within a workflow project or activity library), and, at this point, no parent activity has been defined. Without this check, you cannot actually build the assembly that contains the activity and the validator. This extra step is not needed if the project type is class library.

The last step is to check that the `Message` property has been set to a value other than an empty string. This uses a static method of the `ValidationError` class, which constructs an error that specifies that the property has not been defined.

To add validation support to your `WriteLineActivity`, the last step is to add the `ActivityValidation` attribute to the activity, as shown in the following snippet:

```
[ActivityValidator(typeof(WriteLineValidator))]
public class WriteLineActivity: Activity
{
    .
}
}
```

If you compile the application and then drop a `WriteLineActivity` onto the workflow, you should see a validation error, as shown in Figure 57-9; clicking this error will take you to that property within the property grid.

If you enter some text for the `Message` property, the validation error will be removed, and you can then compile and run the application.

Now that you have completed the activity validation, the next thing to do is to change the rendering behavior of the activity to add a fill color to that activity. To do this, you need to define both an `ActivityDesigner` class and an `ActivityDesignerTheme` class, as described in the next section.

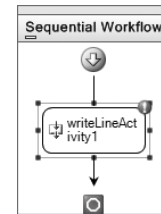


FIGURE 57-9

Themes and Designers

The onscreen rendering of an activity is performed by using an `ActivityDesigner` class, and this can also use an `ActivityDesignerTheme`.

The theme class is used to make simple changes to the rendering behavior of the activity within the Workflow Designer:

```
public class WriteLineTheme: ActivityDesignerTheme
{
    /// <summary>
    /// Construct the theme and set some defaults
    /// </summary>
    /// <param name="theme"></param>
    public WriteLineTheme(WorkflowTheme theme)
    : base(theme)
    {
        this.BackColorStart = Color.Yellow;
        this.BackColorEnd = Color.Orange;
        this.BackgroundStyle = LinearGradientMode.ForwardDiagonal;
    }
}
```

A theme is derived from `ActivityDesignerTheme`, which has a constructor that is passed a `WorkflowTheme` argument. Within the constructor, set the start and end colors for the activity, and then define a linear gradient brush, which is used when painting the background.

The `Designer` class is used to override the rendering behavior of the activity. In this case, no override is necessary, so the following code will suffice:

```
[ActivityDesignerTheme(typeof(WriteLineTheme))]
public class WriteLineDesigner: ActivityDesigner
{
}
```

Note that the theme has been associated with the `Designer` by using the `ActivityDesignerTheme` attribute.

The last step is to adorn the activity with the `Designer` attribute:

```
[ActivityValidator(typeof(WriteLineValidator))]
[Designer(typeof(WriteLineDesigner))]
public class WriteLineActivity: Activity
{
}
```

With this in place, the activity is rendered as shown in Figure 57-10.

With the addition of the `Designer` and the theme, the activity now looks much more professional. A number of other properties are available on the theme — such as the pen used to render the border, the color of the border, and the border style.

By overriding the `OnPaint` method of the `ActivityDesigner` class, you can have complete control over the rendering of the activity. Be sure to exercise restraint here, because you could get carried away and create an activity that doesn't resemble any of the other activities in the toolbox.

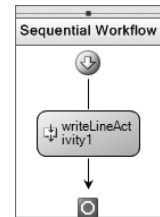


FIGURE 57-10

One other useful override on the `ActivityDesigner` class is the `Verbs` property. This allows you to add menu items on the context menu for the activity. It is used by the `Designer` of the `ParallelActivity` to insert the `Add Branch` menu item into the activities context menu and also the `Workflow` menu. You can also alter the list of properties exposed for an activity by overriding the `PreFilterProperties` method of the `Designer` — this is how the method parameters for the `CallExternalMethodActivity` are surfaced into the property grid. If you need to do this type of extension to your `Designer`, you should run Red Gate's `Reflector` (available from <http://reflector.red-gate.com>) and load the workflow assemblies into it to see how Microsoft has defined some of these extended properties.

This activity is nearly done, but now you need to define the icon used when rendering the activity and also the toolbox item to associate with the activity.

ActivityToolboxItem and Icons

To complete your custom activity, you need to add an icon. You can optionally create a class deriving from `ActivityToolboxItem` that is used when displaying the activity in the toolbox within Visual Studio.

To define an icon for an activity, create a 16×16 pixel image and include it in your project. When it has been included, set the build action for the image to `Embedded Resource`. This will include the image in the manifest resources for the assembly. You can add a folder to your project called `Resources`, as shown in Figure 57-11.

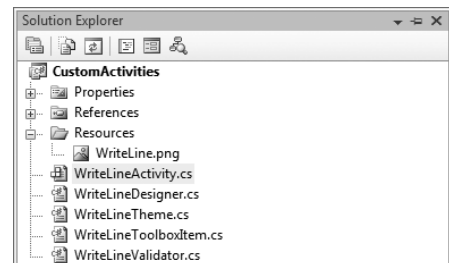


FIGURE 57-11

Once you have added the image file and set its build action to `Embedded Resource`, you can then attribute the activity as shown in the following snippet:

```
[ActivityValidator(typeof(WriteLineValidator))]
[Designer(typeof(WriteLineDesigner))]
[ToolboxBitmap(typeof(WriteLineActivity), "Resources.WriteLine.png")]
public class WriteLineActivity: Activity
{
    .
}
```

The `ToolboxBitmap` attribute has a number of constructors defined, and the one being used here takes a type defined in the activity assembly and the name of the resource. When you add a resource to a folder, its name is constructed from the namespace of the assembly and the name of the folder that the image resides within — so the fully qualified name for the resource here is `CustomActivities.Resources.WriteLine.png`. The constructor used with the `ToolboxBitmap` attribute appends the namespace that the type parameter resides within to the string passed as the second argument, so this will resolve to the appropriate resource when loaded by Visual Studio.

The last class you need to create is derived from `ActivityToolboxItem`. This class is used when the activity is loaded into the Visual Studio toolbox. A typical use of this class is to change the displayed name of the activity on the toolbox — all of the built-in activities have their names changed to remove the word “Activity” from the type. In your class, you can do the same by setting the `DisplayName` property to “WriteLine.”

```
[Serializable]
public class WriteLineToolboxItem: ActivityToolboxItem
{
    /// <summary>
    /// Set the display name to WriteLine - i.e. trim off
    /// the 'Activity' string
    /// </summary>
    /// <param name="t"></param>
    public WriteLineToolboxItem(Type t)
        : base(t)
    {
        base.DisplayName = "WriteLine";
    }

    /// <summary>
    /// Necessary for the Visual Studio design time environment
    /// </summary>
    /// <param name="info"></param>
    /// <param name="context"></param>
    private WriteLineToolboxItem(SerializationInfo info,
        StreamingContext context)
    {
        this.Deserialize(info, context);
    }
}
```

The class is derived from `ActivityToolboxItem` and overrides the constructor to change the display name; it also provides a serialization constructor that is used by the toolbox when the item is loaded into the toolbox. Without this constructor, you will receive an error when you attempt to add the activity to the toolbox. Note that the class is also marked as `[Serializable]`.

The toolbox item is added to the activity by using the `ToolboxItem` attribute as shown:

```
[ActivityValidator(typeof(WriteLineValidator))]
[Designer(typeof(WriteLineDesigner))]
[ToolboxBitmap(typeof(WriteLineActivity), "Resources.WriteLine.png")]
[ToolboxItem(typeof(WriteLineToolboxItem))]
public class WriteLineActivity: Activity
{
    .
}
```

With all of these changes in place, you can compile the assembly and then create a new workflow project. To add the activity to the toolbox, open a workflow and then display the context menu for the toolbox and click Choose Items.

You can then browse for the assembly containing your activity, and once you have added it to the toolbox, it will look something like Figure 57-12. The icon is somewhat less than perfect, but it's close enough.

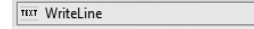


FIGURE 57-12

You revisit the `ActivityToolboxItem` in the next section on custom composite activities, because there are some extra facilities available with that class that are necessary only when adding composite activities to the design surface.

