"Office development using managed code has hit new strides with Visual Studio 2008, and personally, I can't wait to take advantage of the answers I find in this book to build great applications."

—From the Foreword by Ken Getz, senior consultant, MCW Technologies

Visual Studio Tools for Office 2007

VSTO for Excel, Word, and Outlook

Microsoft .NET DEVELOPMENT SERIES

Eric Carter
Eric Lippert

Library of Congress Cataloging-in-Publication Data

Carter, Eric.
Visual Studio tools for Office 2007 : VSTO for Excel, Word, and Outlook / Eric Carter, Eric Lippert. — 2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-321-53321-0 (pbk. : alk. paper)
1. Microsoft Visual BASIC. 2. BASIC (Computer program language) 3. Microsoft Visual studio. 4. Microsoft Office. I. Lippert, Eric. II. Title.

QA76.73.B3C3452 2009
005.13'3—dc22 2009000638

ISBN-13: 978-0-321-53321-0
ISBN-10: 0-321-53321-6
Text printed in the United States on recycled paper at Courier in Stoughton, Massachusetts.
First printing, February 2009

Foreword

I FACE THE CHALLENGE of composing a foreword to this particular book with some amount of trepidation. Let's face it: The names on the cover of this book inspire some amount of awe. It is humbling to know that one's words will introduce what one believes is to be the seminal work on a given topic, and believe me, I'm relatively sure that this one will meet those lofty goals. When approached with the invitation to grace the front matter of the book, my first response was to wonder what I could possibly add; couldn't they find some luminary at Microsoft to preface the book? It seems, however, that an outside voice adds some credence to the proceedings, so, dear reader, I speak meekly in the presence of greatness.

First, a little about me (it's the last chance I'm going to get in this short piece): I've been lurking about, programming Office in its various guises, for upward of 10 years. I've written a lot about the wonders, and gotchas, of Office development, and survived the glory years surrounding Office 2000, when it looked like Office might finally make a successful integrated development platform. Around 2001, it became clear that no matter how hard I and like-minded folks wanted Office to become a respected development standard, it just wasn't going to make it with VBA as the programming language.

With the release of Visual Studio Tools for Office 2003, it finally looked like we had made some progress. No longer relegated to the 1990s, Office developers could embrace .NET and all its goodness, taking advantage of managed code, code-access security, xcopy deployment, and all the rest

that .NET supplied. I loved this product, but it never really reached critical mass with the developer community. Most likely, the fact that you could only use COM-based controls on documents, and the fact that the product supplied no design-time experience at all, made it a slow starter.

Around that time, I remember very clearly sitting down at some Microsoft event and meeting Eric Carter. I didn't really know who he was at the time (and he certainly didn't know anything about me), but he seemed nice enough, and we chatted for several hours about Office development in general and about VSTO in specific. Only later did I learn that he was high up in the development side of the product. (I spent hours worrying that I had said something really stupid while we were chatting. Hope not.) We began a long correspondence, in which I've more often than not made it clear that I've got a lot to learn about how .NET and Office interact. I've spent many hours learning from Eric's blog, and Eric Lippert's blog is just as meaty. If you are spending time doing Office development, make sure you drop by both:

http://blogs.msdn.com/ericlippert/
http://blogs.msdn.com/eric_carter/

I spent some measurable hours perusing the draft copy of this book and in each chapter attempted to find some trick, some little nugget, that I had figured out on my own that didn't appear in the book. I figured that if I was going to review the book, I should add something. The result: I was simply unable to find anything missing. Oh, I'm sure you'll find some little tidbit that you've figured out that won't appear here, but in my quick pass, I wasn't able to. I thought for sure I would catch them on something. Alas, I failed. And, I suppose, that's a good thing, right? Every time I thought I had them in a missing trick, there it was, right there in print. What that means is that you'll have the best possible reference book at your fingertips. Of course, you need to get your expectations set correctly; it's simply not possible, even in a 60-page chapter, to describe the entirety of the Excel or Word object model. But E&E have done an excellent job of pointing out the bits that make the biggest impact on .NET development.

If you're reading this foreword before purchasing the book, just do it. Buy the thing. If you've already bought it, why are you reading this? Get to the heart of the matter—skip ahead, and get going. You can always read this stuff later. There's a considerable hill ahead of you, and it's worth the climb. Office development using managed code has hit new strides with the release of Visual Studio 2008, and personally, I can't wait to take advantage of the answers I find in this book to build great applications.

—Ken Getz, senior consultant for MCW Technologies

Preface

IN 2002 THE first release of Visual Studio .NET and the .NET Framework was nearing completion. A few of us at Microsoft realized that Office programming was going to miss the .NET wave unless we did something about it.

What had come before was Visual Basic for Applications (VBA), a simple development environment integrated into all the Office applications. Each Office application had a rich object model that was accessed via a technology known as COM. Millions of developers identified themselves as "Office developers" and used VBA and the Office COM object models to do everything from automating repetitive tasks to creating complete business solutions that leveraged the rich features and user interface of Office. These developers realized that their users were spending their days in Office. By building solutions that ran inside Office, they not only made their users happy, but also were able to create solutions that did more and cost less by reusing functionality already available in the Office applications.

Unfortunately, because of some limitations of VBA, Office programming was starting to get a bad rap. Solutions developed in VBA by small workgroups or individuals would gain momentum, and a professional developer would have to take them over and start supporting them. To a professional developer, the VBA environment felt simple and limited, and of course, it enforced a single language: Visual Basic. VBA embedded code in every customized document, which made it hard to fix bugs and update solutions because a bug would get replicated in documents across the

enterprise. Security weaknesses in the VBA model led to a rash of worms and macro viruses that made enterprises turn VBA off.

Visual Studio .NET and the .NET Framework provided a way to address all these problems. A huge opportunity existed not only to combine the richness of the new .NET Framework and developer tools with the powerful platform that Office has always provided for developers, but also to solve the problems that were plaguing VBA. The result of this realization was Visual Studio Tools for Office (VSTO).

The first version of VSTO was simple, but it accomplished the key goal of letting professional developers use the full power of Visual Studio .NET and the .NET Framework to put code behind Excel 2003 and Word 2003 documents and templates. It let professional developers develop Office solutions in Visual Basic and C#. It solved the problem of embedded code by linking a document to a .NET assembly instead of embedding it in the document. It also introduced a new security model that used .NET code-access security to prevent worms and macro viruses.

The second version of VSTO, known as VSTO 2005, was even more ambitious. It brought with it functionality never available to the Office developer before, such as data binding and data/view separation, design-time views of Excel and Word documents inside Visual Studio, rich support for Windows Forms controls in the document, the ability to create custom Office task panes, server-side programming support against Office—and that's just scratching the surface. Although the primary target of VSTO is the professional developer, that does not mean that building an Office solution with VSTO is rocket science. VSTO makes it possible to create very rich applications with just a few lines of code.

The third version of VSTO, which this book focuses on, shipped as a core feature of Visual Studio 2008. It is sometimes said that it takes Microsoft three versions to get something right, and we truly feel that this version of VSTO has the most amazing support for Office programming that Microsoft has ever built. In VSTO, you can now build add-ins for all the major Office applications; you can build application-level custom task panes; you can customize the new Office Ribbon; you can modify Outlook's UI using Forms Regions, and you can easily deploy everything you

build using ClickOnce. The Office 2007 applications themselves are more extensible and provide many new programmability features.

If you've been reluctant to use VSTO because of the issues in previous versions—such as the difficulty of deployment, the nonsupport of VSTO in the Visual Studio Professional SKU, and the limited support for add-ins—we're happy to tell you that these issues have been fixed in the third version of VSTO.

This book tries to put in one place all the information you need to succeed using VSTO to program against Word 2007, Excel 2007, and Outlook 2007. It introduces the Office 2007 object models and covers the most commonly used objects in those object models. In addition, this book helps you avoid some pitfalls that result from the COM origins of the Office object models. This book also provides necessary backround for developers using VSTO to customize Visio 2007, Publisher 2007, PowerPoint 2007, and InfoPath 2007. Although it doesn't specifically focus on these applications, it teaches how to use the VSTO add-in model, how to create custom task panes and ribbons, and how to code against Office object models using C#.

This book also provides an insider view of all the rich features of VSTO. We participated in the design and implementation of many of these features; therefore, we can speak from the unique perspective of living and breathing VSTO for the past six years. Programming Office using VSTO is powerful and fun. We hope you enjoy using VSTO as much as we enjoyed writing about it and creating it.

—Eric Carter
—Eric Lippert
January 2009

16
Working with Outlook Form Regions

Introduction to Form Regions

In Outlook 2007, developers have the ability to extend the Outlook UI by creating a special kind of Outlook extension called an Outlook form region. *Form regions* are used primarily to customize Inspector windows, which we introduced in Chapter 10, "Working with Outlook Events." *Inspector windows* are the Outlook windows that appear when you double-click an Outlook item—a mail item in your inbox or a task in a task list, for example. With form regions you can do things like add pages to the Inspector window, replace all the existing pages in an Inspector window with your own page, or dock some custom UI onto an existing page. You can also use a certain type of Outlook form region (an Adjoining form region) to customize the reading pane in Outlook Explorer windows.

Creating a New Form Region

To begin our exploration of Outlook form regions, let's create a simple one by using Visual Studio 2008. Start by creating a new Outlook add-in project by choosing File > New > Project. In the New Project dialog box that appears, create a new Outlook 2007 add-in, as shown in Figure 16-1.

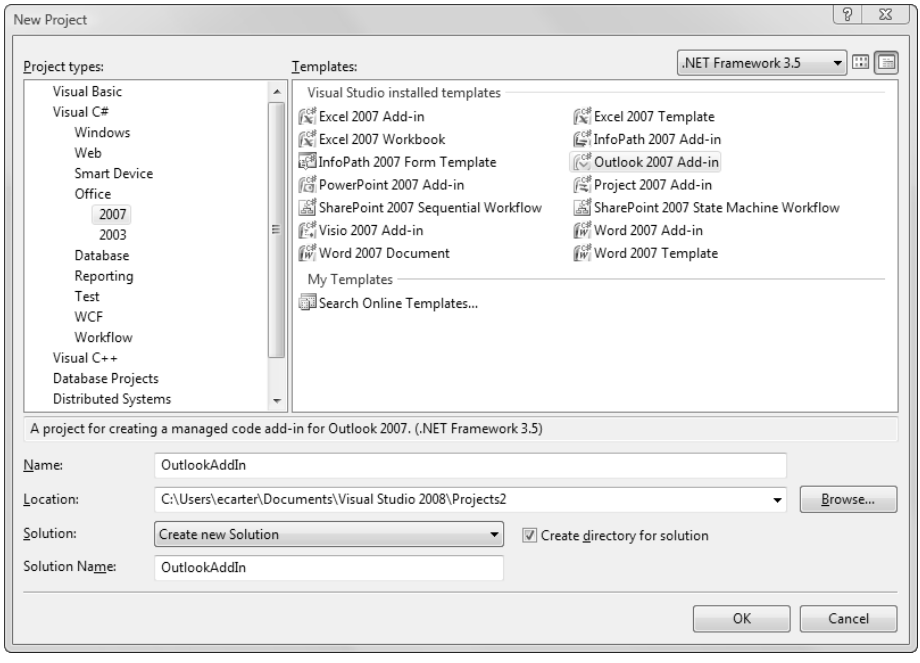

Figure 16-1: Creating a new Outlook 2007 add-in.

Now, in your new add-in project, choose Project > Add New Item. Click the Office category to filter to show just the Office-specific items. In the list of Office items, click Outlook Form Region, as shown in Figure 16-2. Name the form region—just use the default name FormRegion1. Then click the Add button.

A wizard appears, as shown in Figure 16-3. The first step in the wizard is to decide whether you want to create an Outlook form region or import a form region that was previously designed in Outlook with Outlook's built-in form designer. For this introduction, click Design a New Form Region. This option lets you use Windows Forms and keeps our editing experience within Visual Studio. Later in the chapter we show you how to use the Outlook built-in form designer, as well as discuss when you might want to use Outlook's form designer instead of Windows Forms.

After you decide whether to design a new form region with Windows Forms or to import an existing Outlook form region designed in Outlook, click the Next button to move to the second page of the wizard, shown in Figure 16-4, which allows you to pick the type of form region you want to create.

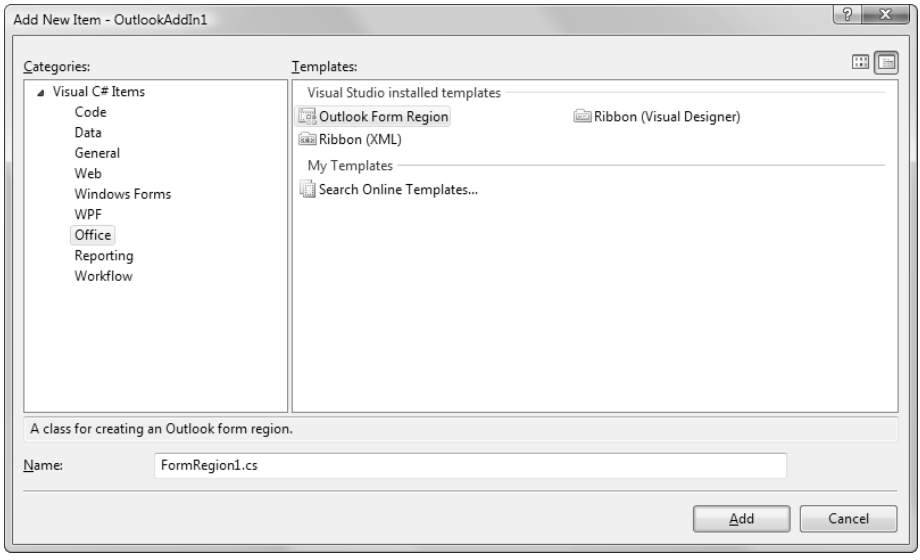

Figure 16-2: Adding an Outlook form region to an Outlook 2007 add-in project.

