

■ 15 ■

Writing Code in InfoPath

Getting Started

Welcome to the first chapter in Part II! The advanced chapters are the next steps for form designers who are comfortable with creating complex InfoPath forms without code. As we venture into the facets of programming the InfoPath platform, you'll learn how to create even more dynamic and sophisticated form templates than the design-mode user interface allows. But to really take advantage of the added value these advanced topics offer, we highly recommend that you know the material presented in the first part of this book.

In this chapter, we'll start by showing you how to add code to a new or existing InfoPath form template. InfoPath supports two classes of code: script and managed code. In this chapter, we'll use **Visual Studio 2005 Tools for Applications** (VSTA) and **Microsoft Script Editor** (MSE), which are the default programming environments for writing form code. Given the prevalence and ease of use of the .NET Framework, we'll emphasize managed code (specifically, C#) over scripting. We'll also cover several options that configure the code-authoring environment.

Then we'll introduce the **InfoPath object model** (OM) and tell you how using it can enhance your form templates. Once you understand the high-level objectives of using the OM and its event-based model, we will then look at how to start using it in a form template. Our discussion will show the various ways to create event handlers through the InfoPath design

mode. We'll also look at the `EventManager` object and how it hooks up the different types of InfoPath events.

Three groups of events belong to the InfoPath platform: form, control, and XML events. **Form events** include general form actions that aren't specific to any controls or data. Signing, saving, and submitting are all examples of form events. **Control events** allow your code to run when, for example, a user clicks a `Button` control when filling out the form. The last type of event is based on the XML data behind the form: **XML data events** (or just **XML events** or **data events**). There are three distinct states of XML data when form data is changed; each state has its own event to add custom code. We'll learn about these states and explain when you would want to use them.

After going over XML events, we'll give some helpful advice on working with the `XPathNavigator` class. An `XPathNavigator`, part of the .NET Framework, uses a cursor-based approach to access and modify the data source. If you are accustomed to working with the `XmlDocument` and `XmlNode` classes from .NET or Microsoft XML Core Services (MSXML), or if you're just getting started with programming XML, this section is for you.

We'll also look at **writing script** (specifically **JScript**), instead of managed code, with InfoPath. There are certain scenarios when using script is advantageous, for example, when programming a custom task pane. We take a look at the pros and cons of using script later in the chapter.

Next on the docket is a sampling of OM methods and properties commonly found in complex, real-world form templates. To show the OM in action, we'll dedicate the last third of this chapter to designing and filling out a new MOI Consulting sample form. In parallel, we'll learn some tips and tricks for working with InfoPath form code.

Finally, if you're migrating from InfoPath 2003 and are accustomed to working with its OM, there is a short learning curve to jump onto the InfoPath 2007 bandwagon. To help in this transition, we've sprinkled notes throughout this chapter when an InfoPath 2003 OM method or property has been renamed or removed or exhibits a different behavior in InfoPath 2007.

Writing Code Behind a Form

Adding code to a new or existing InfoPath form might be a big decision in the design process. But the steps to start writing code behind a form template are pretty easy. The only up-front decision concerns what programming

TABLE 15.1: Programming Languages and Available InfoPath Object Model Versions

Language	InfoPath OM Version
JScript	2003 SP1
VBScript	2003 SP1
C#	2007 managed
C# (InfoPath 2003)	2003 SP1 managed
Visual Basic	2007 managed
Visual Basic (InfoPath 2003)	2003 SP1 managed

language you prefer. Table 15.1 shows the available programming languages. Scripting languages, including JScript and VBScript, use the 2003 OM, while managed code can use the updated InfoPath 2007 OM. Since this book is focused on InfoPath 2007, we'll concentrate on the new 2007 managed OM.

Forms Services

Browser-enabled form templates support only the 2007 managed OM in C# or Visual Basic.

Terminology

Talking about code “behind” a form template isn't always very clear. That's because there are many different ways to say the same thing; that is, each way is interchangeable with another. The following terms, unless otherwise noted in the book, refer to the code included in a form template that runs when the form is filled out:

- Form code
- Form template code
- Business logic

Settings Related to Adding Code

Before adding code, let's choose the language in which we'll do our programming. The default language is Visual Basic. But if you want to choose another language, go to the *Form Options* item on the *Tools* menu and select the *Programming* category. The *Programming language* section, at the bottom of the dialog shown in Figure 15.1, lists various form template programming options.

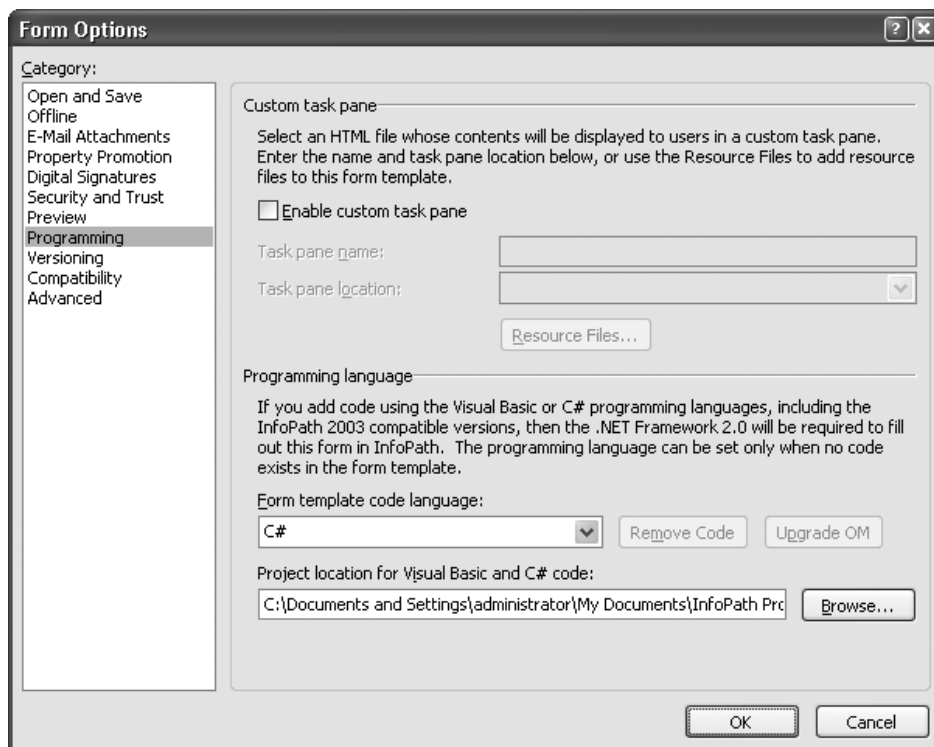


FIGURE 15.1: Programming category in the Form Options dialog

Options available in the *Programming language* section includes the *Form template code language* drop-down, *Remove Code* and *Upgrade OM* buttons, and a path for C# and Visual Basic .NET projects. When code already exists in the form, the *Form template code language* drop-down is unavailable, but the *Remove Code* button is enabled. Clicking *Remove Code* opens the dialog in Figure 15.2 so you can confirm removal.

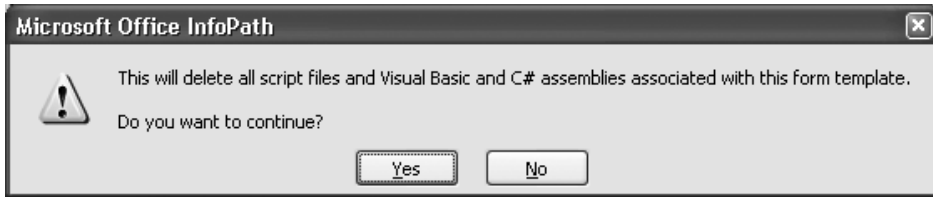


FIGURE 15.2: Confirmation dialog shown before removing code

■ **WARNING** Removing Code Is a One-Way Operation

You cannot undo the action of removing form code.

■ **TIP** Removing Managed Code

If the form template has managed code, only the assembly (.dll) and debugger (.pdb) files are removed. The Visual Studio project itself with the source code (which is not included in the form template .xsn file) is not deleted.

The *Upgrade OM* button is available when a form template with managed code was created with a version of InfoPath older than 2007. Upgrading your managed form code from an older version of InfoPath is highly recommended so you can take advantage of the much improved OM (which is also the programming focus of this book!). Since the InfoPath 2007 scripting OM is the same as older versions of InfoPath, a pre-2007 form template with script doesn't need upgrading. Clicking the *Upgrade OM* button for a template with managed code opens the dialog shown in Figure 15.3.



FIGURE 15.3: Upgrading code from a form template compatible with InfoPath 2003

734 ■ **Chapter 15: Writing Code in InfoPath**

When upgrading the OM, InfoPath will ask you to save your form template (to preserve your old template with the InfoPath 2003 form code) and then upgrade your code. All project references will be automatically updated to use the InfoPath 2007 object model. If you look at the form code after an upgrade, you'll notice most of it will be in gray text, instead of black. This is because the upgrade process essentially comments out your code by using `#if/#endif` compiler directive statements. First, the entire file's contents (within the namespace block) is essentially duplicated but enclosed in one large `#if/#endif` at the end of the file. (InfoPath uses the `InfoPathManagedObjectModel` symbol for the `#if` conditional directive.) Next, you'll notice that a new method called `InternalStartup` replaces the old `_Startup` and `_Shutdown` methods (there is no equivalent `_Shutdown` method in the 2007 OM). The `InternalStartup` method is automatically populated with hookups for all of the event handlers you had in your old form template. You'll also see that all of the event handlers still exist but are defined with the updated event definitions (e.g., `field3_OnAfterChange` becomes `field3_Changed`—we'll learn more about this later in the chapter) and parameters types. The code you had in each of the original event handlers is copied into the new event handler but is enclosed in `#if/#endif` compiler directives. Listing 15.1 shows the InfoPath 2003 code; Listing 15.2 shows the same code after being upgraded to the version 2007 OM.

LISTING 15.1: InfoPath 2003 C# Form Code Before Upgrading the OM Version

```
// The following function handler is created by Microsoft Office
// InfoPath. Do not modify the type or number of arguments.
[InfoPathEventHandler(MatchPath="/my:myFields/my:field1",
EventType=InfoPathEventType.OnValidate)]
public void field1_OnValidate(DataDOMEvent e)
{
    // Write your code here.
    IXMLDOMNode field1 =
thisXDocument.DOM.selectSingleNode("/my:myFields/my:field1");
    thisXDocument.UI.Alert("My foo is " + field1.text);
}
```

LISTING 15.2: Form Code from Listing 15.1 After Upgrading to the InfoPath 2007 OM Version

```
public void field1_Validating(object sender, XmlValidatingEventArgs e)
{
    #if InfoPathManagedObjectModel
        // Write your code here.
        IXMLDOMNode field1 =
thisXDocument.DOM.selectSingleNode("/my:myFields/my:field1");
        thisXDocument.UI.Alert("My foo is " + field1.text);
    #endif
}
```

NOTE Legacy: Hooking Up Events in Form Code

InfoPath 2003 relied on a C# attribute (specifically, the `InfoPathEventHandler` attribute) to decorate a method as an event handler. This syntax was changed in InfoPath 2007 to conform to a more recognizable .NET-standard style.

NOTE Script Code Does Not Create a Project

Adding script code does not create a project. Instead, InfoPath adds a JavaScript (.js) or VBScript (.vbs) file as a resource to the form template.

The final option shown in Figure 15.1 is the *Project location for Visual Basic and C# code* text box. It defaults to the InfoPath Projects folder under the current user's My Documents folder, but any path can be used. The *Project location* text box cannot be edited if a script language is chosen or if managed code is already being used in the form template.

TIP Saving a Form Template with Managed Code

If a form template already has an associated project with managed code, saving the template as a new name (not publishing) will copy the associated project folder to a folder with the same name as the newly saved form template. As a result, saving a template with a new name essentially checkpoints your entire project.

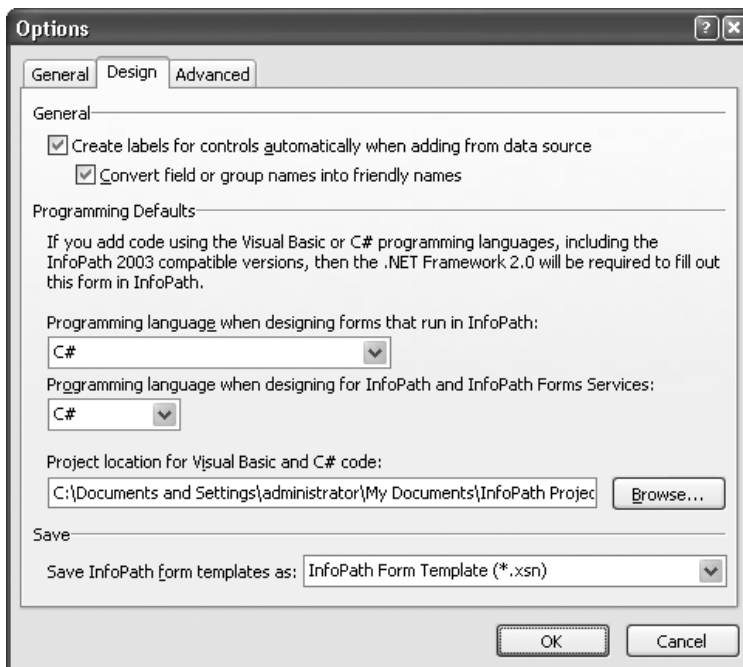


FIGURE 15.4: Programming options, which are saved as defaults for all new form templates

All of the settings shown in Figure 15.1 apply only to the current form template. If, for example, you keep changing from Visual Basic to C# as the language for your form templates, the *Options* dialog shown in Figure 15.4 offers relief. The *Programming Defaults* section contains some settings similar to those shown in Figure 15.1, but unlike those, these settings (in the *Programming Defaults* section) persist as defaults for all new form templates.

Forms Services

The *Programming language when designing for InfoPath and InfoPath Forms Services* drop-down applies when you add code to a browser-enabled form. Since Forms Services supports only the InfoPath 2007 managed code OM, *C#* and *Visual Basic* are the only available options.

Adding Code to a Form Template

To add managed code to your new or existing form, simply choose a form or data event that you want to **sink**. (Sinking an event means an event handler is created in your code that will now be invoked to handle a specific event.) You can find a list of prominent form and data events, such as *Loading Event* and *Changed Event*, on the *Programming* fly-out menu from the *Tools* menu. Alternatively, the *Programming* fly-out menu also has an entry called *Microsoft Visual Studio Tools for Applications* (or *Microsoft Script Editor* if you're using script) that will open the code editor without creating an event handler. Figure 15.5 shows the VSTA environment after we selected *Loading Event* from the *Programming* submenu.

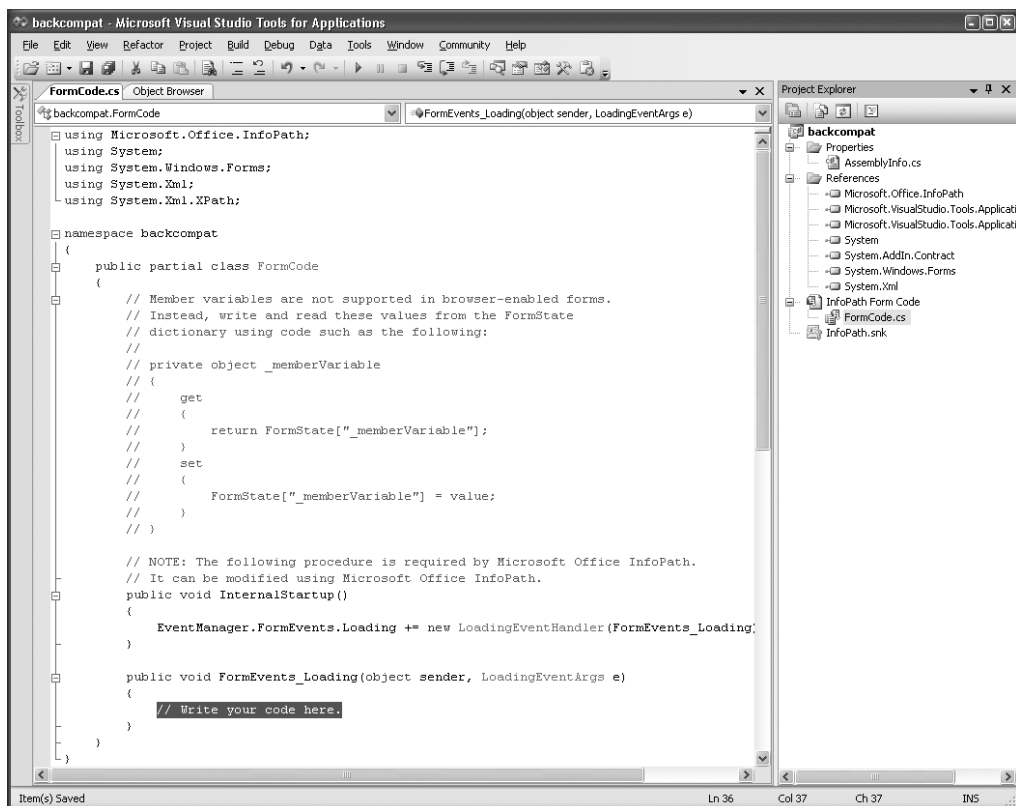


FIGURE 15.5: VSTA development environment

NOTE Save Before Adding Code

If you're adding code to a new form template, the template must first be saved. Attempting to add code to an unsaved template will result in InfoPath prompting you to save before continuing. Remember that saving is not publishing (see Chapter 9), so this is not the published form name your users will see.

No IntelliSense Information in the Object Browser or ToolTips

Selecting an InfoPath object model class, method, or property in the object browser or hovering over an object in code may not show IntelliSense information. This is because the files with information for IntelliSense are not in the same directory as the Microsoft.Office.InfoPath.dll assembly.

To fix this problem, you need to copy two files. Start by navigating to the %programfiles%\Microsoft Office\Office12 directory in Windows Explorer. Next, copy the Microsoft.Office.InfoPath.xml file in the Office12 directory to the following path: %systemdrive%\WINDOWS\assembly\GAC_MSIL\Microsoft.Office.InfoPath\12.0.0.0_71e9bce111e9429c. From the same Office12 directory, find the Microsoft.Office.Interop.InfoPath.SemiTrust.xml file, but this time copy it to %systemdrive%\WINDOWS\assembly\GAC\Microsoft.Office.Interop.InfoPath.SemiTrust\11.0.0.0_71e9bce111e9429c.

Filling Out and Debugging a Form with Code

While there's nothing particularly special about filling out a form that has code, it's helpful to become familiar with the basics of the VSTA environment and how it integrates with InfoPath design mode. Considering that we now have two development environments, InfoPath for designing the form and VSTA for writing the form's code, it makes sense to minimize the number of times we switch from one application to another.

Before we fill out our blank form that simply sinks the `Loading` event, let's add a single line of code that runs when our form is loaded. Since

we've already asked InfoPath to create the method that sinks the `Loading` event (called `FormEvents_Loading` in Figure 15.5), all we need to do is put the code in the area with the comment `// Write your code here`. One of the simplest operations we can do is to show a dialog box message. Let's add this single line of form code in the `FormEvents_Loading` method:

```
MessageBox.Show("Hello, InfoPath!");
```

NOTE Legacy: `XDocument.UI.Alert`

The InfoPath 2003 method `XDocument.UI.Alert` no longer exists in the InfoPath OM. The .NET Framework method `MessageBox.Show` (in the `System.Windows.Forms` namespace) is its replacement.

Now that we have some code that runs when the form loads, let's look at ways we can start filling out the form. (A sample named `LoadingMessageBox` is included with the samples for this chapter on the book's Web site.) Since we just added code in VSTA, it would be nice to preview the form without having to go back to InfoPath. This is possible if we select the *Start Debugging* item from the *Debug* menu (the shortcut key is F5). VSTA then performs the following actions:

1. Compiles the form code and builds the assembly
2. Opens the form in a preview window
3. Attaches the debugger to the preview process to facilitate debugging

Forms Services

Showing dialogs, such as message boxes, in forms running in the browser is not supported. In fact, the `System.Windows.Forms` namespace is not automatically referenced in the form code.

NOTE What Is an Assembly?

An **assembly** is .NET code compiled and built as Microsoft Instruction Language (MSIL) in an application extension (.dll) file.

■ WARNING Understanding Failed Compilation

The first two steps could potentially fail for a few reasons. If compiling fails, it's likely that you have made a syntax error in your code. Should the form fail to preview, it's possible that the form template itself has problems (such as a rule or data connection that fails when the form opens) or that some code you wrote (e.g., in the `Loading` event handler) encountered an error when executing.

If you wanted only to compile your code and build the assembly, you can use the *Build* menu or hit the Ctrl+Shift+B keyboard shortcut. (You may recognize this shortcut. It's coincidentally the same key combination used by InfoPath for the *Preview Form* command.) Building the form code updates the assembly that InfoPath uses when running the form. So the next time you preview the form, it will use the last successfully built assembly.

Debugging Form Code

As you would expect from a full-fledged development environment, modern conveniences of debugging are at your disposal. Features such as breakpoints, breaking on specific handled and unhandled exceptions, and variable watches are just a few of the assorted debugging options in VSTA.

To see how debugging works with code running behind an InfoPath form, let's set a breakpoint on `MessageBox.Show`. A breakpoint is a debugging mechanism used to pause execution of your code. One way to enable a breakpoint is to put your cursor on the line of code you want to pause at and hit F9. After setting your breakpoint, hit F5 to start debugging the form in a preview window. After the window appears but before the form is rendered, focus jumps to the VSTA window and highlights the line with your breakpoint. The form is running but halted until you allow execution to continue by hitting F5 or by stepping over code.

Forms Services

Debugging a form running in the browser requires debugging on the server itself (or remotely debugging to it) and attaching to the `w3wp.exe` process. This is conceptually similar to debugging an ASP.NET Web application. Since there is no concept of previewing a browser form, you must debug while filling out a form.

Depending on your computer's speed and the complexity of your form code, previewing while debugging with VSTA might be a little slow sometimes. And we don't blame you if you want to speed things up! However, it's not really possible to preview the form from VSTA without debugging enabled. If you want to run the form without debugging, preview the form from InfoPath instead.

TIP Building Is Automatic

There's no need to build the project in VSTA when previewing from InfoPath. InfoPath always asks VSTA to build the project, even if the VSTA environment isn't open!