To recap, you've created a custom `WriteLineActivity`, added validation logic by creating the `WriteLineValidator`, created a designer and a theme with the `WriteLineDesigner` and `WriteLineTheme` classes, created a bitmap for the activity, and lastly created the `WriteLineToolboxItem` class to alter the display name of the activity.

Custom Composite Activities

There are two main types of activity. Activities that derive from `Activity` can be thought of as callable functions from the workflow. Activities that derive from `CompositeActivity` (such as `ParallelActivity`, `IfElseActivity`, and the `ListenActivity`) are containers for other activities. Their design-time behavior is considerably different from simple activities in that they present an area on the Designer where child activities can be dropped.

In this section, you create an activity that you can call the `DaysOfWeekActivity`. This activity can be used to execute different parts of a workflow based on the current date. You might, for instance, need to execute a different path in the workflow for orders that arrive over the weekend than for those that arrive during the week. In this example, you learn about a number of advanced workflow topics, and by the end of this section, you should have a good understanding of how to extend the system with your own composite activities. The code for this example is also available in the 03 CustomActivities solution.

To begin, you create a custom activity that has a property that will default to the current date/time. You will allow that property to be set to another value that could come from another activity in the workflow or a parameter that is passed to the workflow when it executes. This composite activity will contain a number of branches — these will be user defined. Each of these branches will contain an enumerated constant that defines which day(s) that branch will execute. The following example defines the activity and two branches:

```
DaysOfWeekActivity
    SequenceActivity: Monday, Tuesday, Wednesday, Thursday, Friday
    <other activities as appropriate>
    SequenceActivity: Saturday, Sunday
    <other activities as appropriate>
```

For this example, you need an enumeration that defines the days of the week — this will include the `[Flags]` attribute (so you can't use the built-in `DayOfWeek` enum defined within the `System` namespace, because this doesn't include the `[Flags]` attribute).

```
[Flags]
[Editor(typeof(FlagsEnumEditor), typeof(UITypeEditor))]
public enum WeekdayEnum: byte
{
    None = 0x00,
    Sunday = 0x01,
    Monday = 0x02,
    Tuesday = 0x04,
    Wednesday = 0x08,
    Thursday = 0x10,
    Friday = 0x20,
    Saturday = 0x40
}
```

Also included is a custom editor for this type, which will allow you to choose enum values based on check boxes. This code is available in the download.

With the enumerated type defined, you can take an initial stab at the activity itself. Custom composite activities are typically derived from the `CompositeActivity` class, because this defines, among other things, an `Activities` property, which is a collection of all subordinate activities.

```
public class DaysOfWeekActivity: CompositeActivity
{
    /// <summary>
    /// Get/Set the day of week property
    /// </summary>
    [Browsable(true)]
    [Category("Behavior")]
    [Description("Bind to a DateTime property, set a specific date time,
        or leave blank for DateTime.Now")]
    [DefaultValue(typeof(DateTime), "")]
    public DateTime Date
    {
        get { return (DateTime)
            base.GetValue(DaysOfWeekActivity.DateProperty); }
        set { base.SetValue(DaysOfWeekActivity.DateProperty, value); }
    }

    /// <summary>
    /// Register the DayOfWeek property
    /// </summary>
    public static DependencyProperty DateProperty =
        DependencyProperty.Register("Date", typeof(DateTime),
            typeof(DaysOfWeekActivity));
}
```

The `Date` property provides the regular getter and setter, and we've also added a number of standard attributes so that it displays correctly within the property browser. The code, though, looks somewhat different from a normal .NET property, because the getter and setter are not using a standard field to store their values, but instead are using what's called a `DependencyProperty`.

The `Activity` class (and therefore this class, because it's ultimately derived from `Activity`) is derived from the `DependencyObject` class, and this defines a dictionary of values keyed on a `DependencyProperty`. This indirection of getting/setting property values is used by WF to support binding; that is, linking a property of one activity to a property of another. As an example, it is common to pass parameters around in code, sometimes by value, sometimes by reference. WF uses binding to link property values together — so in this example, you might have a `DateTime` property defined on the workflow, and this activity might need to be bound to that value at runtime. You see an example of binding later in the chapter.

If you build this activity, it won't do much; indeed it will not even allow child activities to be dropped into it, because you haven't defined a `Designer` class for the activity.

Adding a Designer

As you saw with the `WriteLineActivity` earlier in the chapter, each activity can have an associated `Designer` class, which is used to change the design-time behavior of that activity. You saw a blank `Designer` in the `WriteLineActivity`, but for the composite activity you need to override a couple of methods to add some special case processing:

```
public class DaysOfWeekDesigner: ParallelActivityDesigner
{
    public override bool CanInsertActivities
        (HitTestInfo insertLocation, ReadOnlyCollection<Activity> activities)
    {
        foreach (Activity act in activities)
```

```

        {
            if (!(act is SequenceActivity))
                return false;
        }

        return base.CanInsertActivities(insertLocation, activitiesToInsert);
    }

    protected override CompositeActivity OnCreateNewBranch()
    {
        return new SequenceActivity();
    }
}

```

This Designer derives from `ParallalActivityDesigner`, which provides you with good design-time behavior when adding child activities. You will need to override `CanInsertActivities` to return false if any of the dropped activities is not a `SequenceActivity`. If all activities are of the appropriate type, you can call the base class method, which makes some further checks on the activity types permitted within your custom activity.

You should also override the `OnCreateNewBranch` method that is called when the user chooses the Add Branch menu item. The Designer is associated with the activity by using the `[Designer]` attribute, as shown here:

```

[Designer(typeof(DaysOfWeekDesigner))]
public class DaysOfWeekActivity: CompositeActivity
{
}

```

The design-time behavior is nearly complete; however, you also need to add a class that is derived from `ActivityToolboxItem` to this activity, because that defines what happens when an instance of that activity is dragged from the toolbox. The default behavior is simply to construct a new activity; however, in the example you also want to create two default branches. The following code shows the toolbox item class in its entirety:

```

[Serializable]
public class DaysOfWeekToolboxItem: ActivityToolboxItem
{
    public DaysOfWeekToolboxItem(Type t)
        : base(t)
    {
        this.DisplayName = "DaysOfWeek";
    }

    private DaysOfWeekToolboxItem(SerializationInfo info,
                                   StreamingContext context)
    {
        this.Deserialize(info, context);
    }

    protected override IComponent[] CreateComponentsCore(IDesignerHost host)
    {
        CompositeActivity parent = new DaysOfWeekActivity();
        parent.Activities.Add(new SequenceActivity());
        parent.Activities.Add(new SequenceActivity());

        return new IComponent[] { parent };
    }
}

```

As shown in the code, the display name of the activity was changed, a serialization constructor was implemented, and the `CreateComponentsCore` method was overridden.

This method is called at the end of the drag-and-drop operation, and it is where you construct an instance of the `DaysOfWeekActivity`. In the code, you are also constructing two child sequence activities, because this gives the user of the activity a better design-time experience. Several of the built-in activities do this, too — when you drop an `IfElseActivity` onto the design surface, its toolbox item class adds two branches. A similar thing happens when you add a `ParallelActivity` to your workflow.

The serialization constructor and the `[Serializable]` attribute are necessary for all classes derived from `ActivityToolboxItem`.

The last thing to do is associate this toolbox item class with the activity:

```
[Designer(typeof(DaysOfWeekDesigner))]
[ToolboxItem(typeof(DaysOfWeekToolboxItem))]
public class DaysOfWeekActivity: CompositeActivity
{
}
```

With that in place, the UI of your activity is almost complete, as you can see in Figure 57-13.

Now, you need to define a property on each of the sequence activities shown in Figure 57-13, so that the user can define which day(s) the branch will execute. There are two ways to do this in Windows Workflow: you can create a subclass of `SequenceActivity` and define it there, or you can use another feature of dependency properties called `Attached Properties`.

You will use the latter method, because this means that you don't have to subclass but instead can effectively extend the sequence activity without needing the source code of that activity.

