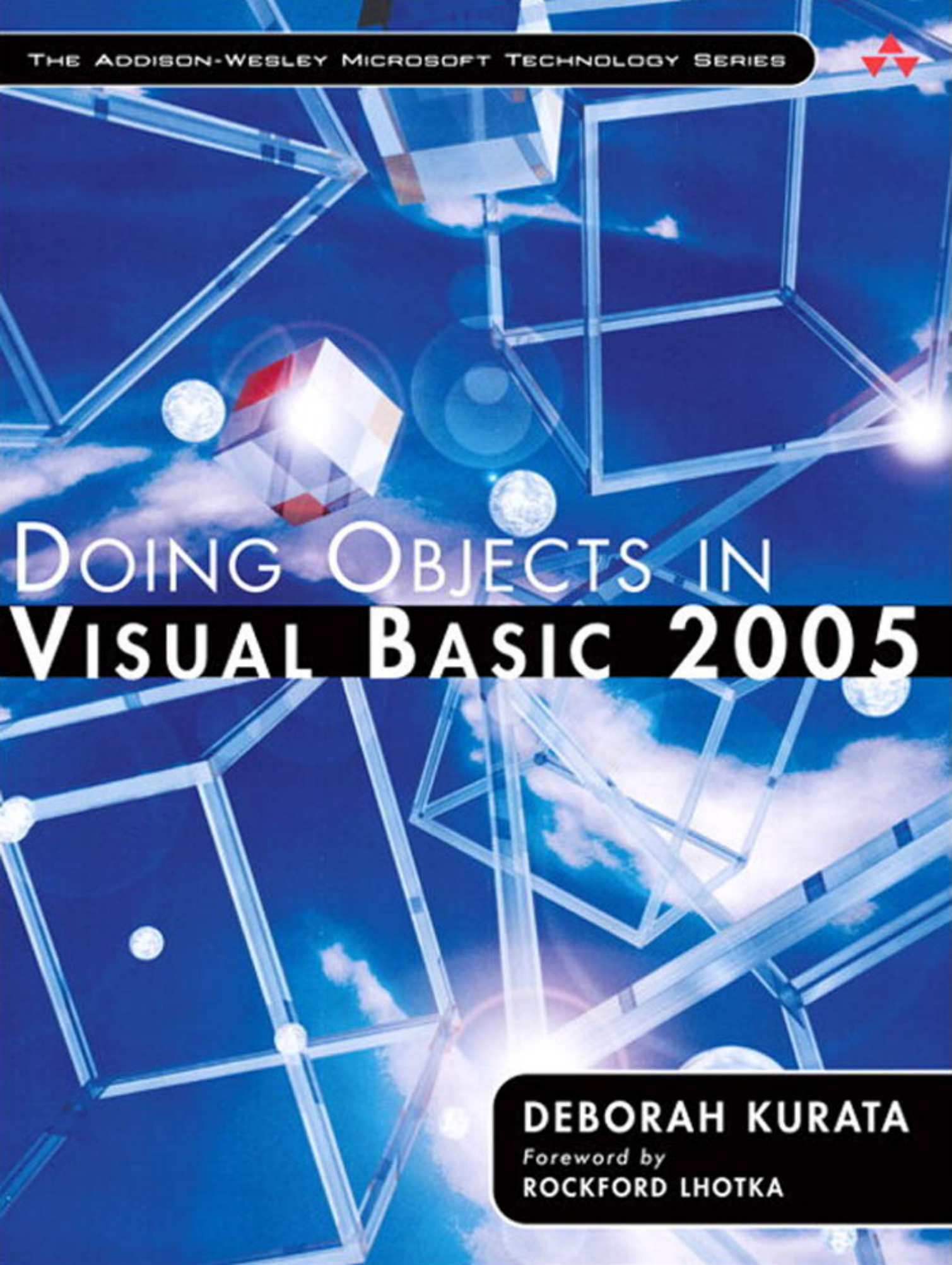
THE ADDISON-WESLEY MICROSOFT TECHNOLOGY SERIES



DOING OBJECTS IN VISUAL BASIC 2005

DEBORAH KURATA

Foreword by
ROCKFORD LHOTKA

Praise for *Doing Objects in Visual Basic 2005*

"*Doing Objects in Visual Basic 2005* is one of the few books that I've seen that lays the proper object-oriented foundation to make new Visual Basic.NET developers as well as VB6 veterans successful in moving to the .NET Framework."
—Paul Ballard, *President, Rochester Consulting Partnership, Inc.*