Debugging code behind an InfoPath form is the same as debugging other types of programs, such as a Windows application or an ASP.NET Web site. With that said, we won't cover the details of debugging strategy in this book. MSDN has various articles on debugging. One such article referenced in the Appendix discusses debugging in Visual Studio, while another talks about debugging script code.

NOTE No Edit and Continue

Form code cannot be modified in VSTA while you're actively debugging a form.

The InfoPath Object Model

From a high level, a form template is constructed in design mode by building the contents of its views and data sources. Besides static aspects of the form (such as the color theme and number of views), many form and data-specific features are dynamic. These dynamic features reveal themselves as the user fills out the form. Examples of form features include those that allow the user to switch a view or submit the form. Invoking these features requires the user to initiate the action. Form features like changing views or submitting, of course, shouldn't affect the form's data. On the other hand, data-specific form features work directly from or on the XML data.

742 ■ Chapter 15: Writing Code in InfoPath

Some examples of data features include conditional formatting and data validation, both of which are activated depending on various data in the form. We're making the clear distinction between form and data features because this mirrors the InfoPath programming dichotomy.

Forms Services

There are actually two different object models available for writing code behind a form template. This chapter concentrates primarily on the Microsoft.Office.InfoPath.dll assembly for InfoPath forms. When you're designing a browser-enabled form template, an assembly of the same name but in a different location is used. This alternate assembly (in the InfoPathOM folder in OFFICE12) defines the browser-enabled OM to be used with both InfoPath and Forms Services. It restricts form code to a subset of the full OM normally available to the InfoPath program.

We're brushing over the concepts of form and data features because they are two main classifications of features that define the InfoPath event model. Having an event model for a programming platform means that you write your code within event handlers. An **event handler** is simply a method with a specific signature that is registered for a particular purpose with InfoPath. We'll learn about registering methods as event handlers when we look at the `EventManager` object.

Form Events

When we called `MessageBox.Show` in the earlier sample form, we wrote our code in the method sinking the Loading event. InfoPath fires this event every time our form template is loaded. We can also sink other form events. Table 15.2 presents a full list of form events, including where they are created from design mode and when they fire when a user fills out the form.

■ TIP InfoPath Programming Paradigm

InfoPath offers a pure event-based programming platform where your code runs only when something specific happens in the form.

TABLE 15.2: Form Events Exposed by InfoPath

Form Event	UI Entry Point	When Is It Fired?
Context Changed	<i>Tools Programming</i>	When control focus changes the XML context
Loading	<i>Tools Programming</i>	Every time the form is opened
Merge	<i>Tools Form Options Advanced</i>	When forms are merged
Save	<i>Tools Form Options Open and Save</i>	When the user saves the form
Sign	<i>Tools Programming</i>	When the form is signed (entirely or partially)
Submit	<i>Tools Submit Options</i>	When the main submit connection is invoked
Version Upgrade	<i>Tools Form Options Programming</i>	When an XML form is being opened whose version is earlier than that of the form template
View Switched	<i>Tools Programming</i>	After switching to a different view

NOTE Test Form Events for Yourself

A sample form called FormEvents (included with the other samples for this chapter) sinks all form events.

Forms Services

The Context Changed, Merge, Save, and Sign events are not available in browser-enabled form templates.

Instead of providing details here about all the form events listed in Table 15.2, we'll use them in samples throughout this chapter. The events

themselves aren't interesting. For example, the `Save` event is called when the form is saved. What is more exciting than the events is what code you can write when handling each event. After talking about XML data events, we'll exhibit a sample form template from our friends at MOI Consulting that shows off much of the InfoPath OM. The sample form will tie together various form events listed in Table 15.2 with properties and methods throughout the object model.

XML Data Events

Some of the most useful events to sink in a form template are those from the XML data source. Given that InfoPath is a strongly data-driven platform, it would make sense that changes in the data can be tracked at a granular level of detail. We will see in a moment that sinking data events is much more involved than one might think.

How Data Changes

Before you learn about the various events fired for XML data, you need to know *how* data can actually get changed. If you don't know how data gets changed, events will be firing all over the place for your data source, and such event notifications won't really make sense. (For a preview of this seemingly crazy behavior, see the `XmlDataEvents` sample form template.) The classic case of changing data is a user simply typing something into a Text Box control. Don't forget about the binding mechanics behind controls and the data source. When data is changed in a Text Box, it is not the Text Box that is changing but rather the data source field in which the Text Box is bound. Thus, the fact that some data is changing means that some item was modified in the data source.

Let's not limit ourselves to thinking only about element fields bound to a Text Box. (It doesn't matter what controls are bound to a data source node.) Attribute fields exhibit behavior similar to that of elements. And how about groups in the data source? A group can't really change, but it can be removed or inserted. Fields can be removed and inserted, too.

What about some other ways fields and groups can be changed? Here's a list of scenarios that is by no means complete:

- Inserting a row in a Repeating Table
- Removing an Optional Section

- Creating a rule action to set a field's value
- Querying a data connection, such as a Web service
- Signing form data
- Merging forms
- Running form code (which can change the data source, too!)

Event Bubbling

Our last topic prior to learning about the actual XML events is to discuss the phenomenon of **event bubbling**. You may be familiar with bubbling of events if you have scripted Internet Explorer with DHTML. In case you're not, let us provide an analogy. Picture a construction crew with crew members on each floor of a 100-story building. Their boss is on the top floor, and crew members are responsible for reporting their status to him. If a crew member on the 85th floor needs to relay a message to the boss, she'll yell it to a crew member on the 86th floor. In turn, that crew member will pass the message to someone on the 87th floor, and so on, until the boss receives the message. If someone on the 1st floor relays a message, it will need to bubble its way up every floor until it reaches the top.

Let's look at the sample form shown in Figure 15.6. The data source shows three nodes: `myFields`, `group1`, and `field1`. (You could say `myFields` is the boss of the construction crew.) To show the concept of event bubbling, we designed our form by going through each node and setting up a `Changing` event handler. InfoPath automatically creates the event handler when you right-click an item in the data source and select *Changing Event* from the *Programming* fly-out menu.

When VSTA opens, the `// Write your code here` text will be highlighted. Simply overwrite that highlighted comment by typing the code snippet shown in Listing 15.3.

LISTING 15.3: Showing a Message Box in the Changing XML Event

```
MessageBox.Show("Changing: Site is " + e.Site.Name + "; Source is  
" + ((XPathNavigator)sender).Name);
```

Forms Services

The `Changing` event is not available in browser-enabled form templates.

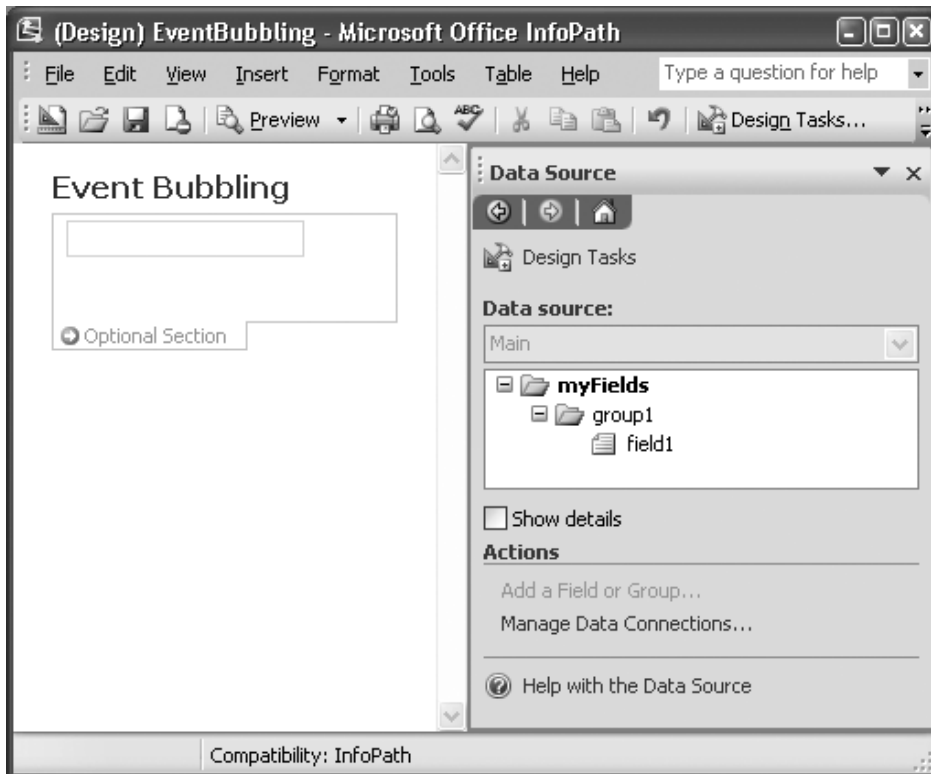


FIGURE 15.6: EventBubbling sample, which shows Site and Sender node names as events bubble up the data source

This line of code shows a dialog box message with the text “Changing:” and adds the node name of the site. The `Site` property on the `e` parameter (which is of type `XmlChangingEventArgs`; `Site` is defined in the OM on the `XmlEventArgs` parent object) yields the `XPathNavigator` of the node where the event is handled. If `field1` changes, the event handlers for `field1`, `group1`, and `myFields` will all be fired (in that order) with the `Site` being the node that’s currently bubbling the event. So the `Site` for `group1` will just be `group1`, and the `Site` values for other nodes will be those same nodes. Alternatively, the `Sender` object tells an event handler who originally caused the event; in our code, we label it as the `Source`. If `field1` is changing, the `Sender` for all event handlers is `field1` no matter which node’s event handler (`group1_Changing` or `myFields_Changing`) is handling the event.

What Is an `XPathNavigator`?

An `XPathNavigator` is a cursor-based reader designed for XPath-based queries to an underlying data source, such as XML. InfoPath 2007 exclusively uses `XPathNavigator` objects to programmatically traverse its data sources. The use of `XmlDocument` and `XmlNode` objects from InfoPath 2003 is essentially deprecated. We dedicate a latter part of this chapter to the `XPathNavigator` and show how to use it with the InfoPath data source.

■ **NOTE** Legacy: `Source`

The `Sender` object was exposed as `Source` on the `XmlEventArgs` object in the InfoPath 2003 OM.

■ **TIP** `Sender` as an Object Parameter

`Sender` is an object because the event handler is made generic to handle a notification from any object, not just an `XPathNavigator`. This new paradigm for `Sender` complies with .NET standards. As a result, we need to explicitly cast the object, which allows us to access `XPathNavigator` members such as the `Name` property.

As you play with the `EventBubbling` sample (included with this chapter's samples), some of the event notifications may surprise you. In particular, inserting or removing the `Optional Section (group1)` shows only one notification: "Changing: Site is my:myFields; Source is my:group1". This is because XML notifications are fired only for the parent of the changing node. You may ask, then, why the `field1` event handler fires when the `Text Box`, bound to `field1`, is changed. Isn't `group1` the parent of `field1`? Yes, but there's more to it.

To clear up the paradox, you must understand the way XML data is stored. A field (element or attribute) doesn't directly contain its own data. The data itself is considered its own separate node—specifically, a **text node**—contained within the field. It follows, then, that if the data (the text

748 ■ **Chapter 15: Writing Code in InfoPath**

node) in `field1` is changing, the `field1` node itself will fire its `Changing` event. This upholds our assertion that the parent of the changing node fires the notification.

Data Source Field Details

Data source fields consist of two separate XML nodes: the field node itself and a text node that holds the field's data. The World Wide Web Consortium (W3C) sets the standard on node types. You can learn more about all types of nodes, including text nodes, at the W3C Web site (as referenced in the Appendix).

Sometimes event bubbling is an unwanted side effect. What if, for example, we had most of our form within the Optional Section from our EventBubbling sample? Do we really want the `group1` event firing every time some data changes? Maybe we just want to know when `group1` is inserted. Unfortunately, there's no way to directly turn off the bubbling behavior, but there is a way to circumvent it when sinking XML events on data source groups. If you want to stop event bubbling when sinking a given group notification, use the code in Listing 15.4 at the top of the event handler.

LISTING 15.4: Code to Stop XML Events from Bubbling Above the Current Event Handler

```
XPathNavigator NavSender = (XPathNavigator)sender;  
bool moveSuccess = NavSender.MoveToParent();  
if (moveSuccess && !NavSender.IsSamePosition(e.Site))  
    return;
```

As you continue in the chapter and through the MOI Consulting sample form toward the latter half, you'll see other examples when we'll suppress the side effects of event bubbling.

NOTE No Nonprogrammatic Option to Cancel Bubbling

There's no built-in concept of canceling bubbling in InfoPath, such as the `cancelBubble` property serves in Internet Explorer.

Data States

Now that we've talked about the ways data can change and how event bubbling works, we're ready to tackle the three states of modified XML data. Every field or group in the main data source supports the general notions we'll discuss, so there are no special cases to call out. It's also consistent that whenever a data source node is modified, it proceeds through three states: *Changing*, *Validating*, and *Changed*.

However, just because these events fire when data changes doesn't mean that we need to concern ourselves with them. As the form template author, you can decide which events you want to sink depending on what you're trying to accomplish. Let's look at each event in turn and see when, why, and how you can use them in your form template code.

■ **NOTE** **Legacy: *OnBeforeChange*, *OnValidate*, and *OnAfterChange***

Changing, *Validating*, and *Changed* were *OnBeforeChange*, *OnValidate*, and *OnAfterChange*, respectively.

Changing Event

The first event in the sequence is probably the least-used XML event, yet it offers a powerful capability: the ability to reject and roll back the changing node. Rejecting a node change in the *Changing* event will show an error dialog to the user filling out the form and make the changed control appear as if it was never modified.

Forms Services

The *Changing* event isn't supported by Forms Services.

■ **NOTE** **Cannot Hide a Rejected Change**

Rejecting a node change in the *Changing* event always shows an error dialog. There is no way to suppress it.

750 Chapter 15: Writing Code in InfoPath

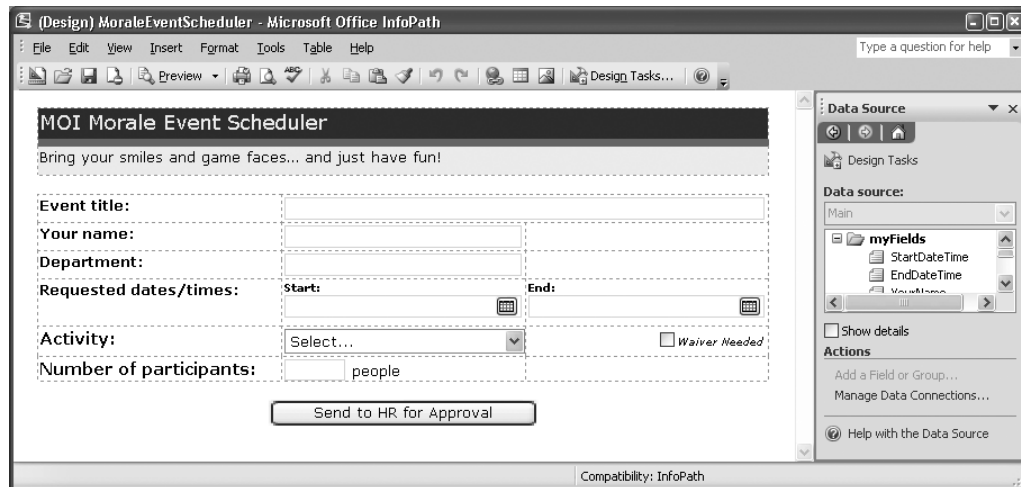


FIGURE 15.7: MOI Consulting morale event scheduler form in design mode

To exhibit the `Changing` event, we'll use the MOI Consulting morale event scheduler form. Figure 15.7 shows the form in design mode. (This sample file's name is `MOIMoraleEventScheduler-Changing`.)

To fill out the form, the user needs to make decisions about dates, times, and the specific activity for the morale event. Only authorized employees are allowed to fill out this form. We could just restrict access to the form altogether, but we don't want to manage the permissions at a file system level. Similarly, we don't want to just conditionally disable submit (e.g., through a rule) because someone could go through the trouble of filling out the form and then become frustrated that submit is blocked. Instead, we want to prompt for a password when someone tries to modify the form. If the password is wrong, the modification is rejected; otherwise, we grant access for the entire session. Thanks to the `Changing` event, we can use an event handler to prompt for a password and gracefully disallow changes in the form based on conditions that we include in our code.

To add a password prompt when someone tries to modify anything in the form, we can leverage event-bubbling behavior to intercept all form changes on the `myFields` document element. We begin by right-clicking on `myFields` in the data source and choosing *Changing Event* from the *Programming* submenu. Listing 15.5 shows the custom form code we use in the `myFields_Changing` event handler to prompt for a password.

LISTING 15.5: Prompting for a Password

```

public void myFields_Changing(object sender, XmlChangingEventArgs e)
{
    // Once a user is validated we won't prompt again
    if (UserIsAuthorized)
        return;

    string password = new MyPasswordDialog().Prompt();

    // This is poor security!
    if (password == "InfoPath!")
    {
        // Allow access
        e.CancelableArgs.Cancel = false;
        // Remember that this user has access
        UserIsAuthorized = true;
        MessageBox.Show("Thank you. Access has been granted for "
            + "this session.", "Access Granted");
    }
    else
    {
        e.CancelableArgs.Cancel = true;
        e.CancelableArgs.Message = "Authorized Users Only";
        e.CancelableArgs.MessageDetails = "Only authorized users "
            + " can schedule morale events.";
    }
}
}

```

Let's analyze this form code and figure out what it does in the `Changing` event for the document element. The first check, which we'll discuss shortly in more detail, determines whether the user has already been authorized by entering the correct password. If not, then we continue in the event handler by using the `MyPasswordDialog` class to prompt the user for a password. The code in this class uses .NET Forms library controls that aren't InfoPath-specific, so we won't discuss this code except to show what happens (Figure 15.8) when calling the `Prompt` method. You can find the code in the `MyPasswordDialog` class of the `MoiEventScheduler-Changing` form template (included with the samples for this chapter).



FIGURE 15.8: Password dialog prompt

■ WARNING Event Handlers in .NET versus InfoPath

The code in the `MyPasswordDialog` class has its own .NET form event handlers. Do not confuse these event handlers with those in InfoPath; they are different mechanisms (even though they look similar) and are not interchangeable. One such .NET form handler, for example, is registered through this code:

```
okButton.Click += new EventHandler(okButton_Click);
```

We discuss the details of registering for InfoPath events in the “Registering Event Handlers” section later in this chapter.

After the dialog prompt appears, the user can click either the *OK* or *Cancel* buttons to dismiss the dialog. If *Cancel* is clicked, an empty password is assigned to the `password` string variable. If *OK* is clicked, `password` gets whatever the user entered in the password text box. Once the password is received, we compare it against the hard-coded password “InfoPath!”. If the password matches, access is granted for editing the form. To do so, we set the `CancelableArgs.Cancel` Boolean (on `XmlChangingEventArgs` variable `e`) property to `false`. This is the default value for the `Cancel` property, but we set it explicitly for the sake of this sample. When the `Cancel` property is `false`, InfoPath does not reject the underlying node change (whatever the change might be).

■ WARNING Hard-Coded Passwords Are Not Recommended

In our sample, the password is hard coded as “InfoPath!”, which is not a recommended practice. This is not secure code and is used here solely for demonstration.

The next line of code (`UserIsAuthorized = true;`) remembers, for the duration of this form-filling session, that the user entered the right password. As a result, we don’t need to prompt the user for the password every time data in the form is changed, even though the `myFields Changing` event is fired. Checking whether the user is already authorized encompasses the first two lines of code in the `myFields Changing` event handler. Let’s look at the code for the `UserIsAuthorized` property (Listing 15.6) and see how it works.

LISTING 15.6: Implementing the `UserIsAuthorized` Property Using `FormState`

```
private const string UserIsAuthorizedStr = "UserIsAuthorized";

/// <summary>
/// Remembers if this user is authorized.
/// </summary>
private bool UserIsAuthorized
{
    get
    { return (bool)FormState[UserIsAuthorizedStr]; }
    set
    { FormState[UserIsAuthorizedStr] = value; }
}
```

The `UserIsAuthorized` property uses InfoPath's **FormState** object to remember a Boolean value that is `true` if the user is authorized and `false` otherwise. `FormState` is the preferred means to maintaining global state in your form code. When a user enters the correct password in the morale event form, we want to remember that the user is validated so he or she doesn't need to enter the password again.

`FormState` implements the `IDictionary` interface, meaning it holds key-value pairs of any type. In the `UserIsAuthorized` property, we're storing a string ("UserIsAuthorized") as the key name and a Boolean (`true` or `false`) as the value. However, we could potentially use anything as the key name despite unique strings or integers being the most common key types. The value can also be any object, but unlike the name, the value is typically a nonprimitive object such as a generic `List` (e.g., of type `string`), `DataConnection`, or any other types, which need not be InfoPath objects. To learn more about the `FormState` object as an `IDictionary` interface, see the related MSDN article referenced in the Appendix.

To get `FormState` working as expected, it must first be initialized with the name-value pair you want to use. The best place to initialize `FormState` is in the `Loading` event handler of the form. Recall that you can sink the `Loading` event by choosing *Loading Event* from the *Programming* fly-out menu of the *Tools* menu. The code snippet in Listing 15.7 shows the initialization of the `UserIsAuthorized` name (the `UserIsAuthorizedStr` variable is the constant string value of "UserIsAuthorized") with a default value of `false`.

754 ■ **Chapter 15: Writing Code in InfoPath****LISTING 15.7: Initializing FormState with UserIsAuthorized as false**

```
public void FormEvents_Loading(object sender, LoadingEventArgs e)
{
    FormState.Add(UserIsAuthorizedStr, false);
}
```

An efficient and organized approach to use the `FormState` object is to expose it by wrapping it as a property. This is exactly what we did with `UserIsAuthorized`. As a property implementation, `UserIsAuthorized` appears to be a simple class-wide variable when you use it throughout your code. Another advantage is that the dictionary aspects of the `FormState` object are abstracted away. As a result, we recommend using properties (with set and get accessors) only to manage any name-value pairs stored in `FormState`.

Forms Services

Why not just use class-wide variables to store state? Browser-enabled form templates do not allow the use of class-scoped variables. Similar to an ASP.NET page or Web service, Forms Services form code does not persist state between client HTTP requests. The `FormState` object is the only way for browser-enabled templates to persist data for the duration of a given session.