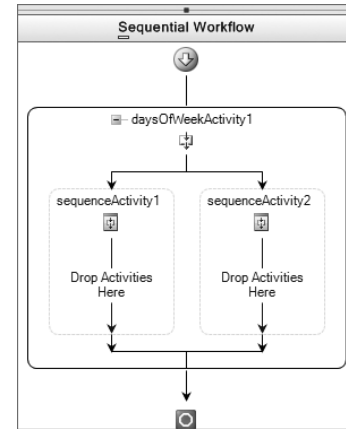


FIGURE 57-13

Attached Properties

When registering dependency properties, you can call the `RegisterAttached` method to create an attached property. An attached property is one that is defined on one class but is displayed on another. So here, you define a property on the `DaysOfWeekActivity`, but that property is actually displayed in the UI as attached to a sequential activity.

The code in the following snippet shows a property called `Weekday` of type `WeekdayEnum`, which will be added to the sequence activities that reside within your composite activity:

```
public static DependencyProperty WeekdayProperty =
    DependencyProperty.RegisterAttached("Weekday",
        typeof(WeekdayEnum), typeof(DaysOfWeekActivity),
        new PropertyMetadata(DependencyPropertyOptions.Metadata));
```

The final line allows you to specify extra information about a property. In this instance, it is specifying that it is a `Metadata` property.

Metadata properties differ from normal properties in that they are effectively read only at runtime. You can think of a `Metadata` property as similar to a constant declaration within C#. You cannot alter constants while the program is executing, and you cannot change `Metadata` properties while a workflow is executing.

In this example, you wish to define the days that the activity will execute, so you could in the Designer set this field to “Saturday, Sunday”. In the code emitted for the workflow, you would see a declaration as follows (I have reformatted the code to fit the confines of the page):

```
this.sequenceActivity1.SetValue
    (DaysOfWeekActivity.WeekdayProperty,
        ((WeekdayEnum)((WeekdayEnum.Sunday | WeekdayEnum.Saturday))));
```


In addition to defining the dependency property, you will need methods to get and set this value on an arbitrary activity. These are typically defined as static methods on the composite activity and are shown in the following code:

```
public static void SetWeekday(Activity activity, object value)
{
    if (null == activity)
        throw new ArgumentNullException("activity");
    if (null == value)
        throw new ArgumentNullException("value");

    activity.SetValue(DaysOfWeekActivity.WeekdayProperty, value);
}

public static object GetWeekday(Activity activity)
{
    if (null == activity)
        throw new ArgumentNullException("activity");

    return activity.GetValue(DaysOfWeekActivity.WeekdayProperty);
}
```

You need to make two other changes in order for this extra property to show up attached to a *SequenceActivity*. The first is to create an *extender provider*, which tells Visual Studio to include the extra property in the sequence activity. The second is to register this provider, which is done by overriding the *Initialize* method of the Activity Designer and adding the following code to it:

```
protected override void Initialize(Activity activity)
{
    base.Initialize(activity);

    IExtenderListService iels = base.GetService(typeof(IExtenderListService))
        as IExtenderListService;

    if (null != iels)
    {
        bool extenderExists = false;

        foreach (IExtenderProvider provider in iels.GetExtenderProviders())
        {
            if (provider.GetType() == typeof(WeekdayExtenderProvider))
            {
                extenderExists = true;
                break;
            }
        }
        if (!extenderExists)
        {
            IExtenderProviderService ieps =
                base.GetService(typeof(IExtenderProviderService))
                    as IExtenderProviderService;
            if (null != ieps)
                ieps.AddExtenderProvider(new WeekdayExtenderProvider());
        }
    }
}
```

The calls to *GetService* in the preceding code allow the custom Designer to query for services proffered by the host (in this case Visual Studio). You query Visual Studio for the *IExtenderListService*, which provides a way to enumerate all available extender providers, and if no instance of the *WeekdayExtenderProvider* service is found, then query for the *IExtenderProviderService* and add a new provider.

The code for the extender provider is shown here:

```
[ProvideProperty("Weekday", typeof(SequenceActivity))]
public class WeekdayExtenderProvider: IExtenderProvider
{
    bool IExtenderProvider.CanExtend(object extendee)
    {
        bool canExtend = false;

        if ((this != extendee) && (extendee is SequenceActivity))
        {
            Activity parent = ((Activity)extendee).Parent;

            if (null != parent)
                canExtend = parent is DaysOfWeekActivity;
        }

        return canExtend;
    }

    public WeekdayEnum GetWeekday(Activity activity)
    {
        WeekdayEnum weekday = WeekdayEnum.None;

        Activity parent = activity.Parent;

        if ((null != parent) && (parent is DaysOfWeekActivity))
            weekday = (WeekdayEnum)DaysOfWeekActivity.GetWeekday(activity);

        return weekday;
    }

    public void SetWeekday(Activity activity, WeekdayEnum weekday)
    {
        Activity parent = activity.Parent;

        if ((null != parent) && (parent is DaysOfWeekActivity))
            DaysOfWeekActivity.SetWeekday(activity, weekday);
    }
}
```

An extender provider is attributed with the properties that it provides, and for each of these properties it must provide a public `Get<Property>` and `Set<Property>` method. The names of these methods must match the name of the property with the appropriate *Get* or *Set* prefix.

With the preceding changes made to the Designer and the addition of the extender provider, when you add a sequence activity within the Designer (shown here as `sequenceActivity1`), you will see the properties in Figure 57-14 within Visual Studio.

Extender providers are used for other features in .NET. One common one is to add tooltips to controls in a Windows Forms project — this registers an extender and adds a `Tooltip` property to each control on the form.

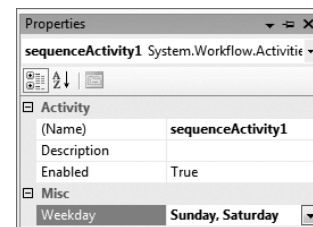


FIGURE 57-14

WORKFLOWS

Up to this point, the chapter has concentrated on activities but has not discussed workflows. A workflow is simply a list of activities, and indeed a workflow itself is just another type of activity. Using this model simplifies the runtime engine, because the engine just needs to know how to execute one type of object — that being anything derived from the `Activity` class.

Each workflow instance is uniquely identified by its `InstanceId` property — this is a `Guid` that can be assigned by the runtime, or this `Guid` can be provided to the runtime by your code. A common use of this is to correlate a running workflow instance with some other data maintained outside of the workflow, such as a row in a database. You can access the specific workflow instance by using the `GetWorkflow(Guid)` method of the `WorkflowRuntime` class.

Two types of workflows are available with WF — sequential and state machine.

Sequential Workflows

The root activity in a sequential workflow is the `SequentialWorkflowActivity`. This class is derived from `SequenceActivity`, which you have already seen, and it defines two events that you can attach handlers to as necessary. These are the `Initialized` and `Completed` events.

A sequential workflow starts executing the first child activity within it, and typically continues until all other activities have executed. There are a couple of instances when a workflow will not continue through all activities — one is if an exception is raised while executing the workflow, and the other is if a `TerminateActivity` exists within the workflow.

A workflow may not be executing at all times. For example, when a `DelayActivity` is encountered, the workflow will enter a wait state and can be removed from memory if a workflow persistence service is defined. Persistence of workflows is covered in “The Persistence Service” section later in this chapter.

State Machine Workflows

A state machine workflow is useful when you have a process that may be in one of several states, and transitions from one state to another can be made by passing data into the workflow.

One example is when a workflow is used for access control to a building. In this case, you may model a door class that can be closed or open, and a lock class that can be locked or unlocked. Initially when you boot up the system (or building!), you start at a known state — for sake of argument, assume that all doors are closed and locked, so the state of a given door is *closed locked*.

When an employee enters his or her building access code at the front door, an event is sent to the workflow, which includes details such as the code entered and possibly the user ID. You might then need to access a database to retrieve details such as whether that person is permitted to open the selected door at that time of day, and assuming that access is granted, the workflow would change from its initial state to the *closed unlocked* state.

From this state, there are two potential outcomes — the employee opens the door (you know this because the door also has an open/closed sensor), or the employee decides not to enter because he has left something in his car, and so after a delay you relock the door. The door could revert to its *closed locked* state or move to the *open unlocked* state.

From here, assume that the employee enters the building and then closes the door. Again, you would then like to transition from the *open unlocked* state to *closed unlocked*, and again, after a delay, would then transition to the *closed locked* state. You might also want to raise an alarm if the door was *open unlocked* for a long period.

