CHAPTER 5

Building .NET Managed Components for COM+

Modern software systems are component-based. Components make large, enterprise-wide software systems easier to manage since system functionality is divided among several components. VB.NET and all other .NET-supported languages have the ability to create these components, which can be used and reused in a variety of projects, including (but not limited to) ASP.NET projects, Windows applications, and unmanaged code (which is code that executes outside the .NET Framework).

This chapter covers the following topics:

- Managed code (the code that runs on the .NET Framework) and runtime environments
- The Common Language Runtime and its role in the .NET Framework
- Just-in-time compilation of managed code
- Code assemblies
- COM+ Component Services and its role in the .NET Framework
- Creation of managed components using VB.NET
- Serviced components, which take advantage of the services provided by COM+ Component Services

5.1 The Concept of Managed Code Execution

Before we get into making .NET components, we need to discuss a .NET concept called *managed code*. Managed code is any source code you compile that targets the .NET Framework. All the code we've examined so far in this book (ASP.NET, console applications, and so on) has been managed code.

The .NET Framework is a runtime environment in which code executes. Internal tasks, such as allocating and freeing memory, and software services (like the kinds discussed in Chapter 4) are handled by the .NET Framework as well. In general terms, a runtime environment enables user programs to take advantage of services provided by the host operating system. A runtime environment also supplies a layer of abstraction between user code and those services, through either an API or some other type of interface. Almost any program you write, regardless of the platform or language, involves interaction with a runtime environment. (An exception to this are programs written in assembly language, in which case the programmer is calling on the services of the microprocessor and memory storage in a direct, low-level fashion.)

Many programs written for the Windows platform use the C++ programming language and its runtime library. Before other languages and development systems became available, this was the only choice for developers making applications for Windows. The C/C++ runtime code shipped as a part of Windows as a series of dynamic link libraries (known as DLLs or `.dll` files). As a result, many Windows applications could be distributed with as little as one file, the `.exe` file that contained the main program code. Since no other files were required, many developers referred to these types of applications as *native code applications* since the preinstalled runtime code was quite small.

As new languages became available for developing Windows applications, new runtime environments needed to be developed. VB developers using version 6.0 or earlier might be aware of special support files that must be installed on the deployment computer in order for VB applications to run. These `.dll` files make up the VB runtime environment. Like the C/C++ runtime code mentioned above, it provides a code wrapper around operating system internals and services.

Some runtime environments provide an additional layer of abstraction over another existing runtime environment. This creates an execution environment that can exist on multiple platforms. Programs targeted to such a runtime don't compile to machine object code. Instead, they compile to another language, referred to as an *intermediate language*. The runtime then executes this intermediate language using an engine built for a particular operating system. The most popular type of intermediate language system is Java. The intermediate language for Java is referred to as *bytecode*. The .NET Framework works in a similar manner. Compiled Microsoft .NET code is referred to as *Microsoft Intermediate Language* (MSIL).

5.2 The Common Language Runtime

The runtime environment for the .NET Framework is called the *Common Language Runtime* (CLR). Managed code execution happens inside the CLR space. The goal of the CLR is to provide an environment that includes language integration, exception handling, security, versioning, deployment, debugging, profiling, and component interaction. Most importantly, all of these features need cross-language support. In other words, all the features mentioned must work in the same manner regardless of the language used.

Metadata makes cross-language integration possible. When you compile .NET managed code, the metadata gets stored along with the object code. Metadata describes to the CLR various types of information (for example, data types, members, and references) used in the code. This data is used by the CLR to "manage" the code execution by providing such services as memory allocation, method invocation, and security enforcement. It also eases deployment since references to other objects are included along with metadata. This ensures that your application contains all up-to-date versions of all dependent components.

5.2.1 The Common Type System

Along with providing information about managed code through metadata, the CLR implements a series of data types that are cross-language compatible. That system of data types is known as the Common Type System (CTS). CTS data types include simple value types, classes, enumerated value types, interfaces, and delegates.

Simple value data types include primitive types as well as user-defined types. Primitive types include integers, Boolean values, and strings. These types are included in the `System` namespace. The data types used thus far in the VB.NET programs in this book are also included in this namespace. When you use these primitive types in your programs, the language you use may already have an equivalent native data type that corresponds to a .NET Framework type. Table 5-1 shows some data types and their VB.NET native language equivalents.

Occasionally you may want to define your own data types. You can do this by using the native features of the language with which you're working. If you're using VB.NET, you can use the `Structure` statement to define a structure. This custom type needs to be type-safe for the CLR, so it's no coincidence that it inherits from the `ValueType` .NET class.

Table 5-1 Primitive Data Types and VB.NET Equivalents

.NET Data Type Class Name (`System` Namespace)	VB.NET Data Type
`Byte`	`Byte`
`SByte`	*Not supported*
`Int16`	`Short`
`Int32`	`Integer`
`Int64`	`Long`
`UInt16`	*Not supported*
`UInt32`	*Not supported*
`UInt64`	*Not supported*
`Single`	`Single`
`Double`	`Double`
`Object`	`Object`
`Char`	`Char`
`String`	`String`
`Decimal`	`Decimal`
`Boolean`	`Boolean`

TECHTALK: STRUCTURES AND CLASSES
Structures and classes are quite similar. They both have members, constructors, events, properties, fields, and constants. They can both implement interfaces as well.
There are some differences. Structures don't allow for inheritance; therefore they are referred to as *sealed*.

Structures can't have any constructor code, that is, you can't define an overloaded `New()` subroutine with your own initialization code like you can with a class.

When structures are used in procedure calls, the structure is passed by value (like primitive data types are). Classes, on the other hand, are always passed by reference. Whether to use classes or structures for your user-defined data types depends mostly on the complexity of your data types. Structures are useful for defining relatively simple data types for which the members do not use much memory and no custom initialization code is required. If your design requires more than this, consider defining your custom data type as a class.

Let's quickly look at a VB.NET structure definition. Structure definitions are placed outside of procedure definitions. You can define them at the module level, as shown below.

Code Sample 5-1 Build instructions: `vbc /r:system.dll cs5-01.vb`

```
Module Module1

❶ Structure Student
    Dim FirstName As String
    Dim LastName As String
    Dim SSN As String
    Dim ClassRank As Integer
  End Structure

  Sub Main()

    Dim udtStudent As New Student()

❷   With udtStudent
      .FirstName = "Matt"
      .LastName = "Crouch"
      .SSN = "888-88-1234"
      .ClassRank = 2
    End With
```

```
   End Sub

End Module
```

The example shows a typical structure definition. The members are enclosed in the `Structure . . . End Structure` block that begins in line ❶. The VB.NET `With . . . End With` statement in line ❷ is used to save some typing. It allows you to refer to the individual members of the structure without fully qualifying the names of the structure members.

Since the VB.NET structures you define automatically inherit from `System.ValueType`, you can treat the value type as a `ValueType` object. As a demonstration, let's create a function that lists all the members of an arbitrary structure at runtime.

Code Sample 5-2 Build instructions: `vbc /r:system.dll cs5-02.vb`

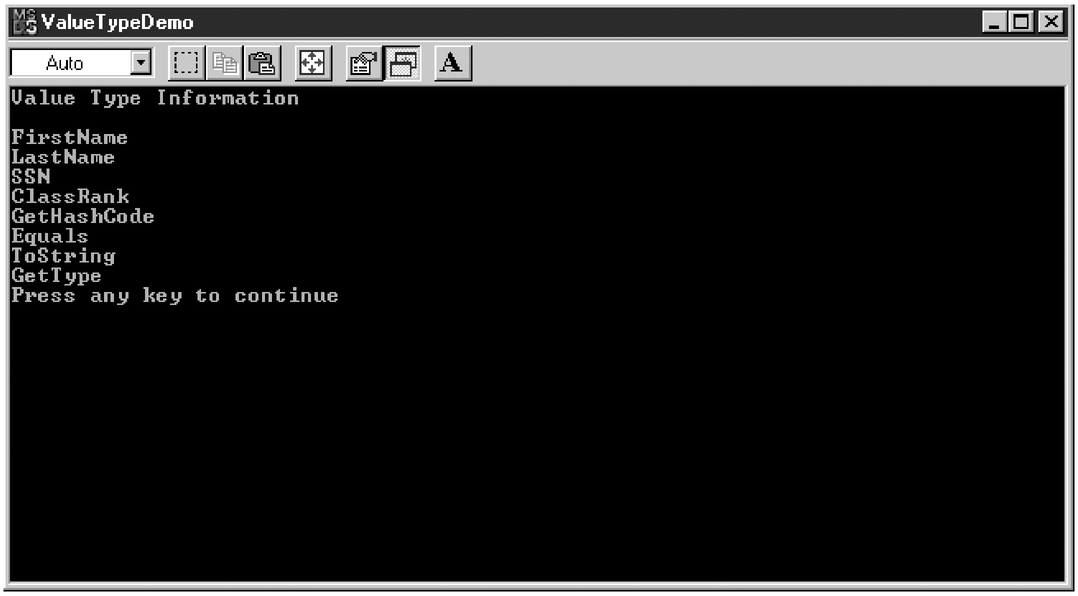
```
Public Sub ValueTypeDemoFunction(ByVal udt As ValueType)

   Dim mi() As MemberInfo
   Dim srmMemberInfo As MemberInfo

   Dim typTmp As Type
❸  typTmp = udt.GetType(udt.ToString())
❹  mi = typTmp.GetMembers()

   Console.WriteLine("Value Type Information" & _
                     Chr(13) & Chr(10))

   For Each srmMemberInfo In mi
     Console.WriteLine(srmMemberInfo.Name)
   Next

End Sub
```

The function `ValueTypeDemoFunction()` takes a `ValueType` object as its parameter. Thus we can pass a VB.NET structure to this function. The `GetType()` function in line ❸, which is a member of the `System.Object` namespace, returns a `System.Type` object. We use the returned `System.Type` object in line ❹ to get the member names (an array of `System.Reflection.MemberInfo` objects).