Let's continue to study the code behind the `myFields Changing` event. After verifying the correct password and setting `UserIsAuthorized` to `true`, the code calls `MessageBox.Show` as positive visual feedback to the user that he or she is authorized. What happens if, instead, the user enters the wrong password or clicks *Cancel*? The code that's executed includes the `else` block of the `myFields_Changing` event code shown earlier in Listing 15.5.

This code represents the essence of the `Changing` event. Of the three XML events, this is the only time when modification to node data can be rejected. If we do not cancel the `Changing` event, the change to the data will undoubtedly occur. Setting the `CancelableArgs.Cancel` property (from

the `XmlChangingEventArgs` variable `e`) to `true` will, after the `Changing` event handler has finished, cause InfoPath to reject the change of node data. Canceling the `Changing` event will kill all subsequent notifications for this node, so the `Validating` and `Changed` events will not fire. If the `Message` and/or `MessageDetails` properties are assigned, they are used in an error dialog that signifies the change was rejected. If at least the `Message` property is not set before the event handler returns, the default message is shown: "Invalid update: A custom constraint has been violated." As a courtesy to your users, you should consider providing a more informative message, such as we did for the `MoiEventScheduler-Changing` sample (Figure 15.9).

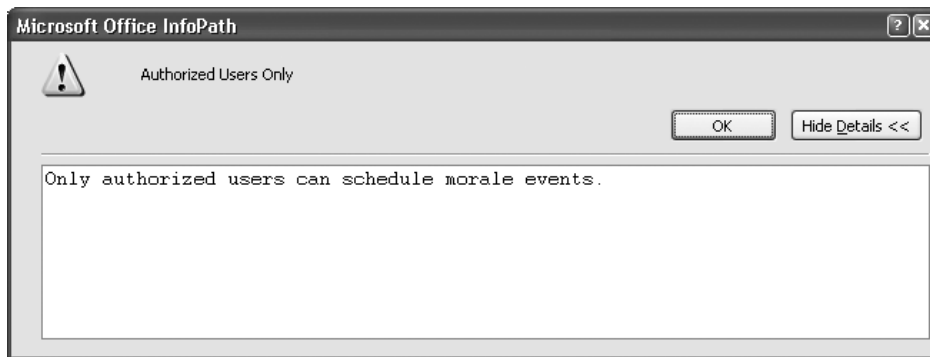


FIGURE 15.9: Custom error message that appears when a `Changing` event handler is canceled

NOTE Only changing Is Cancelable

The `Changing` event is the only XML event that can be canceled. Canceling the `Changing` event suppresses the data change from propagating to the `Validating` and `Changed` events. Many form events are also cancelable if the event's argument inherits from the `CancelEventArgs` .NET Framework class.

Once an authorized user enters the correct password and fills out the form, the last step is to click the *Send to HR for Approval* button, which sends e-mail to the morale contacts in the human resources department.

■ TIP Watch for Bugs That Circumvent Validation in Your Form

Clicking the *Send to HR for Approval* button will not fire the `myFields` event handler, nor will it fire any event handler that we have set up. As a result, the submission is allowed even though the user doesn't have access and the form is empty. To fix this bug with our form, we can conditionally disable the button while the *Event title* field is empty. (*Event title* is convenient to use, although any field or combination of fields could have been used instead.) Later in this chapter, we'll look at how to sink a Button click event so we could handle this in code if so desired.

You now have a good understanding of the `Changing` event. Before we move on to the next XML event, you should know about a couple of general restrictions regarding the `Changing` event. First, not all OM calls can be made from this event handler. For example, switching views programmatically (which is covered later in this chapter) is disallowed. If an invalid OM call is made in a `Changing` handler, InfoPath throws an exception, which automatically cancels the event. This exception manifests itself to the user as a visual error (Figure 15.10). A complete list of unsupported OM calls during XML events appears in Table 15.3.

Another limitation of the `Changing` event is that the entire main data source is read-only. Any changes to the data using an `XPathNavigator` (or any other means) will also throw an exception similar to the one shown in Figure 15.10. The reasoning behind a read-only main data source is that the

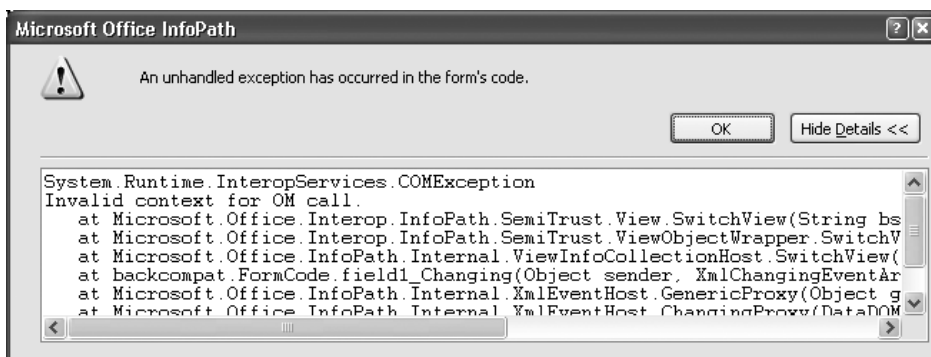


FIGURE 15.10: Error that occurs when calling unsupported OM during the `Changing` event

TABLE 15.3: Invalid Object Model Calls During XML Events

Event	Invalid OM	Comments
Changing	SwitchView Changing the MainDataSource object	MainDataSource is read-only.
Validating	SwitchView Changing the MainDataSource object	MainDataSource is read-only.
Changed	None	

Changing event is the first of three XML events that are fired for a node change. If the data source could be changed during this time, it would follow that two XML events may happen concurrently: Changing another node would kick off its XML events while this node still hasn't finished committing its data. As a result of disallowing changes at this time, InfoPath handles all XML operations in a synchronous, linear, and therefore predictable manner.

Validating Event

Should the Changing event not be canceled, next in the XML event order is Validating. As its name suggests, this is the time to perform validation on the new data. Once the Validating event handler is called to run your code, the node's new data has already been committed. This means that the underlying main data source has assumed the new node value. Consequently, there is no way to roll back the change at this point. The role of the Validating event is to evaluate the new data and **report an error**, if necessary. All of the same restrictions from the Changing event apply to the Validating event (main data source is read-only, some OM cannot be called, and so on).

■ **TIP** The Change Is Accepted in validating

Our assumption in the Validating event is that the change in data was allowed and supposed to occur. If the change in a node's data is to be rolled back, it should have happened in the Changing event.

758 ■ Chapter 15: Writing Code in InfoPath

Reporting an error does not actually affect the data. Instead, InfoPath maintains a list of nodes with data validation errors. These errors may surface visually on controls bound to the nodes with errors. Some controls, such as Section controls, do not support displaying error visuals. See Chapter 5 for more on data validation.

■ TIP Iterating Through Validation Errors

You can go through all validation errors, even if they aren't visible, by using the *Go to Next Error* and *Show Error Message* items on the *Tools* menu when filling out the form.

Forms Services

The *Go to Next Error* and *Show Error Message* features are also available on forms filled out in the browser. Hit Ctrl+Alt+R to go to the next error. Use Ctrl+Shift+I to show a message box with error details.

To put the `Validating` event into motion, we'll use the MOI Consulting morale event form that we used in the `Changing` event discussion. (This sample is called `MoiEventScheduler-Validating`.) For this sample, we want to check the dates and times of the proposed activity. We want to apply some validation rules when these values change. Here's what we want to ensure.

1. The requested start date/time is later than the end date/time.
2. The start time is between 8 A.M. and 4 P.M.; the end time is between 9 A.M. and 10 P.M.
3. An event may not last longer than one day if a waiver is necessary.

Let's begin by looking at the code that implements these validation rules by sinking the `StartDateTime Validating` event (Listing 15.8).

LISTING 15.8: Sinking the StartDateTime Validating Event

```

public void StartDateTime_Validating(
    object sender, XmlValidatingEventArgs e)
{
    // Get Start and End date-times
    XPathNavigator root = MainDataSource.CreateNavigator();
    XPathNavigator start = e.Site;
    XPathNavigator end = root.SelectSingleNode(
        "/my:myFields/my:EndDateTime", NamespaceManager);

    DateTime startDT, endDT;
    // Can we read the start?
    if (!DateTime.TryParse(start.Value, out startDT))
        return; // InfoPath handles invalid DateTime values

    const string error2a = "Start time is between 8 AM and 4 PM";
    if (startDT.Hour < 8 /*8am*/ || startDT.Hour > 12 + 4 /*4pm*/)
        e.ReportError(start, false /*siteIndependent*/, error2a);

    // Can we read the end?
    if (!DateTime.TryParse(end.Value, out endDT))
        return; // InfoPath handles invalid DateTime values

    CheckDateTimes(e, startDT, endDT);
}

```

The first line of code in the `StartDateTime_Validating` event handler uses the `MainDataSource` and calls `CreateNavigator`, which returns an `XPathNavigator`. (We discuss the `XPathNavigator` in detail later in this chapter.) The root `XPathNavigator` is used to get the `EndDateTime` node from the main data source. We conveniently use `e.Site` to get the `StartDateTime` but could have instead used `SelectSingleNode` as we did with the `EndDateTime` field.

TIP Check Whether Changes Are Allowed

You could call `MainDataSource.ReadOnly` to check whether the main data source currently allows changes.

NOTE Legacy: thisXDocument.DOM

`thisXDocument.DOM` is now `this.MainDataSource` (or just `MainDataSource` since this object is implicit). Many of the methods that were under `thisXDocument` are now inherited from the `Microsoft.Office.InfoPath.XmlFormHostItem` class and thus available as inherited class members.

760 ■ **Chapter 15: Writing Code in InfoPath**

To perform validation on the start date and time, we get the `Start` field value into a .NET `DateTime` structure. The `Validating` event is called *before* InfoPath performs schema data type validation, so when our code runs we can't assume that the user entered a valid date and time value. To verify the value, we use the `TryParse` method of `DateTime` to determine whether the `Start` field is a valid date and time. If it's invalid, we simply return from the method and let InfoPath do the work to report the appropriate data type validation error.

.NET DateTime Class

`DateTime` (see the Appendix for a reference to documentation on MSDN) offers a safe and convenient way to accurately and reliably parse date/time, date, or time values.

■ TIP Schema and .NET Data Type Correlation

All XML Schema data types have corresponding .NET data types. Moreover, most .NET data types (except string, of course) have a `TryParse` method. See MSDN for a table of XML Schema and corresponding .NET data types (as referenced in the Appendix).

After verifying that the start date and time value is valid, we check the first part of condition 2 in our list of validation requirements. If the hour is before 8 A.M. or after 4 P.M., we call `e.ReportError` to report a data validation error. The default version of `ReportError` takes three arguments: an `XPathNavigator` where the error occurs, whether or not the error is site independent, and an error message that's shown to the user.

The one parameter that's not straightforward is the Boolean `siteIndependent` flag. Its name is somewhat of a misnomer and, in our opinion, a little confusing. You should always pass `false` to the `siteIndependent` parameter unless the error node itself is repeating. If it's repeating, passing `false` means that the error is targeting a specific node. Passing `true` tells InfoPath that all repeating instances of a node are involved in the error. InfoPath uses this information to determine when to automatically delete

the error. When the error is site independent, a change in any of the nodes that repeat will remove the error; a site-dependent error, on the other hand, will not clear itself until that specific node is modified.

■ **NOTE** **ReportError Overloads**

The other versions of `ReportError` are similar to the `Errors.Add` method, which we'll discuss in the "Form Errors" subsection.

Since the `EndTime_Validating` event is almost identical to the `StartTime_Validating` form code in Listing 15.8, we'll omit it for brevity. You'll notice, however, that there's a call to a `CheckDateTimes` method. This code was added because both the `StartTime_Validating` and `EndTime_Validating` events call into it, thus eliminating the need for duplicate code. Listing 15.9 shows the form code for `CheckDateTimes`.

LISTING 15.9: Implementation of the `CheckDateTimes` Method

```
private void CheckDateTimes(
    XmlValidatingEventArgs e, DateTime startDT, DateTime endDT)
{
    const string error1 =
        "Requested Start date/time is later than the End date/time.";
    const string error3 =
        "An event may not last longer than 1 day if a waiver is "
        + "necessary.";
    if (startDT > endDT)
        e.ReportError(e.Site, false /*siteIndependent*/, error1);

    if (((TimeSpan)endDT.Subtract(startDT)).Days >= 1 && WaiverNeeded)
        e.ReportError(e.Site, false /*siteIndependent*/, error3);
}
```

Remember that if a waiver is needed, the event may not last longer than one day. (The `Days` member of the `TimeSpan` class returns the number of whole days.) We use a `WaiverNeeded` property to check the `WaiverNeeded` node. Listing 15.10 shows the code.

762 ■ Chapter 15: Writing Code in InfoPath

LISTING 15.10: The WaiverNeeded Property

```
private bool WaiverNeeded
{
    get
    {
        XPathNavigator root = MainDataSource.CreateNavigator();
        XPathNavigator waiver = root.SelectSingleNode(
            "/my:myFields/my:WaiverNeeded", NamespaceManager);
        return waiver.ValueAsBoolean;
    }
}

```

Let's not forget to discuss what happens if someone chooses a morale event to last more than one day and then checks the *Waiver Needed* Check Box control! Checking *Waiver Needed* fires XML events on the *WaiverNeeded* node and bubbles up to the root. So our *Validating* event handlers aren't run. The result is that the violation of an event lasting longer than one day with a required waiver is not caught. While there are a few remedies to this problem, we took the approach of sinking the *Validating* event of the *WaiverNeeded* node, as shown in Listing 15.11.

LISTING 15.11: Sinking the WaiverNeeded Validating Event to Handle Checking of the Waiver Needed Check Box

```
public void WaiverNeeded_Validating(
    object sender, XmlValidatingEventArgs e)
{
    // Get Start and End date-times
    XPathNavigator root = MainDataSource.CreateNavigator();
    XPathNavigator start = root.SelectSingleNode(
        "/my:myFields/my:StartDateTime", NamespaceManager);
    XPathNavigator end = root.SelectSingleNode(
        "/my:myFields/my:EndDateTime", NamespaceManager);

    DateTime startDT, endDT;
    // Can we parse the Start and End values?
    if (!DateTime.TryParse(end.Value, out endDT)
        || !DateTime.TryParse(start.Value, out startDT))
        return; // InfoPath handles invalid DateTime values

    // Only do start and end common checks if date-times are both valid
    if (DateTimesAreValid())
        CheckDateTimes(e, startDT, endDT);
}

```

The code for `DateTimesAreValid` checks the `Errors` collection to see if we've already reported errors (using `ReportError`). Listing 15.12 shows the implementation.

LISTING 15.12: Checking the `Errors` Collection for Reported Start or End Errors

```
private bool DateTimesAreValid()
{
    // Are the Start and End date-times valid?
    FormError[] errs = Errors.GetErrors(FormErrorType.SystemGenerated);
    foreach (FormError err in errs)
    {
        if (err.Site.Name == "my:StartDateTime"
            || err.Site.Name == "my:EndDateTime")
        {
            return false;
        }
    }
    return true;
}
```

■ **NOTE** Validation Always Happens on Form Load

We mentioned that the `Validating` event is fired after `Changing` but before `Changed`. While this is true, there is one exception. Only `Validating` happens (without `Changing` or `Changed`) when the user creates a new or opens an existing form. (See the `FormCodeAndRules` sample for a demonstration.) If you report an error when the form is first validating, the form will still open with the validation error as expected. In previous versions of InfoPath, the form fails to load if a `Validating` event was canceled or threw an unhandled exception in its form code.

InfoPath allows for a maximum of one error per node at any given time, so it's not possible to report multiple errors on the same node simultaneously. If we report two errors on, say, the `StartDateTime` node, the first would be overwritten. As you can see from the `Validating` event and the `CheckDateTimes` shared method, our form code has the potential to report more than one error on a given node. A case where multiple errors can be reported on `Start`, for example, is when the start time is not between 8 A.M. and 4 P.M. and the start date is later than the end date. Since errors are overwritten, only the last error will be reported and seen by the user. No ill effects

occur by reporting multiple errors; after the user corrects the error, the other (formally overwritten error) will be exposed. (This obviously assumes that conditions for reporting each error can be satisfied independently.)

■ **TIP** Accelerate Your Learning When Programming InfoPath

If you want to experiment with how this `Validating` code behaves, you can use the debugger to step through the code and use the *Immediate* debug window to run any code on-the-fly.

By the way, did you notice any bugs in our form code? If you played a little with the sample form, you might have noticed some special cases. One bug creeps in if the start time is after the end time and the valid field is modified in order to trigger the XML events for that node. Here is one way to reproduce the problem.

- Set `Start` to January 1, 2002.
- Set `End` to January 1, 2001.
- (Touch the `Start` field.) Set `Start` to January 2, 2002.

Now both `Start` and `End` have validation errors for `Start` being later than `End`. Fixing either `Start` or `End` to be valid will not remove the other error because that field was not edited. `ReportError` will remove the reported error only when the node that has the error is modified. How can we fix this case of a duplicate error? Since the main data source is read-only during the `Validating` event, it's not possible to touch the other node to force the XML events to be called. Could we go through the `Errors` collection and use the `Errors.Delete` method to remove the troublesome form error? Unfortunately, this isn't possible either. To explain, we need to talk more about InfoPath form errors as exposed via the OM.

■ **NOTE** “Duplicate Error” Bug

There's a way to fix our form code bug to prevent the “duplicate error” predicament. We'll leave it as an exercise for you to implement the fix. (Here's a hint: Allow only one, but not both, of the `DateTime` fields—either `Start` or `End`—to have a common error at any given time. A common error is one that involves both `Start` and `End` and could be reported on either node.)

Form Errors

From an end-user's perspective, and even from the standpoint of the form designer, a validation error is, well, just a validation error! But when you're programming InfoPath, the details of validation are essential when working with form errors. To address our question about going through the `Errors` collection and using `Errors.Delete` to remove a form error, we need to first discuss the three types of form errors.

When a validation error occurs when a user fills out a form, the error is classified as one of the following types: `SchemaValidation`, `SystemGenerated`, or `UserDefined`. `SchemaValidation` errors are reported only by the XML parsing engine (MSXML for InfoPath, `System.Xml` on Forms Services). `SystemGenerated` errors are a result of calling `ReportError`. A `UserDefined` error is generated when form code manually adds a new `FormError` to the `Errors` collection. The three error types correspond with the values of the `FormErrorType` enumeration.

Listing 15.13 shows an example of adding a `UserDefined FormError`.

LISTING 15.13: Adding a UserDefined FormError to the Errors Collection

```
XPathNavigator NumPeopleNav = MainDataSource.CreateNavigator();
NumPeopleNav = NumPeopleNav.SelectSingleNode(
    "/my:myFields/my:NumberOfPeople", NamespaceManager);
Errors.Add(NumPeopleNav, "TooManyPeople",
    "No more than 10 people can attend.");
```

This code first gets the `XPathNavigator` to the main data source root. Next, we select the `NumberOfPeople` node. Finally, an error is added by `Errors.Add`. The first parameter is the `XPathNavigator` of the node in which the error is associated, followed by the internal (unexposed to the user) error name, and finally the short error message. Override versions of this method add the following parameters: `messageDetails`, `errorCode`, and `errorMode`. The `messageDetails` parameter provides additional information to the user if he or she right-clicks the error visualization and requests to see details. `errorCode` is an arbitrary integer for your internal and personal use. Finally, the `errorMode` parameter takes a value from the `ErrorMode` enumeration: `Modal` or `Modeless`. If `Modal` is used, an error is immediately shown in an alert-style dialog that blocks the user until it's dismissed. `Modeless`, on the other hand, does not show a dialog and is the default behavior.

766 ■ Chapter 15: Writing Code in InfoPath

Calling the `Errors.Add` method can be used in any event handler (including form events) without restrictions. This is unlike the `ReportError` method, which is only available (and thus must be called) within the Validating XML event handler. Why, then, would anyone use `ReportError` versus `Errors.Add`? This brings us to the topic of deleting errors.

There are two ways to manually remove errors from the `Errors` collection. The first way is with the `Delete` method. An error is deleted by passing in either a `FormError` (an item from the `Errors` collection) or an error name (the internal name used in the `Add` method). Only `UserDefined` errors are allowed to be deleted. So, reporting an error using `ReportError` cannot be done easily this way. However, using the `Errors.DeleteAll` method will obliterate all form errors, no matter what their type.

■ WARNING Don't Use ReportError with an Arbitrary Node

When calling `ReportError` on the `XmlValidatingEventArgs` object, be sure that the error is based on the node associated with the `e.Site.XPathNavigator`. If an error is reported on an arbitrary node with `e.ReportError`, InfoPath will not know when to remove the error, which results in it being “stuck” and your form in a bad state.

A useful mechanism for debugging form errors is to add a `Button` control to the view of your form that shows all errors when clicked. To set up the `ClickedEventHandler`, go to the *Button Properties* dialog and click on the *Edit Form Code* button. Listing 15.14 shows our `Button` event handler.

LISTING 15.14: Showing All Errors on a Button Clicked Event

```
public void CTRL11_5_Clicked(object sender, ClickedEventArgs e)
{
    MessageBox.Show("Number of errors: " + Errors.Count);
    string errors = string.Empty;
    foreach (FormError err in Errors)
    {
        errors += "(" + err.FormErrorType + ", "
            + err.Site.Name + ") " + err.Message
            + System.Environment.NewLine;
    }
    MessageBox.Show("Error messages: "
        + System.Environment.NewLine + errors);
}
```



FIGURE 15.11: Show Error Message dialog

Before we move on to the `Changed` event, we have a final word about reporting custom validation errors using the OM. When using `ReportError` or `Errors.Add`, ensure that the error node exists in a view or at least that the user can sufficiently recognize the error and be able to fix it. In Figure 15.11, an error has been added to `MainDataSource.CreateNavigator()` (the root) that the user can never see. It can be seen only by using `Go to Next Error` and then `Show Error Message`. The dialog shown in Figure 15.11 appears when using the `Show Error Message` option.

This is clearly a contrived example because there is no reason to report an error on the root, but the concept remains. The code in Listing 15.15 is written behind a button `Clicked` event to reproduce the behavior shown in Figure 15.11.