Modeling this scenario within Windows Workflow is fairly simple. You need to define the states that the system can be in, and then define events that can transition the workflow from one state to the next. The following table describes the states of the system and provides details of the transitions that are possible from each known state and the inputs (either external or internal) that change the states.

STATE	TRANSITIONS
Closed Locked	This is the initial state of the system. In response to the user swiping her card (and a successful access check), the state changes to closed unlocked, and the door lock is electronically opened.
Closed Unlocked	One of two events can occur when the door is in this state: The user opens the door — you transition to the open unlocked state. A timer expires, and the door reverts to the closed locked state.
Open Unlocked	From this state, the workflow can only transition to closed unlocked.
Fire Alarm	This is the final state for the workflow and can be transitioned to from any of the other states.

One other feature you might want to add to the system is the capability to respond to a fire alarm. When the fire alarm goes off, you would want to unlock all of the doors so that anyone can exit the building, and the fire service can enter the building unimpeded. You might want to model this as the final state of the doors workflow, because from this state the full system would be reset once the fire alarm had been canceled.

The workflow in Figure 57-15 defines this state machine and shows the states that the workflow can be in. The lines denote the state transitions that are possible within the system.

The initial state of the workflow is modeled by the `ClosedLocked` activity. This consists of some initialization code (which locks the door) and then an event-based activity that awaits an external event — in this case, the employee entering his building access code. Each of the activities shown within the state shapes consist of sequential workflows, so we have defined a workflow for the initialization of the system (`CLInitialize`) and a workflow that responds to the external event raised when the employee enters her PIN (`RequestEntry`). If you look at the `RequestEntry` workflow, it is defined as shown in Figure 57-16.

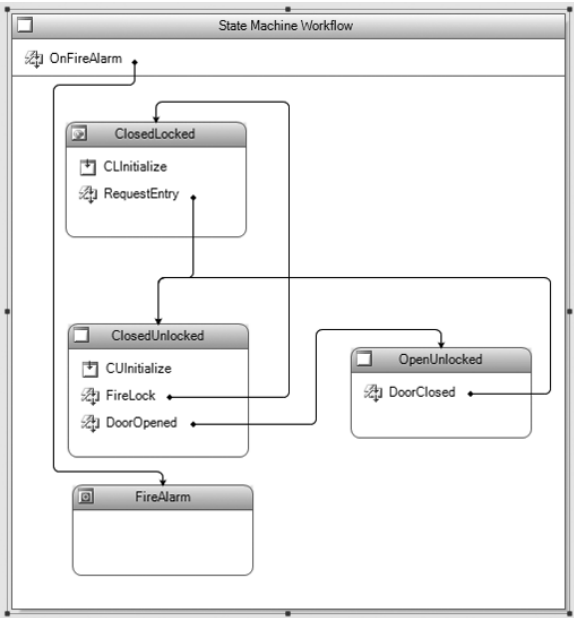


FIGURE 57-15

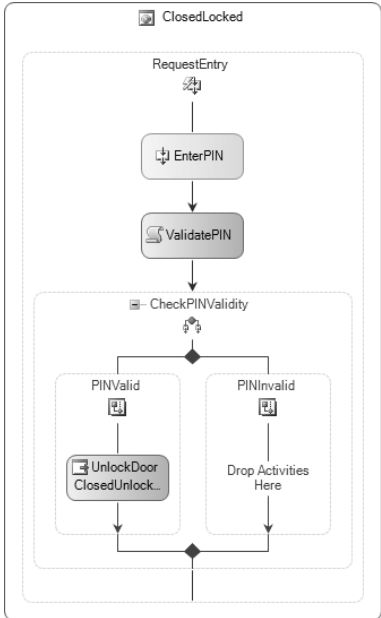


FIGURE 57-16

Each state consists of a number of subworkflows, each of which has an event-driven activity at the start and then any number of other activities that form the processing code within the state. In Figure 57-16, there is a `HandleExternalEventActivity` at the start that awaits the entry of the PIN. This is then checked, and if it is valid, the workflow transitions to the `ClosedUnlocked` state.

The `ClosedUnlocked` state consists of two workflows — one that responds to the door open event, which transitions the workflow to the `OpenUnlocked` state, and another that contains a delay activity that is used to change the state to `ClosedLocked`. A state-driven activity works in a similar manner to the `ListenActivity` shown earlier in the chapter — the state consists of a number of event-driven workflows, and on arrival of an event, just one of the workflows will execute.

To support the workflow, you need to be able to raise events in the system to affect the state changes. This is done by using an interface and an implementation of that interface; this pair of objects is termed an *external service*. The interface used for this state machine is described later in the chapter.

The code for the state machine example is available in the 04 `StateMachine` solution. This also includes a user interface in which you can enter a PIN and gain access to the building through one of two doors.

Passing Parameters to a Workflow

A typical workflow requires some data in order to execute. This could be an order ID for an order-processing workflow, a customer account ID for a payment-processing workflow, or any other items of data necessary.

The parameter-passing mechanism for workflows is somewhat different from that of standard .NET classes, in which you typically pass parameters in a method call. For a workflow, you pass parameters by storing those parameters in a dictionary of name-value pairs, and when you construct the workflow, you pass through this dictionary.

When WF schedules the workflow for execution, it uses these name-value pairs to set public properties on the workflow instance. Each parameter name is checked against the public properties of the workflow, and if a match is found, the property setter is called and the value of the parameter is passed to this setter. If you add a name-value pair to the dictionary where the name does not correspond to a property on the workflow, an exception will be thrown when you try to construct that workflow.

As an example, consider the following workflow, which defines the `OrderID` property as an integer:

```
public class OrderProcessingWorkflow: SequentialWorkflowActivity
{
    public int OrderID
    {
        get { return _orderID; }
        set { _orderID = value; }
    }

    private int _orderID;
}
```

The following snippet shows how you can pass the order ID parameter into an instance of this workflow:

```
WorkflowRuntime runtime = new WorkflowRuntime ();

Dictionary<string,object> parms = new Dictionary<string,object>();
parms.Add("OrderID", 12345);

WorkflowInstance instance = runtime.CreateWorkflow(typeof(OrderProcessingWorkflow), parms);

instance.Start();

. Other code
```

In the example code, you construct a `Dictionary<string, object>` that will contain the parameters you wish to pass to the workflow and then use this when the workflow is constructed. The preceding code includes the `WorkflowRuntime` and `WorkflowInstance` classes, which haven't been described yet but are discussed in the "Hosting Workflows" section later in the chapter.

Returning Results from a Workflow

Another common requirement of a workflow is to return output parameters, which might then be used to record data within a database or other persistent storage.

Because a workflow is executed by the workflow runtime, you can't just call a workflow using a standard method invocation — you need to create a workflow instance, start that instance, and then await the completion of that instance. When a workflow completes, the workflow runtime raises the `WorkflowCompleted` event. This is passed contextual information about the workflow that has just completed and contains the output data from that workflow.

So, to harvest the output parameters from a workflow, you need to attach an event handler to the `WorkflowCompleted` event, and the handler can then retrieve the output parameters from the workflow. The following code shows an example of how this can be done:

```
using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())
{
    AutoResetEvent waitHandle = new AutoResetEvent(false);
    workflowRuntime.WorkflowCompleted +=
        delegate(object sender, WorkflowCompletedEventArgs e)
        {
            waitHandle.Set();
            foreach (KeyValuePair<string, object> parm in e.OutputParameters)
            {
                Console.WriteLine("{0} = {1}", parm.Key, parm.Value);
            }
        };

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

    waitHandle.WaitOne();
}
```

You have attached a delegate to the `WorkflowCompleted` event, and within this you iterate through the `OutputParameters` collection of the `WorkflowCompletedEventArgs` class passed to the delegate and display the output parameters on the console. This collection contains all public properties of the workflow. There is actually no notion of specific output parameters for a workflow.

Binding Parameters to Activities

Now that you know how to pass parameters into a workflow, you also need to know how to link these parameters to activities. This is done via a mechanism called binding. In the `DaysOfWeekActivity` defined earlier, there was a `Date` property that could be hard-coded or bound to another value within the workflow. A bindable property is displayed in the property grid within Visual Studio, as shown in Figure 57-17. In the image the `Date` property is bindable, as indicated by the small icon to the right of the property name.