Crouch05_393-458.ps 4/23/02 5:59 PM Page 399

If we modify Code Sample 5-1 to add a call to `ValueTypeDemoFunction()` as shown below in boldface text, we'll obtain output similar to Figure 5-1.

```
Sub Main()

   Dim udtStudent As New Student()

   With udtStudent
     .FirstName = "Matt"
     .LastName = "Crouch"
     .SSN = "888-88-1234"
     .ClassRank = 2
   End With

   ValueTypeDemoFunction(udtStudent)

End Sub
```



```
Value Type Information

FirstName
LastName
SSN
ClassRank
GetHashCode
Equals
ToString
GetType
Press any key to continue
```

Figure 5-1 Output of the ValueType example

5.2.2 Just-in-Time Code Compilation

Managed code cannot be executed directly by the CPU. It must be converted to native executable code before running. Just-in-time (JIT) compilation compiles MSIL code right at the moment it is needed. Optimizations exist in the JIT compiler to ensure that only code planned for execution gets compiled. The JIT compiler also performs security checks and verifies type-safety.

5.2.3 Code Assemblies

I've mentioned the topic of component-based system architectures before, and now it's time to introduce the .NET Framework concept of this idea. The component (that is, the unit of reuse) in the .NET Framework is the *assembly*. An assembly is a collection of files, typically `.dll` files and any others relating to the assembly, such as resource files. The assembly *manifest* contains metadata relating to version information, security attributes, and external code references. It also contains information on how the pieces in the assembly relate to each other. The assembly manifest, therefore, constructs a logical DLL around the assembly elements.

5.2.4 Application Domains

Modern software systems run applications that are isolated from the internal execution of the operating system and other programs. The reason for this is to protect the operating system from crashing if the application attempts to access memory being used by another application. Another situation that could cause a crash is an internal error in the application that causes the operating system to crash. Fortunately, all versions of Windows offer process protection to prevent this problem. Each application running under Windows has its own memory space, and memory used by other running applications is not visible from any other application.

The .NET Framework extends the capabilities of protected process spaces by building this functionality into the CLR. These protected spaces are known as *application domains*. In addition to the fault tolerance that process isolation provides, application domains can enforce security policies, thereby granting or denying users and groups the right to run the application. Application domains also consume fewer system resources than traditional Windows processes because they can provide fault tolerance by taking advantage of the inherit type-safety of the .NET Framework code.

Crouch05_393-458.ps 4/23/02 5:59 PM Page 401

5.3 COM+ Component Services

The .NET Framework leverages many existing Windows services to make it a more robust application environment. A particular technology that deserves attention is COM+ Component Services. These technologies were the predecessors to the .NET Framework. To see how COM+ Component Services fits into the .NET Framework arena, let's explore a little about these technologies.

5.3.1 Overview of COM

The Component Object Model (COM) was designed to address the shortcomings of conventional object-oriented languages like C++ and traditional binary software distribution of code. COM is about not a particular type of software but rather a philosophy of programming. This philosophy is manifested in the COM specification. The specification explicitly states how a COM object should be constructed and what behaviors it should have.

COM objects are roughly equivalent to normal classes, but COM defines how these objects interact with other programs at the binary level. By *binary*, I mean compiled code, with all the methods and member variables of the class already built into an object. This "binary encapsulation" allows you to treat each COM object as a "black box." You can call the black box and use its functionality without any knowledge of the inner workings of the object's implementation. In the Windows environment, these binary objects (COM objects) are packaged as either DLLs or executable programs. COM is also backed by a series of utility functions that provide routines for instantiating COM objects, process communication, and so on.

COM was the first methodology to address object-oriented software reuse. COM has enjoyed great commercial success; many third-party software vendors provide COM objects to perform a wide range of tasks, from e-mail to image processing. COM is also highly useful for creating components called *business objects*. Business objects are COM objects in the strict sense, but they are used to encapsulate business rules and logic. Typically these business objects are tied to database tables. The objects move around the database according to the business rules implemented in the COM object.

Generally, several smaller business objects work together to accomplish a larger task. To maintain system integrity and to prevent the introduction of erroneous data into the application, transactions are used. A software service called Microsoft Transaction Server (MTS) is used to manage these transactions. We'll cover the function of MTS (and its successor, COM+) in Section 5.3.3.

5.3.2 Overview of Transactions

Simply stated, a transaction is a unit of work. Several smaller steps are involved in a transaction. The success or failure of the transaction depends on whether or not all of the smaller steps are completed successfully. If a failure occurs at any point during a transaction, you don't want any data changes made by previous steps to remain. In effect, you want to initiate an "undo" command, similar to what you would do when using, say, a word processor. A transaction is *committed* when all steps have succeeded. A failed transaction causes a *rollback* to occur (the "undo" operation).

Well-designed transactions conform to *ACID* principles. ACID is an acronym for Atomicity, Consistency, Isolation, and Durability.

- *Atomicity* means that either the operation that the component performs is completely successful or the data that the component operates on does not change at all. This is important because if the transaction has to update multiple data items, you do not want to leave it with erroneous values. If a failure occurs at any step that could compromise the integrity of the system, the changes are undone.
- *Consistency* deals with preserving the system state in the case of a transaction failure.
- *Isolation* means that a transaction acts as though it has complete control of the system. In effect, this means that transactions are executed one at a time. This process keeps the system state consistent; two components executed at the same time that operate on the same data can compromise the integrity of the system.
- *Durability* is the ability of a system to return to any state that was present before the execution of a transaction. For example, if a hard drive crash occurs in the middle of a transaction, you can restore the original state from a transaction log stored on another disk to which the system recorded.

A classic example of a transaction operation is a bank transfer that involves a transfer of funds from one account to another (a credit and a debit). Such a transaction moves through the following steps.

1. Get the amount to be transferred, and check the source account for sufficient funds.
2. Deduct the transfer amount from the source account.
3. Get the balance of the destination account, and add the amount to be transferred to the balance.
4. Update the destination account with the new balance.

Suppose a system failure occurs at step 4. The source account had the transfer amount deducted but the amount was not added to the destination account. Therefore, the money from the source account gets lost. Clearly, this is not good because the integrity of the database has been damaged.

Each of the account transfer's steps can be checked for success or failure. If a failure occurs before all values have been updated, the program needs to undo the deduction made to the source account. That's a rollback. If every step succeeds, the program needs to apply all the changes made to the database. That's when a commit operation is performed.

5.3.3 Automatic Transactions

Transactions have been in widespread use since the early days of enterprise computing. Many database systems include internal support for transactions. Such database systems contain native commands to begin, abort, and commit transactions. This way, several updates to database data can be made as a group, and in the event of a failure, they can be undone. Using a database's internal transaction-processing system is referred to as *manual transaction processing*.

Automatic transactions differ from manual transactions because automatic transactions are controlled by a system external to the database management system (DBMS). Earlier versions of Windows (95/98/NT) provide automatic transaction services using Microsoft Transaction Server (MTS). MTS works by coordinating database updates made by COM components grouped into a logical unit called a *package*. An MTS package defines the boundary of the transaction. Each component in the package participates in the transaction. After a component performs a piece of work (such as updating the database), it informs MTS that it successfully (or unsuccessfully) performed its share of the transaction. MTS then makes a determination to continue based on the success of the last component's signal of success or failure. If the transaction step was unsuccessful, the transaction is aborted immediately, and MTS instructs the DBMS to undo any changes made to data. If the step was successful, the transaction continues with the other steps. If all steps execute successfully, MTS commits the transaction and tells the DBMS to commit changes to the data.

5.3.4 COM+ Applications

With the release of Windows 2000 came the next version of COM, dubbed *COM+*. COM+'s *raison d'être* is the unification of COM and MTS. COM+

also offers performance improvements over MTS by implementing technologies such as *object pooling*, which maintains an active set of COM component instances. Other performance-enhancing features include *load balancing*, which distributes component instances over multiple servers, and *queued components*, which uses Microsoft Message Queue Server to handle requests for COM+ components.

The services that were formerly provided by MTS are known as *COM+ Component Services* in the COM+ model. COM+ Component Services works in a similar manner to MTS. Packages are now referred to as *COM+ applications*. Participating transactional components are grouped into applications in the same way components were grouped into packages under MTS.

Individual COM+ components in an application can be assigned different levels of involvement in an automatic transaction. When setting up COM+ applications, each component can have the levels of automatic transaction support shown in Table 5-2.

Table 5-2 COM+ Automatic Transaction Support Levels

Transaction Support	Description
`Disabled`	No transaction services are ever loaded by COM+ Component Services.
`Not Supported`	This is the default setting for new MTS components. Execution of the component is always outside a transaction regardless of whether or not a transaction has been initiated for the component.
`Supported`	You may run the component inside or outside a transaction without any ill effects.
`Required`	The component must run inside a transaction.
`Required New`	The component needs its own transaction in which to run. If the component is not called from within a transaction, a new transaction is automatically created.

5.3.5 COM+ Security

Security is of paramount importance, especially for applications intended to run on the Internet. In the past, programming security features into an Internet application was largely a manual effort. Often it consisted of custom security schemes that did not necessarily leverage the existing security infrastructure provided by the operating system. Besides being difficult to maintain, such security systems are typically costly to develop.

COM+ Component Services provides a security infrastructure for applications that uses Windows 2000/XP users and groups. COM+ security is declarative, which means you designate which users and groups have permission to access a COM+ application. This is done by defining *roles* for application access.

A role is a defined set of duties performed by particular individuals. For example, a librarian can locate, check out, and shelve books. The person fulfilling the librarian role is permitted to perform such duties under the security policies defined for that role. An administrator is responsible for assigning users and groups to roles. The roles are then assigned to a COM+ application.

This *role-based security* is not only easy to implement (it can be done by the system administrator) but it also typically doesn't require the programmer to work on the components to implement any security code. When a call is made to a component running under COM+ Component Services, COM+ checks the user/group identity of the caller and compares it against the roles assigned to the component. Based on that comparison, the call is allowed or rejected.