LISTING 15.15: Adding an Error That the User Can Never See

```
Errors.Add(this.CreateNavigator(),
           "Error", "This is a bad error because it can't be seen!");
```

Changed Event

Last in the series of XML events fired on a modified node, and all of its parents, is the `Changed` event. Its name is quite representative of the state involving the modified node data: The change has already occurred. So what's the point of sinking and handling this event in custom form code? That's a good question, and we have a great answer. You can do pretty much anything your heart desires! Unlike `Changing` and `Validating`, the

768 ■ **Chapter 15: Writing Code in InfoPath**

Changed event can modify the main data source. You can also ask InfoPath to switch to another view—something that was also disallowed during Changing and Validating. Regard the Changed event as your instrument to completely react to a change in form data in whatever way necessary. That's certainly powerful!

Like us, you might find yourself wanting to use form code for simple things in the Changed event. Here are some examples: showing a dialog box, setting a node's value, querying a data source, or even switching to another view. The Rules feature was designed to cater to these scenarios so you wouldn't have to use the Changed event. As a matter of fact, rules fire at nearly the same time as the Changed event. With rules in mind, let's take another look at the order of "events" that occur when a node is modified: Changing, Validating, rules, Changed. (See the FormCodeAndRules sample form template.) So rules could be a direct substitution for the Changed event in simple scenarios.

Our recommendation is to leverage the simplistic declarative nature of rules when possible because code is harder and more expensive to maintain. If rules won't fully suffice for your needs, you can tack on a Changed event. If it doesn't help to split the work between rules and the Changed event, you could try to put some logic in the Validating event to precede rules. Should that not work, for example, because you need to modify the value of a node (recall that the main data source during Validating is read-only) before rules, you may need to reimplement the rule actions into your form code. Even though the likelihood of reimplementing rules into code is low, most rules constitute no more than about two or three lines of code.

To demonstrate the Changed event, we'll add some final touches to the MOI morale event scheduler form. (See the MoiEventScheduler-Changed sample.) Say that we've received feedback from the managers that we need to gather names of participants for an accurate head count. The names should be sorted in alphabetical order by last name.

Let's figure out what we need to do in our form to fulfill the managers' requirements. First we need controls in the view to capture the names of the participants. Since we'll be sorting by last name, it's easiest to separate first and last names into their own Text Box controls. The Text Box controls will be within a Repeating Table control so that multiple participant instances can be inserted and filled out.

Number of participants:		2	people
Last Name	First Name		
Green	Hagen		
Roberts	Scott		

Insert item

Send to HR for Approval

FIGURE 15.12: Supporting entry of last and first names in a Repeating Table control

With the new controls in the view, our form now supports the entry of many first and last names. Figure 15.12 shows how that part of the view appears in design mode. Now that we have the controls bound to the data source, how do we sort by the `LastName` field? Before we approach the answer, let's investigate and understand the problem.

First, there are two ways to sort data in InfoPath: in the view or in the data. Sorting content in the view simply orders the visual presentation of the data; it does not actually touch the data in the XML. With XSLT driving the presentation layer of InfoPath, you could modify the view (.xsl file) to include an `xsl:sort` directive. (See the W3Schools Web site referenced in the Appendix for specifics.) Since this is a book about using InfoPath and working with XML data, we're taking the data approach to sorting. Sometimes sorting the data is a requirement because the form XML will be used in another application that already expects the data to be sorted.

NOTE View-Based Sorting

Sorting in the XSL is almost always faster in terms of absolute performance. Unfortunately, sorting in the view may not always yield the results you expect. For example, if the data you're sorting is bound to more than one control, you'll need to add sorting functionality to each of the controls. Moreover, view-based sorting is not supported in browser-enabled form templates because it requires manual hand-editing of the view XSL file. Sorting the form data will work similarly in client and browser-enabled form templates.

To approach sorting the Repeating Table data containing participants, we need to first determine what will trigger the sort. Given that we're sorting by last name, it seems appropriate to fire off the sort whenever the `LastName` field is changed. Despite having `LastName` trigger the sort, we

770 ■ **Chapter 15: Writing Code in InfoPath**

don't want to actually sort the `LastName` field. (If we did so, we'd move last names without the rest of the row data!) We want to sort the `Participant` repeating group under the `Participants` group. Consider the data source bound to the controls in Figure 15.12. We can see that the parent node of field `LastName` is a repeating group named `Participant`, and its parent node is a group named `Participants`. The code snippet listening to the `LastName` field is shown in Listing 15.16. To focus more on the scenario and less on the sorting algorithm, we'll use a simple bubble sort to order the `Participant` groups based on the `LastName` field. In a real-world scenario, you'd want to use the .NET Framework to efficiently sort the data.

NOTE Sinking Secondary Data Source Events

Secondary data sources allow sinking only the `Changed` event. Without any validation on secondary data sources, there are no equivalent concepts of the `Changing` or `Validating` events to reject or report validation errors, respectively.

LISTING 15.16: Sorting Items in the Repeating Table When the `LastName` Field Is Changed

```
public void LastName_Changed(object sender, XmlEventArgs e)
{
    if (e.UndoRedo)
        return;
    // resort the list
    if (e.Operation == XmlOperation.ValueChange)
    {
        XPathNavigator senderNav = sender as XPathNavigator;
        senderNav.MoveToParent(); // move to 'Participant'
        senderNav.MoveToParent(); // move to 'Participants'
        SortChildNodes(senderNav);
    }
}
```

WARNING Sorting Every Time `LastName_Changed` Is Called

It's very important to call `SortChildNodes` only when the value of `LastName` actually changes. Removing the `if` statement that checks for `ValueChange` will result in the `LastName_Changed` event firing again (within itself) when nodes are moved for the sorting operation.

The `Changed` event starts with two comments generated by InfoPath. These comments basically say that the main data source (i.e., OM property `MainDataSource`) should not be modified yet. That's because if the operation that caused the `LastName` field to change is an undo or redo, our code shouldn't handle it. InfoPath takes care of all undo and redo operations automatically. If you still wanted some code to run during an undo or redo, keep in mind that the main data source is read-only. Finally, after the check for `e.UndoRedo`, we can write our code!

Forms Services

Since there is no concept of undo or redo, the `UndoRedo` property is always `false` when a browser-enabled form template is running in the browser.

We begin by getting the `Participants` group. To get there, we could have just used `MainDataSource` to get the root and then `SelectSingleNode` with the XPath `/my:myFields/my:Participants`. Instead, we wanted to show how the `XPathNavigator` can be used as a cursor to traverse the data source. The `sender` object in all XML events will always be the `XPathNavigator` behind the specific event handler. So in this case we know that `sender` will be `LastName`. (Recall that `e.Site` is the `XPathNavigator` that initiated the event; if the event handler is sinking group XML events, `e.Site` can be any child node.) After we move to the `Participants` group, we pass the navigator to the `SortChildNodes` method, which does the actual bubble sort. Listing 15.17 shows the implementation of the sort.

The `SortChildNodes` method first needs to know the total number of participant items to sort. To get the number of `Participant` groups, we count the result of selecting all `Participant` children of the `Participants` group. The two `for` loops account for the bubble sort iterations. To compare the last names of participant `j` and participant `j+1`, we select the nodes by using XPath with a **predicate filter**. A predicate filter is signified by square brackets within an XPath expression. Passing a simple numerical value like `[1]` is a shortcut for `[position()=1]`. (You can find more on the XPath Language at the W3C site, as referenced in the Appendix.) `String.Compare` performs a culture-specific string comparison that returns `1` if `child1`'s value is lexicographically later than that of `child2`. If

772 ■ Chapter 15: Writing Code in InfoPath

LISTING 15.17: Sorting Participants (by LastName) in the Data Source

```

public void SortChildNodes(XPathNavigator parent)
{
    const string Prtpnt = "Participant";
    XPathNodeIterator childNodes =
        parent.SelectChildren(Prtpnt, parent.NamespaceURI);
    int numberChildren = childNodes.Count;
    for (int i = 1; i <= numberChildren; i++)
        for (int j = 1; j <= numberChildren - i; j++)
            {
                XPathNavigator child1, child2;
                child1 = parent.SelectSingleNode(
                    "my:" + Prtpnt + "[" + j + "]/my:LastName",
                    NamespaceManager);
                child2 = parent.SelectSingleNode(
                    "my:" + Prtpnt + "[" + (j+1) + "]/my:LastName",
                    NamespaceManager);
                if (string.Compare(child1.Value, child2.Value) > 0)
                    {
                        child1.MoveToParent();
                        child2.MoveToParent();
                        XPathNavigator temp = child1.Clone();
                        child1.ReplaceSelf(child2);
                        child2.ReplaceSelf(temp);
                    }
            }
}

```

that's the case, we want to swap the positions of the `child1` and `child2` parent nodes. (The parent nodes of `child1` and `child2` represent entire rows in the Repeating Table.) Since `XPathNavigator` objects are simply pointers to the underlying data source, we cannot assign new objects to them. Instead, navigators expose methods such as `Clone`, to make a deep copy, and `ReplaceSelf`, to swap out (and remove) the node for another. We use a combination of these methods to swap the positions of `child1` and `child2` Participant groups.

■ **NOTE** XPath Is 1-Based

Are you wondering why the bubble sort loops start with 1 instead of 0? Repeating items are addressed in XPath by using a 1-based index.

Since the `SortChildNodes` method is called only from the `LastName_Changed` event handler, it runs only when a `LastName` field is modified. What if the form template is opened with an already existing form whose underlying participants data is unsorted? Currently, it won't be sorted. However, by sinking the `Loading` form event, you could call `SortChildNodes` to ensure the data is sorted from the start. If a form is opened with a huge amount of data that's sorted in descending (Z to A) order (the worst case for the bubble sort algorithm), loading performance could be lethargic. Consider showing the user a "Please wait" message if this is a concern. However, we recommend using a more efficient algorithm such as merge or binary tree sorting, which take less time to run in their worst cases.

Forms Services

When posting data back to the server takes a long time (which includes running custom form code), Forms Services automatically shows the "Sending data to the server" busy dialog over the form in the browser.

■ TIP Concatenating Strings by Selecting Multiple Text Nodes

If you want to sort by last name and then by first name, you don't need to add any special code. A shortcut is to select the `Participant` itself instead of the `LastName` node. The `Value` property of a group node is the result of concatenating the `Value` of each child text node under that group. Try removing the appropriate code in the sample to see the values of `child1` and `child2`. Remember that text nodes exist *between* field nodes as white space to format the XML itself!

Multiple Event Notifications for XML Event Handlers

A single XML event handler can be fired multiple times during a data source change. In the past few examples, we didn't see this behavior because we've been using string data types. Some other data types, however, require the XML data to be decorated in particular ways. These so-called decorations require managing hidden XML on the changing element itself to accommodate its behaviors. For example, for an integer field to be blank, the hidden

attribute node called `xsi:nil` must be set to `true`. (This is an XML data type requirement to which InfoPath must adhere for an integer field's data to be considered valid.) If the blank integer field is set to some value (which may or may not be a numerical value), the `xsi:nil` attribute is removed. As a result, any XML events on this integer node are fired twice: the first time for removing `xsi:nil`, the second for changing the value of the field itself. The order of notifications in this case, with the sender in parenthesis, is `Changing (xsi:nil)`, `Validating (xsi:nil)`, `Changed (xsi:nil)`, `Changing (my:integer)`, `Validating (my:integer)`, and `Changed (my:integer)`.

TIP Integer and `xsi:nil`

To witness firsthand how `xsi:nil` is involved in field changes, you can save a form with a blank integer field, and then save it again as a different name when it has a value. Open the resulting `.xml` file in a text editor such as Notepad to see the `xsi:nil` decoration. You can use the `MultipleNotifications` sample (included with the samples for this chapter) for this purpose as well as to explore other concepts involving multiple notifications.

The `MultipleNotifications` sample form, shown in Figure 15.13, contains several Text Box controls bound to different data types. The `String` field is the standard case; it exists only for comparison. The `Integer`, `RichText`, and `DateTime` fields all show various forms of multiple notifications. `Integer` and `DateTime` are similar in their use of `xsi:nil`, while `RichText` is a little different. The `RichText` field fires not only regular change notifications but also HTML element insertions and deletions. For example, underlining in the Rich Text Box control will fire an event from an HTML node named `u`. (The `<u>` tag in HTML represents markup for underlined content.)

Multiple notifications may cause your code to run twice or more, depending on the data source change. Running code in your XML event handlers many times may not cause any problems. You might not even know it's running a few times! But in some cases, running code several times could cause serious problems. One of the most common issues is poor performance. In our `Changed` event sample, we used a sorting algorithm to

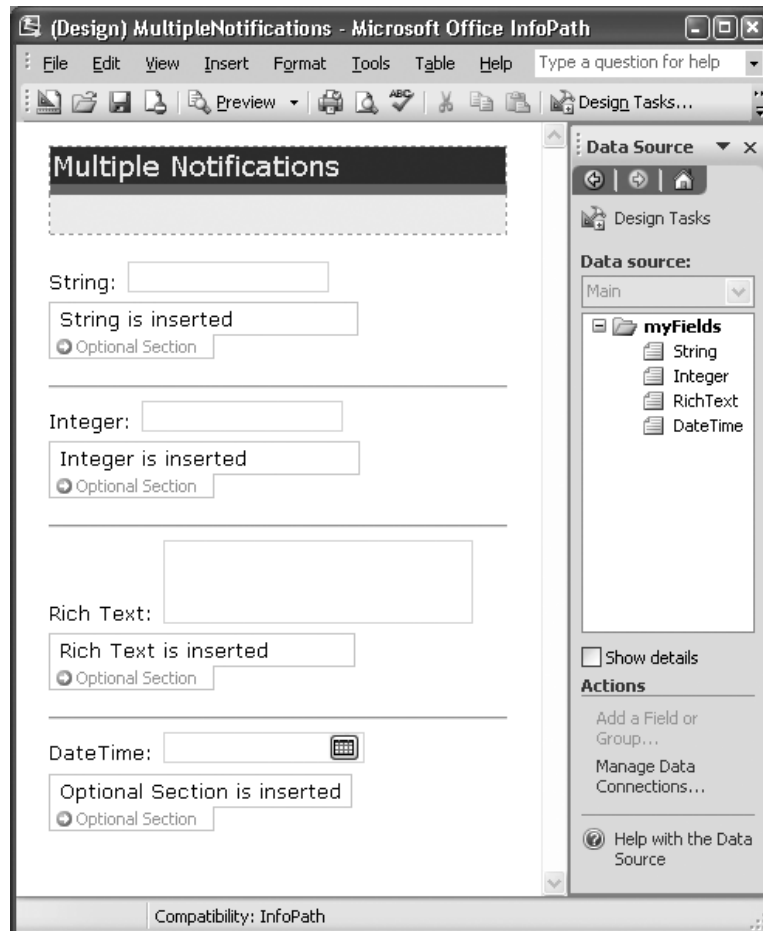


FIGURE 15.13: MultipleNotifications sample form in design mode

shuffle the main data source. If there were many items, the algorithm could take a long time to run. But without handling multiple notifications properly, the wait time can double or worse! Can we do anything to handle only the notifications we care about? Let's take another look at the MultipleNotifications sample to learn about OM that can help us make distinctions between different operations.

As you fill out the sample MultipleNotifications form, you'll notice that a message box appears after each change (Figure 15.14).

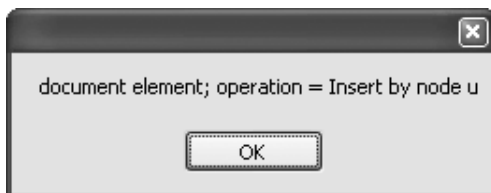


FIGURE 15.14: One of the notifications when adding underline to a rich text (XHTML) field

This dialog appears because a `Changed` event handler is listening to the document element (`myFields`) group. Because it's on the document element, this handler receives all notifications throughout the data source by the means of event bubbling. Let's look at the call to the `MessageBox` class in Listing 15.18 to see how we can get the operation by name.

LISTING 15.18: Generating Visual Notifications for Data Source Events

```
string nodeName = ((XPathNavigator)sender).Name;
MessageBox.Show("document element; operation = "
    + e.Operation + " by node " + nodeName);
```

The `e` variable in the `Changed` event is of type `XmlEventArgs`, but the same `Operation` property is also available on `XmlChangingEventArgs` and `XmlValidatingEventArgs`. `Operation` is of the `XmlOperation` enumeration type. Its allowable values include `Delete`, `Insert`, `None`, or `ValueChange`. By playing with the `MultipleNotifications` sample form, you will see `Delete`, `Insert`, and `ValueChange` operations. `None` is used in other cases where an event is fired for reasons other than inserts, deletes, or changes in value. (In a moment, we'll briefly look at one of the cases when `Operation` is `None`.) When writing code for any of your XML event handlers, one good approach is to handle only those `Operation` values that your code is designed to handle. Alternatively, you can reuse the code snippet shown in Listing 15.18 in your XML event handlers to show the same `MessageBox` as Figure 15.14 whenever a notification occurs. Then you can check for specific conditions and bail (by simply using `return`) from a handler if it's a multiple notification that you want to ignore.

Another out-of-band notification happens whenever a new or existing form is opened: The `Validating` event is fired on every item in the data

source. This occurs to ensure that the data is in a good state before the form opens and even before the `Loading` form event fires. To handle this specific case in your `Validating` event handler, `e.Operation` will always be set to `None`.

Using `XPathNavigator` Objects

If you're aspiring to be an expert in writing form code behind InfoPath forms, the `XPathNavigator` object is absolutely crucial to understand. It is the only gateway to accessing, modifying, and working with the main and secondary data sources. In the samples and code snippets we've looked at thus far, you've learned how to do basic node selection and get a node's value. We now present advanced node selection and data modification tips and tricks that are useful when designing some form templates.

■ **WARNING** Watch for Infinite Change Loops

We'll discuss how to add, replace, and remove nodes in the data source. Remember that as these operations occur, the corresponding XML event handlers will fire! If you modify a node in an event handler that is also handling that node's change, your code will be in an infinite loop. Modifying a node in its own handler still fires the event handler! Consider using `OldValue` and `NewValue` from the `e` parameter to detect this case.

Finding the XPath of a Node

At a loss to figure out how to get the XPath of a node in a data source? Just find the field or group in the *Data Source* task pane, right-click, and choose *Copy XPath*. Now paste it inside double quotes as the `xpath` parameter to an `XPathNavigator.Select` method.

But what about more complicated XPaths that don't represent a simple data source binding? Try using an Expression Box control. Take your time to fiddle with the XPath in design mode and see if it evaluates properly when previewing the form. Once you settle on the correct XPath in the Expression Box, you can copy and paste it into your code.

Using the `NamespaceManager` Parameter

Are you trying to use one of the various `Select` methods (such as `SelectSingleNode`), only to get an `XPathException` with the details “Namespace Manager or `XsltContext` needed”? This means the XPath you’re passing uses at least one namespace prefix, such as `my:`, `dfs:`, `ns1:`, or another “prefix:name” format. You need to pass a resolver parameter (of type `IXmlNamespaceResolver`). If you aren’t very familiar with using `XPathManager` and selecting nodes yet, don’t worry too much about what a resolver does. This parameter is given to you by InfoPath as `NamespaceManager` (available through IntelliSense on the `this` object). It never hurts to simply pass it to node selection methods whenever it’s listed as a parameter—you can still provide it even if you don’t have namespace prefixes in your XPath.

■ TIP No Namespace? No `NamespaceManager`!

The `NamespaceManager` parameter is not necessary when you select nodes that don’t have a namespace. They can appear as `:FirstName` (with a semicolon but without a prefix) in the data source.

Some selection methods, such as `SelectDescendants` and `SelectAncestors`, do not accept `IXmlNamespaceResolver`. Instead, they require a `namespaceURI` parameter as a string. The `namespaceURI` is the actual namespace behind the prefix. For example, the `my` prefix resolves to a namespace such as `http://schemas.microsoft.com/office/infopath/2003/myXSD/2006-01-22T20:55:55`. To figure out the namespace string in your case, go to the *Details* tab of the *Field or Group Properties* dialog for the node you want to select. If you have already selected a node (e.g., the document element) that has the same namespace prefix as a node you want to select, you can get the `namespaceURI` easily via the existing node `XPathNavigator.NamespaceURI` string property. If there’s no namespace listed, the node is not in a namespace, and you won’t need to pass `namespaceURI`. If you must pass `namespaceURI`, use `String.Empty`.

TIP Don't Hard-Code the Namespace URI

Writing robust form code that doesn't hard-code the namespace URI is preferred. Using literal namespace URI strings will break your code if the underlying data source is converted to use a different namespace.

NOTE Secondary Data Sources Are Similar

All of the concepts discussed here also apply to secondary data sources.

Selecting Multiple Nodes

There's no need to use `SelectSingleNode` for every node you want to select. If you're working with a repeating field or group and using `SelectSingleNode`, there's no good way to determine how many of the repeating item exist. To get a set of fields or groups that are repeating, use the `Select` method of the `XPathNavigator`. In the code shown in Listing 15.19 (and in the corresponding sample form template, called `SelectingMultipleNodes`), we're displaying the values of every field in a Bulleted List control. The `XPathNodeIterator` object represents a collection of nodes that we can iterate through by using the `MoveNext` method and the `Current` property. The `Count` and `CurrentPosition` properties (not shown in this sample) on the iterator are useful in `for` loops.

LISTING 15.19: Selecting Multiple Nodes Using XPathNodeIterator

```
string message = String.Empty;
XPathNavigator root = MainDataSource.CreateNavigator();
XPathNodeIterator nodes = root.Select("/my:myFields/my:group1/my:field1",
NamespaceManager);
while (nodes.MoveNext())
{
    message += nodes.Current.Value + System.Environment.NewLine;
}
MessageBox.Show(message);
```

Deleting Nodes

You might think that you can use similar code shown in Listing 15.19 to delete multiple nodes. Could you just call `nodes.Current.DeleteSelf()` within the while loop? Yes, but you'll delete only the first node. Why? Since

780 ■ Chapter 15: Writing Code in InfoPath

nodes is an `XPathNodeIterator`, it does not support being modified while used as an iterator. As soon as it detects itself being modified, the `MoveNext` method prematurely returns `false`.

The proper approach to deleting a series of sibling nodes is to use the `DeleteRange` method. By selecting the first and last nodes within a range, the `DeleteRange` method will remove those two nodes and all nodes in between. To determine which nodes are the first and last for a repeating field or group, we'll use a predicate filter. Recall that passing a simple number as a predicate filter indexes into the 1-based collection of nodes. The last node is found by using the XPath functions `position` and `last`. The code in Listing 15.20 shows the call to `DeleteRange` from the first node with the last passed as a parameter. (The sample file is named `DeletingNodes`.)