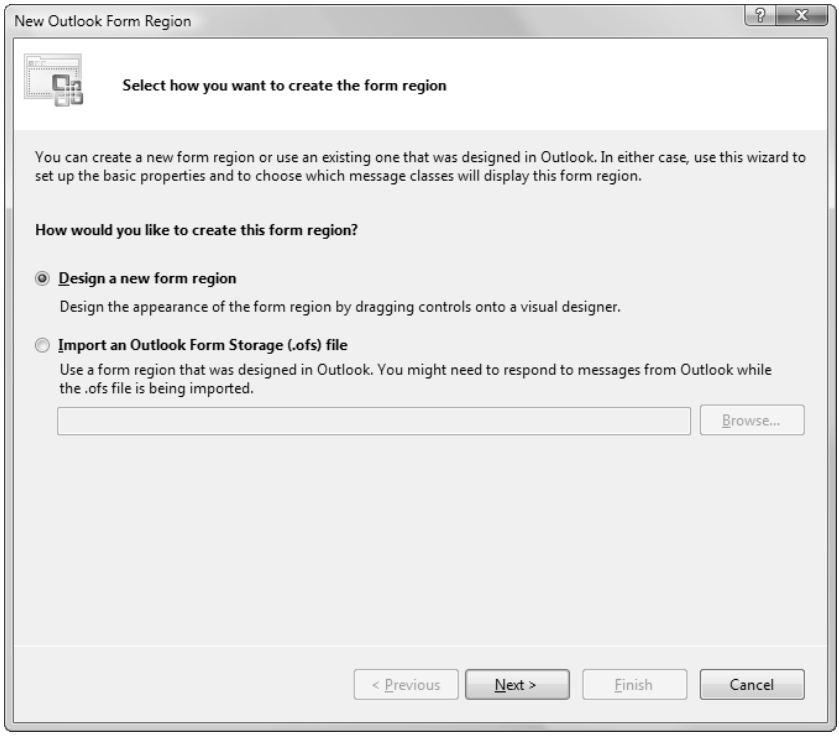

Figure 16-3: Selecting the form technology to use to create the form region.

To understand the types of form regions that are available in Figure 16-4, we must take a step back and discuss Inspector windows in some additional detail. Form regions are used primarily in Outlook Inspector windows. An Outlook Inspector window can have multiple pages associated with it, and Ribbon buttons are used to switch between the pages associated with a particular Inspector window. Consider the Inspector window that appears when you double-click an Outlook task, as shown in Figure 16-5.

Figure 16-5 has two Ribbon buttons in the Show group: Task and Details. In Figure 16-5 the Task button is selected and the Task page is displayed. The Task page is the default page for the Task Inspector window and is displayed first whenever a task is opened. If you click the Details button, the view changes to the Details page, as shown in Figure 16-6.

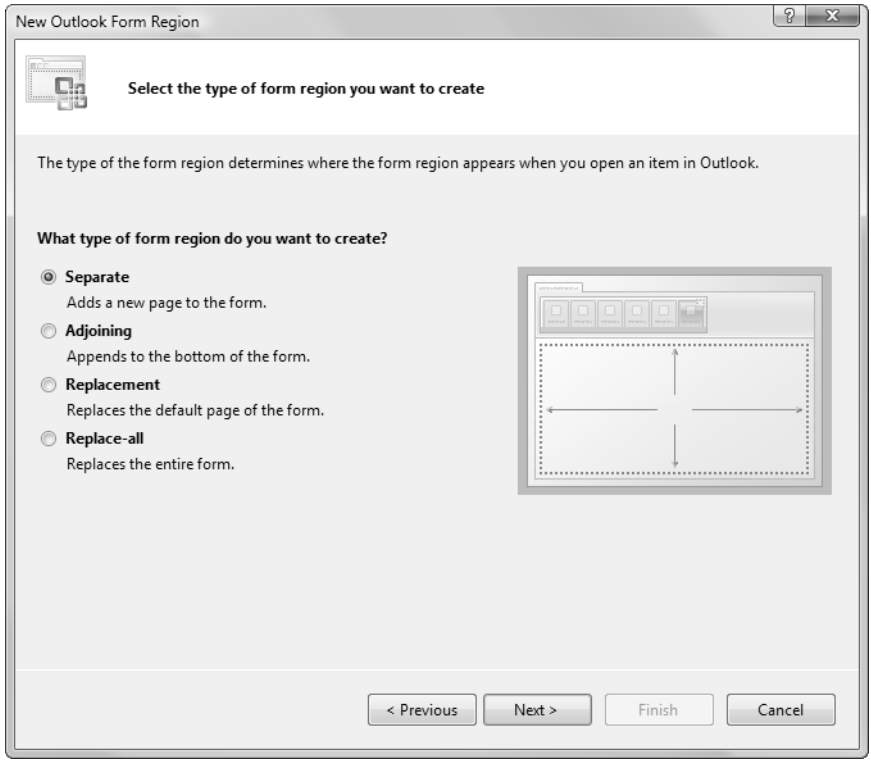

Figure 16-4: Selecting the type of form region to create: Separate.

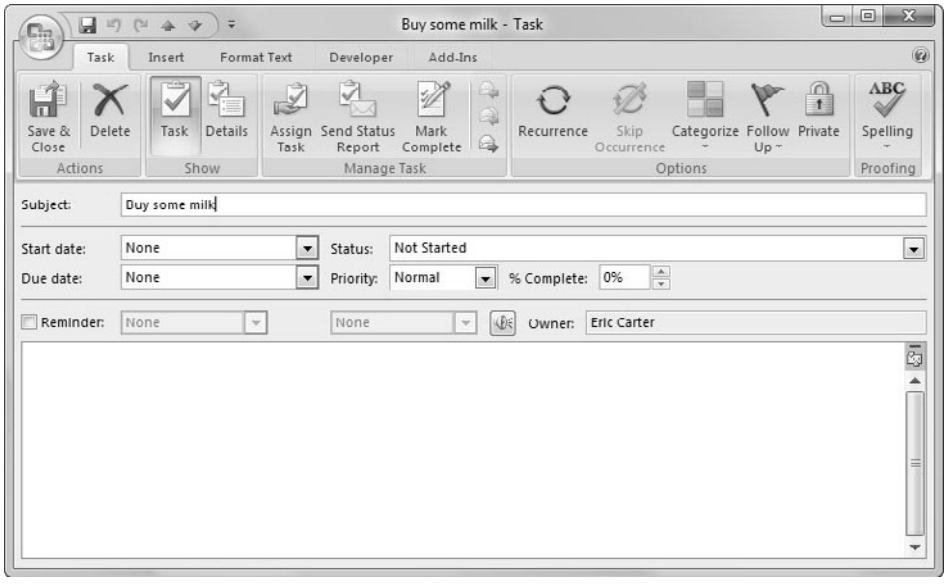

Figure 16-5: A task Inspector window with the Task page selected.

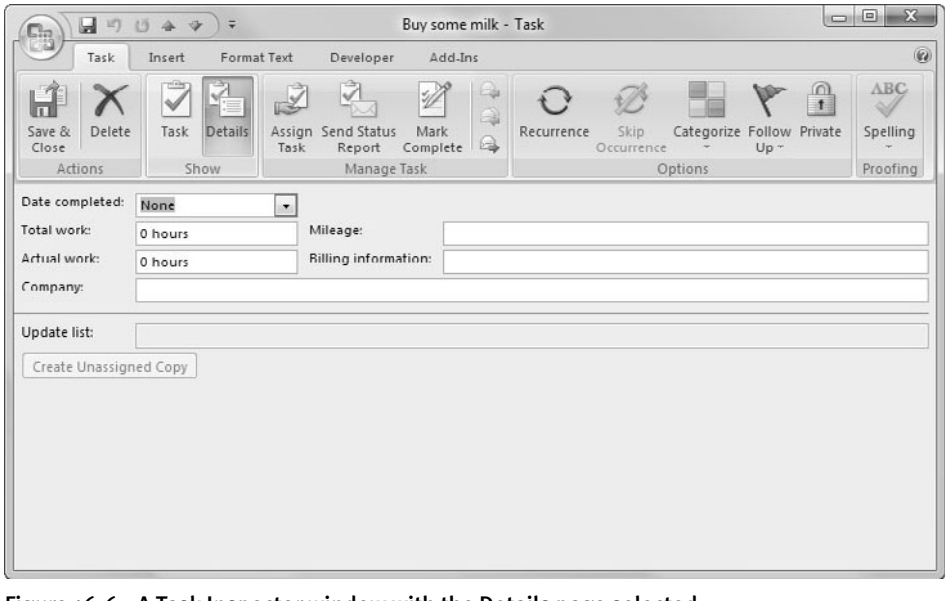

Figure 16-6: A Task Inspector window with the Details page selected.

With this background, you're ready to go back to Figure 16-4 and make sense of the options. A Separate form region adds a new page (and a new Ribbon button to activate that page) to an Inspector window. So you could add a new page to the Task Inspector window to show something like subtasks that must be completed to finish the main task. In Figure 16-4 the wizard also displays a nice graphic to help you remember what a Separate form region is. In this case the graphic emphasizes that you get a new Ribbon button to display the new page, and you have complete control of the new page that is shown.

Figure 16-7 shows what the wizard displays when you select Replacement instead of Separate as the type of form region. A Replacement form region allows you to replace the default page of the Inspector window. So in the task example, you could replace the Task page (the default page for a Task Inspector window), but the Details page would still be available.

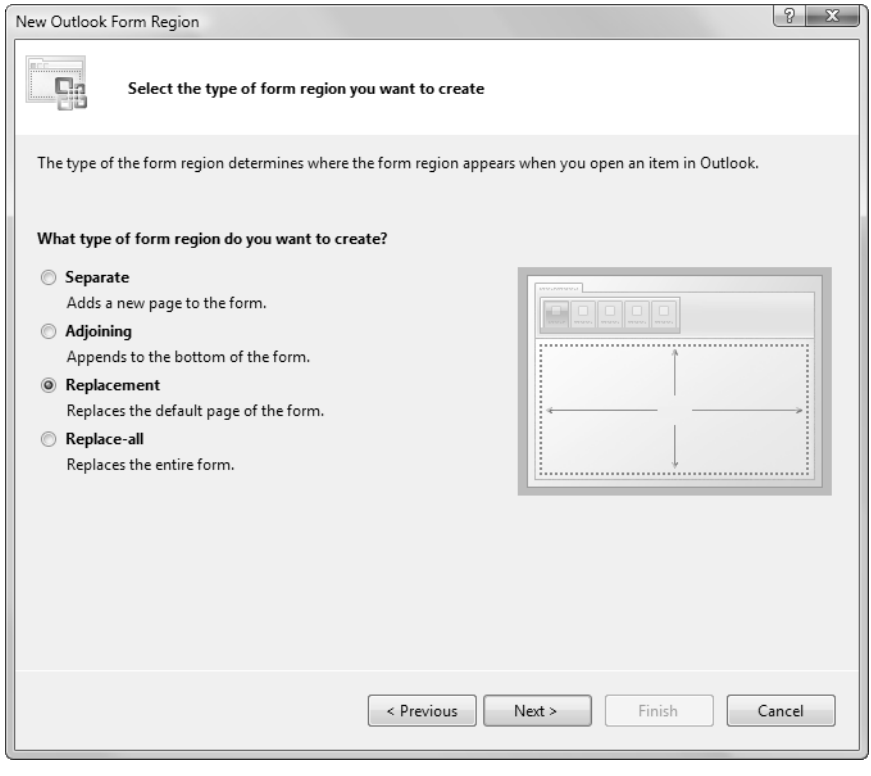

Figure 16-7: Selecting the type of form region to create: Replacement.

Figure 16-8 shows what the wizard displays when you select Replace-All as the type of form region. A Replace-All form region allows you to replace all available pages and make your page available only in the Inspector window. So in the task example, you could replace both the Task page and the Details page; your page would be the only page displayed in the Inspector window.

When you think about Replacement and Replace-All form region types, you realize that replacing the default pages for an Outlook item type is a pretty powerful capability—actually too powerful, in a way, because you could change the default page for an Outlook item type, such as a task, and implement a new default page that prevents the user from editing key data associated with that task. You may forget to provide a way to set the priority of a task in your Replacement or Replace-All form region, for example. Indeed, the creators of Outlook didn't want to give you quite that much power, enough to possibly break key functionality of Outlook.

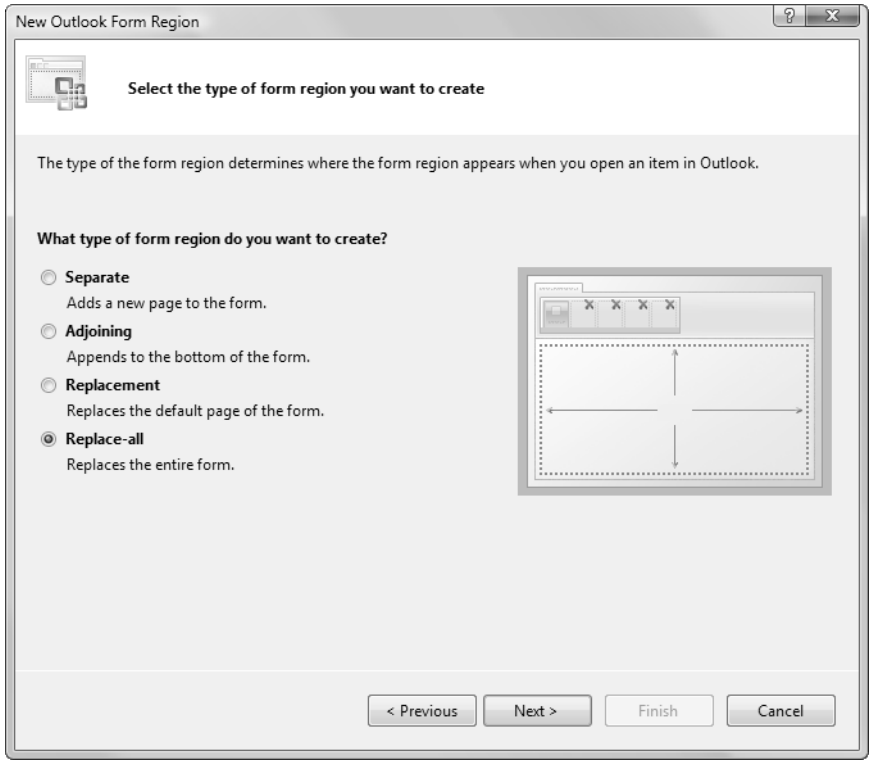

Figure 16-8: Selecting the type of form region to create: Replace-All.