You can provide additional security checking by using *procedural security*. This type of security is implemented programmatically using special .NET classes designed for interaction with COM+ Component Services.

5.3.6 .NET Classes and COM+ Component Services

Thus far, our discussions about COM+ Component Services, transactions, and security deal specifically with COM+ components. COM+ predated the .NET Framework and has had much success in enterprise-wide applications developed using Microsoft Visual Studio 6.0. But how does .NET fit into all of this?

COM+ still remains a dominant technology and is a significant part of Windows. The architecture for .NET managed components was designed to take advantage of all the features COM+ Component Services has to offer (object pooling, transaction processing, security, and so on) by providing classes to implement those features. These concepts are very important when developing Web applications, too.

5.4 Using VB.NET to Develop Managed Components

In this section I'll present the concepts you need to understand to build managed components using VB.NET.

5.4.1 Defining Namespaces

Namespaces are a way to hierarchically organize class names. Namespaces become especially useful when the number of classes available to you is quite large. You've been exposed to namespaces quite a bit already in this book. Whenever you write VB.NET code, you use .NET classes in the `System` namespace. For example, when you use the `Console.WriteLine()` command, you are using an assembly called `Console` that exists in the `System` namespace. In the code, `Console` is not fully qualified because it is assumed to be a part of the `System` namespace based on the uniqueness of the name `Console`. You could fully qualify the statement this way:

```
System.Console.WriteLine( . . . )
```

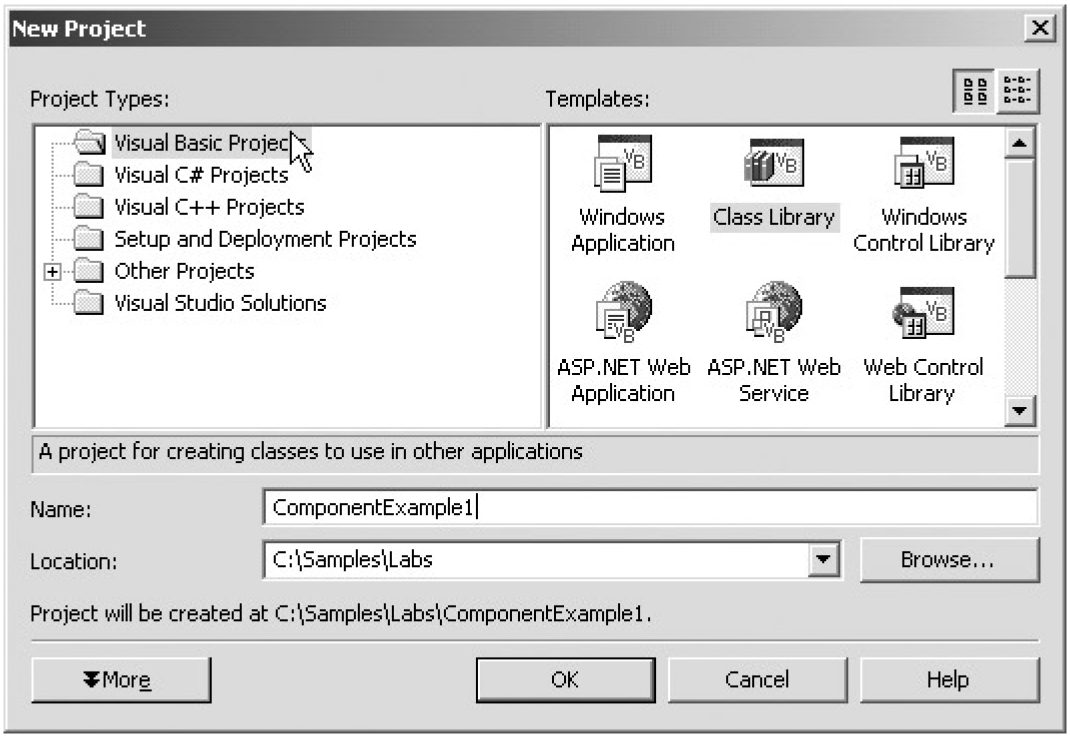
When you create a new .NET assembly (program), a default root namespace is assigned to the assembly. This name is typically the name of your project. However, you are free to assign your own namespace names.

5.4.2 Using the Class Library

Managed components start with a VS.NET project called a *class library*. The class library puts the infrastructure in place for creating an assembly for packing a component. Figure 5-2 shows the selections used to create a new class library project.

You can specify several project properties and options for class library projects. Of particular interest are the root namespace and the assembly name. VB.NET creates default names for both of these options. You can view and modify them by selecting the **Project→Properties** menu. This opens the Property Pages dialog shown in Figure 5-3.

By default, VB.NET assigns the name you specified for the project as the names for the assembly and the root namespace. You can override these default names with better, more descriptive ones. Given these new designations, references to classes inside the assembly would be made in the follow-



A project for creating classes to use in other applications

Name: ComponentExample1

Location: C:\Samples\Labs

Project will be created at C:\Samples\Labs\ComponentExample1.

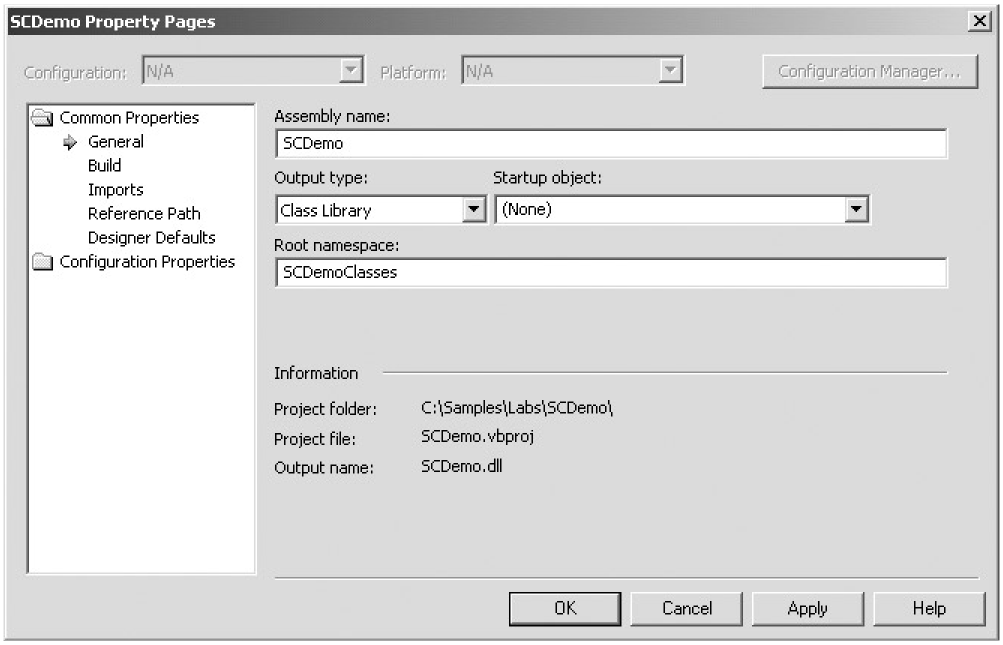
Figure 5-2 Creating a new class library project in VB.NET

ing manner. (This example shows the usage of `Dim` to declare an object for a particular class inside the assembly.)

```
Dim objMyObject As objMyExample.SomeClassInTheAssembly
```

.NET applications that you write need to include a reference to each assembly you wish to use inside the main application. You can add a reference to an assembly by right-clicking on the References folder (see Figure 5-4) and selecting Add Reference . . . . VS.NET displays a dialog with a list of available assemblies (see Figure 5-5). Those assemblies that are a part of the .NET Framework distribution are displayed in the list. To add a reference to another assembly, click the Browse . . . button. Then you'll be able to select the assembly file you need.

Crouch05_393-458.ps 4/23/02 5:59 PM Page 408


SCDemo Property Pages

Configuration: N/A Platform: N/A Configuration Manager...

Common Properties
- General
- Build
- Imports
- Reference Path
- Designer Defaults

Configuration Properties

Assembly name: SCDemo

Output type: Class Library

Startup object: (None)