■ TIP Using `DeleteRange`

`DeleteRange` is inclusive, that is, the first and last nodes are included in the deletion.

LISTING 15.20: Deleting Node Siblings by Using `DeleteRange`

```
const string f1 = "/my:myFields/my:group1/my:field1";
XPathNavigator root = MainDataSource.CreateNavigator();
XPathNavigator first =
    root.SelectSingleNode(f1 + "[1]", NamespaceManager);
XPathNavigator last =
    root.SelectSingleNode(f1 + "[position()=last()]",
        NamespaceManager);
first.DeleteRange(last);
```

Learning More about `XPathNavigator`

Start with the article “Accessing XML Data Using `XPathNavigator`” on MSDN (as referenced in the Appendix). You can also search the Web for `XPathNavigator` to find more tutorials and references.

Registering Event Handlers

By now, we've seen that InfoPath automatically creates event handlers when you're designing a form template in design mode. Conveniently, they're also

registered for us with the `EventManager` object in the `InternalStartup` method. Listing 15.21 shows an example of registering an event handler with an XML event.

LISTING 15.21: Registering an XML Event Handler to the Changing Event of the UseEmail Node

```
EventManager.XmlEvents [  
    "/my:myFields/my:LoginInfo/my:UseEmail"].Changing +=  
    new XmlChangingEventHandler(UseEmail_Changing);
```

The `EventManager` is a class-wide member whose purpose is limited to tracking event registration. Its use is restricted to the `InternalStartup` method where all registrations must occur. Any registrations outside of `InternalStartup` will cause failures when a user fills out the form (specifically, a `SecurityException` is thrown).

TIP InternalStartup Is Restricted to Registration

The `InternalStartup` method is meant only for registering event handlers with InfoPath. Any calls to the InfoPath OM (in the `Startup` or `Shutdown` methods) will fail when a user fills out the form. Even though you can write other code in this method, it's strongly discouraged. The form may not be fully initialized, which could result in intermittent issues and undefined behaviors.

Form Code Is Separated into Partial Classes

The `EventManager` object is defined in the `FormCode.Designer.cs` file. This file, in conjunction with `FormCode.cs`, completely defines the partial class where your form code is written. To see `FormCode.Designer.cs`, enable the *Show All Files* item on the *Project* menu. This file is auto-generated by InfoPath, and manual editing of its contents is not supported.

During our discussions in this chapter, we've talked about each class of event: form, XML, and control. We briefly discussed the various form events (we'll use form events extensively later). Afterward, we thoroughly

TABLE 15.4: Examples of Advanced XPathS for Registering XML Event Handlers

XPath	Description
/	Selects the XML root; parent of the document element
//my:field1	Selects all descendents (including self) named my:field1
//text()	Selects all text nodes

examined each XML event. InfoPath 2007 supports a single control event: `ButtonEvent`. We used a `Button` control event handler when discussing form errors in the “Validating Event” subsection earlier in this chapter.

Registering events with the `EventManager`, for the most part, isn’t particularly interesting. There is a little flexibility, however, in defining XML event handlers. InfoPath will generate the listener XPath (i.e., the string within the square brackets) based on a field or group in the data source. This is fine for most cases, but sometimes you’ll need a greater level of flexibility. Table 15.4 shows some examples of XPathS that still work as expected when a user fills out the form, even though InfoPath will not generate them.

■ WARNING Complex XPathS for XML Events Are Not Supported

Some complex XPathS are unsupported when you register them for XML events. InfoPath throws an `ArgumentException` when attempting to run the `InternalStartup` method that registers XML events with unsupported XPathS.

Script and the Custom Task Pane

Talking about the InfoPath OM isn’t complete without discussing scripting languages. In its Microsoft Office 2003 infancy, InfoPath supported scripting only with JScript or VBScript. Its object model has changed very little since then, so if you are a legacy InfoPath script writer, you’ll have no problem scripting in InfoPath 2007. Not to disappoint those expert script writers, however, the motivation to use scripting over managed code is dwindling. With the InfoPath managed OM expanded and revamped in 2007, the scripting OM was kept constant. Nevertheless, one major advantage of using

script over managed code is its ability to easily interoperate with the custom task pane. Communicating to and from the task pane from script code doesn't require any special permissions (besides domain trust) and is relatively easy to do.

■ **NOTE** **Supported Script Languages**

Our script samples will be in JScript. VBScript, however, is also supported as a scripting language for form code.

■ **WARNING** **Form Code Language Must Be Homogeneous**

Managed code and script cannot be mixed in an InfoPath form template. In fact, even different languages within managed code (or script) cannot be mixed.

Managed Form Code and Task Pane Script

It's possible to use managed code to communicate to the custom task pane, but it takes some .NET type trickery and a full trust assembly. (Recall from Chapter 11 that a full trust form is different than a fully trusted .NET assembly.) See the InfoPath team blog on the MSDN blogs site, referenced in the Appendix, for details.

Forms Services

Browser-enabled form templates and Forms Services do not support script code.

Custom Task Pane

In the same way InfoPath relies on its own task pane during design mode, you can use your own custom task pane. The custom task pane behaves just like any other task pane. It opens (by default) with your form to the right side of the view but can be moved anywhere or even closed by the user filling out your form. The content of the task pane, as you define it when

784 ■ **Chapter 15: Writing Code in InfoPath**

designing the form, is essentially a Web page in a restricted instance of Internet Explorer. For example, some scripting objects, such as `WScript.Shell`, cannot be created. The task pane, unless you write custom form code, is a disconnected component of the form itself. In a moment we'll see how to tie together the form and the custom task pane.

Forms Services

A custom task pane is supported in a browser-enabled form template, but not when running in the context of a Web browser.

■ NOTE Custom Task Pane Cannot Sink Data Source Events

You cannot bind HTML controls in the custom task pane to the InfoPath data source.

■ WARNING Objects Unsafe for Scripting Are Restricted

Attempting to create restricted ActiveX objects such as `WScript.Shell` results in a "Cannot create automation object" runtime script error.

To add a custom task pane to your form, go to the *Programming* category on the *Form Options* dialog (Figure 15.15). Check the *Enable custom task pane* checkbox to enable the *Task pane name* text box and *Task pane location* drop-down. The name is used in the heading of the task pane. The task pane location can be any Web site URL (e.g., `http://www.moiconsulting.com/taskpane.htm`), local path (e.g., `C:\taskpane.htm`), UNC path (e.g., `\\mycomputer\myshare\taskpane.htm`), or resource file that is included in the form template. In our example, we've added a static HTML page to the form by using the *Resource Files* button.

■ NOTE Custom Task Pane Is Not Shown in Design Mode

The custom task pane does not appear in design mode. In fact, you might forget you have one until you fill out the form!

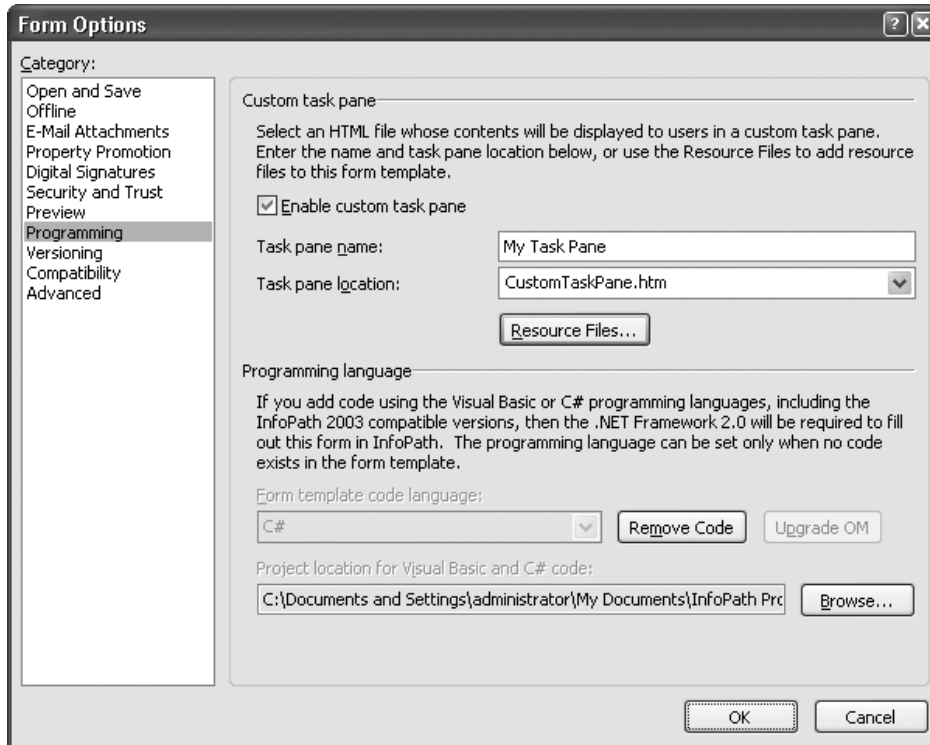


FIGURE 15.15: Setting up a custom task pane in the Form Options dialog

A very useful scenario is to use the custom task pane as an active help reference when users fill out your form. We'll show a sample later in this chapter where we use the custom task pane with the `ContextChanged` event to provide on-demand context-sensitive help.

WARNING Use a Task Pane from Local Resource Files

It's preferred to use a task pane that is included as part of the form template. Referencing an external page, for example, may be prohibited if the form trust level is not sufficient. Moreover, will your users even have permission to access the external site?

Scripting the Custom Task Pane

Designing a form with script is as simple as selecting a scripting language in the *Form Options* dialog's *Programming* category before editing the code.

786 ■ **Chapter 15: Writing Code in InfoPath**

(See “Writing Code Behind a Form” at the beginning of this chapter for details.) If you’ve already started writing managed code, you can use the *Remove Code* button. Unlike managed code and the VSTA project that comes with it, script is edited in the Microsoft Script Editor as a single file. There is no compiling or building with script. Instead, it is interpreted by the scripting engine when the form is opened. The biggest differences between managed code and script include the following.

- The InfoPath 2007 OM does not carry over to scripting languages.
- No IntelliSense (code auto-complete) is available.
- There is no compiling, so code errors are not caught in design mode.
- Script errors reveal themselves when you fill out the form.

Let’s look at a simple sample form that uses script behind the form with a custom task pane (which also includes its own script). As we mentioned, the custom task pane is disconnected from the form itself. Using form code, the task pane can become a more interactive component of the form. Our sample form demonstrates data traveling from the form to the task pane, as well as from the task pane to the form. While we don’t incorporate any specific scenario in the sample, you can picture the form populating a Drop-Down List Box control with a list of articles from a secondary data source. A click on an article in the List Box (handled by the *Changed* event) could fill in a hidden field in the custom task pane HTML and post the data to a server to get the corresponding article details. If you wanted to really tie in the interaction between the task pane and the form, the task pane can directly call in to execute and pass parameters to a method in the form’s script. Let’s look at how the sample, shown in a preview window in Figure 15.16, works behind the scenes. (The sample form template is named *CustomTaskPaneInterop*.)

The first Text Box control, labeled *Field1* in Figure 15.16, is simply bound to a field with the same name in the main data source. Nothing is particularly special about this control. However, the *Set Task Pane Text Box* Button control is hooked up to run some script when clicked. The script essentially takes the value of the *Field1* Text Box and sets the text box in the custom task pane. The code in Listing 15.22 implements this functionality.

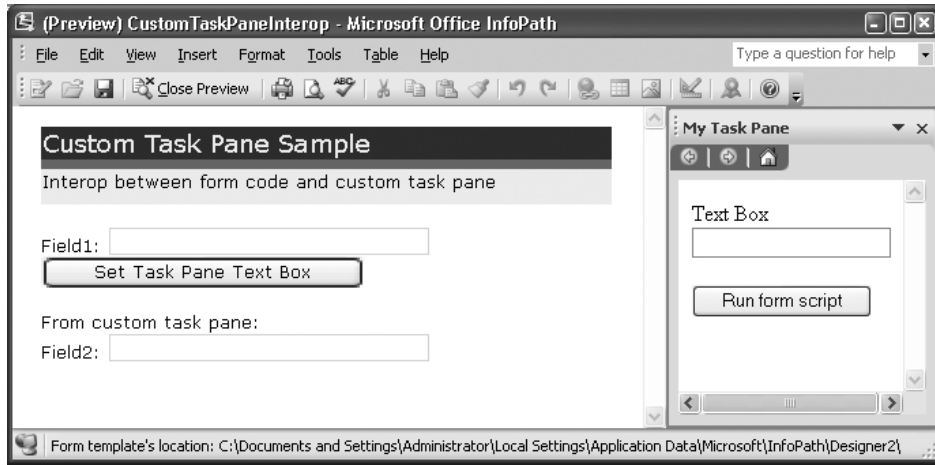


FIGURE 15.16: Filling out the CustomTaskPanelInterop sample form

NOTE Setting Up Events in Script

Sinking an event for a Button control, or any XML or form event for that matter, is the same process as when working with managed form code.

LISTING 15.22: Setting a Value in the Form's Data Source from the Custom Task Pane

```
function CTRL1_5::OnClick(eventObj)
{
    // get the task pane
    var taskPane = XDocument.View.Window.TaskPanes.Item(0);

    // get the html document
    var htmlDocument = taskPane.HTMLDocument;
    var textbox1 = htmlDocument.getElementById("textbox1");

    // get field1 from the main data source
    var myField1 =
        XDocument.DOM.selectSingleNode("/my:myFields/my:field1");

    // assign the value of my:field1 to the task pane text box
    textbox1.value = myField1.text;
}
```

Despite the inline comments, we'll take a closer look at the script to decipher exactly what it is doing. The first step to working with the custom task pane is to get the task pane object. You can always assume the `HTMLTaskPane` object is always the zeroth item in the `TaskPanes` collection.

HTMLTaskPane Object

You can find more information about the `HTMLTaskPane` object on MSDN (as referenced in the Appendix). Additionally, MSDN explains how to wait for the custom task pane to completely load (if it takes a long time loading a large page) and even call methods in the task pane HTML script from the form script.

The next step in the form script code is to get the `HTMLDocument` object. This returns a reference to Internet Explorer's `HTMLDocument` object, which implements the `IHTMLDocument2` interface. We use this object in the next line of script to get the HTML element with ID `textbox1`. As you can probably guess, the `HTMLDocument` object exposes properties and methods that are not part of InfoPath.

■ TIP Useful HTMLTaskPane OM

The `HTMLTaskPane` object has some other useful properties and methods, including the `Navigate` method (accepts a URL), the `HTMLWindow` property (returns `IHTMLWindow2`), the `Type` property (enum `XdTaskPaneType`, which will always return 0, or `xdTaskPane-HTML`, for the custom task pane), and the `Visible` property. The `HTMLWindow` property is level 3 OM, so it's restricted to fully trusted form templates.

The final steps include getting the data source node for `my:field1` and assigning its value to the `textbox1` HTML element in the custom task pane. The final result is that the text box in the task pane gets the value of the *Field1* Text Box control from the InfoPath form.

Programming Internet Explorer

We could write an entire book dedicated to programming Internet Explorer. Actually, we already did! A former publication by coauthor Scott Roberts, *Programming Internet Explorer 5*, was published in 1999 by Microsoft Press.

Let's now look at the HTML behind our custom task pane page, as shown in Listing 15.23. You can see the text box with `id=textbox1` that we're assigning in the script from Listing 15.22.

LISTING 15.23: Calling Form Script from the Custom Task Pane

```
<html>
<head>
  <title>Custom Task Pane</title>
</head>
<body>
Text Box <input id=textbox1 type=text><p>
<input type=button id=runscript onclick="CallFormCode();"
  value="Run form script">
</body>
<script>
function CallFormCode()
{
  window.external.Window.XDocument.UI.Alert(
  'Calling "SetField2" from the custom task pane. ');
  window.external.Window.XDocument.Extension.SetField2(
  textbox1.value);
}
</script>
</html>
```

Now that we've witnessed InfoPath script code changing the HTML content in the task pane, let's see how the task pane can affect the form. In the task pane HTML shown in Listing 15.23, we can see the `CallFormCode` function that is executed when the user clicks the `runscript` button (visible as the *Run form script* button in the custom task pane shown earlier in Figure 15.16). The first thing that the `CallFormCode` function does is call into the InfoPath OM to show an alert dialog. If similar functionality existed in the form script instead of the task pane HTML script, the form script would simply be `XDocument.UI.Alert`. The next line of script uses the `XDocument.Extension` object to access functions in the form script. In this case, the task pane is calling function `SetField2` and passing the value of `textbox1` (the text box in the custom task pane).

NOTE Not All OM Is Available Via `window.external`

Not all InfoPath OM is directly callable through the `window.external.Window.XDocument` object. Access is denied to some parts of the OM for security reasons. A workaround is to use the `XDocument.Extension` object to call a function in the form script that can, in turn, call into any InfoPath OM property or method as allowed by the security level of the form.

Listing 15.24 shows how the `SetField2` function looks. Remember that this function lives in the form script. (Its simplicity may surprise you!)

LISTING 15.24: Implementation of the `SetField2` Method in Form Script

```
function SetField2(val)
{
    var field2 =
        XDocument.DOM.selectSingleNode("/my:myFields/my:field2");
    field2.text = val;
}
```

This JScript code snippet is pretty simple. Despite its plainness, it shows off some of the more interesting capabilities of interacting with the task pane. Since there are no plans to further develop JScript (as well as VBScript) support in InfoPath, there's no guarantee that the script code you write today will work in the next wave of InfoPath programs. We can already see this today: Browser-enabled form templates with script code cannot be published to a server running Forms Services.

You can find more information about programming InfoPath with script languages in any resource that discusses InfoPath 2003 or SP1 scripting. We recommend that you visit MSDN for further reading and research.

Programming InfoPath . . . in Action!

By now you are familiar with the various form events, have a thorough working knowledge of XML events, and know about the single control event (`ButtonClick`). We've seen bits and pieces of the OM throughout this chapter, but we still haven't really used an overwhelming majority of what is available. Those OM objects that appear more often in form events have particularly evaded our focus. The best way to show commonly used InfoPath OM, as well as form events, is to tie everything into a real-world sample.

Forms Services

This sample form template is strictly tied to the full InfoPath object model. In Chapter 17, we'll discuss the shared InfoPath and Forms Services OM in a little more detail.

The MOI Consulting Request Form

For the remainder of this chapter, we'll demonstrate the MOI Consulting request form. This is one of the primary forms used by MOI Consulting to field requests from customers; as such, it is a complex form using a wide variety of InfoPath features including, of course, a generous amount of C# form code. Figure 15.17 shows the *Welcome* view. (This sample is named *MoiConsultingRequest*.)

(Preview) MoiConsultingRequest - Microsoft Office InfoPath


File Edit View Insert Format Tools Table Help

Type a question for help

Submit Close Preview

MOI Consulting Request Form

Our Service is at Your Fingertips Welcome, Administrator!



Request Type

Question about your Account

Request for your Consultant

Time-sensitive / Critical

Other:


Specify a sub-area:

Research

Opinion

References

On-site training



Please contact me:

Yes No

Phone number: 555 1212

Preferred date/time: 8/23/2006 9:33 PM

Now

Your Information

Your Name: John Franks

Company: MOI Consulting International

Customer ID: 123456789 (9 digit "MOID")

To ensure your identity and privacy, please [click here](#) to sign this part of the form and continue...

Alternatively, you can [start over](#).

Form Version: 1.0.0.304 (Cache ID: 63b6eae8584addf3) [What's this?](#)

[Click here to sign this section](#)

Form template's location: C:\Documents and Settings\Administrator\Local Settings\Application Data\Microsoft\InfoPath\Designer2\

MOI Help

Welcome to MOI!

Consulting Request Help

This task pane will update itself as you click on (or tab to) controls in the form.

FIGURE 15.17: Welcome view of the MOI Consulting request form

Filling Out the MOI Consulting Request Form

Let's start learning about the request form by first filling it out. We'll look at the end-user experience and then, afterward, discuss how the form was designed. Since the request sample requires full trust to fill out, we can open the form template in design mode and then click *Preview*. (See Chapter 11 to read more about previewing fully trusted forms.) We'll soon see why the form needs to be fully trusted in the first place.

■ NOTE Web Service Sample Dependency

For this form to run as expected, the `GetSubAreas` Web service must exist at `http://localhost:1369/MoiConsultingRequest/Service.aspx`. Web service code is included in the sample files for this chapter as `GetSubAreas`. You can modify the service location of the sample form template by using the *Data Connections* dialog in design mode.

When the form opens, you may notice a lot of red validation rectangles and asterisks. The form doesn't contain as much validation as it appears to considering that, for example, the Option Button controls for selecting a request type are all bound to the same node. As a result, once you select one of those options, the red validation rectangles on all the Option Button controls in that group disappear.

Let's try toggling the various request type options. As you select different options, two things occur in the form: Items suddenly appear in the subarea List Box, and the custom task pane changes. The subarea List Box changes relative to the request type selected. The custom task pane, on the other hand, jumps to the "Request Type" help topic, which is constant for any request type. (The text shown in the custom task pane changes when you click on various controls in the view. As such, we'll leave it out of the immediate discussion to minimize distractions.) If you select the *Time sensitive/Critical* option, the dialog shown in Figure 15.18 confirms the high-priority request status. Clicking *Yes* in this dialog sets the *Please contact me* control to *Yes* and disables the *No* option.

When the *Please contact me* control is set to *Yes*, the *Phone number* and *Preferred date/time* fields are both required to be filled in. *Phone number* requires a particular format and will show a validation error dialog when it's not

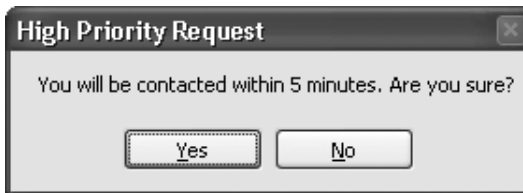


FIGURE 15.18: Dialog that results when the Time-sensitive/Critical request type is selected

satisfied. *Preferred date/time* accepts both a date (in 1/2/2007 format) as well as a time (e.g., 1:23 P.M.), with the default being two days from the current date/time. This field also has many validation constraints. For example, a date/time cannot be chosen in the past, nor can it be within five minutes of the current time. The *Now* Check Box control, situated below the *Preferred date/time* Date Picker, sets the date/time to five minutes from now while emphasizing and disabling the date/time value.