To jump ahead a little, select Replacement or Replace-All as the form region type and then skip two steps ahead in the wizard by clicking the Next button twice. You see the wizard page shown in Figure 16-9, where you determine which Outlook message classes you want this form region to be associated with. When you select Replacement or Replace-All, notice that all the standard message classes (Appointment, Contact, Task, and so on) are grayed out in this dialog box. Outlook won't let you replace the default page or replace all the pages for standard message classes because you may break key features of Outlook. To use Replacement and Replace-All form region types, you must define a custom message class. A custom message class can reuse all the existing functionality of a built-in message class such as Appointment, Contact, or Task and acts as a specialized version of those built-in Outlook item objects. We discuss working with custom message

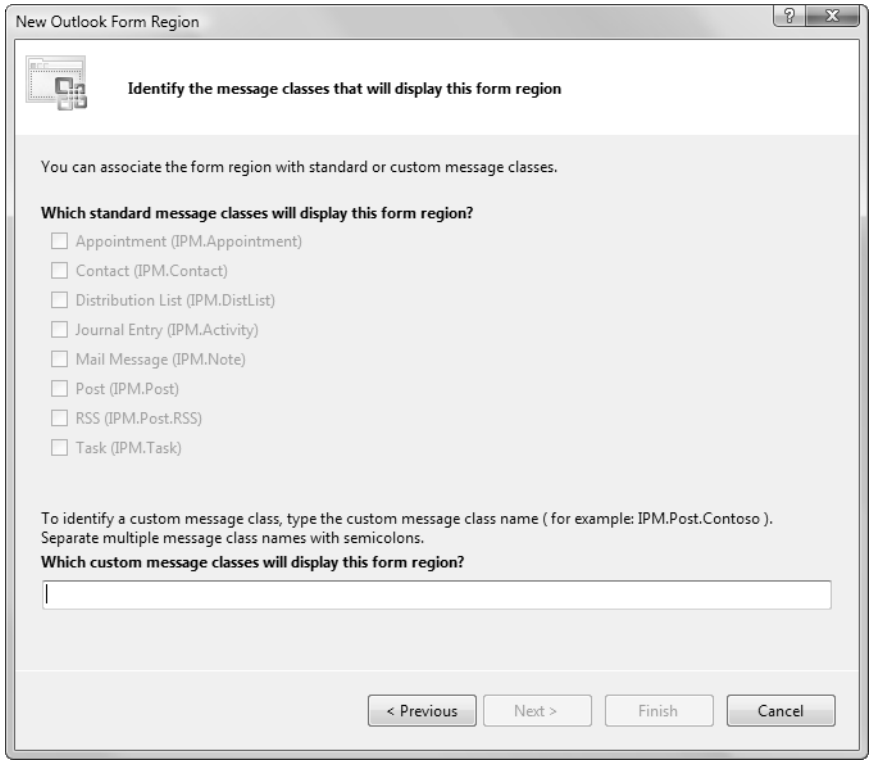

Figure 16-9: Replacement and Replace-All form regions can be associated only with custom message classes.

classes in more detail later in this chapter, in the section "Form Region Types and Custom Message Classes," because you must understand that concept to use Replacement and Replace-All form region types.

Moving back to the page in the wizard where you pick the form region type, consider the final form region type: Adjoining, shown in Figure 16-10. An Adjoining form region is appended to the bottom of the default page for an Inspector. Multiple adjoining form regions can be associated with the same message class, so potentially you can have several Adjoining form regions displayed in one Inspector window's default page. Adjoining form regions have headers that allow them to be collapsed and expanded to make more room in the default page when needed.

Another interesting application of an Adjoining form region is in an Explorer window. Specifically, an Adjoining form region can be used in the reading pane that is displayed in an Explorer window. In much the

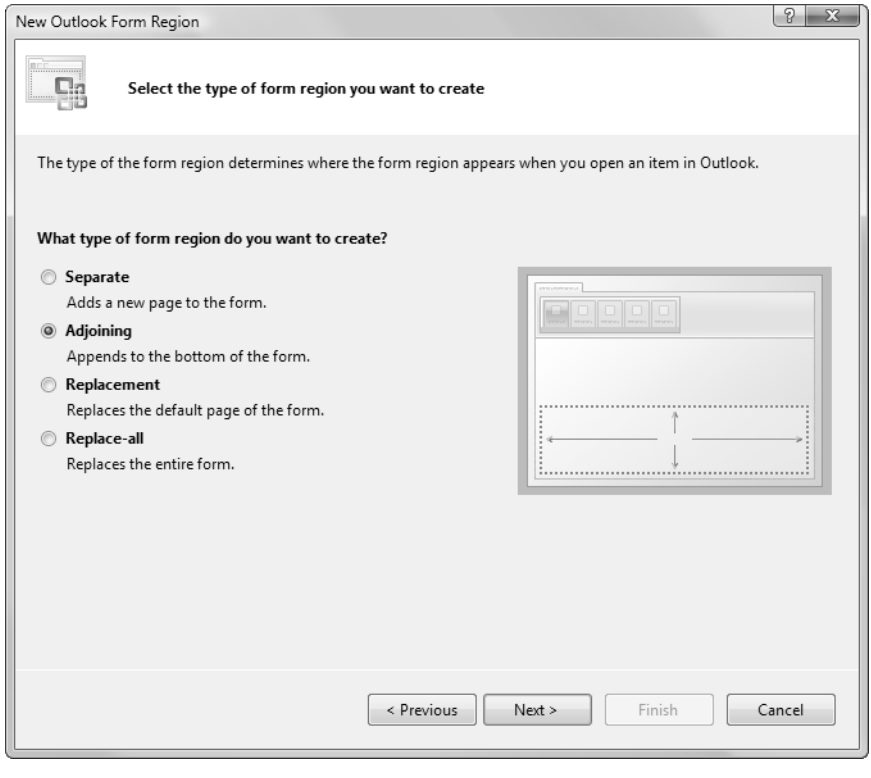

Figure 16-10: Selecting the type of form region to create: Adjoining.

same way that they are used in the default page of an Inspector window, multiple Adjoining form regions can be associated with an Outlook message class and can be displayed in the reading pane. Form regions displayed in the reading pane can also be collapsed to their headers. Replacement and Replace-All form regions can be used in the reading pane as well, although in this case they replace what is shown in the reading page and can be used only for custom message classes.

Now that you're familiar with all the form region types, select Adjoining as the form region type and click the Next button to move to the next page of the wizard, shown in Figure 16-11. In this dialog box, you set the name for the form region that will be displayed in the UI, so pick a friendly name. Title and Description are grayed out because you're creating an Adjoining form region; those options are enabled only for Replacement and Replace-All form region types.

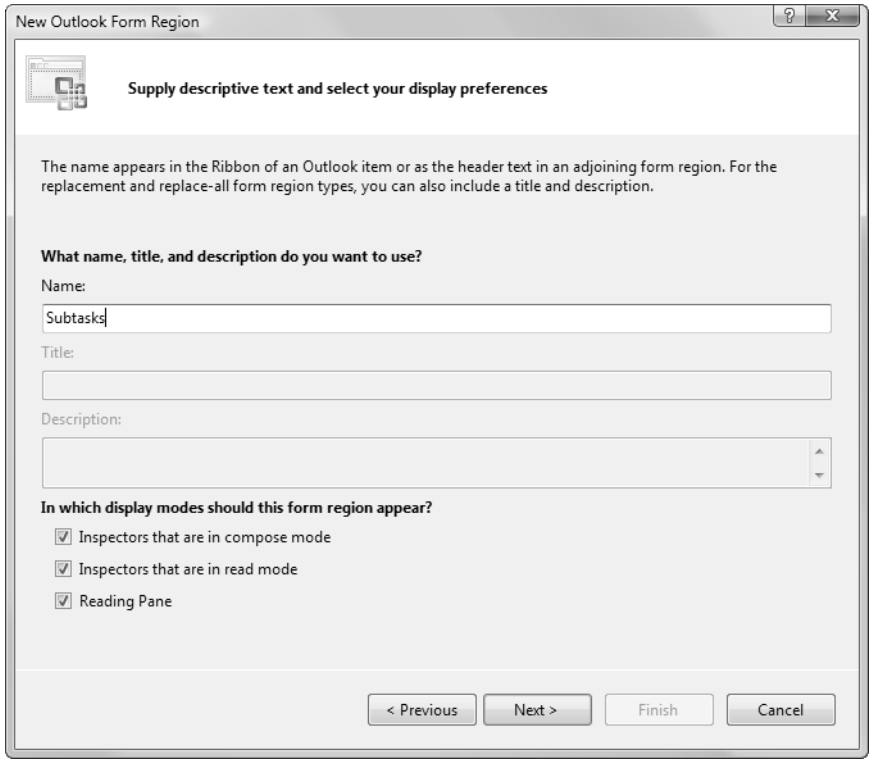

Figure 16-11: Setting descriptive text and display preferences.

This page of the wizard also has three check boxes that specify when the form region is displayed. The first check box sets whether the form region is displayed for an Inspector window that is in compose mode. An Inspector window is in compose mode when you create a new instance of the Outlook item associated with it—when you create a new task, for example. The second check box sets whether the form region is displayed for an Inspector window that is in read mode. An Inspector window is in read mode when you open an existing item—a mail message, for example. Finally, the third check box sets whether to display the form region in reading-pane view.

For this example, keep all the boxes checked and click the Next button to pick which Outlook message classes to associate the form region with, as shown in Figure 16-12. For this example, select Task. Note that you can associate the same form region with multiple built-in Outlook message

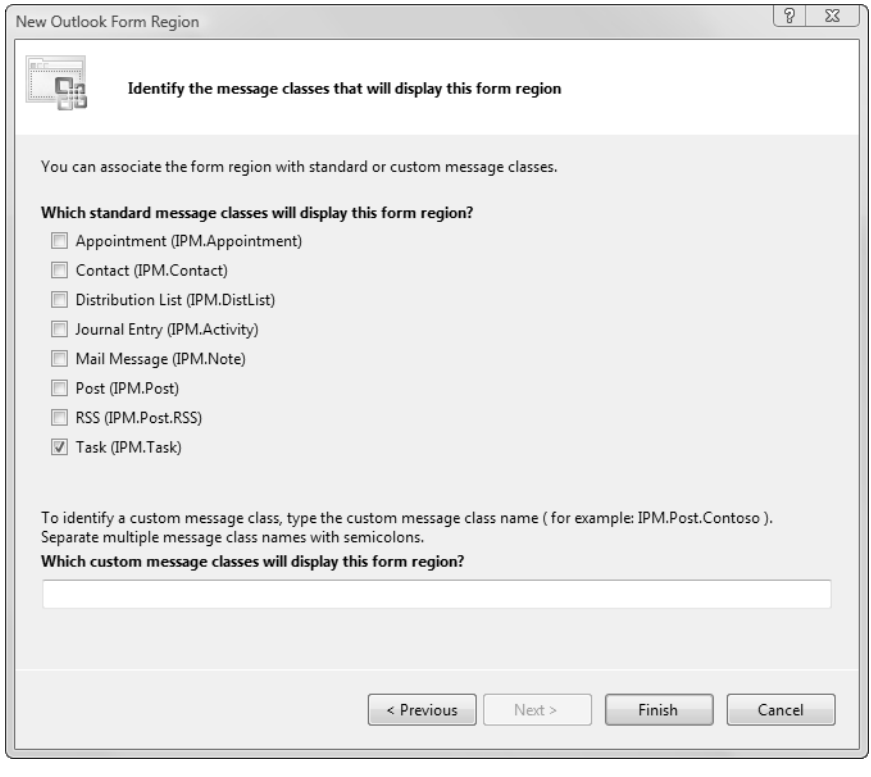

Figure 16-12: Picking which message classes will display a form region.

classes. You could have a form region that displays for both Tasks and Mail messages, for example. You can also associate a form region with custom message classes, which we discuss later in this chapter. As we describe earlier in this section, Replacement and Replace-All form region types can be associated only with custom message classes.

Associate the form region with the built-in Task type, and click the Finish button to exit the wizard. Visual Studio creates a new project item called FormRegion1.cs, as shown in Figure 16-13. It displays a visual designer in which you can drag and drop Windows Forms controls from the toolbox to construct the form region. This visual designer is much like the one you use to design user controls and task panes.

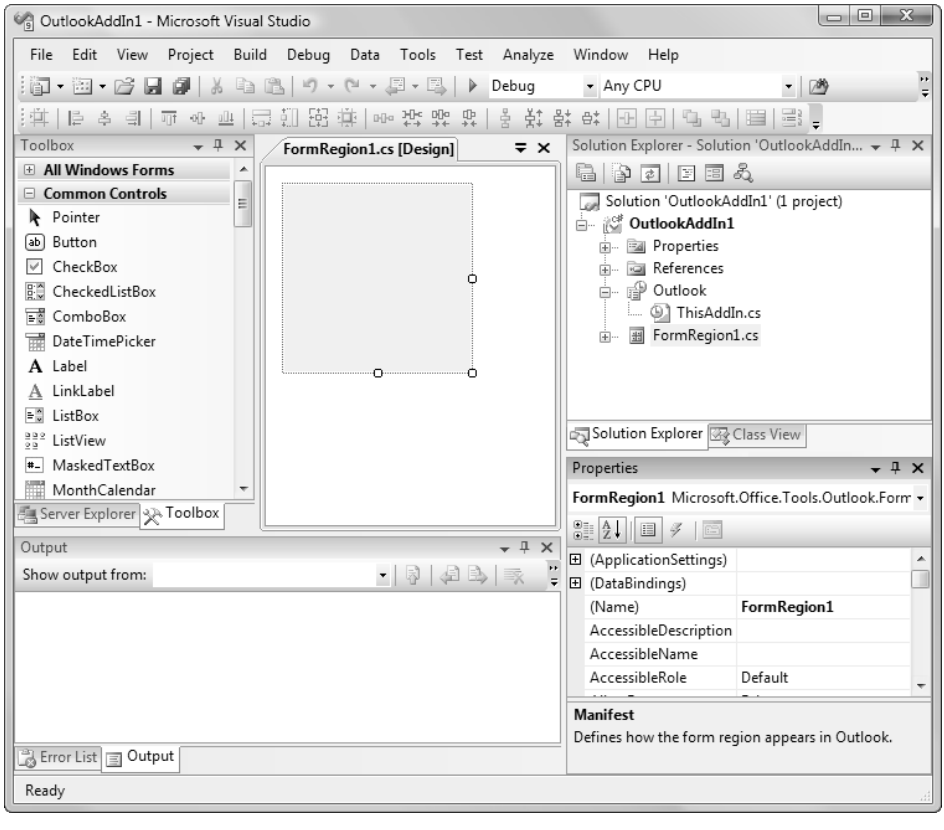

Figure 16-13: The newly created form region project item in visual design view.

Customizing a Form Region

Your goal is to add a form region in which subtasks can be associated with a task. First, drag and drop a list box control and a button to create a new task and delete an existing task. Because the user can resize the form region, use the Anchor property of the controls to anchor the list box to the top, left, bottom, and right, and anchor the buttons to the bottom and left. Figure 16-14 shows the final form region.

Before you go any further, run the add-in project and see what happens. Press F5 to build and run Outlook with the add-in project loaded. If you click a task in a task list and show reading view (by choosing View > Reading Pane > Bottom), you see that the adjoining form region is displayed docked at the bottom of reading-pane view for a task, as shown in Figure 16-15. If you double-click a task, the Adjoining form region is docked at the bottom of the default page for the Inspector window, as shown in Figure 16-16. After you've run your project, if you want to remove the form region and add-in from Outlook, choose Build > Clean.