"Deborah Kurata's *Doing Objects in Visual Basic 2005* is salvation for every programmer beached on the forbidding isle of .NET object-oriented programming. 'Right this way,' she says, leading you confidently into that vaguely menacing interior. Step follows step. Suddenly the daunting and unfamiliar become doable and commonplace. You're productive again. My goodness, you're actually enjoying yourself!"
—Ward Bell, *V.P., Product Management, IdeaBlade, Inc.*

"When is comes to advice on programming objects in Visual Basic, nobody could be better qualified than Deborah Kurata. She's been doing *Doing Objects* since VB4, and she doesn't let us down as we move our classic VB code to the Microsoft.NET platform. From initial analysis and design, through to the final implementation, you'll find everything you need here to take on the Visual Basic 2005 development environment. This book is a must have for every VB.NET developer!"
—Kel Good, *MCT, MCITP, MCPD, Custom Software Development Inc., (www.customsoftware.ca)*

"I've long been frustrated that I couldn't recommend a book on object-oriented fundamentals in .NET. Sure, there were plenty of books on OO syntax. But what good is explanation of syntax if you don't already understand the concepts? At last, we have the successor to the *Doing Objects* series for classic VB, from which so many of us learned how to think about objects and object design, completely rewritten for .NET. If you're a VB 2005 developer who needs to add object orientation to your skill set, this is the book you need."
—Billy Hollis, *author/consultant, Next Version Systems*

"Deborah Kurata does her *Doing Objects* thing again! This is the newest book from Deborah which has been completely rewritten from the ground up for the Visual Basic .Net 2005 developer. Anyone needing a solid foundation in object technology, Visual Basic .Net 2005, and Visual Studio .Net 2005 should read this book. Deborah's presentation of core topics such as class design, object state management, exception handling, events, data binding, validation, data access techniques, and many others is clear, concise, and direct. The clarity of the content, coupled with the hands-on examples make this book an easy read and a must have."
—Ron Landers, *Senior Technical Consultant, IT Professionals, Inc.*

6. Select **(My Application Events)** from the Class Name drop-down at the top left of the Code Editor.
7. Select **UnhandledException** from the Event drop-down at the top right of the Code Editor.
   The following event handler code lines are generated:

```
Private Sub MyApplication_UnhandledException( _
  ByVal sender As Object, _
  ByVal e As Microsoft.VisualBasic. _
  ApplicationServices.UnhandledExceptionEventArgs) _
  Handles Me.UnhandledException

End Sub
```

NOTE: Chapter 4 provides detailed information on building event handlers.

8. Add code to handle the unhandled exception within the event handler.

Any exception that is not handled anywhere else in the application is handled by this code. Most commonly, this code should log the error and display a message to the user.

NOTE: The `UnhandledException` event *never* occurs when you are running within Visual Studio. To test the code in this event, you must run the application outside of Visual Studio by running the executable in the bin directory for the Windows Application project.

With the new `UnhandledException` application event, you can now ensure that your application catches any unexpected exception.

Referencing Projects in a Solution

Each project within a solution is an independent component. You cannot access code in one project from another project without first setting a reference from the one project to the other project. For example, you cannot

call the business logic project from the user interface project unless the user interface project has a reference to the business logic project.

To add a reference to a project:

1. Select the project in Solution Explorer.
   This is the project that needs to reference another project. For example, if the user interface Windows Application project needs to access the business object Class Library project, select the Windows Application project.
2. Open the Project Designer for the project by right-clicking the project and selecting **Properties** from the context menu, *or* select **Project | Properties** from the main menu bar, *or* double-click on the **My Project** folder under the project.
3. Select the **References** tab.
   The References tab of the Project Designer appears, as shown in Figure 3.16.