The next set of fields appears under the *Your Information* region. It includes *Your Name*, *Company*, and *Customer ID* Text Box controls. The only requirement for these fields is that they cannot be blank. Following the *Your Information* region are a couple of Button controls: *click here* and *start over*. Clicking *click here* first causes InfoPath to check for validation errors, but only in this view. If any errors are found, a dialog similar to the one shown in Figure 15.19 appears.



FIGURE 15.19: Dialog that appears when form errors exist in the current view

If there are no validation errors, the *Signatures* dialog appears, asking the user to sign the form data. Successfully signing the form switches to the *Request Details* view; otherwise, the user is informed that the data must be signed to continue (Figure 15.20). Since the views aren't listed on the *View* menu, it is impossible to continue to the next view without signing the form.

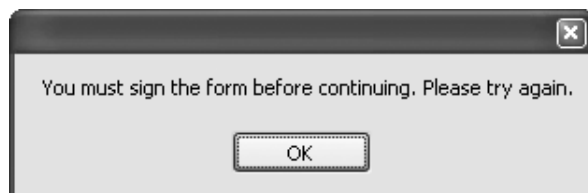


FIGURE 15.20: Informing the user that signing is required to continue

The *start over* Button control, as its name suggests, clears all form data so you can fill out the form from blank. We'll see this Button control again as we continue filling out the form and progress through its different views. When the current view isn't the *Welcome* default view, *start over* carries out the view switch automatically.

Toward the bottom of the form, we see some smaller font text, another Button control, and finally a *Click here to sign this section* link. The text reveals the form version and cache ID values of the form. The version corresponds to the version of the underlying form template. Cache ID, on the other hand, gives the folder name containing the extracted form template files kept on the user's machine. If someone encounters a problem when using the form, having the version readily available can help you determine whether the user is using the most recent template. The cache ID might be useful in rare cases. For example, if the form template cache somehow becomes corrupted, you could instruct the user to delete the cache directory. This forces InfoPath to fetch and cache a fresh copy of the original form template. The *What's this?* Button control simply shows a dialog telling the user that the form version and cache ID are helpful data points for diagnosing problems. Finally, the *Click here to sign this section* link is the standard (noncustomizable) entry point provided by InfoPath when a signable section exists in the view. Clicking it does not sign the form but instead opens the dialog shown in Figure 15.21.



FIGURE 15.21: Dialog that appears after using the Click here to sign this section link in the MOI request form

Once there are no errors, the *click here* Button control has been clicked, and signing has been successful, the view switches to *Request Details* and the task pane disappears (Figure 15.22).

The *Request Details* view contains two input fields: a Rich Text Box for the request description and a Bulleted List for additional details. The Bulleted List is included inside an Optional Section that does not exist by default. Clicking the *Add Additional Details* Button control inserts the Optional Section. Subsequent clicks on the same Button control insert new Bulleted List items. After completing this part of the form, we can click *Continue* to move forward. Clicking *Back*, however, takes you back to the *Welcome* view.

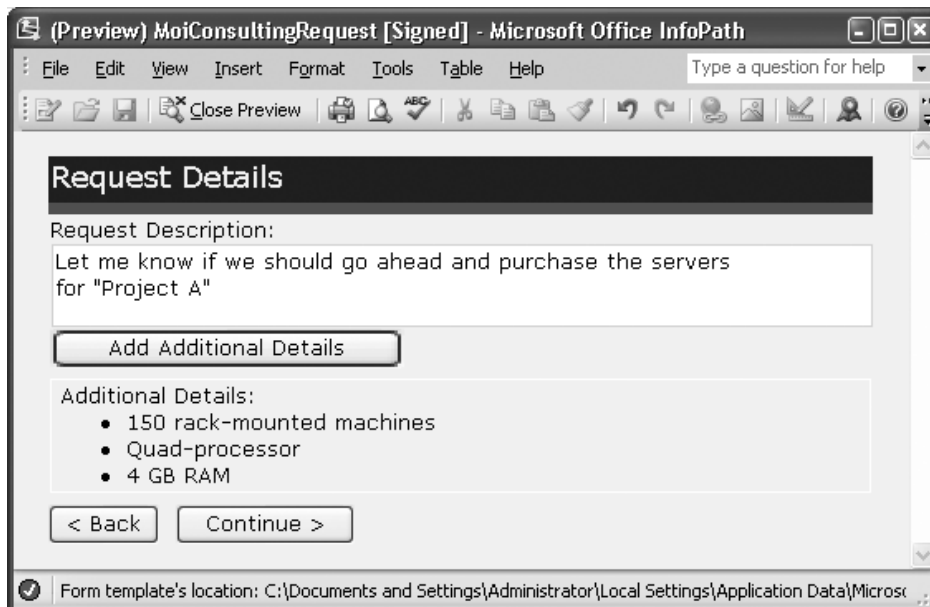


FIGURE 15.22: Request Details view of the MOI Consulting request form



(Preview) MoiConsultingRequest [Signed] - Microsoft Office InfoPath

File Edit View Insert Format Tools Table Help Type a question for help

Please Review the Following Information

Request Type ConsultantRequest

Contact Me yes

Contact Info (123) 555-1212

Current User Administrator

Sub Area Research

Preferred Date 2/1/2007

Request Description Let me know if we should go ahead and purchase the servers for "Project A"

Additional Details

- 150 rack-mounted machines
- Quad-processor
- 4 GB RAM

<-- Start Over <- Go Back < Submit >

Form template's location: C:\Documents and Settings\Administrator\Local Settings\Application Data\Microsof...

FIGURE 15.23: Confirm view of the MOI Consulting request form

No data on the *Welcome* view can change because it has been signed and, as a result, is read-only. The only options are to use the *click here* or *start over* Button controls.

After clicking *Continue* on the *Request Details* view, a dialog appears asking us to confirm the data we've entered. This is the (appropriately named) *Confirm* read-only view. Figure 15.23 shows the *Confirm* view, which summarizes the data we've entered thus far. Nothing can be changed unless we click *Back* to change the data through the *Welcome* or *Request Details* views.

Clicking *Submit* takes us to the final view (Figure 15.24). The form was "submitted" by being saved to the location shown in the link. From here, we can click either *Close this Form* or *Create a New Request*. The form-filling session is complete.

Unfortunately, we can't cover every possible scenario in this form. But we suggest that you spend some time playing with this sample and getting accustomed to its behavior. Having a better understanding of the details in

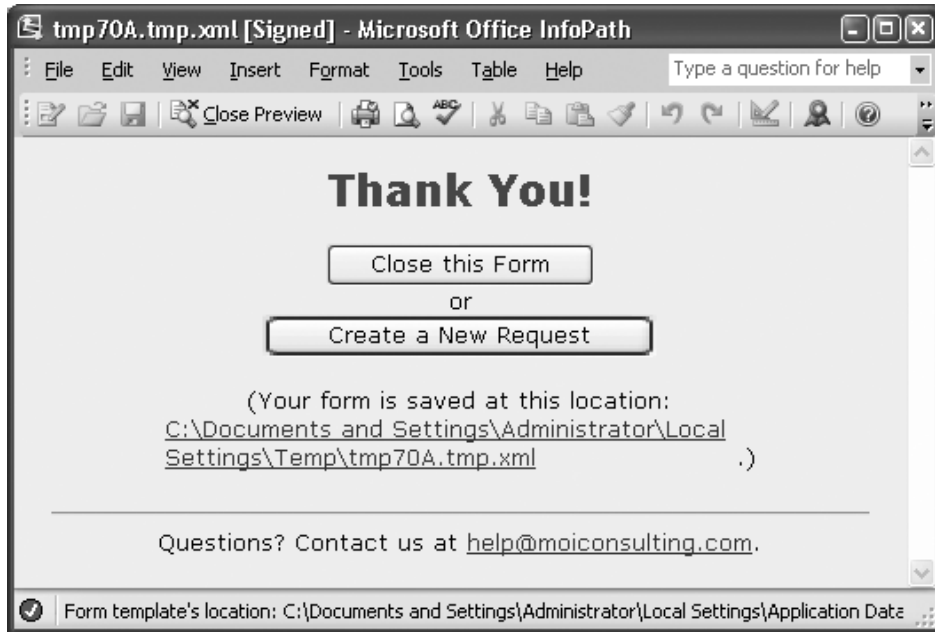


FIGURE 15.24: Thank You view of the MOI Consulting request form

this form will benefit you when we discuss its detailed design and underlying form code implementation.

Designing the MOI Consulting Request Form

Are you anxious to learn how to design this form template? For the remainder of the chapter, we'll go through many lines of form code as well as relevant thought processes used to develop and refine the form's behavior. We'll introduce new OM objects, including their properties and methods, as we encounter their use in the code behind the form.

NOTE Taking a Scenario-Based Focus

It's not our intention to cover the precise steps of designing this form. Our goal is to focus on the scenario and higher-level operatives that contribute to a complex template. Since this chapter is about programming InfoPath, we'll obviously dedicate more time to form code.

Gathering Requirements and Designing the Visual Layout

To begin designing the MOI Consulting request form, we first gather our data requirements. The correlation between the form and our data requirements is reflected in each control in the *Welcome* and *Request Details* views. Each control essentially maps to one of our data-gathering requirements. Note that the data requirements do not yet involve creating the data source or even designing the form. When sitting down to design this form, we compile a list of data to collect from users. This list is essentially reflected in the main data source and bound to controls in the view; so for this purpose, you can infer our data requirements by browsing the form views and data source in design mode. We'll create a first draft of the data source in the next step.

After realizing what data we need to collect, we begin laying out the *Welcome* view. InfoPath uses two-dimensional flow layout, so we insert *Table with Title* and *Three-Column Table* layout tables (found on the *Layout* task pane) as the foundation for allowing us to accurately position our content. Then we choose a color scheme to spice up the boring black-and-white form. However, we could choose the color scheme later since it can be applied at any time.

The next step in our design approach is to insert various controls from the *Controls* task pane to satisfy our data requirements. At this point, we don't necessarily care about the data source. Our goal is to visually lay out controls that correspond with our data requirements; we'll deal with the details of the data source later. As we insert controls, we try to logically group them within adjacent table cells. Some crafty work with the *Merge Cells* and *Split Cells* toolbar buttons as well as changes to the height and width of individual cells eventually give us a pleasing visual layout. To help identify controls, we type labels adjacent to them.

At this point you might realize that there are too many controls on the view, or that it's just too busy. This is where multiple views come in handy. We decide to use a wizard style of form (with *Next* and *Back* buttons) because the user should confirm the data entered across more than one view. Finally, we add a terminating view (*Thank You*) to make it very clear that the form had been filled out and submitted successfully.

Organizing the Data Source

Once we have hashed out the visual layout and end-user form-filling scenarios, the form looks like it's 90% ready for production. Truth be told, we

have barely scratched the surface in terms of the total amount of work that needs to be done! The next step is determining the structure of the main data source and the names of its fields and groups. Since we've already created an unstructured version of the data source by dragging all of the controls into the view, this step is mostly a matter of moving and renaming fields. However, we need to consider some other data source changes to support the visual design of the form.

One such change in the data source, for example, is related to having a wizard-like form. We want the user to continue to the next view only if the current view is free of validation errors. (We'll see how to check for form errors in the "Adding Form Code" section later in this chapter.) The easiest way to achieve such behavior is to gather all data source fields and groups that you want to validate under a common parent. This is easiest to do by creating groups that map to the views in the form. In our case, only the *Welcome* and *Request Details* views collect data from users. Once we create groups that correspond with these views, we can move each node into its appropriate group.

Another task of data source editing involves renaming the generic node names (`field1`, `group1`, and so on) to something meaningful. Remember the text labels we created for the newly inserted controls when visually setting up the form? We've found it easiest to mimic those names in the field or group name with PascalCasing. Many benefits are realized by properly naming data source nodes. One benefit that we didn't realize until it "just worked" is displaying the raw field names to the user to identify specific errors (as shown earlier in Figure 15.19). Another big win is hooking up the context-sensitive help in the custom task pane. As you can see, naming your nodes properly can make form development easier when you add advanced form features.

■ TIP Modifying Other Data Source Properties

While you're renaming data source nodes, also consider whether they cannot be blank and whether the data type should be more restrictive. For data types such as integer, you can click the *Format* button to define a display format. If you're not sure, you can always change the properties later, but it's quicker to make changes in a single pass.

800 ■ **Chapter 15: Writing Code in InfoPath**

What if you want to use some data in your form but really don't have anywhere to put it (such as the form version or cache ID)? Consider adding that data node under the document element. In our case, let's add `FormVersion`, `CacheId`, and `SavedFileLocation` as attribute fields to the document element. These fields aren't edited by the user, yet they're set through form code and are displayed to the user.

■ WARNING Do Not Store Sensitive Information in the Main Data Source

Storing sensitive data in the main data source is not recommended, even if you are not displaying the fields in the view. Simply saving the form and viewing the content in a text editor (unless the form uses Information Rights Management) will reveal all form data in plain text.

This form also contains a data connection (and corresponding secondary data source) for populating the subarea List Box items. We can create a very simple Web service (whose code is available in the `GetSubAreas` sample file) that takes a request type as a parameter and returns an array of subarea strings. The data connection that we create to query this Web service does not have the *Automatically retrieve data when form is opened* checkbox checked. This is because we're going to set the parameter value and perform the query manually in form code.

Finally, we want to make sure that the data entered by the user on the *Welcome* view is digitally signed for proof that this person, indeed, filled out the request. We enable signing the `Welcome` group by inserting it into the view as a Section control, essentially dragging the entire view (except for the Table with Title at the top) into the Section, and turning on the option that allows the Section to be signed. Alternatively, we could have added the `Welcome` group to the set of signable data in the *Form Options* dialog. However, we want the user to get visual feedback that the data is digitally signed with his or her own certificate. The fallout from showing the digital signature in the view leads to having the *Click here to sign this section* link available for users to click. Since we want to check for form errors in the *Welcome* view before a user can sign the data and switch to the *Request Details* view, we want to block this link from doing any signing. We'll see how to handle this in the "Adding Form Code" section.

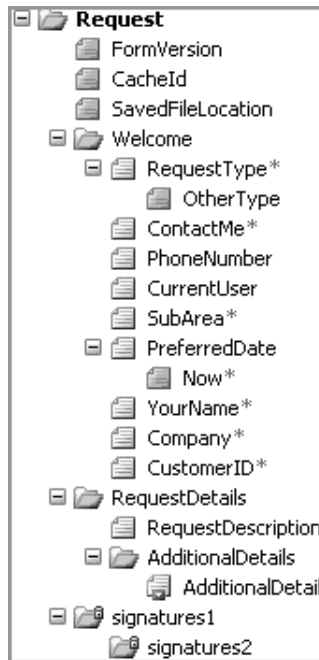


FIGURE 15.25: Main data source for the MOI Consulting request form template

Figure 15.25 shows the main data source for the MOI Consulting request form template.

Adding Logic without Code

The next major step after getting most of the data source in place is to add logic to our form. We could implement all of our form's logic by just writing code, but that's not the best approach. You might not expect us to recommend this (especially in this chapter), but you should consider using rules before form code when adding logic. Form code not only takes longer for you to implement but also is more difficult to maintain. It is prone to errors and security flaws, despite how good we think we are at programming. Rules, on the other hand, can be maintained by almost anyone and have been well tested for use in countless situations. Moreover, rules are declaratively defined as part of the form definition, so there is neither code to compile nor an assembly to build.

802 ■ **Chapter 15: Writing Code in InfoPath**

Here is a sampling of the rules used in the MOI Consulting request form.

- When `ContactMe` is changed, set `Now` to `false`.
- When `Now` is checked, set `PreferredDate` to `now()` plus five minutes.
- When `RequestType` is `SensitiveCritical`, set `ContactMe` to `yes`.
- Clicking the *What's this?* Button control on the *Welcome* view shows a dialog box message.
- Clicking the *Next* or *Back* Button controls takes the user to the next view or the previous view, respectively.

As you can see, rules define an “if X then do Y” behavior that can easily replace a few lines of form code. Rules can also have multiple conditions and multiple actions that may be tedious to implement using code. Writing the code isn’t the only burden—there’s also the need to handle special cases, for example, whether a field node exists or is just empty. If you’re skeptical that rules save time and energy, we leave it as an exercise for you to reimplement all rules as form code in the MOI request form.

More logic that we can’t forget to add to our form involves data validation and conditional formatting. Using InfoPath’s built-in pattern builder to define data validation constraints basically defeats the purpose of doing any basic regular expression matching in the `Validating` event handler. We also use extensive data validation on the `PreferredDate` field; errors will be reported under any of these conditions:

- If `ContactMe` is `yes` and `PreferredDate` is blank
- If `PreferredDate` is within an hour of `now` and `RequestType` is not `SensitiveCritical` and `ContactMe` is `yes`
- If `PreferredDate` is less than five minutes from `now` and `ContactMe` is `yes`

Conditional formatting in this sample form makes controls read-only or hidden under certain conditions in the data. For example, we use conditional formatting to hide the `OtherType` Text Box when `RequestType` is not set to `Other`. Condition disabling is used for both the *Please contact me* (bound to field `ContactMe`) Option Button controls as well as the `PhoneNumber` and `PreferredDate` fields under certain circumstances. We didn’t conditionally

disable the *click here* Button control, but we could have done so by checking some field (perhaps `enableClickHere`) in the data source that does not exist in the view. In the right circumstances, we could set this field to a specific value through form code, which would allow the user to click the Button control. We won't take this approach because we believe that having a disabled control would frustrate users. Keeping the Button control enabled might tempt some users to click it too early, but we offer feedback (in this case) through data validation. That's where we show the error dialog shown earlier in Figure 15.19.

Adding Form Code

When making the decision to add code, it shouldn't only be a question of whether you already tried using rules and data validation. You should also be sure to solidify names and the structure of the data source, as we mentioned earlier. As we start to add form code, we should have already convinced ourselves that between rules and data validation we still cannot get the desired form behavior. Although this is the guidance we give to our readers, we do not completely adhere to this practice in the MOI request form. We'll call out when we're violating our own principles, but keep in mind that it's only to show the breadth of writing form code with the InfoPath OM.

Naming Your Nodes and Buttons Besides assigning real names to data source fields and groups, giving real ID values to all Button controls (in the *Button Properties* dialog) is worth the small time investment. This seemingly tedious task is important, if, of course, you want your form code to be readable and maintainable! But how do data source names, structure, and Button ID values correlate with code behind the form? Take a look at our `InternalStartup` method in Listing 15.25 and see for yourself.

Without proper field and group names in the data source, the XPath expressions used for hooking up any XML events would be meaningless. Moreover, the event handler method names are also based on the nodes whose events they're sinking. It's possible to change the names of nodes in the data source after creating XML event handlers, but InfoPath will update only the XPath expression. The method name does not change unless you do it manually. Furthermore, any code you already wrote that depends on a

804 ■ Chapter 15: Writing Code in InfoPath

LISTING 15.25: InternalStartup Definition for the MOI Consulting Request Form Template

```

public void InternalStartup()
{
    EventManager.XmlEvents[
        "/my:Request/my:Welcome/my:RequestType"].Changing +=
        new XmlChangingEventHandler(RequestType_Changing);
    EventManager.FormEvents.Loading +=
        new LoadingEventHandler(FormEvents_Loading);
    ((ButtonEvent)EventManager.ControlEvents["Start_Over"]).Clicked +=
        new ClickedEventHandler(Start_Over_Clicked);
    ((ButtonEvent)EventManager.ControlEvents[
        "click_here_to_sign"]).Clicked +=
        new ClickedEventHandler(click_here_to_sign_Clicked);
    EventManager.FormEvents.ContextChanged +=
        new ContextChangedEventHandler(FormEvents_ContextChanged);
    EventManager.XmlEvents[
        "/my:Request/my:Welcome/my:RequestType"].Changed +=
        new XmlChangedEventHandler(RequestType_Changed);
    EventManager.FormEvents.ViewSwitched +=
        new ViewSwitchedEventHandler(FormEvents_ViewSwitched);
    ((ButtonEvent)EventManager.ControlEvents[
        "Add_Additional_Details"]).Clicked +=
        new ClickedEventHandler(Add_Additional_Details_Clicked);
    EventManager.FormEvents.Submit +=
        new SubmitEventHandler(FormEvents_Submit);
    EventManager.FormEvents.Sign +=
        new SignEventHandler(FormEvents_Sign);
}

```

particular data source structure (e.g., using `XPathNavigator` objects to select nodes) could break. It's easy to see that getting your data source right the first time saves time and problems in the long run!

Unlike XPath expressions that automatically update when the data source changes, Button ID values do not enjoy the same luxury. If the Button ID is changed when a `Clicked` event handler is already set up, that particular Button control will no longer fire the event handler. With default Button IDs such as `CTRL1_5` (and whatever other senseless IDs Button controls can have), looking at a method in form code such as `CTRL1_5_Clicked` is frustrating. Furthermore, if your form has many Button controls, it's not easy to determine which ID maps to a given Button. The best way to understand which `Clicked` event handlers listen to which Button controls is to give each

Button's ID value a meaningful name. We visit each Button's properties dialog in the MOI request form, copy the display label, paste it into the ID box, and then replace spaces with underscore characters. (A Button control ID must begin with a letter and contain only alphanumeric characters and underscores.) This practice, as you can see in Listing 15.25, makes the Clicked event handler methods easily recognizable.

Showing a Custom Dialog with Buttons One of the first form behaviors you may have noticed is a confirmation dialog (shown earlier in Figure 15.18) that appears when you select the *Time-sensitive/Critical* request type. The code that implements this behavior (Listing 15.26) shouldn't introduce anything new in terms of the Changing XML event. However, it does show how to use the `MessageBox.Show` static method to show buttons and get back the user's selection. The Rules feature can only show a dialog with an OK button. In this case, we want user feedback, which requires the use of form code.

LISTING 15.26: Using `MessageBox.Show` to Get User Feedback

```
public void RequestType_Changing(object sender, XmlChangingEventArgs e)
{
    if (e.NewValue == "SensitiveCritical"
        && e.Operation == XmlOperation.ValueChange)
    {
        DialogResult result = MessageBox.Show(
            "You will be contacted within 5 minutes. Are you sure?",
            "High Priority Request", MessageBoxButtons.YesNo);
        if (result != DialogResult.Yes)
            e.CancelableArgs.Cancel = true;
    }
}
```

Showing Read-Only Properties from the OM Another interesting form feature you may have noticed is the use of dynamic, read-only data in parts of the view. Specifically, we show the user name in the upper right, and in the lower right we show the form version and, in parentheses, the cache ID (refer back to Figure 15.17). How did we get this data? It comes straight from various InfoPath OM properties. But to show this data in the view, we need to set values of fields in the data source that are bound to Expression Box controls. We could use read-only Text Box controls, but Text Box controls

806 ■ **Chapter 15: Writing Code in InfoPath**

have an opaque white background, which makes them stand out (and beg to be filled out) against the soft peach view background. Since these controls are not to be filled out by the user yet still need to exist in the data source, we add `FormVersion` and `CacheId` nodes under the document element as attribute fields and put `CurrentUser` within the `Welcome` group as an element field. (Refer back to Figure 15.25 for the main *Data Source* task pane.)