Let's examine the adjoining form region a little more. First, notice that the Name you specified in Figure 16-11 is displayed as the caption above the Adjoining form region. To the left of the form region caption is a -/+ button that expands and collapses the form region. In Figure 16-17 you see what an Adjoining form region looks like when it is collapsed. Remember that several Adjoining form regions could be displayed in one Inspector window or reading pane; the ability to expand and collapse them is important, because it allows the end user to manage screen real estate.

Also, notice that when you resize the reading pane or the Inspector window, the form region has a default height. When the user adjusts the

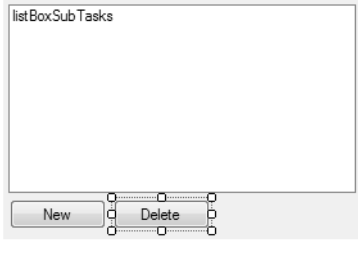

Figure 16-14: A simple form region.

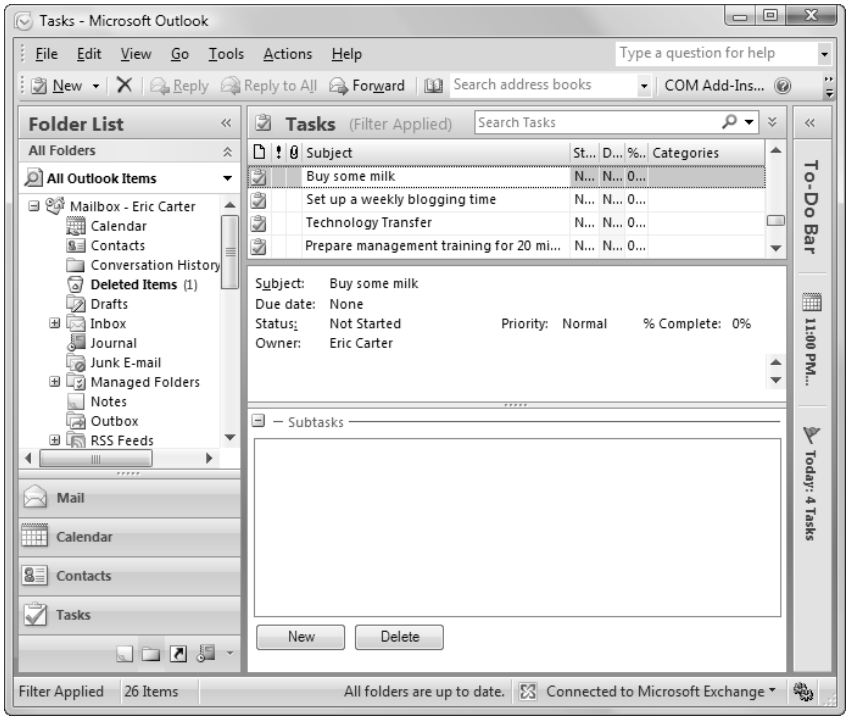

Figure 16-15: An Adjoining form region in the reading pane.

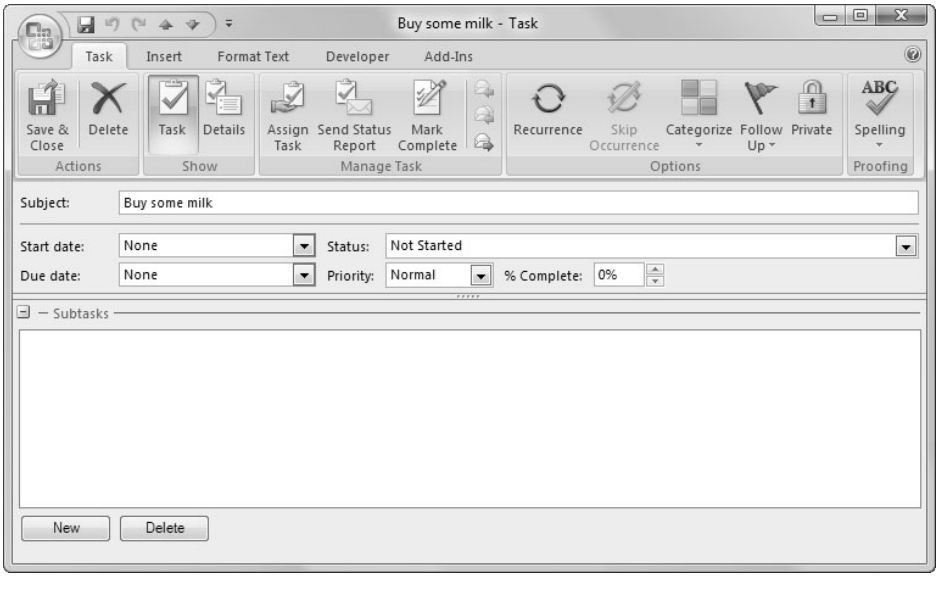

Figure 16-16: An Adjoining form region in the default page of an Inspector window.

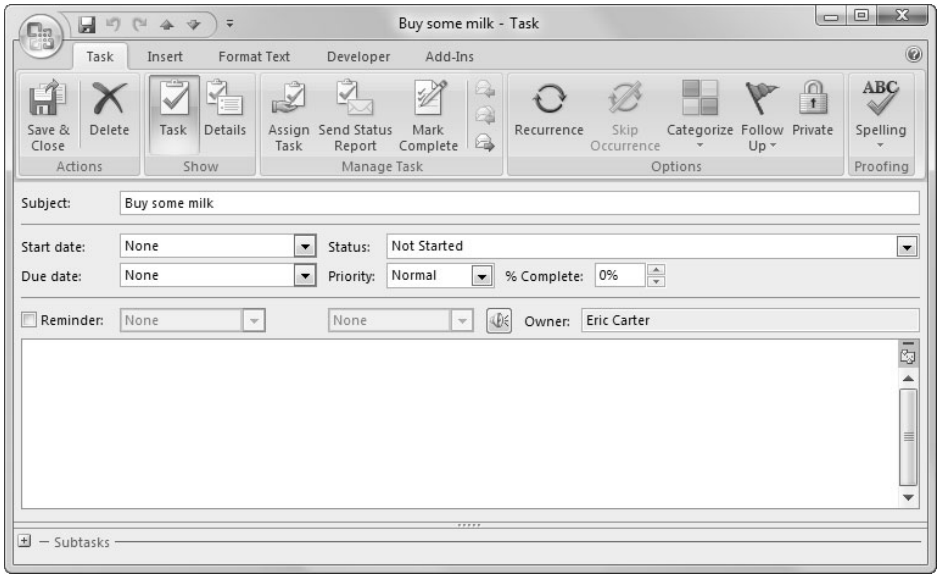

Figure 16-17: A collapsed Adjoining form region.

size of the form region, Outlook remembers the height and uses that height the next time the reading view is displayed. If you size the window small enough that the default height of the form region can't be displayed, a vertical scroll bar appears, as shown in Figure 16-18. This minimum height represents the height you set when you designed the form region. To have a smaller or larger minimum height, simply adjust the height of the visual design surface for the form region inside Visual Studio.

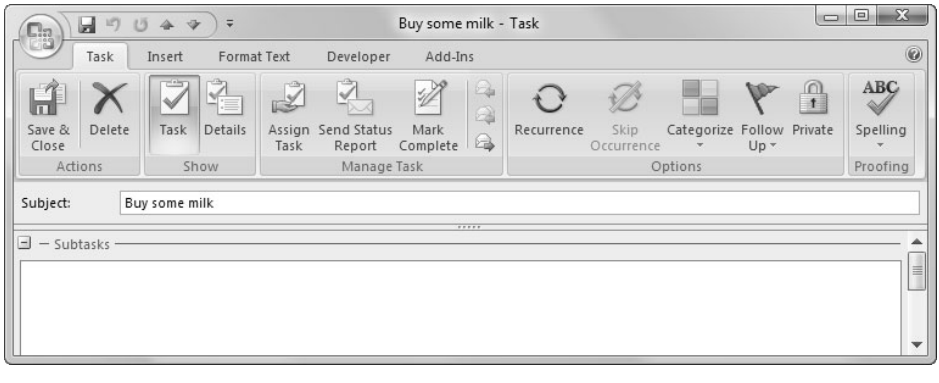

Figure 16-18: The effect of default height on the form region's vertical scroll bar.

Now exit Outlook and go back to the add-in project to put some code behind the form region. Right-click FormRegion1.cs in the Solution Explorer, and choose View Code from the context menu. The default code for a form region is shown in Listing 16-1. There are three event handlers of interest in our class FormRegion1. The first is actually in a nested class called FormRegion1Factory. This nested class provides a method called FormRegion1Factory_FormRegionInitializing where you can write code to decide whether to show the form region for a given Outlook item. The FormRegionInitializing event handler is passed a parameter `e` of type FormRegionInitializingEventArgs that can be used to get the Outlook item that the form region is about to be shown for (e.OutlookItem) and to cancel the showing of the form region if necessary by setting e.Cancel to `true`. Don't hold a reference to the Outlook item (e.OutlookItem) that is about to be shown; it is provided for use only during the event handler.

The form region class itself (FormRegion1) has a FormRegionShowing event handler that is invoked before the form region is displayed (but too late to prevent the display of the form region altogether; that is what FormRegionInitializing is for). In the FormRegionShowing event handler, you can write code to initialize your form region. In this event handler, you can use the property this.OutlookItem to access the Outlook item associated with the form region.

When the form region is closed, the FormRegionClosed event handler is invoked. This event handler is a good place to save any changes made to the Outlook item by your form region and to do any final cleanup.

Listing 16-1: The Default Code in a New Windows Forms-Based Form Region

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Office = Microsoft.Office.Core;
using Outlook = Microsoft.Office.Interop.Outlook;

namespace OutlookAddIn1
{
  partial class FormRegion1
  {
    #region Form Region Factory
    [Microsoft.Office.Tools.Outlook.
```

```
    FormRegionMessageClass(Microsoft.Office.Tools.Outlook.
    FormRegionMessageClassAttribute.Task)]
  [Microsoft.Office.Tools.Outlook.
    FormRegionName("OutlookAddIn1.FormRegion1")]
  public partial class FormRegion1Factory
  {
    // Occurs before the form region is initialized.
    // To prevent the form region from appearing, set e.Cancel
    // to true. Use e.OutlookItem to get a reference to the
    // current Outlook item.
    private void FormRegion1Factory_FormRegionInitializing(
      object sender, Microsoft.Office.Tools.Outlook.
      FormRegionInitializingEventArgs e)
    {
    }
  }
  #endregion

  // Occurs before the form region is displayed.
  // Use this.OutlookItem to get a reference to the current
  // Outlook item. Use this.OutlookFormRegion to get a reference
  // to the form region.
  private void FormRegion1_FormRegionShowing(object sender,
    System.EventArgs e)
  {
  }

  // Occurs when the form region is closed.
  // Use this.OutlookItem to get a reference to the current
  // Outlook item. Use this.OutlookFormRegion to get a reference
  // to the form region.
  private void FormRegion1_FormRegionClosed(object sender,
    System.EventArgs e)
  {
  }
 }
}
```

Listing 16-2 shows a simple implementation for the subtasks form region. You don't need to write any code in FormRegionInitializing because you always want to display your form region. In FormRegionShowing, write some code to get a custom UserProperty object from the Outlook item with which the form region is associated. The custom UserProperty we will associate with the Outlook item will have the identifier "SubTasks" You'll use this custom UserProperty to store the subtasks that are edited by the

form region. If the UserProperty isn't associated with the Outlook item yet, create the UserProperty for the Outlook item in FormRegionInitializing. The "SubTasks" user property contains a string value that contains subtasks delimited by a new line. You parse any subtasks that are in the string and populate the list box for the form region with the subtasks.

In FormRegionClosed, you do the reverse: Grab all the entries out of the list box and concatenate them into a string in which subtasks are separated by new lines. If the subtasks have been changed, set the "SubTasks" UserProperty's value to the new string and save the associated Outlook item.

Finally, a simple implementation for the Add button just adds the current time as a new subtask; a complete implementation would include a dialog box with an edit box in which the user could type a subtask description. The Delete button deletes the selected list item.

Listing 16-2: Form Region Code for a Simple Subtasks Form Region Based on Windows Forms

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Office = Microsoft.Office.Core;
using Outlook = Microsoft.Office.Interop.Outlook;

namespace OutlookAddIn1
{
  partial class FormRegion1
  {
    Outlook.TaskItem task;
    Outlook.UserProperty subtasks;

    #region Form Region Factory
    [Microsoft.Office.Tools.Outlook.
      FormRegionMessageClass(Microsoft.Office.Tools.Outlook.
      FormRegionMessageClassAttribute.Task)]
    [Microsoft.Office.Tools.Outlook.
      FormRegionName("OutlookAddIn1.FormRegion1")]
    public partial class FormRegion1Factory
    {
      // Occurs before the form region is initialized.
      // To prevent the form region from appearing, set e.Cancel
      // to true. Use e.OutlookItem to get a reference to the
      // current Outlook item.
```

```
  private void FormRegion1Factory_FormRegionInitializing(
    object sender, Microsoft.Office.Tools.Outlook.
    FormRegionInitializingEventArgs e)
  {
  }
}
#endregion

// Occurs before the form region is displayed.
// Use this.OutlookItem to get a reference to the current
// Outlook item. Use this.OutlookFormRegion to get a reference
// to the form region.
private void FormRegion1_FormRegionShowing(object sender,
  System.EventArgs e)
{
  task = this.OutlookItem as Outlook.TaskItem;
  if (task != null)
  {
    // Check for custom property SubTasks
    subTasks = task.UserProperties.Find("SubTasks", true);
    if (subTasks == null)
    {
      subTasks = task.UserProperties.Add("SubTasks",
        Outlook.OlUserPropertyType.olText, false,
        Outlook.OlUserPropertyType.olText);
    }
  }

  // Convert string
  string subTasksString = subTasks.Value.ToString();
  if (!String.IsNullOrEmpty(subTasksString))
  {
    string[] delimiters = new string[1];
    delimiters[0] = System.Environment.NewLine;
    string[] tasks = subTasksString.Split(delimiters,
      StringSplitOptions.RemoveEmptyEntries);
    for (int i = 0; i < tasks.Length; i++)
    {
      listBoxSubTasks.Items.Add(tasks[i]);
    }
  }
}

// Occurs when the form region is closed.
// Use this.OutlookItem to get a reference to the current
// Outlook item. Use this.OutlookFormRegion to get a reference
// to the form region.
private void FormRegion1_FormRegionClosed(object sender,
  System.EventArgs e)
```

```
    {
      if (subTasks == null || task == null)
        return;

      string oldTasks = subTasks.Value.ToString();
      StringBuilder builder = new StringBuilder();

      foreach (object o in listBoxSubTasks.Items)
      {
        string t = o as string;
        if (!String.IsNullOrEmpty(t))
        {
          builder.AppendLine(t);
        }
      }

      string newTasks = builder.ToString();

      if (!String.IsNullOrEmpty(newTasks) &&
        !String.IsNullOrEmpty(oldTasks))
      {
        if (newTasks.CompareTo(oldTasks) == 0)
          return; // no changes
      }

      subTasks.Value = newTasks;
      task.Save();
    }

    private void buttonNew_Click(object sender, EventArgs e)
    {
      // Just add current time as a subtask for simplicity
      listBoxSubTasks.Items.Add(
        System.DateTime.Now.ToShortTimeString());
    }

    private void buttonDelete_Click(object sender, EventArgs e)
    {
      if (listBoxSubTasks.SelectedItem != null)
      {
        listBoxSubTasks.Items.RemoveAt(
          listBoxSubTasks.SelectedIndex);
      }
    }
  }
}
```

When you run the form region, it displays as before, but now the Add and Delete buttons work, and you can add subtasks (set to the current time) to the current task.