Root namespace: SCDemoClasses

Information

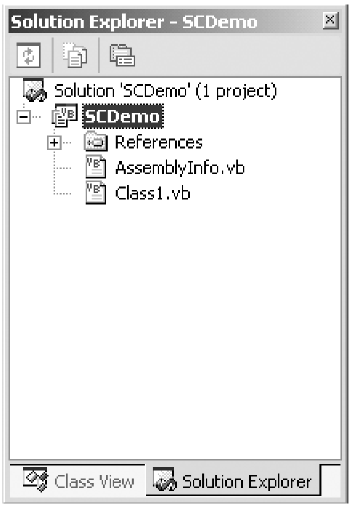
Project folder: C:\Samples\Labs\SCDemo\

Project file: SCDemo.vbproj

Output name: SCDemo.dll

OK Cancel Apply Help

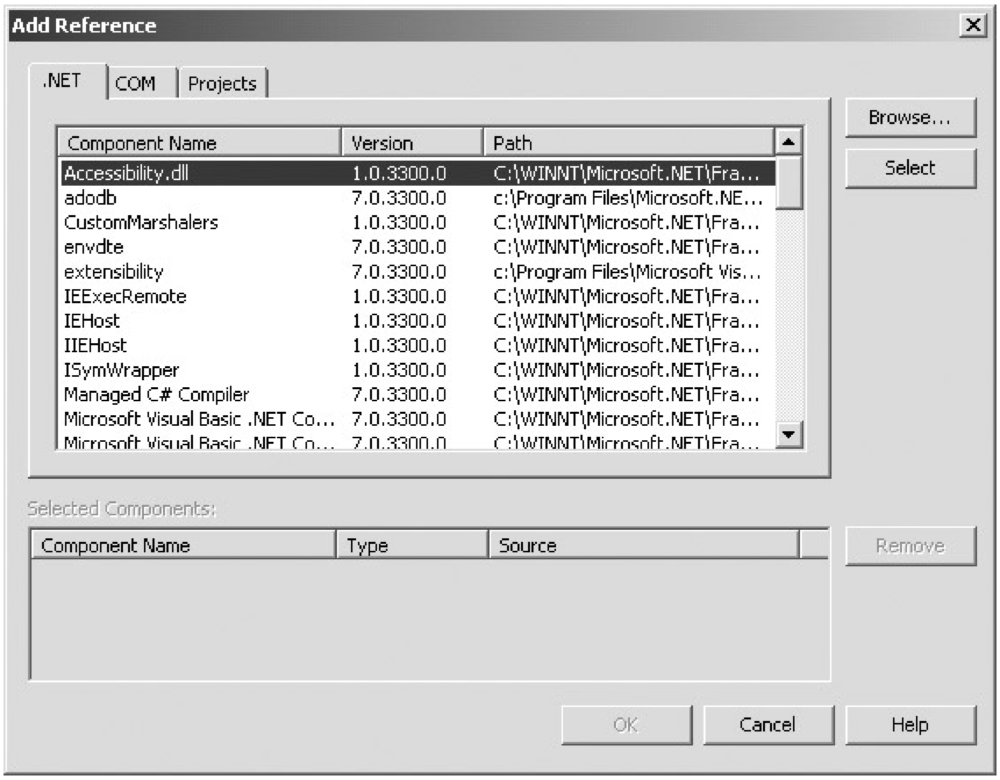
Figure 5-3 The Property Pages dialog


Solution Explorer - SCDemo

Solution 'SCDemo' (1 project)
- SCDemo
  - References
  - AssemblyInfo.vb
  - Class1.vb

Class View | Solution Explorer

Figure 5-4 The References folder



Component Name	Version	Path
Accessibility.dll	1.0.3300.0	C:\WINNT\Microsoft.NET\Fra...
adodb	7.0.3300.0	c:\Program Files\Microsoft.NE...
CustomMarshalers	1.0.3300.0	C:\WINNT\Microsoft.NET\Fra...
envdte	7.0.3300.0	C:\WINNT\Microsoft.NET\Fra...
extensibility	7.0.3300.0	c:\Program Files\Microsoft Vis...
IEExecRemote	1.0.3300.0	C:\WINNT\Microsoft.NET\Fra...
IEHost	1.0.3300.0	C:\WINNT\Microsoft.NET\Fra...
IIEHost	1.0.3300.0	C:\WINNT\Microsoft.NET\Fra...
ISymWrapper	1.0.3300.0	C:\WINNT\Microsoft.NET\Fra...
Managed C# Compiler	7.0.3300.0	C:\WINNT\Microsoft.NET\Fra...
Microsoft Visual Basic .NET Co...	7.0.3300.0	C:\WINNT\Microsoft.NET\Fra...
Microsoft Visual Basic .NET Co...	7.0.3300.0	C:\WINNT\Microsoft.NET\Fra...

Component Name	Type	Source

Figure 5-5 The Add Reference dialog

5.4.3 Using Component "Plumbing" Code

When class library projects are started in VB.NET, some stub code is generated for you. A new, "empty" component's "plumbing" code looks something like this:

```
Public Class Class1

End Class
```

The VS.NET IDE adds this code for you for an empty class called `Class1`. A `.vb` file is also generated that contains the class's code (which VS.NET should be displaying in the front-most window). Give the class another name (in the example below, we'll name it `CTaxComponent`) by

Crouch05_393-458.ps 4/23/02 5:59 PM Page 410

changing it in the editor and then renaming the class file in the Solution Explorer to `CTaxComponent`. The code should now say:

```
Public Class CTaxComponent

End Class
```

5.4.4 Adding Initialization Code

Whenever your component is instantiated, the `New()` subroutine is called. Any initialization code you require goes here. As an illustration, suppose your class contains a private member variable that you wish to have an initial default value when the component is instantiated. Such initialization code could look like the following.

```
Private m_dblTaxRate As Double

Public Sub New()
   MyBase.New

   ' Specify default percent tax rate
   m_dblTaxRate = 5

End Sub
```

5.4.5 Creating Methods

In Chapter 2 we discussed classes and several class concepts, such as methods. When creating class libraries and components, you'll notice that the same principles apply to components. Methods are simply member functions and subroutines. In order for other .NET applications to access methods you define, you need to expose them by marking them for public access. Here's a public method definition and implementation to illustrate.

```
Public Function CalculatePriceAfterTax( _
   ByVal Amount As Double) As Double

   CalculatePriceAfterTax = Amount * (1 + (m_dblTaxRate / 100))

End Function
```

5.4.6 Creating Properties

Component properties are created in the same way as properties for regular classes. Like methods, they should be marked for public access. The property in the sample code below sets or gets the value of the private class member variable used in the previous example.

```
Public Property TaxRate() As Double
   Get
      TaxRate = m_dblTaxRate
   End Get
   Set(ByVal Value As Double)
      m_dblTaxRate = Value
   End Set
End Property
```

5.4.7 Using the Class Library in an Application

The preceding sections described all the steps required to create a class library that can function as an independent component in another application. Now the class library needs to be tested. You can create another .NET application that references the assembly and write code to call the `CTaxComponent` class. The code below is a simple console application that uses the class library.

```
Imports System
Imports SCDemoClasses
Imports Microsoft.VisualBasic

Module Module1

   Sub Main()
      Dim objTaxComponent As New CTaxComponent()
      Dim dblItemPrice As Double = 9.99
      Dim dblTotalPrice As Double

      With objTaxComponent
         .TaxRate = 7
         dblTotalPrice = _
            .CalculatePriceAfterTax(dblItemPrice)
      End With

      Console.WriteLine("{0} after tax is {1}", _
         FormatCurrency(dblItemPrice), _
```

```
            FormatCurrency(dblTotalPrice))

    End Sub

End Module
```

This code produces the following output (assuming U.S. currency):

```
$9.99 after tax is $10.69
```

> **WARNING:** Before this program can compile and run, a reference to the assembly DLL, `SCDemo` (used in the example), must be added as a reference to the hosting application (the sample console application) using the **Add Reference . . .** command from the menu.

5.5 Serviced Components

Building components and class libraries is a good way to organize your code, but the main advantage of writing class libraries and components for your application is to gain the benefits of COM+ Component Services. With COM+ Component Services, your components can take advantage of automatic transaction processing, object pooling, and role-based security. Components that use COM+ Component Services are called *serviced components*.

Setting up a component to take advantage of these features is not difficult. It requires creating a class library project as shown earlier plus creating several component classes to handle the business logic. Let's discuss an example scenario that uses serviced components.

Suppose that we are writing an ordering and inventory system for a supermarket. The system will track inventory as well as receive orders. Three different types of users will work with this system: Suppliers, Supervisors, and Receiving Clerks. Suppliers enter any orders that we expect to receive into our database. Any products that are ordered must be products that the store normally carries. Supervisors are authorized to add new product types to the database. Receiving Clerks and Supervisors are authorized to receive orders. Line items received from an order will be immediately reflected as available inventory for the supermarket. Figure 5-6 shows the database schema used for

Crouch05_393-458.ps 4/23/02 5:59 PM Page 413

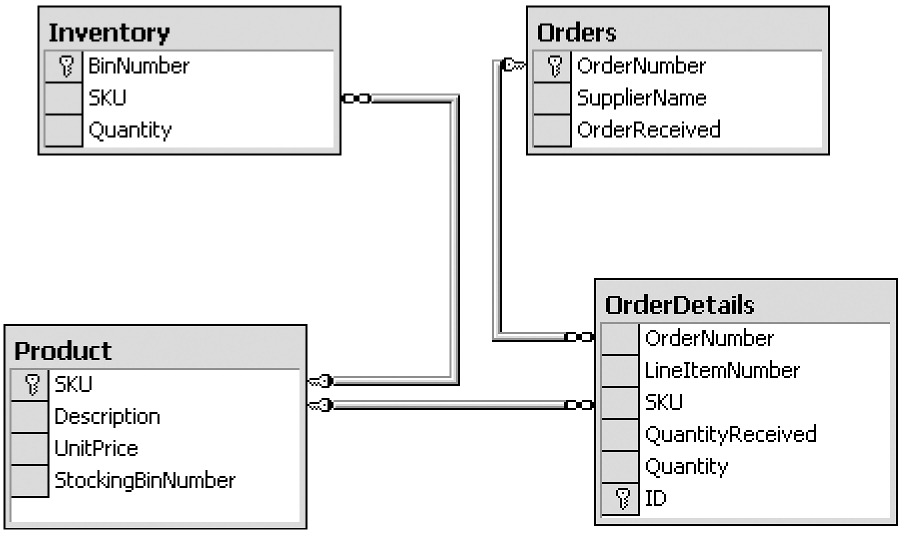


Figure 5-6 Database diagram of a supermarket ordering and inventory system

the ordering and inventory system, showing tables, columns, and foreign key relationships among columns. (Note that this system is simplified for the purposes of the example.) To run the code samples, you will need to create a SQL Server database with these specifications, tables, and relationships. Tables 5-3 through 5-6 list the table column names and data types for the four tables shown in Figure 5-6.

Table 5-3 Column Names and Data Types for the `Inventory` Table

Column Name	Data Type
`BinNumber`	`Varchar(255)`
`SKU`	`Varchar(255)`
`Quantity`	`Int`

Table 5-4 Column Names and Data Types for the `Product` Table

Column Name	Data Type
`SKU`	`Varchar(255)`
`Description`	`Varchar(255)`
`UnitPrice`	`Decimal`
`StockingBinNumber`	`Varchar(255)`

Table 5-5 Column Names and Data Types for the `Orders` Table

Column Name	Data Type
`OrderNumber`	`Varchar(255)`
`SupplierName`	`Varchar(255)`
`OrderReceived`	`Bit`

Table 5-6 Column Names and Data Types for the `OrderDetails` Table

Column Name	Data Type
`OrderNumber`	`Varchar(255)`
`LineItemNumber`	`Int`
`SKU`	`Varchar(255)`
`QuantityReceived`	`Int`
`Quantity`	`Int`
`ID`	`Identity`

Crouch05_393-458.ps 4/23/02 5:59 PM Page 415

TIP: Consult your SQL Server documentation about creating a new database and setting up tables if you are not familiar with the process. Recruit the help of a database administrator if you need assistance in obtaining authorization to create a database (if you're in an enterprise/shared database environment). We will discuss database access in detail in Chapter 7.