Since setting the `FormVersion`, `CacheId`, and `CurrentUser` fields is a one-time occurrence when the form opens, the code seems to fit nicely in the `Loading` event. We also have some other code that must be run either when the form first opens or during other one-time operations. The code that needs to run when the form opens does a check to see if the form that's opening is new from the template or an existing `.xml` file. In the MOI request form, we don't want existing forms to be reopened. In fact, let's disable the *Save* and *Save As* options (through the *Open and Save* category of the *Form Options* dialog) to ensure that the form can't be saved.

A Form to Be Filled Out in One Sitting Why do we go through the trouble to discourage saving and prevent opening an existing form? The request form isn't meant to be saved. Its purpose is to allow a user to quickly fill out one request during a single sitting. We can check to see whether the form is new or not by looking at the `New` property. In the form code shown in Listing 15.27, if the form is not new (i.e., it is an existing saved form—and who knows how the user got it?), we cancel the `Loading` event. Similar to canceling the `Changing` event, which we discussed earlier in this chapter, canceling the `Loading` event forces InfoPath to fail loading the form.

NOTE **UserName VERSUS LoginName**

Getting the user's name with `Application.User.UserName` returns the Active Directory (AD) display name. If AD is unavailable, the nondomain portion of the login name is returned. Alternatively, `Application.User.LoginName` always returns the `DOMAIN\LoginName` identity of the user. Note that using `Application.User.LoginName` requires a fully trusted form.

LISTING 15.27: The Loading Event Handler for the MOI Consulting Request Form

```
const string _startOverOuterXml = "startOverOuterXml";
const string _startOverGetSubAreas = "startOverGetSubAreas";
/* . . . */
public void FormEvents_Loading(object sender, LoadingEventArgs e)
{
    // Only allow new forms (i.e., no saved forms can be opened).
    // We could add more logic to check the user role or specific data
    // in the form before rejecting it.
    if (!this.New)
    {
        e.CancelableArgs.Cancel = true;
        e.CancelableArgs.Message = "I'm sorry, you must fill out a "
            + " new request. Previous requests cannot be honored.";
        return;
    }

    XPathNavigator root = MainDataSource.CreateNavigator();
    XPathNavigator user = root.SelectSingleNode(
        "/my:Request/my:Welcome/my:CurrentUser", NamespaceManager);
    user.SetValue(Application.User.UserName);
    XPathNavigator uri = root.SelectSingleNode(
        "/my:Request/@my:CacheId", NamespaceManager);
    uri.SetValue(this.Template.CacheId);

    XPathNavigator version = root.SelectSingleNode(
        "/my:Request/@my:FormVersion", NamespaceManager);
    version.SetValue(this.Template.Version);

    // Cache the Entire data source for the "Start Over" feature.
    // The StartOverOuterXml property will return this saved data.
    XPathNavigator docElem = root.SelectSingleNode("");
    FormState.Add(_startOverOuterXml, docElem.OuterXml);

    // Cache secondary data source.
    XPathNavigator getSubAreasRoot =
        DataSources["GetSubAreas"].CreateNavigator();
    XPathNavigator getSubAreasDomElem =
        getSubAreasRoot.SelectSingleNode("");
    FormState.Add(_startOverGetSubAreas, getSubAreasDomElem.OuterXml);

    // Set the default view.
    ViewInfos.Initial = ViewInfos["Welcome"];
}
```

808 ■ Chapter 15: Writing Code in InfoPath



FIGURE 15.26: Error dialog that appears when user attempts to open an existing MOI request form

If a user tries to open an existing form, an error dialog appears (Figure 15.26), and then InfoPath closes the form-filling session.

“Start Over” Feature and Setting a Default View Another nifty form feature in this form is the ability to start over when filling out the form. You can find Button controls labeled *start over* or *Create a New Request* in three of the four views. All of these Button controls call into the same clicked event handler. But before we look at that particular event handler, we need to first consider a couple other one-time operations that are included in the `Loading` event. The comments in Listing 15.27 (labeled with comments `Cache the Entire data source for the "Start Over" feature` and `Cache secondary data source`) show caching of both the main and `GetSubAreas` data sources. We store the data by getting the document element’s outer XML and saving it in the `FormState` object. Without doing these prerequisites, the “start over” feature would not work. This will become clearer in a moment as we see the code behind starting over. The last line in the `Loading` event sets the view that appears when the form opens. Despite setting the *Welcome* view as the default in design mode, this setting is a little redundant. But since calling the `SwitchView` method (a member of `ViewInfos`) is not supported and will throw an exception in the context of the `Loading` event, it’s helpful to know how to set a default view as the form opens. (Calling `SwitchView` within the `Merge` form event is also disallowed.)

■ **TIP** **Setting the Default View Dynamically**

You could add additional logic (e.g., using `if` statements) to go to a specific view based on criteria in the form (such as a field value) or other factors (such as the user’s name or role).

The “start again” feature we added to our form came out of our feeling that it’s easier to start over by clicking a button than it is to manually close and reopen a form. To facilitate starting again, we should think about the various states of the form at the time it’s invoked.

- We can be on any view.
- There could be validation errors.
- The form could be blank to begin with.
- The data may be digitally signed.

Since we could be on any view, starting again means we should go back to the *Welcome* view. As a result, the first OM call is to `SwitchView`. If we’re already on the *Welcome* view, this code has no effect. So there’s no need to check what view you’re on with `this.CurrentView` before switching. We threw the next two bullet points regarding validation errors and a blank form as curve balls! These form states do not affect the ability for our “start again” code to perform as expected. Neither validation errors nor the blank form (which still has validation errors) hinders us in modifying the data source.

When we first wrote the `Start_Over_Clicked` Button handler (see Listing 15.28), we didn’t check to see if the form was digitally signed. If the form is signed, clearing the main data source fails. Remember that a signed data block is always read-only after a signature is applied. Seeing as it could be a potential security issue if form code were allowed to remove a signature, we cannot do it programmatically. To accommodate this scenario, we still switch to the *Welcome* view but bail out by asking the user to manually remove the signature and try again. That’s exactly what we do within the first `if` statement in Listing 15.28.

If the form isn’t digitally signed, we have the green light for clearing the data sources. Remember how we cached the main and secondary data sources in the `Loading` event? Specifically, we took the document element outer XML strings and added them to the `FormState` object. As you can see in Listing 15.28, we call `ReplaceSelf` on the document element node (of both the main and `GetSubAreas` data sources) and pass our cached copy of the outer XML. Since our cached copies of the data sources are snapshots of the initial form’s data sources, we can rest assured that they’re valid according to the schema and that the form will appear similar to a newly created form. The

810 ■ Chapter 15: Writing Code in InfoPath

code in Listing 15.29 supports the `Start_Over_Clicked` method by exposing our `FormState` data wrapped in properties for consistent data access.

LISTING 15.28: Implementation of the Start Over Button Click Event

```
public void Start_Over_Clicked(object sender, ClickedEventArgs e)
{
    // Every button ID called "Start_Over" calls into this handler
    ViewInfos.SwitchView("Welcome");

    // Tell the user to remove his or her digital signature
    if (this.Signed)
    {
        MessageBox.Show("Cannot start over because the form "
            + "data is signed. Please go to the Digital Signatures "
            + "dialog and remove your signature. Then click this "
            + "button again to continue.");
        return;
    }

    // Clear the main data source
    XPathNavigator docElem =
        MainDataSource.CreateNavigator().SelectSingleNode("*");
    docElem.ReplaceSelf(StartOverOuterXml);

    // Clear any secondary data sources
    XPathNavigator subAreasRoot =
        DataSources["GetSubAreas"].CreateNavigator();
    XPathNavigator subDocElem = subAreasRoot.SelectSingleNode("*");
    subDocElem.ReplaceSelf(StartOverGetSubAreas);
}
```

■ NOTE Using `FormState` to Persist Session Data

To recall an earlier discussion in this chapter, the `FormState` object holds name-value pairs of any serializable types. Its purpose is to maintain state in form code from the time a specific form is opened for filling until it is closed. The `Loading` event is well suited for initializing `FormState` objects.

LISTING 15.29: Persisting the Default State of XML for Main and Secondary Data Sources

```
const string _startOverOuterXml = "startOverOuterXml";
const string _startOverGetSubAreas = "startOverGetSubAreas";
/* ... */
private string StartOverOuterXml
{
    get
```

```
{
    if (FormState.Contains(_startOverOuterXml))
        return FormState[_startOverOuterXml] as string;
    throw new Exception(
        "Form load did not load properly before Starting Over!");
}
}
private string StartOverGetSubAreas
{
    get
    {
        if (FormState.Contains(_startOverGetSubAreas))
            return FormState[_startOverGetSubAreas] as string;
        throw new Exception(
            "Form load did not load properly before Starting Over!");
    }
}
```

Checking for Errors and Adding a Digital Signature Now that we've seen how the form can start over, we should learn about the *click here* Button control. As its surrounding text hints, clicking this Button will "sign this part of the form and continue." But that's not all. Earlier we made it a requirement that before the user can sign and switch to the next view, there must be no pending form errors within the *Welcome* group. That means all required or invalid fields with validation errors in the *Welcome* view must be fixed. To determine whether a form error exists on an item within the *Welcome* group, we use the `XPathNavigator.IsDescendent` method. As you'll see in the code, we ask the *Welcome* node whether a specific node is a descendent; if so, we've found a node of interest. If errors are found, we show a dialog listing the exact errors with a corresponding corrective action. Figure 15.27 shows the dialog (which also



FIGURE 15.27: Dialog for clearly presenting to the user any form errors

812 ■ Chapter 15: Writing Code in InfoPath

appeared earlier as Figure 15.19). Generating such a detailed dialog is very easy. But ultimately, to make the dialog helpful for users, you must rely on field names that closely resemble any text labels adjacent to the corresponding controls.

As you look at the code snippet shown in Listing 15.30, you'll see that we first check whether the form data is signed. This will be true, by the way, if the form is partially or wholly signed. If signing has already occurred, we don't need to check for form errors. For signing to occur in the first place, we must have already validated that there are no errors through this same code path! Since we're also sinking and special-casing the `Signing` event (as you'll soon see), there's no other way for the form to get signed.

LISTING 15.30: Code for the Button That Starts Validation on the Default View of the MOI Consulting Request Form

```
public void click_here_to_sign_Clicked(
    object sender, ClickedEventArgs e)
{
    if (this.Signed)
    {
        ViewInfos.SwitchView("RequestDetails");
        return;
    }

    // 1. Check for errors
    XPathNavigator root = MainDataSource.CreateNavigator();
    XPathNavigator welcome = root.SelectSingleNode(
        "/my:Request/my:Welcome", NamespaceManager);

    string welcomeErrors = string.Empty;
    foreach (FormError error in Errors)
    {
        // Is this error under the "Welcome" group?
        if (welcome.IsDescendant(error.Site))
            welcomeErrors += error.Site.LocalName + ": "
                + error.Message + System.Environment.NewLine;
    }

    // If there are errors, report them and stop
    if (!string.IsNullOrEmpty(welcomeErrors))
    {
        MessageBox.Show("Please fix these errors: "
            + System.Environment.NewLine + System.Environment.NewLine
            + welcomeErrors, "Please Fill or Fix Some Data");
        return;
    }

    // Tell the Signed event it's okay to allow signing
    SigningAllowed = true;
}
```

```
// Continue with signing
foreach (SignedDataBlock sig in SignedDataBlocks)
    // Sign only this block
    if (sig.XPath.Expression.EndsWith("my:Welcome"))
    {
        // Calling Sign requires a full trust form
        sig.Sign();

        // Was it actually signed or canceled?
        if (sig.Signatures.Count > 0)
            ViewInfos.SwitchView("RequestDetails");
        else
            MessageBox.Show("You must sign the form "
                + "before continuing. Please try again.");
        break;
    }
    SigningAllowed = false;
}
```

The code for going through errors is quite easy. First, the `FormErrorsCollection` object is exposed by the `this.Errors` (or just `Errors`) property. Since `FormErrorsCollection` inherits from the `ICollection` interface, iterating through each `FormError` object is performed by a `foreach` loop. Of the properties on the `FormError` object, the most interesting to us are `Site` (yielding the `XPathNavigator` of the node with the error) and `Message`. `Error` will always provides a nonempty short error message, while `MessageDetails` might be more verbose. `MessageDetails` is often empty unless a specific data validation constraint provides details. In this sample, `MessageDetails` is empty for all errors except for a few that we defined through InfoPath design mode, such as the `PhoneNumber` and `PreferredDate` fields. To detect whether there are any errors to show, we start with an empty `welcomeErrors` string. If it's still empty after looping through any form errors, the code skips over showing a dialog because there are no errors.

A moment ago, we said that we're sinking the `Signing` event to programmatically determine when signing is permitted. Essentially, we want the code from Listing 15.30 (but nothing else) to initiate signing the form. Other ways to add a signature include clicking the *Click here to sign this section* link in the view as well as using the *Digital Signatures* dialog accessed from the *Tools* menu. No matter how signing is instigated, the `Signing` event is always fired, which takes over the entire signing process. For our scenario

814 ■ **Chapter 15: Writing Code in InfoPath**

to work properly, we need to essentially block all attempts to sign the form unless the *click here* Button control is the initiator. The `SigningAllowed` Boolean property facilitates our scenario by allowing signing in the `Signing` event only when the property is set to `true`. We'll look at the `Signing` event shortly.

To sign the `Welcome` group in the form, the code iterates through the `SignedDataBlocks` collection. We look for the signed data block whose XPath ends with `my:Welcome` for brevity. The full XPath is `/my:Request/my:Welcome`. Calling the `Sign` method tells InfoPath to show the signing dialog for this specific signed data block. It is not possible to block this dialog and silently sign the form. This is for security reasons. Despite our call to `Sign`, it's up to the user to follow through with signing. Since the dialog could be closed or canceled without signing, the code must check whether a signature really exists. To perform this check, we use the `Count` property of the `Signatures` object. Instead, since there is only one signature, we could have used the `this.Signed` property as we did earlier in this method.

■ WARNING Determine Which Data Block Is Signed

The `this.Signed` property will return `true` if the form has a signature. If you have multiple signed data blocks, the `Count` property on the `Signatures` object of the `SignedDataBlock` itself is sure evidence that the specific block in question was signed.

The `Signing` event is fired immediately before the `Signatures` dialog, which signs the data, appears. We mentioned before that we need to block signing unless the user clicks the *click here* Button control. Since we just reviewed the `click_here_to_sign_Clicked` event handler sinking that Button control, we can see how to arrange this. Let's also look at the `SigningAllowed` property that maintains its Boolean status through the `FormState` object. Listing 15.31 shows the code from the sample form template.

■ NOTE Signing Event Requires Full Trust

Only fully trusted forms can sink the `Signing` event. As a result, you'll need to set the form template's security level to full trust.

LISTING 15.31: Code for the Sign Form Event

```
const string _signingAllowed = "signingAllowed";
/* . . . */
private bool SigningAllowed
{
    get
    {
        if (FormState.Contains(_signingAllowed))
            return (bool)FormState[_signingAllowed];
        return false;
    }
    set
    {
        if (!FormState.Contains(_signingAllowed))
            FormState.Add(_signingAllowed, value);
        else
            FormState[_signingAllowed] = value;
    }
}
/* . . . */
public void FormEvents_Sign(object sender, SignEventArgs e)
{
    if (SigningAllowed)
    {
        Signature thisSignature =
            e.SignedDataBlock.Signatures.CreateSignature();
        thisSignature.Sign();
        e.SignatureWizard = true;
    }
    else
    {
        e.SignatureWizard = false;
        MessageBox.Show("Please use the 'click here' "
            + "button (in the form) to proceed.");
        return;
    }
}
```

If the `SigningAllowed` property is true, we first create a new signature by using the `CreateSignature()` method of the `Signatures` object (which returns a `SignatureCollection`). Calling `Sign` on the newly created `Signature` shows the *Signatures* dialog, which the user can fill out. The code won't continue past `Sign` until the dialog is dismissed. In the case when the user tries to sign the form in an alternate way (such as using the *Digital Signatures* dialog), we show a dialog explaining how our form expects the user to sign the form data. Whether or not the user applied a

816 ■ **Chapter 15: Writing Code in InfoPath**

signature or just closed the dialog is handled by the `Button Clicked` event handler we looked at in Listing 15.30.

Context-Sensitive Help in the Custom Task Pane Another major feature in the MOI request form is the context-sensitive help in the custom task pane. Before we see how the task pane is involved in the code, it's important to learn about the underlying mechanism that causes the task pane to change. The `ContextChanged` event is a form event fired whenever XML context in the current view is changed. The "context" portion of "XML context" is determined by what control in the view has focus. The "XML" part is the specific data source field or group bound to that control. So whenever control focus changes from control A to B, XML context changes if and only if controls A and B are bound to different data source nodes.

To give a concrete example, let's look at the MOI request form. As you click through the `Option Button` controls for the request type, you'll notice that the task pane does not change; it just stays on the "Request Type" help topic. Even though we're clicking on different controls, the data source (`RequestType`) behind those controls is the same. Now try clicking on any other control in the view; as expected, the control in which you clicked is not bound to `RequestType`, and the task pane changes its content.

■ WARNING Context Changed Event Behavior

The `ContextChanged` event is called for all sorts of contexts that you may not expect. For example, clicking in white space in the view (not on a control) results in the document element getting context! We recommend that you try repeating controls as well. The best way to see when the event is called is to put this code into the `Context Changed` event handler: `MessageBox.Show(e.Context.Name);`

Now that we understand how the `ContextChanged` event works, let's apply it to our context-sensitive help system in the task pane. For starters, the task pane content is a single HTML file authored in Word. We used Word's Bookmarks feature to place bookmarks throughout the document. These bookmarks are simply HTML anchor tags with unique names. We're leveraging Internet Explorer's ability to jump to an anchor on an HTML

page by using the # character at the end of the URL, followed by the anchor's name. Thanks to the effort we gave when properly naming the fields and groups in the data source, it's really easy to hook up the `ContextChanged` event handler to the task pane. Assuming that we use the field and group names (such as `RequestType` for the "Request Type" help topic) to bookmark content in the HTML file, the anchor name is given automatically in the `ContextChanged` event! See the implementation of the `Context-Changed` form event in Listing 15.32.

NOTE Custom Task Pane HTML

The HTML file included in the sample includes only simple text. InfoPath does not impose any artificial boundaries in terms of the complexity of the HTML file. A lightweight HTML file was used for the sake of this sample.

TIP Custom Task Pane Navigation and Pane Visibility

We use an optimization to skip navigation if the task pane is not visible. It is not a requirement to have a visible task pane to perform navigation.

LISTING 15.32: Wiring Up the `ContextChanged` Form Event for Context-Sensitive Help in the Task Pane

```
public void FormEvents_ContextChanged(object sender,
ContextChangedEventArgs e)
{
    if (e.ChangeType == "ContextNode")
    {
        HtmlTaskPane htmlTaskPane =
            (HtmlTaskPane) this.TaskPanes[TaskPaneType.Html];
        if (htmlTaskPane.Visible)
            htmlTaskPane.Navigate("MoiConsultingRequest.htm#"
                + e.Context.LocalName);
        return;
    }
}
```

Before we move on to the *Request Details* view that appears after the user clicks the *click here* Button control, let's talk about another data-rich

818 ■ Chapter 15: Writing Code in InfoPath

feature you may have noticed. Selecting an Option Button in the *Request Type* region of the form automatically populates the subarea List Box. This happens via a `Changed` event handler on the `RequestType` field. We chose the `Changed` event for several reasons. First, the data in the `RequestType` field has been confirmed because it made it past the `Changing` and `Validating XML` events. Second, even though we're not doing it, the main data source can be modified. This gives us flexibility in the future if we decide to add functionality that could change other data in the form. In this particular case with the `RequestType` field, we also have a `Changing` event. (We showed this `Changing` event toward the beginning of explaining the MOI request sample form.) In the case where the `Changing` event rejects the user's selection (which is a possibility if he or she chooses the *Time-sensitive/Critical* option), our `Changed` event is never called. Listing 15.33 shows the `Changed` event handler.

■ NOTE Querying Via a Rule Instead of Form Code

In reality, we would have chosen to use a rule on the `RequestType` node. It would set the `requestType` query parameter on the secondary data source (using the *Set a Field's Value* action); then query the Web service connection. We wrote this code to demonstrate form code involving data connections, as well as some other OM properties and methods.

LISTING 15.33: Dynamically Populating the `RequestType` List Box Using a Data Connection

```
public void RequestType_Changed (object sender, XmlEventArgs e)
{
    // Process value changes only for the RequestType field.
    // This code also should not run when filling the OtherType field.
    if (e.UndoRedo
        || e.Operation != XmlOperation.ValueChange
        || e.Site.Name != ((XPathNavigator)sender).Name)
    {
        return;
    }

    // Request Type changed, get an updated list of Sub Areas
    DataSource getSubAreasDS = DataSources["GetSubAreas"];
    // We could have also used DataConnections["GetSubAreas"], below
    WebServiceConnection getSubAreasWSC =
        (WebServiceConnection)getSubAreasDS.QueryConnection;
```

Programming InfoPath . . . in Action! 819

```

// Get the navigator for the GetSubAreas secondary data source
XPathNavigator getSubAreasNav = getSubAreasDS.CreateNavigator();
// Get the requestType param that will be sent to the Web service
XPathNavigator requestTypeNav = getSubAreasNav.SelectSingleNode(
    + "/dfs:myFields/dfs:queryFields/tns:GetSubAreas"
    + "/tns:requestType", NamespaceManager);
// Get the Request Type from the main data source
requestTypeNav.SetValue(e.NewValue);

// Web service timeout is 30s, increase to 60 for slow connections
getSubAreasWSC.Timeout = 60;
// Create a new navigator to capture errors, if any
XmlDocument errorsXmlDoc = new XmlDocument();
XPathNavigator errorsNav = errorsXmlDoc.CreateNavigator();
try
{
    // Query the Web service
    getSubAreasWSC.Execute(
        null /*input*/, null /*output*/, errorsNav /*errors*/);
}
// Silently fail and we'll show the message box in a moment.
// If we didn't do this, InfoPath would show the exception.
catch (Exception) { }

if (errorsNav.HasChildren)
    MessageBox.Show("I'm sorry, an error occurred accessing " +
        getSubAreasWSC.ServiceUrl
        + ". Please select a Request Type again.");
}

```

The Changed event handler for RequestType (Listing 15.33) at first may appear to be a long and complicated method. But most of it merely involves comments or data source operations, which happen to take up more lines of code. Let's parse through this source code to understand what purposes it serves. We left most of the comments in the code to offer a more development-centric viewpoint (since, of course, we added them while writing the code). See the `MoiConsultingRequest` sample code for the entirety of comments.