Form Region Types and Custom Message Classes

Table 16-1 summarizes the behaviors and capabilities of the four types of form regions we introduced in the preceding sections. Figure 16-19 shows what the subtasks form region you created in the introduction looks like when it is changed to a Separate form region type. To make this change in Visual Studio, simply click the form region surface in the visual designer and change the FormRegionType property in the property grid. (This property can be a bit hard to find initially; it is a child property of the expandable Manifest property in the property grid.) Now when you open the Task Inspector, an additional Ribbon button appears in the Show group with the name of the separate form region—in this example, Subtasks. Subtasks is not the default page (Task is the default page), but when you click the Subtasks button, the form region page is displayed.

TABLE 16-1: Behavioral Capabilities of the Four Form Region Types

	Separate	Adjoining	Replacement	Replace-All
Inspector window behavior	Adds a new page	Appends to the bottom of the default page	Replaces the default page	Replaces all pages
Reading pane behavior	N/A	Appends to the bottom of the reading pane	Replaces the reading pane	Replaces the reading pane
Can customize standard built-in message classes	Yes	Yes	No—custom message classes only	No—custom message classes only

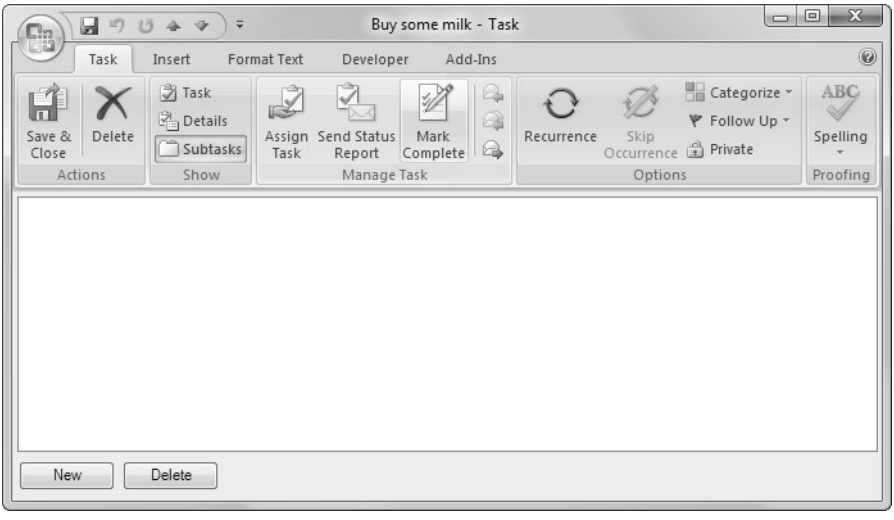

Figure 16-19: A Separate form region version of the Subtasks form region.

Built-In and Custom Message Classes

To convert the example form region to a Replacement or Replace-All form region type, you need to learn a little bit more about built-in and custom message classes. The type of all Outlook items is identified by a `string` value called a message class. Table 16-2 lists the message classes associated with some of the major Outlook item types.

You can define your own custom message class by defining your own message class string. The message class string must begin with a built-in message class string from Table 16-2 to ensure that you inherit the behavior associated with a built-in message class; Outlook does not support having a "baseless" message class that doesn't inherit behavior from a built-in Outlook type. Then you append your own unique identifier to the message class string to create a unique message class string. If you want to create a custom message class based on Outlook's built-in contact type that extends the contact with some information about the Facebook user ID associated with that contact, for example, your message class string might be "IPM.Contact.FacebookAware". The important thing is that your custom message class string start with a built-in message class identifier ("IPM.Contact", for example) and have some additional identifier that

TABLE 16-2: Built-In Outlook Message Classes

Outlook Item Type	Message Class String
Appointment	IPM.Appointment
Contact	IPM.Contact
Distribution List	IPM.DistList
Journal Entry	IPM.Activity
Mail Message	IPM.Note
Post	IPM.Post
RSS Post	IPM.Post.RSS
Sharing Invitation	IPM.Sharing
Task	IPM.Task

won't be picked by another add-in developer. So you might make it more unique by embedding your company name, as in "IPM.Contact.FacebookAwareAddisonWesley".

You can use these unique `string` custom message classes to create Outlook items with the Items.Add method on an Outlook Folder object. You can modify the code of the add-in you created in the introduction to edit the ThisAddIn_Startup method so that it creates an Outlook item with a custom message class based on Task, to be called "IPM.Task.MySubTaskAwareTask". Listing 16-3 shows the new ThisAddIn.cs code file.

Listing 16-3: An Outlook Add-In That Creates a New Outlook Item with a Custom Message Class Based on Task

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Xml.Linq;
using Outlook = Microsoft.Office.Interop.Outlook;
using Office = Microsoft.Office.Core;
```

```
namespace OutlookAddIn1
{
  public partial class ThisAddIn
  {
    private void ThisAddIn_Startup(object sender,
      System.EventArgs e)
    {
      Outlook.Folder taskList =
        Application.Session.GetDefaultFolder(
        Outlook.OlDefaultFolders.olFolderTasks)
        as Outlook.Folder;

      Outlook.TaskItem taskItem = taskList.Items.Add(
        "IPM.Task.MySubTaskAwareTask") as Outlook.TaskItem;

      taskItem.Subject = "IPM.Task.MySubTaskAwareTask Created On "+
        System.DateTime.Now.ToLongDateString();

      taskItem.Save();
    }

    #region VSTO generated code
    private void InternalStartup()
    {
      this.Startup += new System.EventHandler(ThisAddIn_Startup);
    }
    #endregion
  }
}
```

Now that the Add-in creates a new task with a custom message class on startup, you can modify the form region to be a Replacement form region type. To do this, double-click FormRegion1.cs in the Solution Explorer to activate the form region designer. In the Properties window, pick FormRegion1 in the list of controls. Expand the Manifest section of the Properties window, and set the FormRegionType to Replacement.

Now you need to change the FormRegion1Factory so that it associates the form region with the custom message class "IPM.Task.MySubTask-AwareTask" rather than with the built-in message class for a task, "IPM.Task". To do this, you need to edit an attribute of the FormRegion-1Factory class. Looking at the FormRegion1Factory class, you see two custom attributes: FormRegionMessageClass and FormRegionName. Form-RegionMessageClass tells the factory what message class to show the form

region for. Because you associated the form region with a task when you created it in the form region wizard, the FormRegionMessageClass attribute is set to display for the string specified by the constant `Microsoft.Office.Tools.Outlook.FormRegionMessageClassAttribute.Task`. This string is a constant string that is set to "IPM.Task". The FormRegionName attribute is set to the fully qualified name of the form region class—in this case, "OutlookAddIn1.FormRegion1". Both custom attributes are shown here:

```
#region Form Region Factory
[Microsoft.Office.Tools.Outlook.FormRegionMessageClass(
  Microsoft.Office.Tools.Outlook.
  FormRegionMessageClassAttribute.Task)]
[Microsoft.Office.Tools.Outlook.FormRegionName(
  "OutlookAddIn1.FormRegion1")]
public partial class FormRegion1Factory
{
```

Change the FormRegionMessageClass attribute to take the custom message class string "IPM.Task.MySubTaskAwareTask", as follows:

```
#region Form Region Factory
[Microsoft.Office.Tools.Outlook.FormRegionMessageClass(
  "IPM.Task.MySubTaskAwareTask")]
[Microsoft.Office.Tools.Outlook.FormRegionName(
  "OutlookAddIn1.FormRegion1")]
public partial class FormRegion1Factory
{
```

Now when you run the add-in, a new task with custom message class "IPM.Task.MySubTaskAwareTask" is created in the Startup handler for the add-in. When the new task with the custom message class is opened, an Inspector window with the default page replaced is displayed, as shown in Figure 16-20. Note in the Show group on the Ribbon, the default is now the Subtasks form region. The original default page, Task, is no longer visible in the pages that can be shown for the task.

Finally, you can go back to the add-in and use the Properties window to set the FormRegionType to Replace-All. When the add-in is run and a task with the custom message class is opened, the Inspector window has all the pages removed except for your form region, as shown in Figure 16-21.

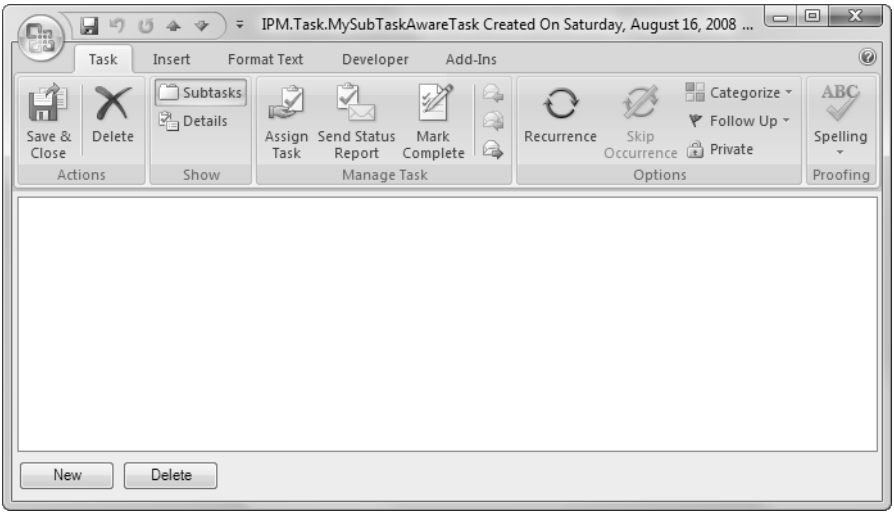

Figure 16-20: A Replacement form region version of the Subtasks form region.

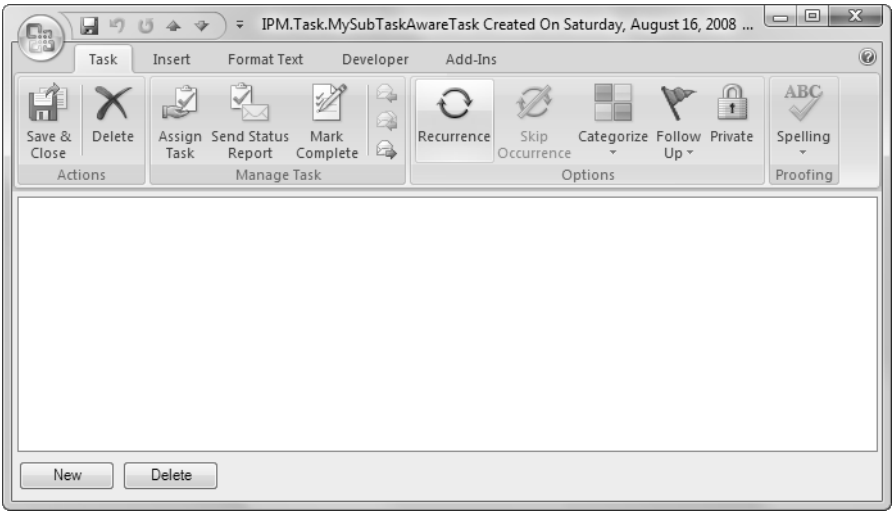

Figure 16-21: A Replace-All form region version of the Subtasks form region.

NOTE

In Figure 16-21, the Show group on the Ribbon is no longer displayed. Because only one page is available to be displayed (your form region), there is no longer a reason to display the Show group, as no other pages can be selected.

Creating an Outlook Forms-Based Form Region

As you saw in the introduction, there are two ways to create form regions in VSTO. The first is to use the Windows Forms designers inside Visual Studio. The second is to use the Outlook Forms Designer inside Outlook. A form region designed with the Outlook Forms Designer is integrated with your .NET code through COM interop. The form region and form controls are COM objects that Visual Studio generates wrappers to communicate with.

If you use Windows Forms, you can use a forms engine that is .NET-based, which may be more familiar to you. The design-time experience for using Windows Forms is much more integrated with Visual Studio. If you use the Outlook Forms Designer, you need to design the form in Outlook and then import it into Visual Studio. If you decide that you want to change the layout or the controls on the form, you have to delete the form region in Visual Studio, go back to Outlook and redesign the form region, and then reimport it into Visual Studio via the wizard.

Although using Windows Forms is more convenient, Outlook Forms have a lot of features that Windows Forms do not. Outlook Forms, for example, have automatic data binding support to bind to properties of the Outlook items with which they are associated. Also, some Outlook Forms controls are more Outlook-like than any of the Windows Forms controls you have available to you. These controls allow you to replicate functionality available in built-in Outlook Inspector windows. Outlook Forms provide controls such as these: the Outlook Body control, which allows you to edit the item body of an Outlook item with the same editor provided for built-in Outlook editors; a Business Card control, which displays the same business-card view that is built into Outlook; and a Category control, which provides a UI for visualizing the categories with which an Outlook item is associated. So in many cases you may pick an Outlook Forms-based form region because it provides more Outlook-aware controls for you to use.

The first step in creating a form region by using Outlook Forms is launching Outlook. Next, choose Tools > Forms > Design a Form to bring up the Design Form dialog box, shown in Figure 16-22. Pick the type of built-in Outlook item type that you want to start from—for this example, Task—and then click the Open button.

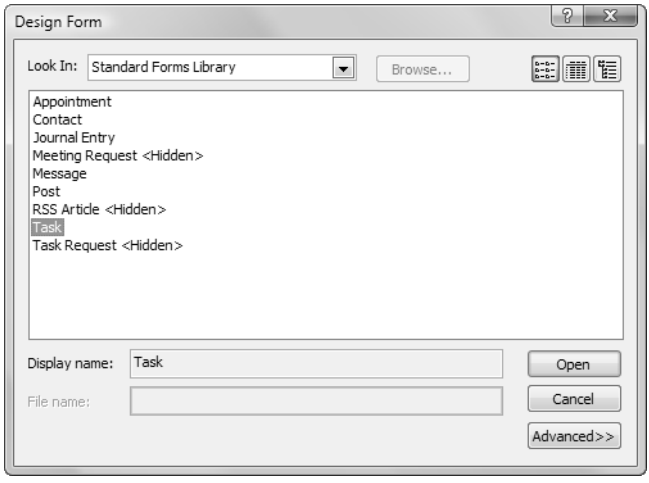

Figure 16-22: The Design Form dialog box in the Outlook Forms Designer.

Next, drop down the Design button and then the Form Region button, and choose New Form Region, as shown in Figure 16-23. (You can also use the Open Form Region command under the Form Region button if you already have a form region in an .OFS file.) A new page titled (Form Region) will appear; you can design your form region in that page.

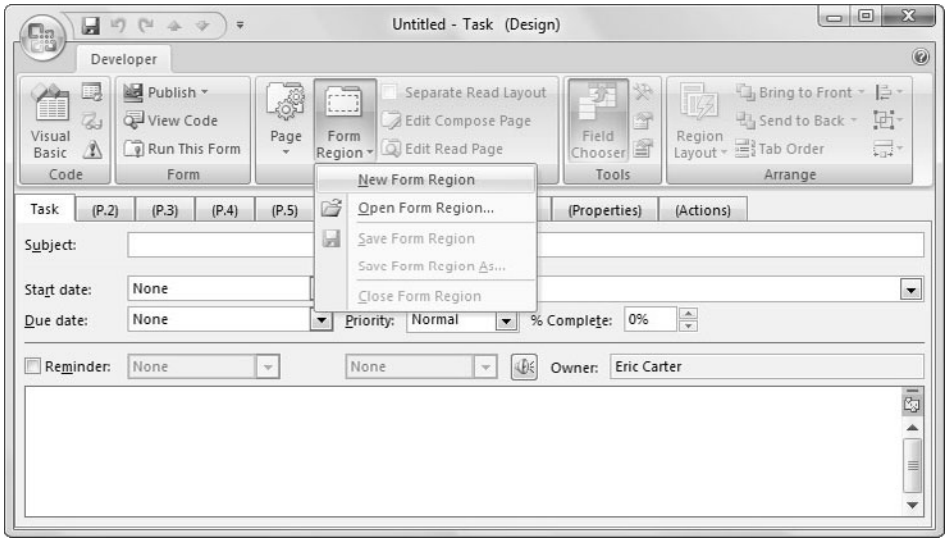

Figure 16-23: Creating a new form region in the Outlook Forms Designer.

The Tools group in the Ribbon allows you to bring up the design-time tools you will need. The Field Chooser tool lets you drag and drop fields into your form region from the Outlook item that the form region will display. These fields are automatically data bound—an advantage over Windows Forms, which require you to write additional code to bind your controls to the Outlook item associated with your form region. Also in the Tools group, the Controls Toolbox button brings up the toolbox, which displays a set of standard controls. The Advanced Properties button displays the properties window, which you can use to set properties for the selected control in the Forms Designer.

The initial set of tools in the Controls toolbox doesn't have any of the cool controls we mentioned earlier, so let's get them added to the toolbox. Right-click a blank area of the Controls toolbox, and choose Custom Controls from the context menu. The Additional Controls dialog box appears, as shown in Figure 16-24. Check all the controls in the list that start with *Microsoft Office Outlook*; then click the OK button.

Figure 16-25 shows the final design environment with all the tools visible and all the additional Outlook controls added to the Controls toolbox.

Now drag some Outlook controls to the design surface. You'll create the same form region you created in the introduction but use Outlook Forms this time. Find the OlkListBox control by hovering over the controls in the

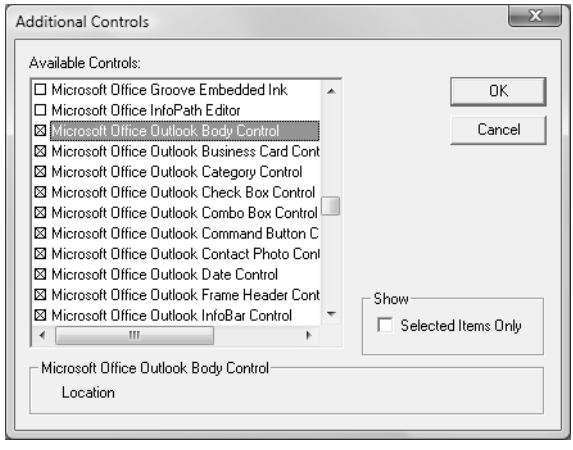

Figure 16-24: Add all the controls that start with Microsoft Office Outlook to the Controls toolbox.

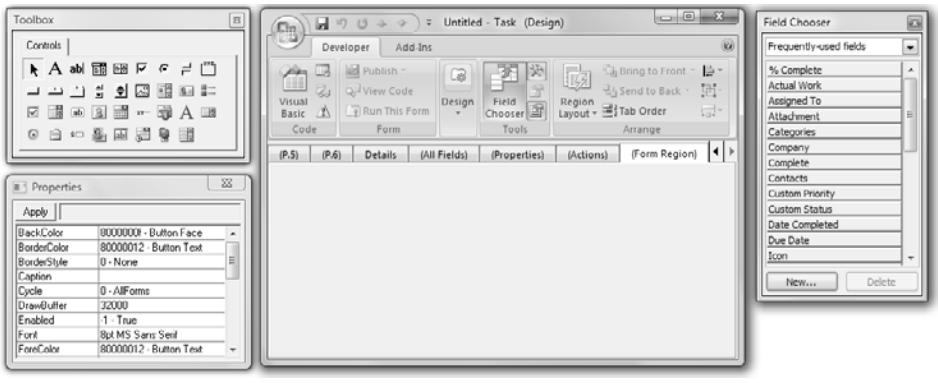

Figure 16-25: The Outlook Forms Designer with the toolbox, Properties window, and Field Chooser tool.

Controls toolbox and finding the control that shows OlkListBox in its tooltip. Drag and drop a OlkListBox, and size it to fill most of the design surface while leaving a strip at the bottom for buttons. Right-click the list-box control you added to the form, and choose Properties from the context menu to bring up the Properties dialog box, shown in Figure 16-26. Click the Layout tab, and drop down the Horizontal combo box to pick Grow/Shrink with Form. This setting allows the list box to size to fill the Inspector window.

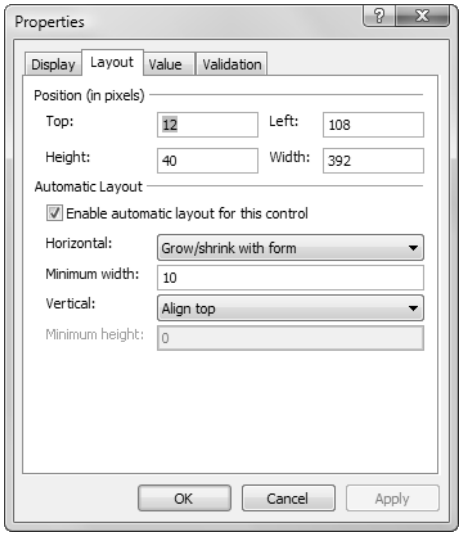

Figure 16-26: The Properties dialog box with the Layout tab.

While the Properties dialog box is open, take some time to explore the rest of it. The Properties dialog box has a Display tab that lets you set the caption of the control, visibility, font, and color. The Layout tab lets you set size and position, as well as several useful autosizing and alignment settings. The Value tab lets you set up an automatic binding to an Outlook item property. Finally, the Validation tab lets you set up some validation rules for the control.

Next, drag two additional Outlook controls onto the design surface. Drag and drop two OlkCommandButton controls at the bottom of the Outlook Form region. The OlkCommandButton will display with a look and feel more consistent with the Outlook UI than with a CommandButton. Right-click each of the OlkCommandButton controls, and choose Properties from the context menu to display the Properties window. In the Display tab, set the caption of one button to Add and the other to Delete. Also, in the Layout tab, set the Vertical drop-down menu to Align Bottom for each of the two buttons to ensure that the buttons stay at the bottom of the form when the form is resized. The final form region should look like Figure 16-27.

With a form region designed, you need to export the form region to an Outlook form region file with a .OFS extension, which then can be imported into Visual Studio. To save the form region as an .OFS file, drop down the Design button; then drop down the Form Region button and choose Save Form Region As to bring up the Save dialog box. Save it as MyFormRegion for this example, as shown in Figure 16-28.

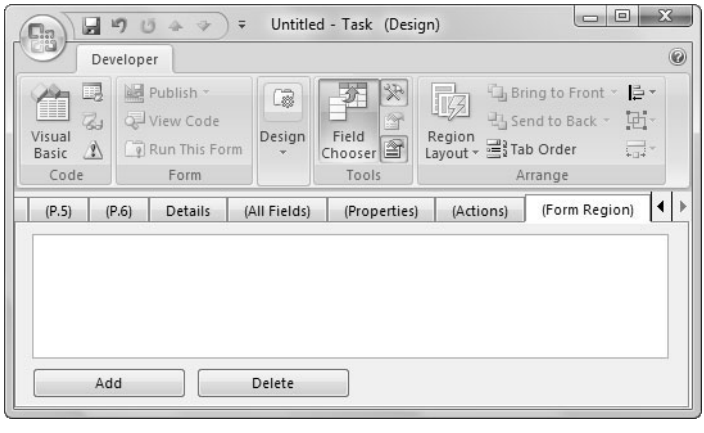

Figure 16-27: A form region designed in the Outlook Form Designer.

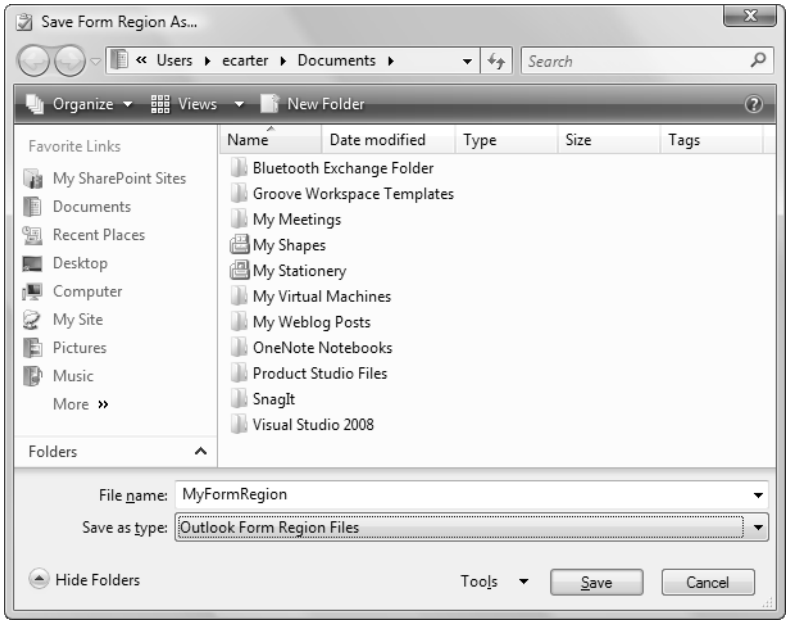

Figure 16-28: Saving the form region to an Outlook form region file (.OFS file).

> **TIP**
>
> At this point, we recommend that you exit Outlook before moving back to Visual Studio. The process of importing a .OFS file from Visual Studio involves Visual Studio starting up Outlook and talking to it to process the OFS file, and we've found that this process works best if you don't already have Outlook already open.

Start Visual Studio, and either create a new Outlook add-in project or open an existing Outlook add-in project. Choose Project > Add New Item. Click the Office category to show just the Office-specific items. In the list of Office items, click Outlook Form Region (refer to Figure 16-2). Name the form region—for this exercise, FormRegion2. Then click the Add button.

In the first page of the New Outlook Form Region wizard that appears, pick Import an Outlook Form Storage (.ofs) File, as shown in Figure 16-29. Click the Browse button, and locate the .OFS file that you saved earlier.

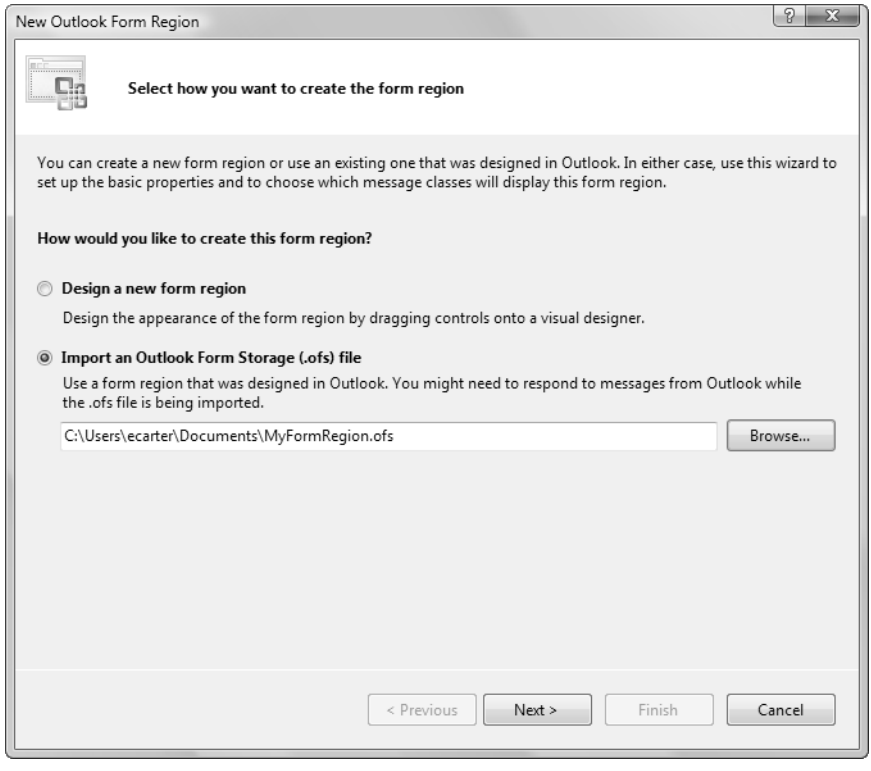

Figure 16-29: Importing an .OFS file in the New Outlook Form Region wizard.

With a .OFS file selected, click the Next button to move to the page where you set the type of the form region. For this example, select Separate as the Form Region type. Then click the Next button. On the next page, set the name you want to use for the form region—for this exercise, Subtasks—and make sure that the check boxes for Inspectors that are in compose mode and Inspectors that are in read mode are checked. Click the Next button to move to the final page. On this page, make sure that only the check box next to Task is checked; then click the Finish button. Visual Studio creates a new project item for the form region.

No visual designer is displayed within Visual Studio—just generated code. As we mention earlier in this chapter, if you want to change the form region, you have to delete the form region code item from your Visual Studio project, go back to Outlook and reopen the .ofs file, modify your form

region, save it an .ofs file, exit Outlook, and then re-create the form region in Visual Studio.

> **TIP**
>
> If you are changing only the layout of the form region, you can edit the .ofs file without regenerating the form region in Visual Studio; just copy the modified .ofs file over the old .ofs file in your project folder.

The code looks very similar to the code for a Windows Forms-based form region. As before, you have a form region class with a nested form region factory class. The form region factory class has a FormRegionInitializing event handler where you can write code to determine whether to show the form region. The event handler is passed a parameter `e` of type FormRegionInitializingEventArgs that can be used to get the Outlook item that the form region is about to be shown for (e.OutlookItem) and to cancel the showing of the form region if necessary by setting e.Cancel to `true`.

The form region class has a FormRegionShowing event handler that is invoked before the form region is displayed (but too late to prevent the display of the form region altogether). In this event handler, you can write code to initialize the form region and use this.OutlookItem to access the Outlook item associated with the form region.

When the form region is closed, the FormRegionClosed event handler is invoked. This event handler is a good place to save any changes made to the Outlook item by your form region and do any final cleanup.

There are also some major differences between a Windows Forms-based form region and an Outlook Forms-based form region. Because there is no design view, no property grid like the one in Windows Forms allows you to interact with key settings—especially the Manifest settings that are editable in a Windows Forms-based form region. To compensate for this deficiency in Outlook Forms-based form regions, VSTO adds a second method called InitializeManifest to the nested form region factory code, as shown in Listing 16-4. In this method, you can modify the code to change the Form region type or any of the other settings that you initially set in the Form Region wizard.

Listing 16-4: The Default Code in a New Outlook Forms-Based Form Region

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Resources;
using System.Text;
using Office = Microsoft.Office.Core;
using Outlook = Microsoft.Office.Interop.Outlook;

namespace OutlookAddIn2
{
  public partial class FormRegion2
  {
    #region Form Region Factory

    [Microsoft.Office.Tools.Outlook.
      FormRegionMessageClass(Microsoft.Office.Tools.Outlook.
      FormRegionMessageClassAttribute.Task)]
    [Microsoft.Office.Tools.Outlook.
      FormRegionName("OutlookAddIn2.FormRegion2")]
    public partial class FormRegion2Factory
    {
      private void InitializeManifest()
      {
        ResourceManager resources =
          new ResourceManager(typeof(FormRegion2));
        this.Manifest.FormRegionType =
          Microsoft.Office.Tools.Outlook.FormRegionType.Separate;
        this.Manifest.Title = resources.GetString("Title");
        this.Manifest.FormRegionName =
          resources.GetString("FormRegionName");
        this.Manifest.Description =
          resources.GetString("Description");
        this.Manifest.ShowInspectorCompose = true;
        this.Manifest.ShowInspectorRead = true;
        this.Manifest.ShowReadingPane = false;

      }

      // Occurs before the form region is initialized.
      // To prevent the form region from appearing, set e.Cancel to
      // true. Use e.OutlookItem to get a reference to the current
      // Outlook item.
      private void FormRegion2Factory_FormRegionInitializing(
        object sender,  Microsoft.Office.Tools.Outlook.
        FormRegionInitializingEventArgs e)
      {
      }
    }
```

```
    #endregion

    // Occurs before the form region is displayed.
    // Use this.OutlookItem to get a reference to the current
    // Outlook item. Use this.OutlookFormRegion to get a reference
    // to the form region.
    private void FormRegion2_FormRegionShowing(object sender,
      System.EventArgs e)
    {
    }

    // Occurs when the form region is closed. Use this.OutlookItem
    // to get a reference to the current Outlook item. Use
    // this.OutlookFormRegion to get a reference to the form/
    // region.
    private void FormRegion2_FormRegionClosed(object sender,
      System.EventArgs e)
    {
    }
  }
}
```