5.6 Building VB.NET Serviced Components

Now it's time for you to use serviced components to build the supermarket ordering and inventory system outlined in the previous section! You'll work through each step in the process, from designing to coding and then testing the components in an ASP.NET application.

LAB 5-1

AN ORDERING AND INVENTORY SYSTEM MADE WITH SERVICED COMPONENTS

STEP 1. Create a new VB.NET class library project.

a. Open VS.NET and select **File→New→Project.**

b. Highlight Class Library and give the new project the name "SupermarketSystem".

STEP 2. Design the components.

a. Consider the design of your system. First, establish the users of the system and their roles:

- Supervisors (adding new products, receiving orders, updating inventory)
- Receiving Clerks (receiving orders)
- Suppliers (shipping goods to the supermarket and supplying order information)

b. Organize these functions into a series of classes for your class library. Four classes will handle the business logic of the different duties that each user will perform: `Product`, `ReceiveOrder`, `UpdateInventory`, and `CreateOrder`. Lab Figure 5-1 shows the classes and the methods of each class.

Adding a new product to the system, updating inventory, receiving an order, and creating an order all involve operations on the database tier of the application. For some of these operations, database updates can occur in multiple tables. It's important to keep data integrity across these tables. You also want to implement security features so the users perform only the functions that their roles designate. You'll see this functionality develop as you code the components.

STEP 3. Write code for component functionality.

In the process of implementing the components, we'll discuss these topics:

- How to use the `System.EnterpriseServices` namespace and one particular class: the `ServicedComponent` class
- How to use class attributes to specify the type of COM+ support you want your components to have
- How to specify an interface from which to call the components
- How to control the transaction inside the components

a. Create the classes outlined in the specifications in Step 2: `Product`, `ReceiveOrder`, `UpdateInventory`, and `CreateOrder`. For each of these classes, right-click the project in the Solution Explorer and select **Add**

Product	ReceiveOrder	UpdateInventory	CreateOrder
Create	GetNextLineItem Receive	Update	Create AddItems

Lab Figure 5-1 The classes and methods of the Supermarket program

Class . . . from the menu. Name the classes as listed above. VS.NET will “stub out” an empty class definition for each class.

b. Building serviced components requires support from COM+ Component Services. The .NET Framework implements this support through classes in the `System.EnterpriseServices` namespace. VS.NET doesn’t include this reference by default, so you’ll need to add it yourself. Select **Project→Add Reference** from the menu and select `System.EnterpriseServices` from the list. Click the Select button and then click OK.

c. Now you’re ready to add component functionality. Start by adding the following code for the `Product` class (`Product.vb`).

LAB CODE SAMPLE 5-1

```
Imports System
Imports System.Reflection
Imports System.EnterpriseServices
Imports System.Data
Imports System.Data.SqlClient

❺ <Assembly: ApplicationName("Supermarket")>
❻ <Assembly: ApplicationActivation(ActivationOption.Library)>
❼ <Assembly: AssemblyKeyFile("KeyFile.snk")>
❽ Namespace Supermarket

❾ Public Interface IProduct
   Function Create(ByVal SKU As String, _
                 ByVal Description As String, _
                 ByVal UnitPrice As Decimal, _
                 ByVal StockingBinNumber As String) _
                 As Boolean
   End Interface

❿ <ConstructionEnabled( _
   [Default]:="Default Construction String"), _
   Transaction(TransactionOption.Required)> _
   Public Class Product
⓫        Inherits ServicedComponent
⓬        Implements Supermarket.IProduct

     Public Sub New()

     End Sub
     Protected Overrides Sub Construct( _
13           ByVal constructString As String)

     End Sub

     Function Create(ByVal SKU As String, _
                 ByVal Description As String, _
                 ByVal UnitPrice As Decimal, _
                 ByVal StockingBinNumber As String) _
                 As Boolean _
                 Implements Supermarket.IProduct.Create

       Dim objCnn As SqlConnection
       Dim objCmd As SqlCommand
       Dim objParam As SqlParameter
       Dim intRowsReturned As Integer

       Try
         objCnn = New SqlConnection()
         objCnn.ConnectionString = _
"Initial Catalog=Supermarket;Data Source=localhost;uid=sa;pwd="
         objCnn.Open()
         objCmd = objCnn.CreateCommand()
         objCmd.CommandText = _
   "INSERT INTO Product " & _
   "( SKU, Description, UnitPrice, StockingBinNumber ) " & _
   "VALUES ( @sku, @description, @unitprice, @stockingbinnumber )"

         objParam = New SqlParameter()
         With objParam
            .ParameterName = "@sku"
            .SqlDbType = SqlDbType.VarChar
            .Direction = ParameterDirection.Input
            .Value = SKU
         End With
         objCmd.Parameters.Add(objParam)

         objParam = New SqlParameter()
         With objParam
            .ParameterName = "@description"
            .SqlDbType = SqlDbType.VarChar
            .Direction = ParameterDirection.Input
            .Value = Description
         End With
         objCmd.Parameters.Add(objParam)
         objParam = New SqlParameter()
         With objParam
            .ParameterName = "@unitprice"
            .SqlDbType = SqlDbType.Decimal
            .Direction = ParameterDirection.Input
            .Value = UnitPrice
         End With
         objCmd.Parameters.Add(objParam)

         objParam = New SqlParameter()
         With objParam
            .ParameterName = "@stockingbinnumber"
            .SqlDbType = SqlDbType.VarChar
            .Direction = ParameterDirection.Input
            .Value = StockingBinNumber
         End With
         objCmd.Parameters.Add(objParam)

         intRowsReturned = _
            objCmd.ExecuteNonQuery()
         Create = True
14           ContextUtil.SetComplete()

       Catch E As Exception
         Create = False
15       ContextUtil.SetAbort()
       Finally
         objCnn.Close()
       End Try

     End Function

   End Class

End Namespace
```

This code implements a serviced component. As mentioned before, a serviced component is a .NET class that uses COM+ Component Services. The class becomes a serviced component when it derives from the `System.EnterpriseServices.ServicedComponent` class. Before we talk more about the `ServicedComponent` class, let’s first investigate the beginning of the code where some assembly-level attributes are declared.

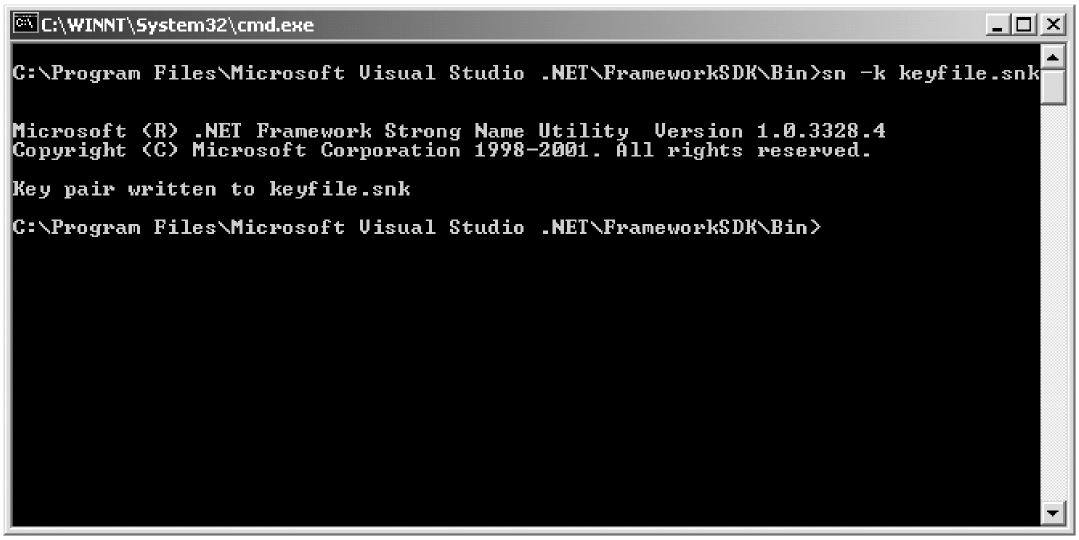
Making a serviced component requires you to provide information to COM+ Component Services about the component's configuration. First you designate to which package, or COM+ application, the component will belong. That designation is made with the `ApplicationName` assembly-level attribute shown in line ❺. The name for the order and inventory application is "Supermarket". COM+ applications are listed in the Component Services Console.

COM+ Component Services provides a runtime environment for assembly components. You can also control where the components are activated—in the same process as the creator of the object (IIS) or in a separate system process (`dllhost.exe`). You can control application activation by specifying the `ApplicationActivation` assembly attribute shown in line ❻. This code specifies `ActivationOption.Library`, which causes components to be activated in the creator's process. So, if you were running these components inside ASP.NET Web Forms, the creating process would be the Web server, IIS. The `ActivationOption.Server` option provides faster performance; this option, which runs the component in a system process, provides more isolation, so the component's execution won't adversely affect the execution of IIS. One advantage to using `ActivationOption.Library` is to make debugging easier.

The final assembly-level attribute, `AssemblyKeyFile`, specifies a shared name for the assembly (see line ❼). The shared name is sometimes referred to as a *strong name*. A shared name ensures that a name assigned to a component is unique. This is accomplished by using a public/private cryptographic key pair to sign each shared component in the assembly. The public key is then published with the assembly. Besides specifying this attribute in the code, you'll need to actually generate the key file specified for the assembly-level attribute. To do this, use `sn.exe`, the Strong Name Utility. Lab Figure 5-2 shows how this is done.