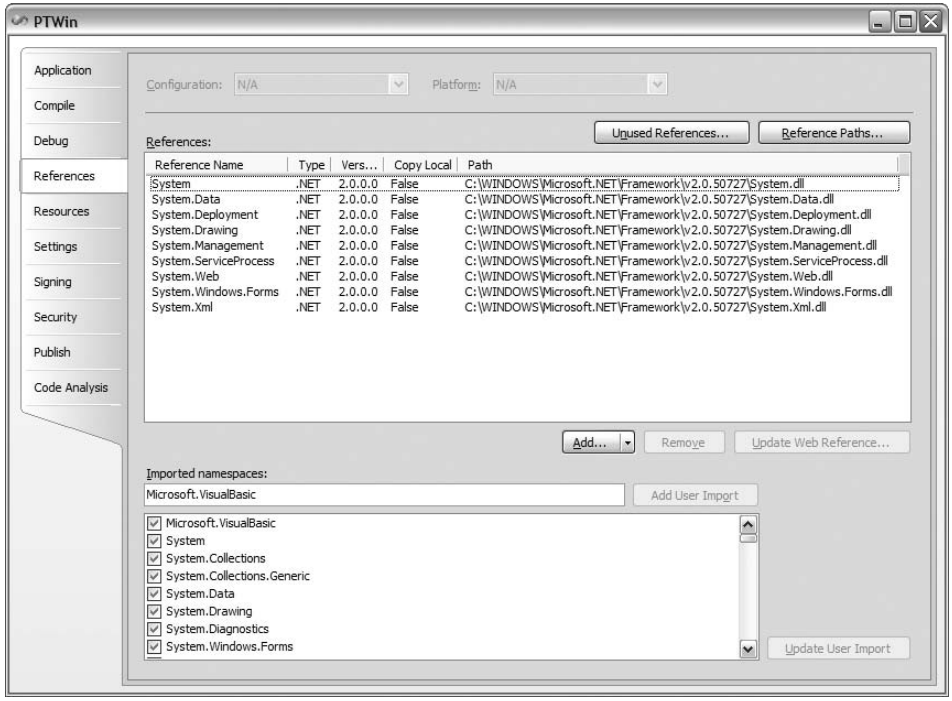


Figure 3.16 The References tab of the Project Designer allows you to define the project references and imports.

4. Click the **Add** button.
   The Add Reference dialog is displayed. This dialog allows you to set a reference to other .NET components, COM components, or other projects within your solution.
5. Click the **Projects** tab.
   The Add Reference dialog Projects tab is shown in Figure 3.17.

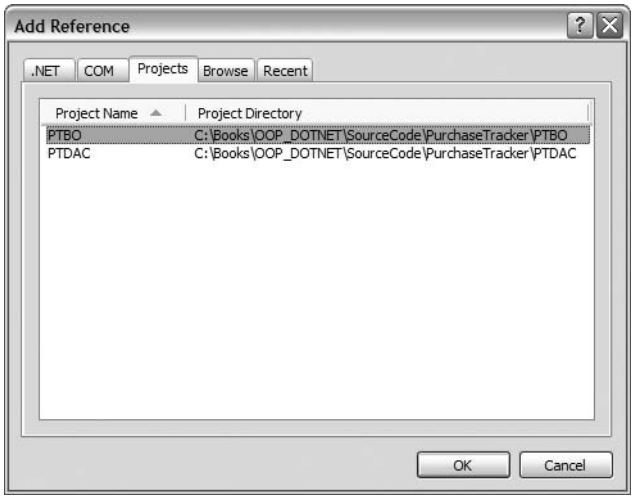


Figure 3.17 The Projects tab of the Add Reference dialog allows you to reference other projects within the solution.

6. Select one or more projects to reference, and click **OK**.
   The project you selected in step 1 can then access classes in the projects selected in this step. For example, select PTBO to allow the project selected in step 1 to access the classes in the business object component. At this point, you can access the classes in the PTBO component using the fully qualified namespace and class name.
   Normally, the user interface component accesses the business logic component, and the business logic component accesses the data access component. So you would set a reference from the user interface project to the business object project and from the business object project to the data access project.
   The selected project(s) are then added to the set of references and to the Imported namespaces list, as shown in Figure 3.18.
7. Optionally, select the project(s) in the **Imported namespaces** list to check them.