Another obvious difference is that the form region class created when you import a .OFS file does not derive from System.Windows.Forms.UserControls. It derives from a class in VSTO called Microsoft.Office.Tools.Outlook.ImportedFormRegion.

As you write your code in the form region class, you will find that VSTO has created member variables for all the controls you used in the .OFS file. For this example, you created a OlkListBox with a default name of OlkListBox1, an OlkCommandButton with a name of OlkCommandButton1, and an OlkCommandButton with a name of OlkCommandButton2. The import of the .OFS file converts the names used in the Outlook Form designer to camel case, so you have three controls named olkListBox1, olkCommandButton1, and olkCommandButton2.

These controls are of types that come from the Microsoft.Office.Interop.Outlook namespace. This namespace has types for many of the built-in Outlook controls. Some of the controls in the toolbox generate types that come from the Microsoft.Vbe.Interop.Forms namespace. Table 16-3 shows the names of the controls in Outlook's Controls toolbox and the .NET types associated with these controls.

TABLE 16-3: Mapping Between Outlook Controls and .NET Types

Name	Type
Microsoft Forms 2.0 CheckBox	Microsoft.Office.Interop.Outlook.OlkCheckBox
Microsoft Forms 2.0 ComboBox	Microsoft.Office.Interop.Outlook.OlkComboBox
Microsoft Forms 2.0 CommandButton	Microsoft.Office.Interop.Outlook.OlkCommandButton
Microsoft Forms 2.0 Frame	Microsoft.Vbe.Interop.Forms.UserForm
Microsoft Forms 2.0 Image	Microsoft.Vbe.Interop.Forms.Image
Microsoft Forms 2.0 Label	Microsoft.Office.Interop.Outlook.OlkLabel
Microsoft Forms 2.0 ListBox	Microsoft.Office.Interop.Outlook.OlkListBox
Microsoft Forms 2.0 MultiPage	Microsoft.Vbe.Interop.Forms.MultiPage
Microsoft Forms 2.0 OptionButton	Microsoft.Office.Interop.Outlook.OlkOptionButton
Microsoft Forms 2.0 ScrollBar	Microsoft.Vbe.Interop.Forms.ScrollBar
Microsoft Forms 2.0 SpinButton	Microsoft.Vbe.Interop.Forms.SpinButton
Microsoft Forms 2.0 TabStrip	Microsoft.Vbe.Interop.Forms.TabStrip
Microsoft Forms 2.0 TextBox	Microsoft.Office.Interop.Outlook.OlkTextBox
Microsoft Forms 2.0 ToggleButton	Microsoft.Vbe.Interop.Forms.ToggleButton

Continues

TABLE 16-3: Mapping Between Outlook Controls and .NET Types *(Continued)*

Name	Type
Microsoft Office Outlook Business Card Control	Microsoft.Office.Interop.Outlook.OlkBusinessCardControl
Microsoft Office Outlook Category Control	Microsoft.Office.Interop.Outlook.OlkCategory
Microsoft Office Outlook Check Box Control	Microsoft.Office.Interop.Outlook.OlkCheckBox
Microsoft Office Outlook Combo Box Control	Microsoft.Office.Interop.Outlook.OlkComboBox
Microsoft Office Outlook Command Button Control	Microsoft.Office.Interop.Outlook.OlkCommandButton
Microsoft Office Outlook Contact Photo Control	Microsoft.Office.Interop.Outlook.OlkContactPhoto
Microsoft Office Outlook Date Control	Microsoft.Office.Interop.Outlook.OlkDateControl
Microsoft Office Outlook Frame Header Control	Microsoft.Office.Interop.Outlook.OlkFrameHeader
Microsoft Office Outlook InfoBar Control	Microsoft.Office.Interop.Outlook.OlkInfoBar
Microsoft Office Outlook Label Control	Microsoft.Office.Interop.Outlook.OlkLabel

TABLE 16-3: Mapping Between Outlook Controls and .NET Types *(Continued)*

Name	Type
Microsoft Office Outlook List Box Control	Microsoft.Office.Interop.Outlook.OlkListBox
Microsoft Office Outlook Option Button Control	Microsoft.Office.Interop.Outlook.OlkOptionButton
Microsoft Office Outlook Page Control	Microsoft.Office.Interop.Outlook.OlkPageControl
Microsoft Office Outlook Recipient Control	Microsoft.Office.Interop.Outlook._DRecipientControl
Microsoft Office Outlook Sender Photo Control	Microsoft.Office.Interop.Outlook.OlkSenderPhoto
Microsoft Office Outlook Text Box Control	Microsoft.Office.Interop.Outlook.OlkTextBox
Microsoft Office Outlook Time Control	Microsoft.Office.Interop.Outlook.OlkTimeControl
Microsoft Office Outlook Time Zone Control	Microsoft.Office.Interop.Outlook.OlkTimeZoneControl
Microsoft Office Outlook View Control	Microsoft.Office.Interop.OutlookViewCtl.ViewCtl
Microsoft Office Outlook Body Control	Microsoft.Office.Interop.Outlook._DDocSiteControl

As you write code against Outlook controls, you will discover that you sometimes need to cast the primary Outlook control types listed in Table 16-3 to either of two different types: Microsoft.Office.Interop.Outlook.OlkControl and Microsoft.Vbe.Interop.Forms.Control. When you cast to an OlkControl, you can set properties to configure Outlook-specific layout and binding options like those that are settable by the Properties dialog box in the Outlook Forms Designer. When you cast to a Control, you can set basic positioning properties that are common to all controls. Remember that before writing code to cast to a Microsoft.Vbe.Interop.Forms.Control, you must ensure that your project has a reference to the Microsoft Forms 2.0 Object Library.

Listing 16-5 is similar to Listing 16-2. The only difference is that it uses Outlook Forms controls, so some of the code for adding and removing items to the OlkListBox is different.

Listing 16-5: Form Region Code for a Simple Subtasks Form Region Based on Outlook Forms

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Resources;
using System.Text;
using Office = Microsoft.Office.Core;
using Outlook = Microsoft.Office.Interop.Outlook;

namespace OutlookAddIn2
{
  public partial class FormRegion2
  {
    Outlook.TaskItem task;
    Outlook.UserProperty subTasks;

    #region Form Region Factory

    [Microsoft.Office.Tools.Outlook.
      FormRegionMessageClass(Microsoft.Office.Tools.Outlook.
      FormRegionMessageClassAttribute.Task)]
    [Microsoft.Office.Tools.Outlook.FormRegionName(
      "OutlookAddIn2.FormRegion2")]
    public partial class FormRegion2Factory
    {
      private void InitializeManifest()
      {
        ResourceManager resources =
          new ResourceManager(typeof(FormRegion2));
```

```
    this.Manifest.FormRegionType = Microsoft.Office.
      Tools.Outlook.FormRegionType.Separate;
    this.Manifest.Title = resources.GetString("Title");
    this.Manifest.FormRegionName = resources.
      GetString("FormRegionName");
    this.Manifest.Description = resources.
      GetString("Description");
    this.Manifest.ShowInspectorCompose = true;
    this.Manifest.ShowInspectorRead = true;
    this.Manifest.ShowReadingPane = false;

  }

  // Occurs before the form region is initialized.
  // To prevent the form region from appearing, set e.Cancel to
  // true. Use e.OutlookItem to get a reference to the current
  // Outlook item.
  private void FormRegion2Factory_FormRegionInitializing(
    object sender, Microsoft.Office.Tools.Outlook.
    FormRegionInitializingEventArgs e)
  {
  }
}

#endregion

// Occurs before the form region is displayed.
// Use this.OutlookItem to get a reference to the current
// Outlook item. Use this.OutlookFormRegion to get a reference
// to the form region.
private void FormRegion2_FormRegionShowing(object sender,
  System.EventArgs e)
{
  this.olkCommandButton1.Click += new
    Microsoft.Office.Interop.Outlook.
    OlkCommandButtonEvents_ClickEventHandler(
    olkCommandButton1_Click);
  this.olkCommandButton2.Click += new
    Microsoft.Office.Interop.Outlook.
    OlkCommandButtonEvents_ClickEventHandler(
    olkCommandButton2_Click);

  task = this.OutlookItem as Outlook.TaskItem;
  if (task != null)
  {
    // Check for custom property SubTasks
    subTasks = task.UserProperties.Find("SubTasks", true);
    if (subTasks == null)
    {
```

```
        subTasks = task.UserProperties.Add("SubTasks",
          Outlook.OlUserPropertyType.olText, false,
          Outlook.OlUserPropertyType.olText);
      }
    }

    // Convert string
    string subTasksString = subTasks.Value.ToString();
    if (!String.IsNullOrEmpty(subTasksString))
    {
      string[] delimiters = new string[1];
      delimiters[0] = System.Environment.NewLine;
      string[] tasks = subTasksString.Split(delimiters,
        StringSplitOptions.RemoveEmptyEntries);
      for (int i = 0; i < tasks.Length; i++)
      {
        olkListBox1.AddItem(tasks[i], i);
      }
    }
  }

  // Occurs when the form region is closed. Use this.OutlookItem
  // to get a reference to the current Outlook item. Use
  // this.OutlookFormRegion to get a reference to the form
  // region.
  private void FormRegion2_FormRegionClosed(object sender,
    System.EventArgs e)
  {
    if (subTasks == null || task == null)
      return;

    string oldTasks = subTasks.Value.ToString();
    StringBuilder builder = new StringBuilder();

    for (int i = 0; i < olkListBox1.ListCount; i++)
    {
      string t = olkListBox1.GetItem(i);
      if (!String.IsNullOrEmpty(t))
      {
        builder.AppendLine(t);
      }
    }

    string newTasks = builder.ToString();

    if (!String.IsNullOrEmpty(newTasks) &&
      !String.IsNullOrEmpty(oldTasks))
    {
```

```
        if (newTasks.CompareTo(oldTasks) == 0)
          return; // no changes
      }

      subTasks.Value = newTasks;
      task.Save();
    }

    // New Button
    void olkCommandButton1_Click()
    {
      // Just add current time as a subtask for simplicity
      this.olkListBox1.AddItem(
        System.DateTime.Now.ToShortTimeString(),
        this.olkListBox1.ListCount);
    }

    // Delete button
    void olkCommandButton2_Click()
    {
      if (this.olkListBox1.ListIndex != -1)
      {
        olkListBox1.RemoveItem(olkListBox1.ListIndex);
      }
    }
  }
}
```

Outlook Form Region Programmability

In this section, we examine in more detail the Outlook form region classes that VSTO creates. As with many other VSTO project items, it uses partial classes to display user code (code edited by the developer of the add-in) and associate it with generated code (code generated by Visual Studio) to build the final Outlook form region class.

The VSTO Form Region Class

In the example from the first section of this chapter, right-clicking FormRegion1.cs and choosing View Code from the context menu shows the user code:

```
partial class FormRegion1
```

You can also see the generated code by expanding FormRegion1.cs and double-clicking the file FormRegion1.Designer.cs, which is a child of

FormRegion1.cs. Here, you see this line of code, showing that a VSTO form region class derives from Microsoft.Office.Tools.Outlook.FormRegionControl:

```
partial class FormRegion1 :
  Microsoft.Office.Tools.Outlook.FormRegionControl
```

Then, if you look at the definition of FormRegionControl, you see that it derives from System.Windows.Forms.UserControl. A form region class is primarily a Windows Forms UserControl with extensions like the implementation of the IFormRegion interface, which is used by VSTO to start and shut down a form region:

```
public class FormRegionControl : UserControl, IFormRegion
```

When you import an .OFS file, the form region class derives from Microsoft.Office.Tools.Outlook.ImportedFormRegion.

The Form Region Factory

The form region factory is mostly an internal implementation detail of how VSTO supports Outlook form regions. You can do some advanced things with custom form region factories, such as having your form region classes in a separate assembly from the add-in or having a single factory to handle multiple form region classes. But outside these advanced scenarios, the form region factory does peek through in one significant way: It exposes the FormRegionInitializing method, which can be handled in your code to prevent a form region from being displayed for a particular Outlook item based on criteria you set. As you might expect, the Factory object creates an instance of your form region class every time an Outlook item requires a form region to be displayed. If you have an Adjacent form region that displays in the reading pane for a list of mail items, for example, each time the selection changes to a different mail item, a new instance of your form region class is created for the current mail item. The factory object is invoked first, and if the FormRegionInitializing method doesn't cancel the creation of the form region by the implementation of the method setting the Cancel property of the `e` parameter to `true`, a new instance of the form region class is created.

When you import an .OFS file, the form region factory also has a method called InitializeManifest in which you can write code to modify settings for the form region, such as the form region type. With Windows Forms-based form regions, you typically modify these form region settings in the Properties window, and no InitializeManifest method is in the form region factory.

Another key element of the form region factory class is the FormRegionMessageClass attribute, which sets the message classes—both built-in and custom—the form region will be displayed for. Listing 16-6 shows the attributes of a form region factory class associated with three built-in message classes (Appointment, Contact, and Note) and one custom message class that derives from Task (IPM.Task.Foo). VSTO provides constant strings in the Microsoft.Office.Tools.Outlook.FormRegionMessageClassAttribute namespace for each of the built-in Outlook message classes. If you want, you can interchange the constant string. The `string` "IPM.Task", for example, is equivalent to the `Microsoft.Office.Tools.Outlook.FormRegionMessageClassAttribute.Task` constant.

Listing 16-6: A Form Region Factory Class Associated with Three Built-In Message Classes and One Custom Message Class via FormRegionMessageClass Attributes

```
[Microsoft.Office.Tools.Outlook.FormRegionMessageClass(
  Microsoft.Office.Tools.Outlook.
  FormRegionMessageClassAttribute.Appointment)]
[Microsoft.Office.Tools.Outlook.FormRegionMessageClass(
  Microsoft.Office.Tools.Outlook.
  FormRegionMessageClassAttribute.Contact)]
[Microsoft.Office.Tools.Outlook.FormRegionMessageClass(
  Microsoft.Office.Tools.Outlook.
  FormRegionMessageClassAttribute.Note)]
[Microsoft.Office.Tools.Outlook.FormRegionMessageClass(
  "IPM.Task.Foo")]
[Microsoft.Office.Tools.Outlook.FormRegionName(
  "OutlookAddIn2.FormRegion3")]
public partial class FormRegion3Factory
```

The Manifest Object

Most of the properties that control how a VSTO form region is displayed by Outlook are found in the FormRegionManifest object returned by the Manifest property of a VSTO form region. Behind the scenes, setting properties on the FormRegionManifest object manipulates an XML manifest

that describes the form region. This manifest is provided to Outlook when the add-in loads. You can modify the properties of the FormRegionManifest object via the property grid for Windows Forms-based form regions by clicking the form region surface in the designer and then using the Properties window to set properties associated with the Manifest property, as shown in Figure 16-30.

To set properties of the manifest object for a Outlook Forms-based form region, write code in the InitializeManifest method of the form region fac-

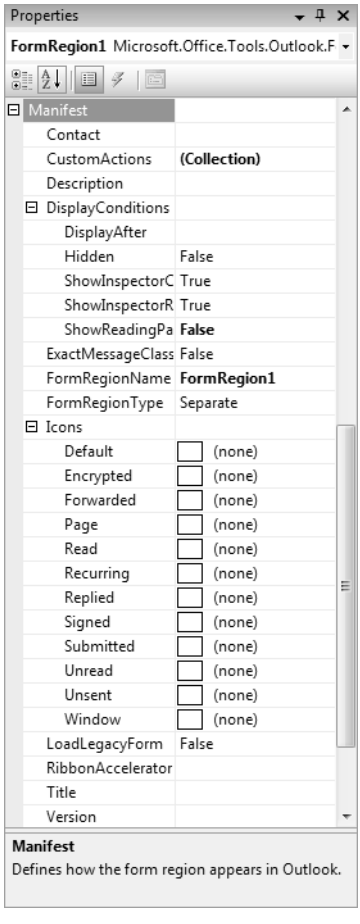

Figure 16-30: Setting properties on the manifest object for a Windows Forms-based form region.

tory. If you try to set properties on the manifest object outside these two mechanisms (the Properties window for Windows Forms and the InitializeManifest method for Outlook Forms), chances are that Outlook will already have asked for the manifest object's settings, and your code will generate an InvalidOperationException. You can use the Locked property of the manifest object to check whether Outlook has already retrieved the settings from the manifest object. If Locked is set to `true`, any code you write against the manifest object will have no effect.

Table 16-4 describes the various properties of the manifest object. The table refers several times to the Choose Form dialog box, which you invoke by dropping down the New button in the Explorer window and choosing Choose Form, as shown in Figure 16-31. The dialog box shown in Figure 16-32 appears, allowing you to pick Replacement or Replace-All form regions (which are associated with custom message classes). This way, an end user can create an Outlook item with a custom message class that you defined and associated with a form region. Table 16-5 describes the icons that can be used by a form region.

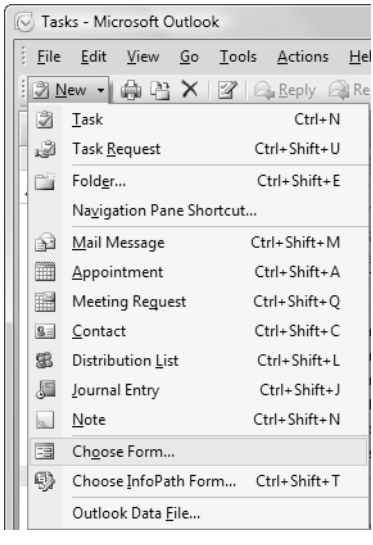

Figure 16-31: Creating a new Outlook Item by using the Choose Form button.

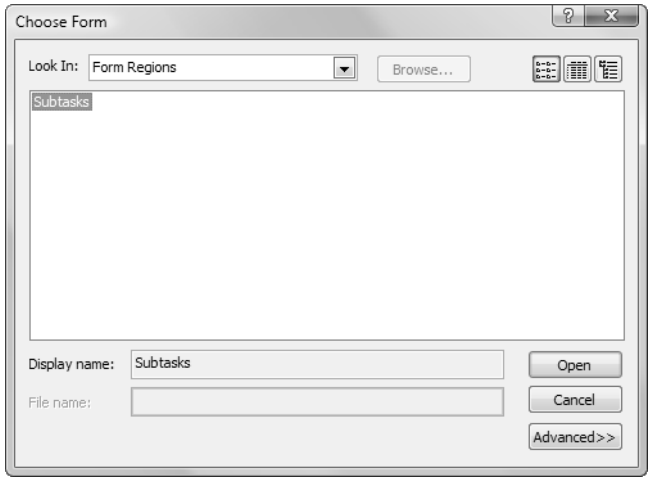

Figure 16-32: The Choose Form dialog box.

TABLE 16-4: Properties of the Manifest Object

Name	Type	What It Does
Contact	`string`	Gets and sets the name used in the Choose Form dialog box for Replacement and Replace-All form regions.
CustomActions	FormRegion-CustomAction-Collection	A collection with custom actions associated with the form region. The custom actions appear in a Custom Actions group in the Ribbon of the Inspector window showing the form region.
Description	`string`	Gets and sets the description used in the Choose Form dialog box for Replacement and Replace-All form regions.
DisplayAfter	`string`	Gets or sets the name of the form region to display before this form region.
ExactMessage-Class	`bool`	If set to `true`, this property prevents a custom message class derived from the message class for the form region from displaying the form region.

TABLE 16-4: Properties of the Manifest Object *(Continued)*

Name	Type	What It Does
FormRegionName	`string`	Gets and sets the name used for the Ribbon button associated with this form region or the header associated with an Adjoining form region.
FormRegionType	FormRegionType	Gets and sets an enum specifying the form region type: Adjoining, Replacement, Replace-All, or Separate.
Hidden	`bool`	If set to `true`, this form region won't be displayed in the Choose Form dialog box.
Icons	FormRegionManifestIcons	Sets the icons used by the form region (see Table 16-5).
Locked	`bool`	Returns `true` if Outlook has already queried the manifest object for its settings. Any code you write against the manifest object after Locked is set to `true` has no effect.
RibbonAccelerator	`string`	Gets and sets the keyboard shortcuts for Separate, Replacement, and Replace-All form regions.
ShowInspectorCompose	`bool`	Gets and sets whether the form region is shown when an Inspector window is in compose mode.
ShowInspectorRead	`bool`	Gets and sets whether the form region is shown when an Inspector window is in read mode.
ShowReadingPane	`bool`	Gets and sets whether a form region is shown for the reading pane.
Title	`string`	Gets and sets the name that appears in the Actions menu and the Choose Form dialog box for Replacement and Replace-All form regions.

TABLE 16-5: Icons on the Manifest Object

Name	What It Does	Applies to Form Region Types
Default	16 × 16-pixel icon that is used by default.	Replacement and Replace-All
Encrypted	16 × 16-pixel icon for encrypted items.	Replacement and Replace-All
Forwarded	16 × 16-pixel icon for forwarded items.	Replacement and Replace-All
Page	Icon used in the Ribbon of an Inspector window for the button that activates the form region. Use a PNG file for this icon.	Separate, Replacement, and Replace-All
Read	16 × 16-pixel icon for read items.	Replacement and Replace-All
Recurring	16 × 16-pixel icon for recurring items.	Replacement and Replace-All
Replied	16 × 16-pixel icon for replied-to items.	Replacement and Replace-All
Signed	16 × 16-pixel icon for digitally signed items.	Replacement and Replace-All
Submitted	16 × 16-pixel icon for items in the Outbox that are submitted for sending.	Replacement and Replace-All
Unread	16 × 16-pixel icon for unread items.	Replacement and Replace-All
Unsent	16 × 16-pixel icon for items in the Drafts folder that are not yet sent.	Replacement and Replace-All
Window	Appears in the notification area and in the Alt+Tab window for Inspector windows displaying the form region. Use a 32 × 32-pixel icon.	Replacement and Replace-All

Other Key Properties and Methods

Several other properties and methods associated with a Outlook form region class created with VSTO are worth pointing out. We've already talked about the OutlookItem property, which returns as an `object` the Outlook Item associated with the Outlook form region. You can cast the `object` returned by the OutlookItem property to the type of Outlook item you expect, based on what built-in or custom message classes your form region is associated with.

The OutlookFormRegion property returns the underlying Microsoft .Office.Interop.Outlook.FormRegion object, which represents your form region in the Outlook object model. Table 16-6 shows some of the key properties and methods on this object.

Globals Support

Whenever you create a form region in a VSTO project, it is added to the Globals object for the project. You can access the currently active form

TABLE 16-6: Key Properties and Methods on the FormRegion Object Returned by the OutlookFormRegion Property

Name	Type	What It Does
Detail	`string`	Gets and sets the name displayed in the header after the display name of an Adjoining form region.
Inspector	Inspector	Returns the Inspector window object associated with the form region.
IsExpanded	`bool`	Read-only property that returns `true` when an Adjoining form region is expanded.
Language	`int`	Returns the locale ID (LCID) for the current language used by Outlook.
Reflow()		Method that forces Outlook to size an Adjoining form region so that all controls are visible.
Select()		Makes the form region the active form region and forces it to be visible.

regions in three ways: You can see all the active form regions for a particular Inspector window, for a particular Explorer window, or for all open windows. You can access only form regions provided by your add-in; you can't access form regions provided by other add-ins by using the Globals object.

Listing 16-7 shows a subroutine that uses all three ways of accessing active form regions to get the count of active form regions associated with the active Explorer window, the count of active form regions associated with the active Inspector window, and the total count of all active form regions.

Listing 16-7: Three Methods of Accessing Active Form Regions: All Active, for an Explorer Window, and for an Inspector Window

```
private void ShowActiveFormRegions()
{
  Outlook.Explorer explorer =
    Globals.ThisAddIn.Application.ActiveExplorer();
  Outlook.Inspector inspector =
    Globals.ThisAddIn.Application.ActiveInspector();

  System.Windows.Forms.MessageBox.Show(
    String.Format("{0} total form regions",
    Globals.FormRegions.Count.ToString()));

  if (explorer != null)
    System.Windows.Forms.MessageBox.Show(
      String.Format("{0} for regions for the active Explorer",
      Globals.FormRegions[explorer].Count.ToString()));

  if (inspector != null)
    System.Windows.Forms.MessageBox.Show(
      String.Format("{0} for regions for the active Inspector",
      Globals.FormRegions[inspector].Count.ToString()));
}
```

Conclusion

With the new form region feature in Outlook 2007, developers have a powerful new way to customize the Outlook UI. The four types of form regions—Separate, Adjacent, Replacement, and Replace-All—provide a wide variety of UI options. In this chapter, you saw how form regions can be created to customize Inspector windows as well as the reading pane.

You also learned about custom message classes, as they are required for using the Replacement and Replace-All form region types.

VSTO supports two form technologies to create form regions with: Windows Forms and Outlook Forms. You saw how Outlook Forms can be created and imported into Visual Studio and how the controls are accessed from managed code. You also dived deeper into the programming model to discover additional ways to customize form regions.

Also in this chapter you saw a simple way in which form regions integrate with the Ribbon: They automatically add a button to switch to the form region page, for example.

In the next chapter, you see how VSTO supports creating a wider range of Ribbon customizations.

Index

C

D

M

P

S

T

U

X

Z