Double-clicking the bind icon will display the dialog shown in Figure 57-18. This dialog allows you to select an appropriate property to link to the `Date` property.

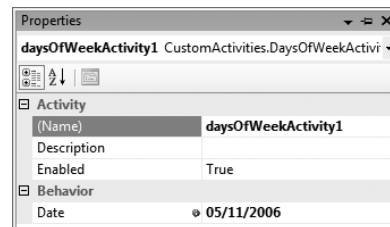


FIGURE 57-17

In Figure 57-18, we have selected the `OrderDate` property of a workflow (`OrderDate` is defined as a regular .NET property on the workflow). Any bindable property can be bound to either a property of the workflow that the activity is defined within or a property of any activity that resides in the workflow above the current activity. Note that the data type of the property being bound must match the data type of the property you are binding to — the dialog will not permit you to bind nonmatching types.

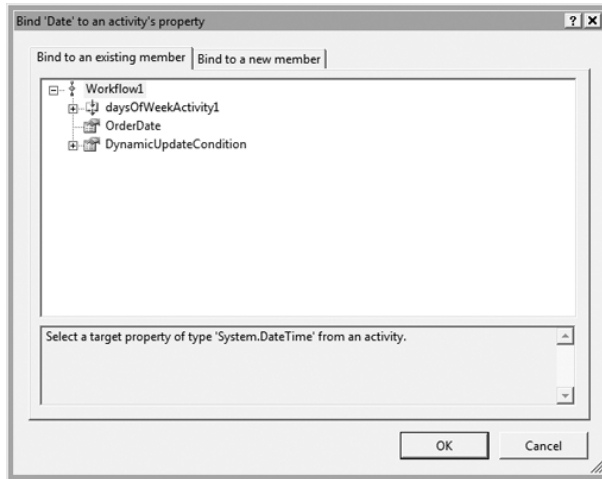


FIGURE 57-18

The code for the `Date` property is shown here to show how binding works and is explained in the following paragraphs:

```
public DateTime Date
{
    get { return (DateTime)base.GetValue(DaysOfWeekActivity.DateProperty); }
    set { base.SetValue(DaysOfWeekActivity.DateProperty, value); }
}
```

When you bind a property in the workflow, an object of type `ActivityBind` is constructed behind the scenes, and it is this “value” that is stored within the dependency property. So, the property setter will be passed an object of type `ActivityBind`, and this is stored within the dictionary of properties on this activity. This `ActivityBind` object consists of data that describes the activity being bound to and the property of that activity that is to be used at runtime.

When reading the value of the property, the `GetValue` method of the `DependencyObject` is called, and this method checks the underlying property value to see if it is an `ActivityBind` object. If so, it resolves the activity to which this binding is linked and then reads the real property value from that activity. If, however, the bound value is another type, it simply returns that object from the `GetValue` method.

THE WORKFLOW RUNTIME

In order to start a workflow, it is necessary to create an instance of the `WorkflowRuntime` class. This is typically done once within your application, and this object is usually defined as a static member of the application so that it can be accessed anywhere within the application.

When you start the runtime, it can then reload any workflow instances that were executing the last time the application was executed by reading these instances from the persistence store. This uses a service called the *persistence service*, which is defined in the following section.

The runtime contains six various `CreateWorkflow` methods that can be used to construct workflow instances. The runtime also contains methods for reloading a workflow instance and enumerating all running instances.

The runtime also has a number of events that are raised while workflows are executing — such as `WorkflowCreated` (raised when a new workflow instance is constructed), `WorkflowIdled` (raised when a workflow is awaiting input such as in the expense-processing example shown earlier), and `WorkflowCompleted` (raised when a workflow has finished).

WORKFLOW SERVICES

A workflow doesn't exist on its own. As described in the previous section, a workflow is executed within the `WorkflowRuntime`, and this runtime provides *services* to running workflows.

A service is any class that may be needed while executing the workflow. Some standard services are provided to your workflow by the runtime, and you can optionally construct your own services to be consumed by running workflows.

This section describes two of the standard services provided by the runtime. It then shows how you can create your own services and some instances of when this is necessary.

When an activity runs, it is passed some contextual information via the `ActivityExecutionContext` parameter of the `Execute` method:

```
protected override ActivityExecutionContext Execute
    (ActivityExecutionContext executionContext)
{
    .
}
}
```

One of the methods available on this context parameter is the `GetService<T>` method. This can be used as shown in the following code to access a service attached to the workflow runtime:

```
protected override ActivityExecutionContext Execute
    (ActivityExecutionContext executionContext)
{
    ICustomService myService = executionContext.GetService<ICustomService>();
    . Do something with the service
}
}
```

The services hosted by the runtime are added to the runtime prior to calling the `StartRuntime` method. An exception is raised if you attempt to add a service to the runtime once it has been started.

Two methods are available for adding services to the runtime. You can construct the services in code and then add them to the runtime by calling the `AddService` method. Or, you can define services within the application configuration file, and these will be constructed for you and added to the runtime.

The following code snippet shows how to add services to the runtime in code — the services added are those described later in this section:

```
using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())
{
    workflowRuntime.AddService(
        new SqlWorkflowPersistenceService(conn, true, new TimeSpan(1,0,0),
                                           new TimeSpan(0,10,0)));
    workflowRuntime.AddService(new SqlTrackingService(conn));
    .
}
```

Here are constructed instances of the `SqlWorkflowPersistenceService`, which is used by the runtime to store workflow state, and an instance of the `SqlTrackingService`, which records the execution events of a workflow while it runs.

To create services using an application configuration file, you need to add a section handler for the workflow runtime and then add services to this section as shown here:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="WF"
      type="System.Workflow.Runtime.Configuration.WorkflowRuntimeSection,
      System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,
      PublicKeyToken=31bf3856ad364e35" />
  </configSections>

  <WF Name="Hosting">
    <CommonParameters/>
    <Services>
      <add type="System.Workflow.Runtime.Hosting.SqlWorkflowPersistenceService,
        System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,
        PublicKeyToken=31bf3856ad364e35"
        connectionString="Initial Catalog=WF;Data Source=.;
        Integrated Security=SSPI;"
        UnloadOnIdle="true"
        LoadIntervalSeconds="2"/>
      <add type="System.Workflow.Runtime.Tracking.SqlTrackingService,
        System.Workflow.Runtime, Version=3.0.00000.0, Culture=neutral,
        PublicKeyToken=31bf3856ad364e35"
        connectionString="Initial Catalog=WF;Data Source=.;
        Integrated Security=SSPI;"
        UseDefaultProfile="true"/>
    </Services>
  </WF>
</configuration>
```

Within the configuration file, you have added the WF section handler (the name is unimportant but must match the name given to the later configuration section) and then created the appropriate entries for this section. The `<Services>` element can contain an arbitrary list of entries that consist of a .NET type and then parameters that will be passed to that service when constructed by the runtime.

To read the configuration settings from the application configuration file, you call another constructor on the runtime, as shown here:

```
using (WorkflowRuntime workflowRuntime = new WorkflowRuntime("WF"))
{
    .
}
}
```

This constructor will instantiate each service defined within the configuration file and add these to the services collection on the runtime.

The following sections describe some of the standard services available with WF.

The Persistence Service

When a workflow executes, it may reach a wait state. This can occur when a delay activity executes or when you are waiting for external input within a `ListenActivity`. At this point, the workflow is said to be *idle* and as such is a candidate for persistence.

Let's assume that you begin execution of 1,000 workflows on your server, and each of these instances becomes idle. At this point, it is unnecessary to maintain data for each of these instances in memory, so it would be ideal if you could unload a workflow and free up the resources in use. The persistence service is designed to accomplish this.

When a workflow becomes idle, the workflow runtime checks for the existence of a service that derives from the `WorkflowPersistenceService` class. If this service exists, it is passed the workflow instance, and the

service can then capture the current state of the workflow and store it in a persistent storage medium. You could store the workflow state on disk in a file, or store this data within a database such as SQL Server.

The workflow libraries contain an implementation of the persistence service, which stores data within a SQL Server database — this is the `SqlWorkflowPersistenceService`. In order to use this service, you need to run two scripts against your SQL Server instance. One of these constructs the schema, and the other creates the stored procedures used by the persistence service. These scripts are, by default, located in the `C:\Windows\Microsoft.NET\Framework\v3.5\Windows Workflow Foundation\SQL\EN` directory.

The scripts to execute against the database are `SqlPersistenceProviderSchema.sql` and `SqlPersistenceProviderLogic.sql`. These need to be executed in order, with the schema file first and then the logic file. The schema for the SQL persistence service contains two tables: `InstanceState` and `CompletedScope`. These are essentially opaque tables, and they are not intended for use outside the SQL persistence service.

When a workflow idles, its state is serialized using binary serialization, and this data is then inserted into the `InstanceState` table. When a workflow is reactivated, the state is read from this row and used to reconstruct the workflow instance. The row is keyed on the workflow instance ID and is deleted from the database once the workflow has completed.

The SQL persistence service can be used by multiple runtimes at the same time — it implements a locking mechanism so that a workflow is accessible by only one instance of the workflow runtime at a time. When you have multiple servers all running workflows using the same persistence store, this locking behavior becomes invaluable.

To see what is added to the persistence store, construct a new workflow project and add an instance of the `SqlWorkflowPersistenceService` to the runtime. The following code shows an example using declarative code:

```
using(WorkflowRuntime workflowRuntime = new WorkflowRuntime())
{
    workflowRuntime.AddService(
        new SqlWorkflowPersistenceService(conn, true, new TimeSpan(1,0,0),
                                          new TimeSpan(0,10,0)));
    // Execute a workflow here.
}
```

If you then construct a workflow that contains a `DelayActivity` and set the delay to something like 10 seconds, you can then view the data stored within the `InstanceState` table. The 05 `WorkflowPersistence` example contains the preceding code and executes a delay within a 20-second period.

The parameters passed to the constructor of the persistence service are shown in the following table.

PARAMETER	DESCRIPTION	DEFAULT
ConnectionString	The database connection string used by the persistence service.	None
UnloadOnIdle	Determines whether a workflow is unloaded when it idles. This should always be set to true; otherwise no persistence will occur.	False
InstanceOwnershipDuration	This defines the length of time that the workflow instance will be owned by the runtime that has loaded that workflow.	None
LoadingInterval	The interval used when polling the database for updated persistence records.	2 Minutes

These values can also be defined within the configuration file.

The Tracking Service

When a workflow executes, it might be necessary to record which activities have run, and in the case of composite activities such as the `IfElseActivity` or the `ListenActivity`, which branch was executed. This data could be used as a form of audit trail for a workflow instance, which could then be viewed at a later date to prove which activities executed and what data was used within the workflow. The tracking service can be used for this type of recording and can be configured to log as little or as much information about a running workflow instance as is necessary.

As is common with WF, the tracking service is implemented as an abstract class called `TrackingService`, so it is easy to replace the standard tracking implementation with one of your own. There is one concrete implementation of the tracking service available within the workflow assemblies — this is the `SqlTrackingService`.

To record data about the state of a workflow, it is necessary to define a `TrackingProfile`. This defines which events should be recorded, so you could, for example, record just the start and end of a workflow and omit all other data about the running instance. More typically, you will record all events for the workflow and each activity in that workflow to provide a complete picture of the execution profile of the workflow.

When a workflow is scheduled by the runtime engine, the engine checks for the existence of a workflow tracking service. If one is found, it asks the service for a tracking profile for the workflow being executed, and then uses this to record workflow and activity data. You can, in addition, define user tracking data and store this within the tracking data store without needing to change the schema.

The tracking profile class is shown in Figure 57-19. The class includes collection properties for activity, user, and workflow *track points*. A track point is an object (such as `WorkflowTrackPoint`) that typically defines a *match location* and some extra data to record when this track point is hit. The match location defines where this track point is valid — so for example, you could define a `WorkflowTrackPoint`, which will record some data when the workflow is created, and another to record some data when the workflow is completed.

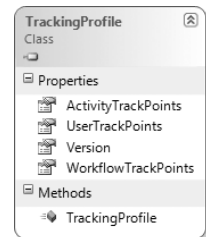


FIGURE 57-19

Once this data has been recorded, it may be necessary to display the execution path of a workflow, as in Figure 57-20. This shows the workflow that was executed, and each activity that ran includes a glyph to show that it executed. This data is read from the tracking store for that workflow instance.

To read the data stored by the `SqlTrackingService`, you could execute queries against the SQL database directly; however, Microsoft has provided the `SqlTrackingQuery` class defined within the `System.Workflow.Runtime.Tracking` namespace for this purpose. The following example code shows how to retrieve a list of all workflows tracked between two dates:

```
public IList<SqlTrackingWorkflowInstance> GetWorkflows
(DateTime startDate, DateTime endDate, string connectionString)
{
    SqlTrackingQuery query = new SqlTrackingQuery (connectionString);

    SqlTrackingQueryOptions queryOptions = new SqlTrackingQueryOptions();
    query.StatusMinDateTime = startDate;
    query.StatusMaxDateTime = endDate;

    return (query.GetWorkflows (queryOptions));
}
```

This uses the `SqlTrackingQueryOptions` class, which defines the query parameters. You can define other properties of this class to further constrain the workflows retrieved.

In Figure 57-20 you can see that all activities have executed. This might not be the case if the workflow were still running or if there were some decisions made within the workflow so that different paths were taken during execution. The tracking data contains details such as which activities have executed, and this data

can be correlated with the activities to produce the image in Figure 57-20. It is also possible to extract data from the workflow as it executes, which could be used to form an audit trail of the execution flow of the workflow.

Custom Services

In addition to built-in services such as the persistence service and the tracking service, you can add your own objects to the services collection maintained by the `WorkflowRuntime`. These services are typically defined using an interface and an implementation, so that you can replace the service without recoding the workflow.

The state machine presented earlier in the chapter uses the following interface:

```
[ExternalDataExchange]
public interface IDoorService
{
    void LockDoor();
    void UnlockDoor();

    event EventHandler<ExternalDataEventArgs> RequestEntry;
    event EventHandler<ExternalDataEventArgs> OpenDoor;
    event EventHandler<ExternalDataEventArgs> CloseDoor;
    event EventHandler<ExternalDataEventArgs> FireAlarm;

    void OnRequestEntry(Guid id);
    void OnOpenDoor(Guid id);
    void OnCloseDoor(Guid id);
    void OnFireAlarm();
}
```

The interface consists of methods that are used by the workflow to call the service and events raised by the service that are consumed by the workflow. The use of the `ExternalDataExchange` attribute indicates to the workflow runtime that this interface is used for communication between a running workflow and the service implementation.

Within the state machine, there are a number of instances of the `CallExternalMethodActivity` that are used to call methods on this external interface. One example is when the door is locked or unlocked — the workflow needs to execute a method call to the `UnlockDoor` or `LockDoor` methods, and the service responds by sending a command to the door lock to unlock or lock the door.

When the service needs to communicate with the workflow, this is done by using an event, because the workflow runtime also contains a service called the `ExternalDataExchangeService`, which acts as a proxy for these events. This proxy is used when the event is raised, because the workflow may not be loaded in memory at the time the event is delivered. So the event is first routed to the external data exchange service, which checks to see if the workflow is loaded, and, if not, rehydrates it from the persistence store and then passes the event on into the workflow.

The code used to construct the `ExternalDataExchangeService` and to construct proxies for the events defined by the service is shown here:

```
WorkflowRuntime runtime = new WorkflowRuntime();
ExternalDataExchangeService edes = new ExternalDataExchangeService();

runtime.AddService(edes);
DoorService service = new DoorService();
edes.AddService(service);
```

This constructs an instance of the external data exchange service and adds it to the runtime. It then creates an instance of the `DoorService` (which itself implements `IDoorService`) and adds this to the external data exchange service.

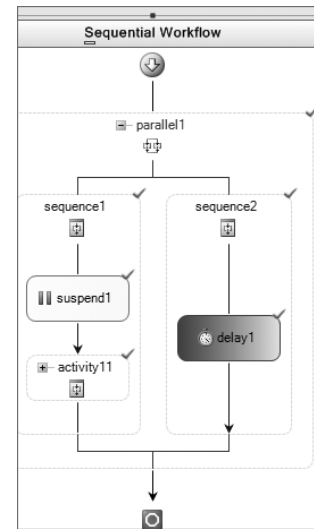


FIGURE 57-20

The `ExternalDataExchangeService.Add` method constructs a proxy for each event defined by the custom service so that a persisted workflow can be loaded prior to delivery of the event. If you don't host your service within the external data exchange service, when you raise events there will be nothing listening to these events, so they will not be delivered to the correct workflow.

Events use the `ExternalDataEventArgs` class, because this includes the workflow instance ID that the event is to be delivered to. If there are other values that need to be passed from an external event to a workflow, you should derive a class from `ExternalDataEventArgs` and add these values as properties to that class.

INTEGRATION WITH WINDOWS COMMUNICATION FOUNDATION

Two activities are available starting with .NET 3.5 that support integration between workflows and WCF. These are the `SendActivity` and the `ReceiveActivity`. The `SendActivity` could more aptly be called the `CallActivity`, because what it does is issue a request to a WCF service and can optionally surface the results as parameters that can be bound within the calling workflow.

Somewhat more interesting, however, is the `ReceiveActivity`. This allows a workflow to become the implementation of a WCF service, so now the workflow is the service. The following example exposes a service using a workflow and also uses the new service test host tool to test the service without having to write a separate test harness.

From the New Project menu in Visual Studio 2010, choose the WCF node and then the Sequential Workflow Service Library entry as shown in Figure 57-21.

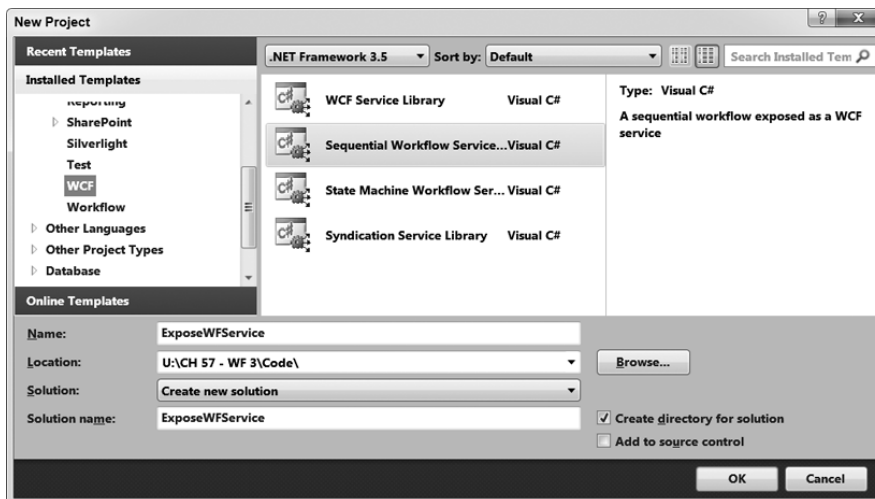


FIGURE 57-21

This will create a library that contains a workflow, as shown in Figure 57-22, an application configuration file, and a service interface. The code for this example is available in the 06 `ExposeWFSERVICE` subdirectory.

The workflow exposes the `Hello` operation of the contract and also defines properties for the arguments passed to this operation, and the return value of the operation. Then, all you need to do is to add code that provides the execution behavior of the service, and your service is complete.

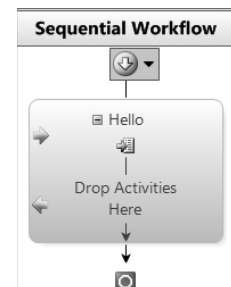


FIGURE 57-22

To do this for the example, drag a `CodeActivity` onto the `ReceiveActivity`, as shown in Figure 57-23, and then double-click that activity to supply the service implementation.

The code shown in the following snippet is all that there is to this service implementation:

```
public sealed partial class Workflow1: SequentialWorkflowActivity
{
    public Workflow1()
    {
        InitializeComponent();
    }

    public String returnValue = default(System.String);
    public String inputMessage = default(System.String);

    private void codeActivity1_ExecuteCode(object sender, EventArgs e)
    {
        this.returnValue = string.Format("You said {0}", inputMessage);
    }
}
```

Because the service contract for the `Hello` operation includes both a parameter (`inputMessage`) and a return value, these have been exposed to the workflow as public fields. Within the code, we have set the `returnValue` to a string value, and this is what is returned from a call to the WCF service.

If you compile this service and hit F5, you will notice a feature of Visual Studio 2010 — the WCF Test Client application, as shown in Figure 57-24.

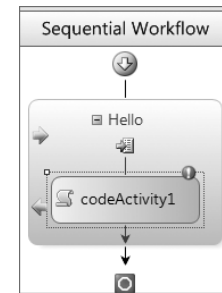


FIGURE 57-23

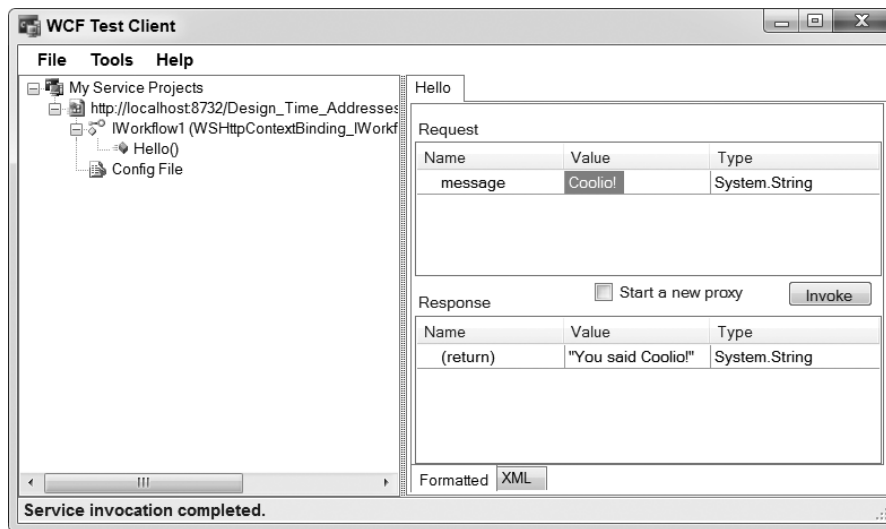


FIGURE 57-24

Here you can browse for the operations that the service exposes, and by double-clicking an operation, the right-hand side of the window is displayed, which lists the parameters used by that service and any return value(s) exposed.

To test the service, enter a value for the message property and click the `Invoke` button. This will then make a request over WCF to the service, which will construct and execute the workflow, call the code activity, which then runs the code-behind, and ultimately return to the WCF Test Client the result from the workflow.

If you wish to manually host workflows as services, you can use the new `WorkflowServiceHost` class defined within the `System.WorkflowServices` assembly. The following snippet shows a minimal host implementation:

```
using (WorkflowServiceHost host = new WorkflowServiceHost
    (typeof(YourWorkflow)))
{
    host.Open();
    Console.WriteLine ( "Press [Enter] to exit" );
    Console.ReadLine();
}
```

Here we have constructed an instance of `WorkflowServiceHost` and passed it the workflow that will be executed. This is similar to how you would use the `ServiceHost` class when hosting WCF services. It will read the configuration file to determine which endpoints the service will listen on and then await service requests.

The next section describes some other options you have for hosting workflows.

HOSTING WORKFLOWS

The code to host the `WorkflowRuntime` in a process will vary based on the application itself.

For a Windows Forms application or a Windows Service, it is typical to construct the runtime at the start of the application and store this in a property of the main application class.

In response to some input in the application (such as the user clicking a button on the user interface), you might then construct an instance of a workflow and execute this instance locally. The workflow may well need to communicate with the user — so, for example, you might define an external service that prompts the user for confirmation before posting an order to a back-end server.

When hosting workflows within ASP.NET, you would not normally prompt the user with a message box but instead navigate to a different page on the site that requested the confirmation and then present a confirmation page. When hosting the runtime within ASP.NET, it is typical to override the `Application_Start` event and construct an instance of the workflow runtime there so that it is accessible within all other parts of the site. You can store the runtime instance in a static property, but it is more usual to store this in application state and provide an accessor method that will retrieve the workflow runtime from application state so that it can be used elsewhere in the application.

In either scenario — Windows Forms or ASP.NET — you will construct an instance of the workflow runtime and add services to it as shown here:

```
WorkflowRuntime workflowRuntime = new WorkflowRuntime();

workflowRuntime.AddService(
    new SqlWorkflowPersistenceService(conn, true, new TimeSpan(1,0,0),
                                     new TimeSpan(0,10,0)));

// Execute a workflow here.
```

To execute a workflow, you need to create an instance of that workflow using the `CreateInstance` method of the runtime. There are a number of overrides of this method that can be used to construct an instance of a code-based workflow or a workflow defined in XML.

Up to this point in the chapter, you have considered workflows as .NET classes — and indeed that is one representation of a workflow. You can, however, define a workflow using XML, and the runtime will construct an in-memory representation of the workflow and then execute it when you call the `Start` method of the `WorkflowInstance`.

Within Visual Studio, you can create an XML-based workflow by choosing the Sequential Workflow (with code separation) or the State Machine Workflow (with code separation) items from the Add New Item dialog. This will create an XML file with the extension `.xoml` and load it into the Designer.

When you add activities to the Designer, these activities are persisted into the XML, and the structure of elements defines the parent/child relationships between the activities. The following XML shows a simple sequential workflow that contains an `IfElseActivity` and two code activities, one on each branch of the `IfElseActivity`:

```
<SequentialWorkflowActivity x:Class="DoorsWorkflow.Workflow1" x:Name="Workflow1"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/workflow">
  <IfElseActivity x:Name="ifElseActivity1">
    <IfElseBranchActivity x:Name="ifElseBranchActivity1">
      <IfElseBranchActivity.Condition>
        <CodeCondition Condition="Test" />
      </IfElseBranchActivity.Condition>
      <CodeActivity x:Name="codeActivity1" ExecuteCode="DoSomething" />
    </IfElseBranchActivity>
    <IfElseBranchActivity x:Name="ifElseBranchActivity2">
      <CodeActivity x:Name="codeActivity2" ExecuteCode="DoSomethingElse" />
    </IfElseBranchActivity>
  </IfElseActivity>
</SequentialWorkflowActivity>
```

The properties defined on the activities are persisted into the XML as attributes, and each activity is persisted as an element. As you can see from the XML, the structure defines the relationship between parent activities (such as the `SequentialWorkflowActivity` and the `IfElseActivity`) and the child activities.

Executing an XML-based workflow is no different from executing a code-based workflow — you simply use an override of the `CreateWorkflow` method that takes an `XmlReader` instance, and then start that instance by calling the `Start` method.

One benefit of using XML-based workflows over code-based workflows is that you can then easily store the workflow definition within a database. You can load up this XML at runtime and execute the workflow, and you can very easily make changes to the workflow definition without having to recompile any code.

Changing a workflow at runtime is also supported whether the workflow is defined in XML or code. You construct a `WorkflowChanges` object, which contains all of the new activities to be added to the workflow, and then call the `ApplyWorkflowChanges` method defined on the `WorkflowInstance` class to persist these changes. This is exceptionally useful, because business needs often change and, for example, you might want to apply changes to an insurance policy workflow so that you send an e-mail to the customers a month prior to the renewal date to let them know their policy is due for renewal. Changes are made on an instance-by-instance basis, so if you had 100 policy workflows in the system, you would need to make these changes to each individual workflow.

THE WORKFLOW DESIGNER

To complete this chapter, we've left the best until last. The Workflow Designer that you use to design workflows isn't tied to Visual Studio — you can rehost this Designer within your own application as necessary.

This means that you could deliver a system containing workflows and permit end users to customize the system without requiring them to have a copy of Visual Studio. Hosting the Designer is, however, fairly complex, and we could devote several chapters to this one topic, but we are out of space. A number of examples of rehosting are available on the Web — we recommend reading the MSDN article on hosting the Designer available at <http://msdn2.microsoft.com/en-us/library/aa480213.aspx> for more information.

The traditional way of allowing users to customize a system is by defining an interface and then allowing the customer to implement this interface to extend processing as required.

With Windows Workflow that extension becomes a whole lot more graphic, because you can present users with a blank workflow as a template and provide a toolbox that contains custom activities that are

appropriate for your application. They can then author their workflows and add in your activities or custom activities they have written themselves.

MOVING FROM WF 3.X TO WF 4

With .NET 4 and Visual Studio .NET 2010 there is a new version of WF, and this new version is not backwards compatible with 3.x. The concepts are much the same; however the implementation is completely different. One upside of this is that you can continue to run 3.x workflows without any modification; however, if you want to take advantage of the new features then you will have to migrate your applications. In this section I cover the main points of that migration process, and provide some suggestions for simplifying the process.



For more detail on WF 4, see Chapter 44, “Windows Workflow Foundation 4.”

Extract Activity Code into Services

The class hierarchy for WF 4 is significantly different from that of WF 3.x; however, the activity class is much the same, so to simplify an upgrade I would recommend removing inline code from an activity and instead moving that into a set of services (such as simple .NET classes that contain static methods).

The nature of data bindings has changed considerably in WF 4, so whereas in WF 3.x you might use dependency properties and/or standard .NET properties, in WF 4 you'll need to use argument objects. The simplest way to migrate your code is to move all processing from within the activity and out into a separate class. As a trivial example, consider the snippet for a WriteLine activity:

```
public class WriteLineActivity : Activity
{
    public string Message { get; set; }

    public override ActivityExecutionStatus Execute
    ( ActivityExecutionContext context )
    {
        Console.WriteLine ( Message ) ;
        return ActivityExecutionStatus.Closed;
    }
}
```

If you wished to convert this to a WF 4 activity, you might wish to create a class as follows:

```
public static class WriteLineService
{
    public static void WriteLine ( string message )
    {
        Console.WriteLine ( message ) ;
    }
}
```

You could then use this inside the current activity (and also unit test it in isolation), and when you upgrade to WF 4 there will be less code to write. While this is a trivial example, it shows how to simplify an upgrade.

Remove Code Activities

Workflow 4 does not include the same code-behind style of activity that existed in 3.x and, therefore, if you have any code in a code-behind file, you should use the previous suggestion to port this into a library in order to be able to use it in WF 4.

Run 3.x and 4 Side by Side

If at all possible, strive to run 3.x workflows alongside 4 workflows until no more 3.x workflows exist. The persistence store and tracking store are different for 4, so a combined system is the easiest way to go. If you have no long-running workflows, then the issue is moot; however, where you do have long-running workflows, it's easiest to leave the old workflows unchanged and ensure that any new workflows created are on 4.

When your application starts a workflow, is it possible to replace this code easily with something that will start a WF 4 workflow? If not, I would recommend adding in a version-aware execution policy into your code, so that you don't have to alter the class calling the workflow in order to schedule a new workflow instance.

Consider Moving State Machines to Flowcharts

Workflow 4 currently does not support state machines — or rather there is no built-in support for state machines as yet. The underlying activity model can handle state machines (and indeed any workflow processing model you could think up); however, in the initial WF 4 release, there is no state machine activity or any design support.

The closest match to the 3.x state machine is the 4 flowchart. It's not an exact match by any means, but it does permit a workflow to jump back to an earlier phase of processing, which is what state machine workflows typically do.

SUMMARY

Windows Workflow will produce a radical change in the way that applications are constructed. You can now surface complex parts of an application as activities, and permit users to alter the processing of the system simply by dragging and dropping activities into a workflow.

There is almost no application that you could not apply workflow to — from the simplest command-line tool to the most complex system containing many hundreds of modules. Although the new communication capabilities of WCF and the new UI capabilities of Windows Presentation Foundation are a great step forward for applications in general, the advent of Windows Workflow will produce a seismic change in the way that applications are developed and configured.

Workflow 3.x is largely superseded by WF 4 in Visual Studio 2010, and this chapter provides some guidance on simplifying the upgrade. If you are planning on using workflow for the first time I recommend starting with version 4 and bypassing the 3.x version entirely.