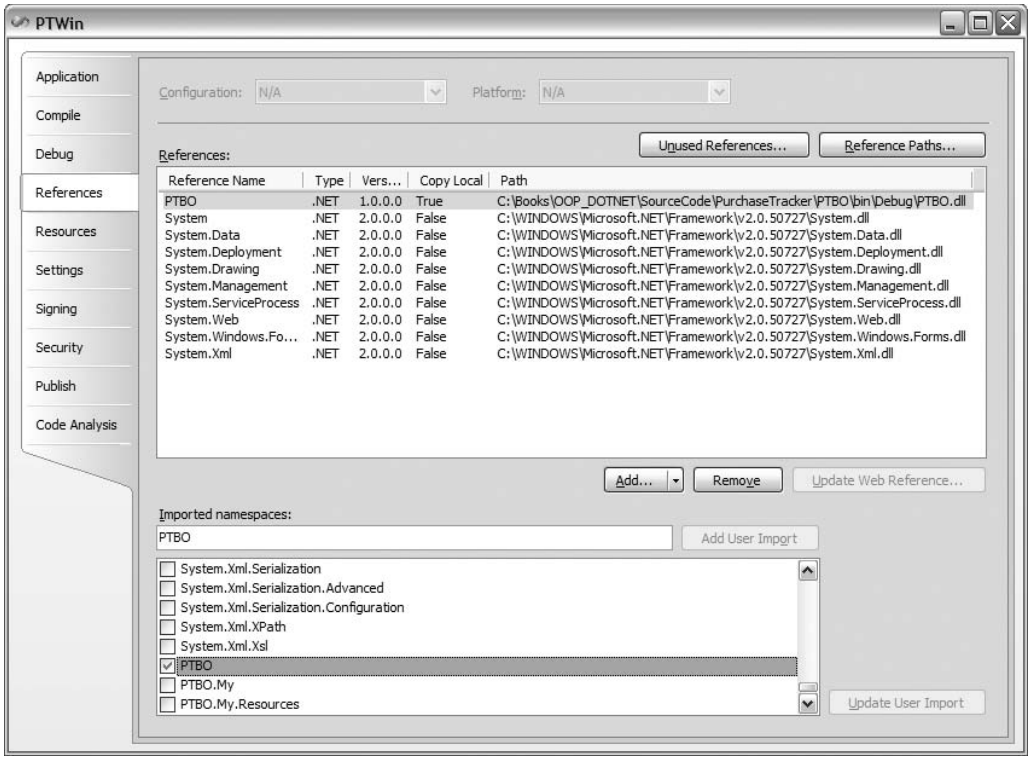


Figure 3.18 Check the projects in the Imported namespaces list to make it easier to access the classes within the project.

NOTE: A problem with Visual Studio sometimes requires that you click the checkbox twice to check the imported namespace.

If you import the namespace for a component, you can access the classes in that component without the fully qualified namespace.

When you set a reference to a component, as shown in step 6, you can access a class in the component by typing the namespace of the component and the name of the class, as follows:

```
PTBO.Customer
```

Visual Studio can find the class using this fully qualified name. But if you have a long namespace or an extensive namespace hierarchy, typing the fully qualified name can get tedious. (See the "Root Namespace" section for more information on namespaces and namespace hierarchies.) It would be much easier if you could just type the name of the class, like this:

```
Customer
```

This is possible if you import the namespace, as shown in step 7. When you import a namespace, Visual Studio can find any class in the namespace without requiring that you qualify the class name with the namespace.

When you type a class name into your code without the fully qualified namespace, Visual Studio resolves the class name by looking through the imported namespaces. If the class is found in one and only one namespace, all is well. If the class does not belong to one of the checked namespaces, or if the same class name is found in multiple namespaces, Visual Studio displays an error.

When you import a namespace using the Project Designer, as described in step 7, you import the namespace for all files in the project. It is also possible to import a namespace for specific project files by using the `Imports` keyword.

To import a namespace in one code file, type the `Imports` statement at the top of the code file before any other declarations. For example, to import the `InStep.PurchaseTracker.AccountingInterface` namespace, use the following `Imports` statement:

```
Imports InStep.PurchaseTracker.AccountingInterface
```

You can then use the name of any class in that namespace without fully qualifying the name, but only within the code file containing the `Imports` statement.

You can also use the Imported namespaces list to import .NET Framework namespaces for a project. Notice in Figure 3.16 that the `System.Diagnostics` namespace is checked by default. This allows you to access the `Debug` class by typing `Debug` instead of `System.Diagnostics.Debug`. You can check any other .NET Framework namespaces to import those as well.

Building Along

For the Purchase Tracker sample application:

- Add a reference from the Windows Application project (**PTWin**) to the business object Class Library project (**PTBO**).
  Be sure to check **PTBO** as an imported namespace.
- Add a reference from the business object Class Library project (**PTBO**) to the data access Class Library project (**PTDAC**).
  Be sure to check **PTDAC** as an imported namespace.

By setting references, you can access the business logic component from the user interface component and the data access component from the business logic component of your application when they are coded in later chapters.

Defining Visual Studio Settings

There is one last thing to do before you start coding your application, and that is defining your desired Visual Studio settings. **Settings** is the general name for all the options you can define using **Tools** | **Options** or set in other dialogs, such as the Find and Replace dialog.

Visual Studio provides a huge number of settings. You can define settings for your Visual Studio environment, your projects, how you want your text editor to look, and so on.

One of the most important settings at this point is your Visual Basic defaults (**VB Defaults**) under the **Projects and Solutions** node, as shown in Figure 3.19.

Option Explicit identifies whether each variable in your application must be declared. If this is set to Off when you create your project, you don't need to declare a variable before using it. If it is set to On when you create your project, you must declare every variable. By default, this is set to On, and it is highly recommended that this remain on.

For example, if Option Explicit is off, you can write code as follows:

```
x = "This is a test"
```

If Option Explicit is on, you must change the code to declare the variable before using it:

```
Dim x As String
x = "This is a test"
```