As we know from studying behaviors of XML events, event bubbling is both our friend and our enemy. It's convenient to use at a higher level in the data source to handle a specific event on a variety of nodes. However, in a case like ours where we want to query via a Web service connection, it is best to restrict this code to running only when necessary. Querying an external data source is a relatively slow ordeal and should be minimized when possible. To adhere to this best practice, we are selective about whether or

820 ■ **Chapter 15: Writing Code in InfoPath**

not our event handler continues processing. Three conditions must hold true for the event to continue with querying the Web service.

1. The user must not have incurred an undo or redo.
2. The operation must be a change in value.
3. The site (where we're listening) and sender (what actually changed) nodes must be the same.

As the comments in the code suggest, the last two conditions are very important in our effort to filter irrelevant event notifications. If the second condition ceased to exist, our code would query the Web service when the user clicks the *start over* Button control. This is because the `RequestType` node would be deleted and then inserted. Surely we wouldn't want to query because the `GetSubAreas` secondary data source will be cleared anyway in the "start over" handler. If the last condition were not in place, our code would still run because of event bubbling. This happens when the user selects the *Other* Option Button control in the *Request Type* region and thus changes the `OtherType` attribute field (bound to the associated Text Box).

In the cases where the user successfully changes the `RequestType` node by clicking on one of the Option Button controls, we want to proceed and query the Web service. The Web service method for getting the list of subareas (called `GetSubAreas`) accepts a string value that matches a `RequestType` enumeration. This enumeration is defined in the Web service itself. If a nonenumerated value is used, the query will fail. The Web service implementation is succinct for the purpose of this sample; it maps the Option Button values to an array of hard-coded strings.

■ NOTE **GetSubAreas Web Service Sample**

You can find the Web service code, `GetSubAreas.cs`, with the samples for this chapter.

Before we do the query, however, we have to take care of a few prequery setup steps. As we know from Chapter 7, secondary query data connections use corresponding secondary data sources to get data to and from the

external source. In this case, the secondary data source (called `GetSubAreas`, which corresponds with its connection counterpart with the same name) is used to populate the List Box bound to the `SubAreas` secondary data source. When we query `GetSubAreas`, the content of `queryFields` is sent to the Web service. The result of the query is returned in the `dataFields` group. If you look in the `queryFields` group (Figure 15.28) of the secondary data source in the *Data Source* task pane, you will see the `requestType` field, which is sent as the parameter to the Web service. `RequestType` node values (such as `AccountQuestion`, `ConsultantRequest`, and so on) are mimicked in the Web service enumeration. As a result, we simply need to copy the value of the `RequestType` node from the main data source to the `GetSubAreas` `requestType` parameter.

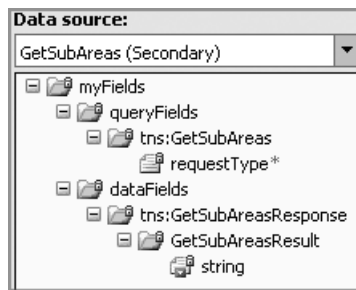


FIGURE 15.28: `GetSubAreas` secondary data source

WARNING No Validation in Secondary Data Sources

`GetSubAreas` is a secondary data source, so there is no data validation. As a result, neither the requirement that the field cannot be blank nor the enumerated data type of the `requestType` field is enforced.

To set the value of the `requestType` parameter, we need to get an `XPathNavigator` to the `GetSubAreas` secondary data source. We use the `DataSources` collection to get the secondary data source by name, and similar to the main data source, we call `CreateNavigator`. Selecting the `requestType` field and setting its value is no different than working with the main data source. If you have trouble finding the correct XPath, we suggest reviewing Chapter 3. We also discuss how to find XPaths in the

822 ■ **Chapter 15: Writing Code in InfoPath**

“Using `XPathNavigator` Objects” section earlier in this chapter. The relevant code corresponds to the code starting where we use a `DataSource` object to the line of code where we set the value of the request type. This code is in the `RequestType_Changed` method shown in Listing 15.33.

■ TIP DataSources (and DataConnections) Collections

The main data source is included in the `DataSources` collection as the empty string (`String.Empty`). Likewise, a main data connection is also included in the `DataConnections` collection. This is important to remember if you’re iterating through these collections or using the `Count` property.

Subjacent to the secondary data source lines of code are the lines for getting the corresponding data connection. (See where we get the `WebServiceConnection` object within the `RequestType_Changed` method.) The easiest way to get the connection is to get it from the `DataSource` object that we used to create the `XPathNavigator`. Remember that every secondary data source, by definition, must have an associated query data connection, so the `QueryConnection` property should always be available. Alternatively, a reference to the Web service connection could be attained through the `DataConnections` collection. This happens in the same fashion in which the secondary data source was found in the `DataSources` collection. Getting a data connection from the collection returns a `DataSource` object that exposes only two useful items: the `Name` property and the `Execute` method. Since we know it’s a Web service connection and we need to access some Web service–specific properties and methods, we need to cast the generic `DataSource` to a `WebServiceConnection`.

The InfoPath OM defines a specific data connection object for each connection type, as shown in Table 15.5. Some commonly used properties and methods are also listed in this table.

■ TIP Execute Override Behavior

Passing `null` for an `XPathNavigator` parameter in an `Execute` override method tells InfoPath to use the default `XPathNavigator` that it would have used without the override.

TABLE 15.5: Types of Data Connections and Their Commonly Used Properties and Methods

Connection Type	Commonly Used OM	Comments
AdoQueryConnection AdoSubmitConnection	Command Connection Timeout	Command can change the SQL query statement; Connection modifies the connection string.
EmailSubmitConnection	AttachmentFileName Introduction Subject Execute (XPathNavigator input)	The Execute override method accepts any data source, including main or secondary, to attach to the mail message.
FileQueryConnection	FileLocation	Receive data from XML document.
FileSubmitConnection	Filename FolderUrl	Submit data to a SharePoint document library. Sets the name of the file and folder location when using SharePoint DAV submit.
SharePointListQueryConnection	SiteUrl Execute (XPathNavigator output)	The Execute override method returns the queried data to the output XPathNavigator provided by form code.
WebServiceConnection	ServiceUrl SoapAction Timeout Execute (XPathNavigator input, XPathNavigator output, XPathNavigator errors)	The Execute override method allows form code to specify optional XPathNavigator objects for providing input when sending data, getting output when receiving, or retrieving SOAP fault error output. The Execute override behaves similarly for both query and submit connections.

824 ■ **Chapter 15: Writing Code in InfoPath**

Now that we've set the `requestType` query parameter and have the Web service connection object, it's almost time to perform the query. However, we have noticed that the server hosting the Web service can sometimes get bogged down from many requests. Sometimes the server can take some time to respond. InfoPath's default 30-second timeout on data connections may not be enough time. Once the timeout is changed, it will be persisted for the life of that connection. So we could have increased the timeout in the `Loading` event instead of immediately before querying the connection. Listing 15.34 shows how to set the `Timeout` property.

LISTING 15.34: Setting the Web Service Connection Timeout Property

```
// Web service timeout is 30s, increase it to 60 for slow connections  
getSubAreasWSC.Timeout = 60;
```

The Web service server has turned out to be quite the unreliable machine. Even with the extended timeout, we see occasional spurious errors when performing queries. InfoPath handles a data connection error by showing a dialog with server details that may not make sense to the user. To make the error case a little less confusing for our users, we want to detect errors when querying the Web service and handle the errors in our form code. We use a `try-catch` block around the Web service connection's `Execute` override method (see Table 15.5) because an exception is thrown when an error occurs. You could put any error-handling logic in the `catch` block; however, we've decided to just check the resulting errors navigator that will be nonempty when an error happens on the server. Listing 15.35 shows the code.

■ TIP Handling Web Service Connection Failures

When a Web service (or other data connection) fails to execute, the resulting server-side error message may reveal implementation details about your external data source. Using a `try-catch` and handling the error in your form code allows you to choose what error messages the user can see.

LISTING 15.35: Capturing Errors from Executing a Web Service Data Connection

```

// Create a new navigator to capture errors, if any
XmlDocument errorsXmlDoc = new XmlDocument();
XPathNavigator errorsNav = errorsXmlDoc.CreateNavigator();
try
{
    // Query the Web service
    getSubAreasWSC.Execute(
        null /*input*/, null /*output*/, errorsNav /*errors*/);
}
catch (Exception)
{
    // Silently fail for now
    // If we didn't do this, InfoPath would show the exception
}
// Did an error occur?
if (errorsNav.HasChildren)
{
    MessageBox.Show("I'm sorry, an error occurred accessing "
        + getSubAreasWSC.ServiceUrl
        + ". Please select a Request Type again.");
}

```

■ TIP Benefits of Using `Execute Overrides`

A try-catch could have been used with the no-argument `Execute()` (instead of the override version) and would have achieved the same effect as in Listing 15.35. The advantages of using the override for getting errors is to perform your own logging or send an administrative alert to track how many errors your users encounter with this particular data connection.

The *Request Details* view, shown in Figure 15.29, gives the user an opportunity to enter specifics of the request.

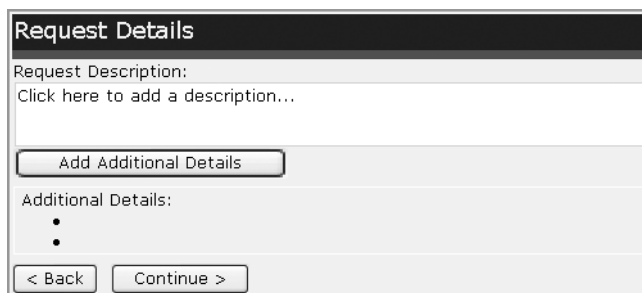


FIGURE 15.29: Request Details view

826 ■ Chapter 15: Writing Code in InfoPath

Did you notice that the task pane disappeared? We want our help task pane to be visible only on the *Welcome* view. Instead of adding logic behind Button clicks that change views, it's much easier to just get notified when the view is actually changed. To do so, we can sink and handle the `ViewSwitched` event. We can also use this event handler (shown in Listing 15.36) for other purposes, such as showing a dialog message on the *Confirm* view.

■ **WARNING** Hiding Does Not Disable the Task Pane

Hiding the task pane by using the `Visible` property does not block the user from deciding to show it again.

LISTING 15.36: Handling the `ViewSwitched` Form Event When Switching to Specific Views

```
public void FormEvents_ViewSwitched(
    object sender, ViewSwitchedEventArgs e)
{
    switch(this.CurrentView.ViewInfo.Name)
    {
        case "Confirm":
            MessageBox.Show("Please confirm the data and click "
                + "Submit to continue.", "Confirm Data");
            break;
        case "Welcome":
            Application.ActiveWindow.TaskPanes[
                TaskPaneType.Html].Visible = true;
            break;
        default:
            Application.ActiveWindow.TaskPanes[
                TaskPaneType.Html].Visible = false;
            break;
    }
}
```

■ **NOTE** Restricted OM Available During the `SwitchView` Form Event

Some OM cannot be called during the `ViewSwitched` event, such as `SwitchView`, of course.

The `RequestDescription` field is a Rich Text Box control, which allows unrestricted HTML input. We did not add any special logic behind this field. However, the *Add Additional Details* Button control offers a list-item approach for more structured request details. By default, the `AdditionalDetails` Bulleted List control does not exist. As we saw earlier when filling out this form, clicking the Button control inserts an Optional Section that contains the Bulleted List. Subsequent clicks on the same Button control insert another item in the Bulleted List. Keep in mind that these are two different controls bound to separate data source items.

The logic behind the *Add Additional Details* Button control is as follows: If the Optional Section (bound to `AdditionalDetails`) exists, then insert it; otherwise, insert another item at the end of the Bulleted List (bound to `AdditionalDetail`). We use two different approaches to facilitate the insertion of these controls. Since the Optional Section is a container control, it can benefit from view-based structural editing operations. Other structurally enabled controls include (but are not limited to) Repeating Table, Repeating Section, File Attachment, and Choice Group and Repeating Choice Group controls. View-based structural editing through the OM uses the `ExecuteAction` method of the `CurrentView` object. The nicety in using `ExecuteAction` over pure data source operations (as we'll use in a moment to insert an item in the Bulleted List) is not needing to concern ourselves (or our code for that matter) with inserting an entire XML subtree in the correct context. For example, if the Optional Section contained many nodes below it, either immediately below or through many depths, the `ExecuteAction` method will take care of inserting all necessary fields and groups. In fact, the `ExecuteAction` method is exactly what InfoPath uses behind the scenes when a user clicks the *Click here to insert* link. Of course, the link doesn't need to exist in the view for form code to successfully call it.

The first argument to `ExecuteAction` is an `ActionType`. This is an enumeration type, and through `IntelliSense`, you can find all of its enumerated values. In the case of an Optional Section, we use the `XOptionalInsert` `ActionType`. The second parameter is the `xmlToEdit` string. This value identifies on which control to perform the `ActionType`. To find the `xmlToEdit` string for a structurally editable control, go to the *Advanced* tab of the control's properties dialog. The *Code* region, shown in Figure 15.30, reveals the `XmlToEdit` value as well as the control's `xOptional` capability.

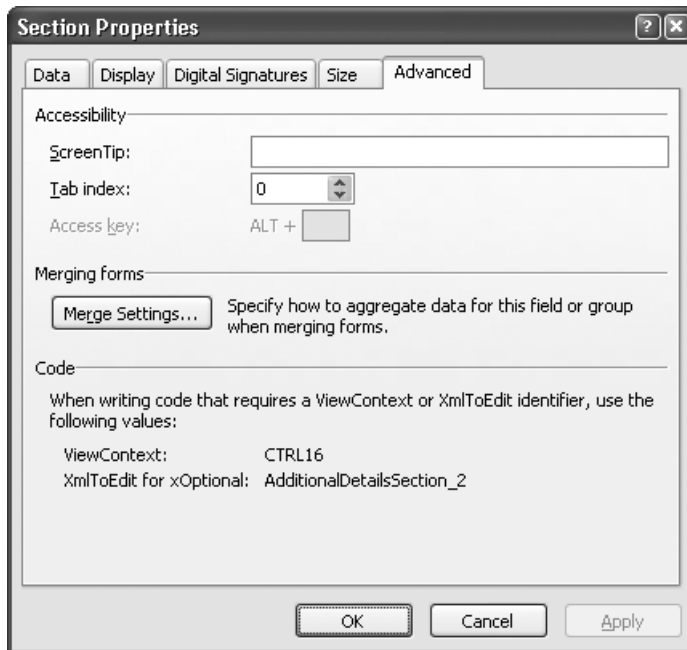


FIGURE 15.30: Finding the `xmlToEdit` and `ActionType` capability of a structurally editable control

The code snippet in Listing 15.37 shows how we use `ExecuteAction()` to insert the `Optional` Section if it does not already exist in the data source.

LISTING 15.37: Using `ExecuteAction` to Insert an `Optional` Section

```
public void Add_Additional_Details_Clicked(
    object sender, ClickedEventArgs e)
{
    // Is AdditionalDetails already inserted?
    XPathNavigator root = MainDataSource.CreateNavigator();
    XPathNavigator additionalDetails = root.SelectSingleNode(
        "/my:Request/my:RequestDetails/my:AdditionalDetails",
        NamespaceManager);

    // If it doesn't exist, we'll insert it
    if (additionalDetails == null)
        // XmlToEdit for xOptional: AdditionalDetailsSection_2
        this.CurrentView.ExecuteAction(
            ActionType.XOptionalInsert, "AdditionalDetailsSection_2");
    // If it already exists, insert an AdditionalDetail
    else
        // Bulleted List doesn't support ExecuteAction since
        // there's no XmlToEdit.
```

```
// We'll insert it using the data source instead.
additionalDetails.AppendChildElement(
    additionalDetails.Prefix, "AdditionalDetail",
    additionalDetails.NamespaceURI, string.Empty);
}
```

If the `Optional Section` control, bound to `AdditionalDetails`, already exists, the `Button` control will insert an `AdditionalDetail` node. Adding this node to the data source will, in turn, add an item to the `Bulleted List` control. Since the `AdditionalDetail` repeating field is a child of the `AdditionalDetails` group and we already have a reference to `AdditionalDetails`, we can easily use the `XPathNavigator` method `AppendChildElement`. The `AppendChildElement` method creates a new data source node and adds it to the end of children nodes relative to the context (`AdditionalDetails`) `XPathNavigator`. If you wanted to assign a value to the newly inserted item, you could pass it instead of the empty string as the last argument to `AppendChildElement`.

TIP Using the `XPathNavigator`'s `AppendChildElement` Method

`AppendChildElement` requires the namespace prefix and URI as parameters. Since the node in which we're appending has the same prefix and URI, we can just reference their values. This lessens the chance of mistyping and is more robust if, say, the namespace or prefix change.

After the user completes the *Request Details* view, the read-only *Confirm* view (refer back to Figure 15.23) displays a summary of gathered form data. Of the three `Button` controls at the bottom of the view (*Start Over*, *Go Back*, and *Submit*), we haven't yet discussed *Submit*. As its name implies, clicking this control submits the form. You learned all about submitting forms in Chapter 8, but the MOI request form does not implement a traditional-style submit to a data connection. Instead, via the *Submit Options* dialog shown in Figure 15.31, we have chosen to submit using our own code. Clicking on the *Edit Code* button in this dialog opens VSTA, hooks up the event in `InternalStartup`, and creates the event handler method called `FormEvents_Submit`.

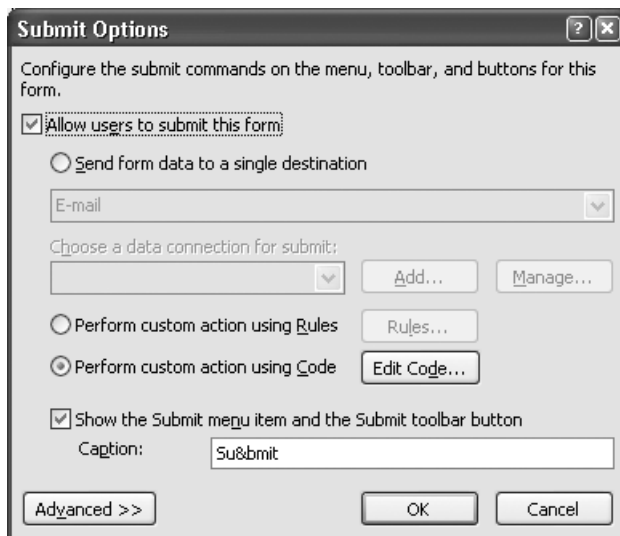


FIGURE 15.31: Setting up the form's main submit to use code

The MOI request form defines the `Submit` event as saving the form to the user's computer. Listing 15.38 shows the implementation for `Submit`. Before doing the save, however, we want to check whether the current form was recovered. We can detect a recovered form by checking the `this.Recovered` Boolean flag. A form's data is recovered if InfoPath suddenly and unexpectedly closed while the form was being filled out. This could happen, for example, if the computer turned off due to a power outage. Because InfoPath quit unexpectedly, the form could be in an indeterminate state. We'd rather be safe and ask the user if he or she wants to continue submitting recovered data. The user may choose to go back and finish filling out parts of the form that weren't completed.

LISTING 15.38: Sinking the `Submit` Form Event

```
public void FormEvents_Submit(object sender, SubmitEventArgs e)
{
    // NOTE: There are no errors if InfoPath allows submit to happen.
    // If this is recovered data, ask the user if we should continue.
    DialogResult result = DialogResult.Yes;
    if (this.Recovered)
        result = MessageBox.Show("This data is recovered. "
            + "Still continue?", "Submitting Recovered Data",
            MessageBoxButtons.YesNo);
    if (result == DialogResult.Yes)
```

```
{
    // Save the form
    string tempFileName = System.IO.Path.GetTempFileName()
        + ".xml";
    this.SaveAs(tempFileName);
    // Fills a link to see where it saved
    XPathNavigator root = MainDataSource.CreateNavigator();
    XPathNavigator savedFileLocation = root.SelectSingleNode(
        "/my:Request/@my:SavedFileLocation", NamespaceManager);
    savedFileLocation.SetValue(tempFileName);
    // Go to the Thank You view
    ViewInfos.SwitchView("ThankYou");
}
else
{
    e.CancelableArgs.Cancel = true;
    e.CancelableArgs.Message = "Submit was canceled.";
    e.CancelableArgs.MessageDetails =
        "Submit again whenever you are ready.";
}
}
```

TIP Accounting for Performance in Form Code

When a single event handler changes both the data source and switches the view, the view switch should be the last operation. Switching the view first is more expensive if the controls bound to the changing data are only visible in the new view.

When submit proceeds successfully, a file is created on the local computer with the .xml extension. The form data is saved by calling `SaveAs` and passing the full path with file name. After saving the file, the user goes to the final *Thank You* view. A hyperlink in this view (bound to `SavedFileLocation`) shows the actual location of the saved form. If calling `SaveAs` failed, say, because the disk is full or read-only, the `Submit` event would also fail. A `try-catch` block could be used around `SaveAs` if you wanted to handle the failure in a special way, such as trying again or asking for an alternate save location.

TIP Save Requires Full Trust

Calling `Save` or `SaveAs` requires a fully trusted form template.

What's Next?

Now that you've been introduced to the InfoPath OM and learned about writing code behind forms, we are positioned to move forward with more advanced programming features. The OM knowledge you've garnered thus far will be used and further developed in the coming chapters. Areas such as hosting, leveraging browser-enabled form templates with Forms Services, and building custom controls using ActiveX all incorporate various aspects of the OM. In the meantime, we encourage you to have fun programming your own advanced InfoPath form templates.