The usage of `sn.exe`, as shown in Lab Figure 5-2, outputs a `.snk` file. This is the file name that you specify in the `AssemblyKeyFile` attribute in the code (change the path to the location of your generated key file). You must complete this step before you compile your project. These assembly-level attributes appear once in the project's code. The class file in which they appear is irrelevant, but they must be declared once and only once in the assembly code.

WARNING: Never let anybody else have access to your `.snk` file. It contains a private key that is for your use only.

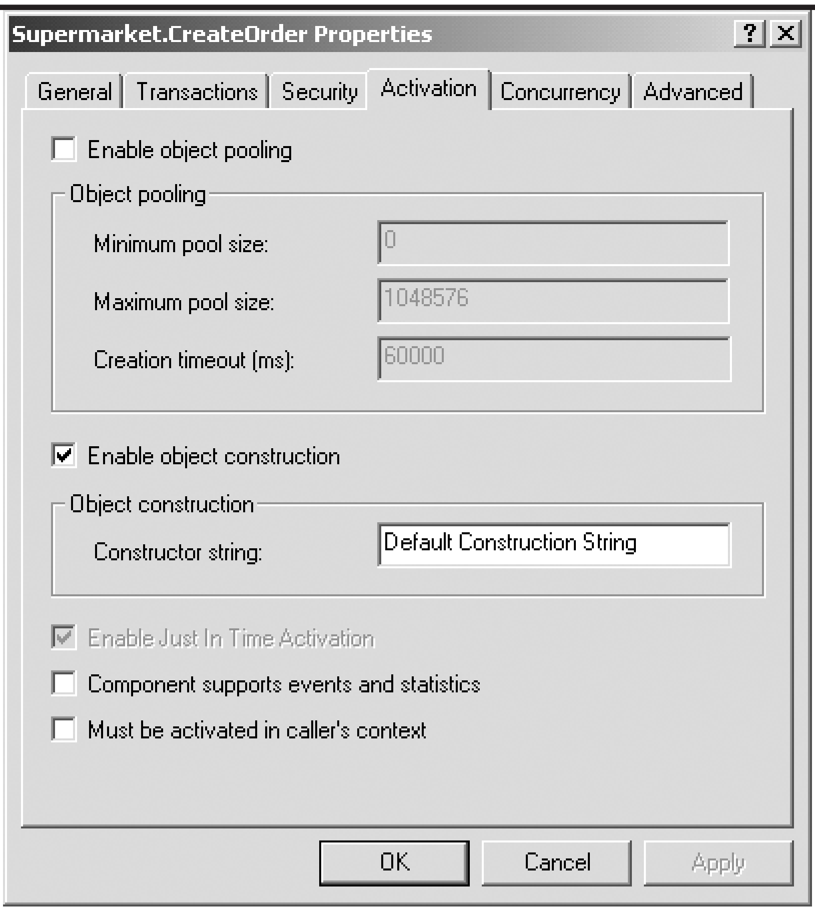


```
C:\Program Files\Microsoft Visual Studio .NET\FrameworkSDK\Bin>sn -k keyfile.snk

Microsoft (R) .NET Framework Strong Name Utility  Version 1.0.3328.4
Copyright (C) Microsoft Corporation 1998-2001. All rights reserved.

Key pair written to keyfile.snk

C:\Program Files\Microsoft Visual Studio .NET\FrameworkSDK\Bin>
```

Lab Figure 5-2 Specifying a strong name for the assembly using `sn.exe`

Let’s move on to the main part of the component code. Line ❽ defines the `Supermarket` namespace, which will contain all the components for the Supermarket application. Line ❾ declares an interface you’ll use in the client application (the ASP.NET Web Form) to call the component. The interface has one function, `Create()`, which you’ll use to set up a new product in the `Product` database table.

The component’s class definition comes next. The code beginning in line ❿ shows two class-level attributes for the `Product` class. The first one, `ConstructionEnabled`, specifies that the class will be able to use COM+ constructor strings. A constructor string is a string passed into the activation procedure of the component. This string can be specified by a system administrator inside the Component Services Console. A constructor string can contain any information, but typically you’ll use it for passing initialization information to the component. If no constructor string is given for the component in the Component Services Console (see Lab Figure 5-3, which shows the Component Services Console dialog box for setting a constructor string), a default constructor string is used by specifying it in the attribute as shown in the continuation of line ❿. The other class-level attribute specifies how the class will participate in transactions. Use the `Transaction` attribute to specify the participation level. In this code, `Transaction` uses



Supermarket.CreateOrder Properties

- [ ] Enable object pooling

Object pooling

Minimum pool size:	0
Maximum pool size:	1048576
Creation timeout (ms):	60000

- [✓] Enable object construction

Object construction

Constructor string:	Default Construction String

- [✓] Enable Just In Time Activation
- [ ] Component supports events and statistics
- [ ] Must be activated in caller's context

Lab Figure 5-3 Specifying a constructor string in Component Services

`TransactionOption.Required`. This indicates that the `Product` component should participate in an existing transaction if one already exists. If no transaction exists, a new transaction will begin and the `Product` component will execute within the boundaries of that transaction.

The class definition continues with the specification of an inherited class and an `Implements` statement. Since this will be a serviced component, it needs to derive from the `ServicedComponent` class, as shown in line ⓫. In line ⓬ the class implements the `IProduct` interface specified earlier.

Line ⓭ provides implementation support for constructor strings. Since the code specified that the component will have support for constructor strings with the class-level attribute `ConstructionEnabled`, here there is an override method for the `Construct()` sub. This method will be called upon object construction, and the constructor string assigned to the component will be available in the `constructString` variable.

Now follows the implementation of the `Create()` method, which accepts a new product's SKU number, product description, unit price, and stocking bin number. The `Create()` method then executes the appropriate ADO.NET code to insert a new row into the `Product` database table. (ADO.NET code will be discussed in Chapter 7. For now, just be aware that the `Product` component contains this method that will be callable from an ASP.NET Web Form.)

Although the discussion of the ADO.NET code details is deferred, it's important to point out two details in the `Create()` method. These are two methods of the `ContextUtil` object, `SetComplete()` in line ⓮ and `SetAbort()` in line ⓯. These methods cast a vote in the current transaction. Typically, when you have verified that all of the code inside a particular component method has executed successfully, a call to `SetComplete()` is made. This tells COM+ Component Services that a unit of work has succeeded and that database integrity and consistency is assured for updates made by the unit of work. It's a signal that the transaction can continue running. Conversely, if a failure occurs (an exception or other user-defined error or condition), the program needs to cast a "fail" vote for the transaction by calling `SetAbort()`. This will cause COM+ Component Services to stop the transaction and roll back any changes made to the database by previous steps in the transaction.

d. Now that you understand the basic pieces of the `Product` class code, add the code for the remaining classes (`CreateOrder`, `UpdateInventory`, and `ReceiveOrder`). The implementations are all different, of course, but they follow pretty much the same conventions as the `Product` component.

LAB CODE SAMPLE 5-2

```
Imports System
Imports System.Reflection
Imports System.EnterpriseServices
Imports System.Data
Imports System.Data.SqlClient


Namespace Supermarket
```

Crouch05_393-458.ps 4/23/02 5:59 PM Page 424

```
 Public Interface ICreateOrder
    Function Create(ByVal OrderNumber As String, _
                    ByVal SupplierName As String) _
                    As Boolean
    Function AddItems(ByVal OrderNumber As String, _
                    ByVal SKU As String, _
                    ByVal Quantity As Integer) _
                    As Boolean
 End Interface

 <ConstructionEnabled( _
 [Default]:="Default Construction String"), _
 Transaction(TransactionOption.Required)> _
 Public Class CreateOrder
    Inherits ServicedComponent
    Implements ICreateOrder

    Public Sub New()

    End Sub

    Public Function Create(ByVal OrderNumber As String, _
                    ByVal SupplierName As String) _
                    As Boolean _
                    Implements ICreateOrder.Create

       Dim objCnn As SqlConnection
       Dim objCmd As SqlCommand
       Dim objParam As SqlParameter
       Dim intRowsAffected As Integer

       Try
          objCnn = New SqlConnection()
          objCnn.ConnectionString = _
"Initial Catalog=Supermarket;Data Source=localhost;uid=sa;pwd="
          objCnn.Open()
          objCmd = objCnn.CreateCommand()

          objCmd.CommandText = _
 "INSERT INTO Orders " & _
 "( OrderNumber, SupplierName, OrderReceived ) " & _
 "VALUES ( @OrderNumber, @SupplierName, @OrderReceived )"

          objParam = New SqlParameter()
```

```
      With objParam
         .ParameterName = "@OrderNumber"
         .SqlDbType = SqlDbType.VarChar
         .Direction = ParameterDirection.Input
         .Value = OrderNumber
      End With
      objCmd.Parameters.Add(objParam)

      objParam = New SqlParameter()
      With objParam
         .ParameterName = "@SupplierName"
         .SqlDbType = SqlDbType.VarChar
         .Direction = ParameterDirection.Input
         .Value = SupplierName
      End With
      objCmd.Parameters.Add(objParam)

      objParam = New SqlParameter()
      With objParam
         .ParameterName = "@OrderReceived"
         .SqlDbType = SqlDbType.Bit
         .Direction = ParameterDirection.Input
         .Value = False
      End With
      objCmd.Parameters.Add(objParam)

      intRowsAffected = objCmd.ExecuteNonQuery()

      Create = True
      ContextUtil.SetComplete()
   Catch E As Exception
      Create = False
      ContextUtil.SetAbort()
   End Try
End Function

Public Function AddItems(ByVal OrderNumber As String, _
                ByVal SKU As String, _
                ByVal Quantity As Integer) _
                As Boolean _
                Implements ICreateOrder.AddItems

   Dim objCnn As SqlConnection
   Dim objCmd As SqlCommand
```

Crouch05_393-458.ps 4/23/02 5:59 PM Page 426

```
        Dim objParam As SqlParameter
        Dim intMaxLineNumber As Integer
        Dim intRowsAffected As Integer
        Dim objTemp As Object

        Try
           objCnn = New SqlConnection()
           objCnn.ConnectionString = _
"Initial Catalog=Supermarket;Data Source=localhost;uid=sa;pwd="
           objCnn.Open()
           objCmd = objCnn.CreateCommand()

           objCmd.CommandText = _
              "SELECT MAX( LineItemNumber ) " & _
              "FROM OrderDetails " & _
              "WHERE OrderNumber = @OrderNumber"
           objParam = New SqlParameter()
           With objParam
              .ParameterName = "@OrderNumber"
              .SqlDbType = SqlDbType.VarChar
              .Direction = ParameterDirection.Input
              .Value = OrderNumber
           End With
           objCmd.Parameters.Add(objParam)

           objTemp = objCmd.ExecuteScalar()
           If TypeOf objTemp Is DBNull Then
              intMaxLineNumber = 1
           Else
              intMaxLineNumber = CType(objTemp, Integer)
              intMaxLineNumber += 1
           End If

           objCmd = objCnn.CreateCommand()
           objCmd.CommandText = _
              "INSERT INTO OrderDetails " & _
              "( OrderNumber, LineItemNumber, SKU, " & _
              "QuantityReceived, Quantity ) VALUES " & _
              "( @OrderNumber, @LineNumber, @SKU, " & _
              "@QuantityReceived, @Quantity )"
           objParam = New SqlParameter()
           With objParam
              .ParameterName = "@OrderNumber"
              .SqlDbType = SqlDbType.VarChar
```

```
   .Direction = ParameterDirection.Input
   .Value = OrderNumber
End With
objCmd.Parameters.Add(objParam)

objParam = New SqlParameter()
With objParam
   .ParameterName = "@LineNumber"
   .SqlDbType = SqlDbType.Int
   .Direction = ParameterDirection.Input
   .Value = intMaxLineNumber
End With
objCmd.Parameters.Add(objParam)

objParam = New SqlParameter()
With objParam
   .ParameterName = "@SKU"
   .SqlDbType = SqlDbType.VarChar
   .Direction = ParameterDirection.Input
   .Value = SKU
End With
objCmd.Parameters.Add(objParam)

objParam = New SqlParameter()
With objParam
   .ParameterName = "@QuantityReceived"
   .SqlDbType = SqlDbType.Int
   .Direction = ParameterDirection.Input
   .Value = 0
End With
objCmd.Parameters.Add(objParam)

objParam = New SqlParameter()
With objParam
   .ParameterName = "@Quantity"
   .SqlDbType = SqlDbType.Int
   .Direction = ParameterDirection.Input
   .Value = Quantity
End With
objCmd.Parameters.Add(objParam)

intRowsAffected = objCmd.ExecuteNonQuery()

AddItems = True
```

```
            ContextUtil.SetComplete()

         Catch E As Exception
            AddItems = False
            ContextUtil.SetAbort()
         Finally
            objCnn.Close()
         End Try

      End Function

   End Class
End Namespace
```

LAB CODE SAMPLE 5-3

```
Imports System
Imports System.Reflection
Imports System.EnterpriseServices
Imports System.Data
Imports System.Data.SqlClient

Namespace Supermarket

   Public Interface IUpdateInventory
      Function Update(ByVal BinNumber As String, _
                      ByVal SKU As String, _
                      ByVal Quantity As Integer) _
                      As Boolean
      Function GetStockingLocation(ByVal SKU As String) _
                      As String
   End Interface

   <ConstructionEnabled( _
   [Default]:="Default Construction String"), _
   Transaction(TransactionOption.Required)> _
   Public Class UpdateInventory
      Inherits ServicedComponent
      Implements IUpdateInventory

      Public Sub New()

      End Sub
```

```
    Public Function GetStockingLocation( _
            ByVal SKU As String) As String _
            Implements IUpdateInventory.GetStockingLocation

       Dim objCnn As SqlConnection
       Dim objCmd As SqlCommand
       Dim objParam As SqlParameter
       Dim objTemp As Object

       Try
          objCnn = New SqlConnection()
          objCnn.ConnectionString = _
"Initial Catalog=Supermarket;Data Source=localhost;uid=sa;pwd="
          objCnn.Open()
          objCmd = objCnn.CreateCommand()

          objCmd.CommandText = _
                "SELECT StockingBinNumber " & _
                "FROM Product WHERE SKU = @SKU"

          objParam = New SqlParameter()
          With objParam
             .ParameterName = "@SKU"
             .SqlDbType = SqlDbType.VarChar
             .Direction = ParameterDirection.Input
             .Value = SKU
          End With
          objCmd.Parameters.Add(objParam)

          objTemp = objCmd.ExecuteScalar()
          If TypeOf objTemp Is DBNull Then
             GetStockingLocation = ""
          Else
             GetStockingLocation = _
                CType(objCmd.ExecuteScalar(), String)
          End If
          ContextUtil.SetComplete()

       Catch E As Exception
          ContextUtil.SetAbort()
          GetStockingLocation = ""
       Finally
          objCnn.Close()
       End Try
    End Function

    Private Function InventoryRecExists( _
            ByVal SKU As String, _
            ByVal StockingBinNumber As String) _
            As Boolean

       Dim objCnn As SqlConnection
       Dim objCmd As SqlCommand
       Dim objParam As SqlParameter
       Dim intRowCount As Integer
       Dim objTemp As Object

       Try
          objCnn = New SqlConnection()
          objCnn.ConnectionString = _
"Initial Catalog=Supermarket;Data Source=localhost;uid=sa;pwd="
          objCnn.Open()
          objCmd = objCnn.CreateCommand()

          objCmd.CommandText = _
                  "SELECT COUNT(*) FROM Inventory " & _
                  "WHERE SKU = @SKU AND " & _
                  "BinNumber = @StockingBinNumber"
          objParam = New SqlParameter()
          With objParam
             .ParameterName = "@SKU"
             .SqlDbType = SqlDbType.VarChar
             .Direction = ParameterDirection.Input
             .Value = SKU
          End With
          objCmd.Parameters.Add(objParam)

          objParam = New SqlParameter()
          With objParam
             .ParameterName = "@StockingBinNumber"
             .SqlDbType = SqlDbType.VarChar
             .Direction = ParameterDirection.Input
             .Value = StockingBinNumber
          End With
          objCmd.Parameters.Add(objParam)

          objTemp = objCmd.ExecuteScalar()
          If TypeOf objTemp Is DBNull Then
```

```
             intRowCount = 0
          Else
             intRowCount = CType(objTemp, Integer)
          End If

          If intRowCount > 0 Then
             InventoryRecExists = True
          Else
             InventoryRecExists = False
          End If

          ContextUtil.SetComplete()

       Catch E As Exception
          InventoryRecExists = False
          ContextUtil.SetAbort()
       Finally
          objCnn.Close()
       End Try

    End Function

    Private Sub UpdateInventoryRecord( _
                ByVal BinNumber As String, _
                ByVal SKU As String, _
                ByVal Quantity As Integer)

       Dim objCnn As SqlConnection
       Dim objCmd As SqlCommand
       Dim objParam As SqlParameter
       Dim intRowCount As Integer

       Try
          objCnn = New SqlConnection()
          objCnn.ConnectionString = _
"Initial Catalog=Supermarket;Data Source=localhost;uid=sa;pwd="
          objCnn.Open()
          objCmd = objCnn.CreateCommand()

          objCmd.CommandText = "UPDATE Inventory " & _
               "SET Quantity = Quantity + @Quantity " & _
               "WHERE BinNumber = @BinNumber AND SKU = @SKU"

          objParam = New SqlParameter()
```

```
            With objParam
               .ParameterName = "@Quantity"
               .SqlDbType = SqlDbType.Int
               .Direction = ParameterDirection.Input
               .Value = Quantity
            End With
            objCmd.Parameters.Add(objParam)

            objParam = New SqlParameter()
            With objParam
               .ParameterName = "@BinNumber"
               .SqlDbType = SqlDbType.VarChar
               .Direction = ParameterDirection.Input
               .Value = BinNumber
            End With
            objCmd.Parameters.Add(objParam)

            objParam = New SqlParameter()
            With objParam
               .ParameterName = "@SKU"
               .SqlDbType = SqlDbType.VarChar
               .Direction = ParameterDirection.Input
               .Value = SKU
            End With
            objCmd.Parameters.Add(objParam)

            intRowCount = objCmd.ExecuteNonQuery()
            ContextUtil.SetComplete()

         Catch E As Exception
            ContextUtil.SetAbort()
         Finally
            objCnn.Close()
         End Try
      End Sub

      Private Sub InsertInventoryRecord( _
              ByVal BinNumber As String, _
              ByVal SKU As String, _
              ByVal Quantity As Integer)

         Dim objCnn As SqlConnection
         Dim objCmd As SqlCommand
         Dim objParam As SqlParameter
```

Crouch05_393-458.ps 4/23/02 5:59 PM Page 433

```
        Dim intRowCount As Integer

        Try
           objCnn = New SqlConnection()
           objCnn.ConnectionString = _
"Initial Catalog=Supermarket;Data Source=localhost;uid=sa;pwd="
           objCnn.Open()
           objCmd = objCnn.CreateCommand()

           objCmd.CommandText = _
                "INSERT INTO Inventory " & _
                "( BinNumber, SKU, Quantity ) VALUES " & _
                "( @BinNumber, @SKU,  @Quantity )"

           objParam = New SqlParameter()
           With objParam
              .ParameterName = "@BinNumber"
              .SqlDbType = SqlDbType.VarChar
              .Direction = ParameterDirection.Input
              .Value = BinNumber
           End With
           objCmd.Parameters.Add(objParam)

           objParam = New SqlParameter()
           With objParam
              .ParameterName = "@SKU"
              .SqlDbType = SqlDbType.VarChar
              .Direction = ParameterDirection.Input
              .Value = SKU
           End With
           objCmd.Parameters.Add(objParam)

           objParam = New SqlParameter()
           With objParam
              .ParameterName = "@Quantity"
              .SqlDbType = SqlDbType.Int
              .Direction = ParameterDirection.Input
              .Value = Quantity
           End With
           objCmd.Parameters.Add(objParam)

           intRowCount = objCmd.ExecuteNonQuery()
           ContextUtil.SetComplete()
```

```
         Catch E As Exception
            ContextUtil.SetAbort()
         Finally
            objCnn.Close()
         End Try
      End Sub

      Public Function Update(ByVal BinNumber As String, _
              ByVal SKU As String, _
              ByVal Quantity As Integer) _
              As Boolean _
              Implements IUpdateInventory.Update

         Dim objCnn As SqlConnection
         Dim objCmd As SqlCommand
         Dim objParam As SqlParameter
         Dim strStockingLocation As String
         Dim intRowsAffected As Integer

         Try
            If InventoryRecExists(SKU, BinNumber) Then
               UpdateInventoryRecord( _
                    BinNumber, _
                    SKU, _
                    Quantity)
            Else
               InsertInventoryRecord( _
                    BinNumber, _
                    SKU, _
                    Quantity)
            End If

            Update = True
            ContextUtil.SetComplete()

         Catch E As Exception
            Update = False
            ContextUtil.SetAbort()
         End Try

      End Function
   End Class
End Namespace
```

LAB CODE SAMPLE 5-4

```
Imports System
Imports System.Reflection
Imports System.EnterpriseServices
Imports System.Data
Imports System.Data.SqlClient

Namespace Supermarket

   Public Interface IReceiveOrder
      Function GetNextLineItem(ByVal OrderNumber As String, _
                      ByRef SKU As String) As Integer
      Function Receive(ByVal OrderNumber As String, _
                      ByVal SKU As String, _
                      ByVal LineNumber As Integer, _
                      ByVal QuantityReceived As Integer) _
                      As Boolean
   End Interface

   <ConstructionEnabled( _
   [Default]:="Default Construction String"), _
   Transaction(TransactionOption.Required)> _
   Public Class ReceiveOrder
      Inherits ServicedComponent
      Implements IReceiveOrder

      Public Sub New()

      End Sub

      Private Sub UpdateOrderDeatils( _
                  ByVal OrderNumber As String, _
                  ByVal LineNumber As Integer, _
                  ByVal QuantityReceived As Integer)

         Dim objCnn As SqlConnection
         Dim objCmd As SqlCommand
         Dim objParam As SqlParameter
         Dim objSQLDr As SqlDataReader
         Dim intRowsAffected As Integer

         Try
            objCnn = New SqlConnection()
```

```
          objCnn.ConnectionString = _
"Initial Catalog=Supermarket;Data Source=localhost;uid=sa;pwd="
          objCnn.Open()
          objCmd = objCnn.CreateCommand()

          objCmd.CommandText = _
"UPDATE OrderDetails " & _
"SET QuantityReceived = " & _
"QuantityReceived + @QuantityReceived " & _
"WHERE OrderNumber = " & _
"@OrderNumber AND LineItemNumber = @LineNumber"

          objParam = New SqlParameter()
          With objParam
             .ParameterName = "@QuantityReceived"
             .SqlDbType = SqlDbType.Int
             .Direction = ParameterDirection.Input
             .Value = QuantityReceived
          End With
          objCmd.Parameters.Add(objParam)

          objParam = New SqlParameter()
          With objParam
             .ParameterName = "@OrderNumber"
             .SqlDbType = SqlDbType.VarChar
             .Direction = ParameterDirection.Input
             .Value = OrderNumber
          End With
          objCmd.Parameters.Add(objParam)

          objParam = New SqlParameter()
          With objParam
             .ParameterName = "@LineNumber"
             .SqlDbType = SqlDbType.Int
             .Direction = ParameterDirection.Input
             .Value = LineNumber
          End With
          objCmd.Parameters.Add(objParam)

          intRowsAffected = objCmd.ExecuteNonQuery()

          ContextUtil.SetComplete()

       Catch E As Exception
```

```
            ContextUtil.SetAbort()
         Finally
            objCnn.Close()
         End Try

      End Sub

      Public Function GetNextLineItem( _
                ByVal OrderNumber As String, _
                ByRef SKU As String) _
                As Integer Implements _
                IReceiveOrder.GetNextLineItem

         Dim objCnn As SqlConnection
         Dim objCmd As SqlCommand
         Dim objParam As SqlParameter
         Dim objSQLDr As SqlDataReader

         Try

            objCnn = New SqlConnection()
            objCnn.ConnectionString = _
"Initial Catalog=Supermarket;Data Source=localhost;uid=sa;pwd="
            objCnn.Open()
            objCmd = objCnn.CreateCommand()

            objCmd.CommandText = _
                "SELECT MAX(LineItemNumber), " & _
                "SKU FROM OrderDetails od, Orders o " & _
                "WHERE od.OrderNumber = @OrderNumber AND " & _
                "o.OrderReceived = 0 AND " & _
                "o.OrderNumber = od.OrderNumber GROUP BY SKU"

            objParam = New SqlParameter()
            With objParam
               .ParameterName = "@OrderNumber"
               .SqlDbType = SqlDbType.VarChar
               .Direction = ParameterDirection.Input
               .Value = OrderNumber
            End With
            objCmd.Parameters.Add(objParam)

            objSQLDr = objCmd.ExecuteReader()
            objSQLDr.Read()
            If Not objSQLDr.IsDBNull(0) Then
               GetNextLineItem = objSQLDr.GetInt32(0)
               SKU = objSQLDr.GetString(1)
            Else
               GetNextLineItem = -1
               SKU = ""
            End If

            ContextUtil.SetComplete()
            objSQLDr.Close()

         Catch E As Exception
            GetNextLineItem = -1
            SKU = ""
            ContextUtil.SetAbort()
         Finally
            objCnn.Close()
         End Try

      End Function

      Public Function Receive(ByVal OrderNumber As String, _
                      ByVal SKU As String, _
                      ByVal LineNumber As Integer, _
                      ByVal QuantityReceived As Integer) _
                      As Boolean _
                      Implements IReceiveOrder.Receive

         Dim objCnn As SqlConnection
         Dim objCmd As SqlCommand
         Dim objParam As SqlParameter
         Dim objInvUpdate As IUpdateInventory
         Dim strBinNumber As String

         Try
            UpdateOrderDeatils(OrderNumber, _
                    LineNumber, _
                    QuantityReceived)

            objInvUpdate = New UpdateInventory()
            strBinNumber = _
               objInvUpdate.GetStockingLocation(SKU)
            If objInvUpdate.Update(strBinNumber, _
                    SKU, QuantityReceived) Then
               Receive = True
               ContextUtil.SetComplete()
            Else
               Receive = False
               ContextUtil.SetAbort()
            End If

         Catch E As Exception
            Receive = False
            ContextUtil.SetAbort()
         End Try
      End Function

   End Class
End Namespace
```

e. Now that you've entered the main code for the Supermarket COM+ application, it's time to compile it. Choose **Build→Build Solution** from the menu to create the assembly file. In the next step, you'll create a project that references that assembly.

STEP 4. Create the ASP.NET application.

a. Create a new VB ASP.NET project called "SupermarketWeb".

b. Add a reference to `System.EnterpriseServices` to the project by right-clicking the project icon and selecting **Add Reference . . .** from the pop-up menu.

c. Add four new Web Forms (`.aspx` files) and their code-behind files (`.vb` files) to the project, named as follows:

```
Lcs5-5.aspx, Lcs5-5.aspx.vb
Lcs5-6.aspx, Lcs5-6.aspx.vb
Lcs5-7.aspx, Lcs5-7.aspx.vb
Lcs5-8.aspx, Lcs5-8.aspx.vb
```

These Web Forms will test the different components and their functionality.

Product Component Test Web Form

The HTML for the Lcs5-5.aspx file appears below.

LAB CODE SAMPLE 5-5

```
<%@ Page Language="vb"
   AutoEventWireup="false"
   src="Lcs5-05.aspx.vb"
   Inherits="WebForm1"
   Transaction="RequiresNew"%>
<html>
<head>
<title>Create New Product</title>
</head>
<body>

<form id="Form1"
   method="post"
   runat="server">

<p>SKU:
<asp:TextBox
   id=txtSKU
   runat="server">
</asp:TextBox></p>

<p>Description:
<asp:TextBox
   id=txtDescription
   runat="server">
</asp:TextBox></p>

<p>Unit Price:
<asp:TextBox
   id=txtUnitPrice
   runat="server">
</asp:TextBox></p>

<p>Stocking Location:
<asp:TextBox
   id=txtStockLoc
   runat="server">
</asp:TextBox></p>
```

```
<p>
<asp:Button
   id=cmdAddProduct
   runat="server"
   Text="Add Product">
</asp:Button>

<asp:CompareValidator
   id=CompareValidator1
   runat="server"
   ErrorMessage="You must enter a price (number)"
   Type="Double"
   ControlToValidate="txtUnitPrice"
   Operator="DataTypeCheck">
</asp:CompareValidator></p>

</form>
</body>
</html>
```

ASP.NET Web Forms can run inside the context of a COM+ Component Services transaction. In this ASP.NET Web application, the Web Forms call the serviced components in response to events raised by Web Controls (buttons clicked and so on). Since the ASP.NET Web Form is the initiator of calls made into a COM+ application, the Web Form is considered the root of the transaction. It has the “final vote” as to whether or not the transaction succeeds or fails.

In order to designate that the Web Form will participate in a transaction, you need to set the `Transaction` page attribute (highlighted in bold above). This code specifies the level as `RequiresNew`. This means that the page will always begin a new transaction for any units of work executed during the lifetime of the page.

Here is the code-behind file, `Lcs5-5.aspx.vb`, for the preceding Web Form.

```
Imports SupermarketSystem.Supermarket
Imports System.EnterpriseServices
Imports System.Reflection
Imports System.ComponentModel

Public Class WebForm1
      Inherits System.Web.UI.Page
      Protected WithEvents cmdAddProduct As _
   System.Web.UI.WebControls.Button
      Protected WithEvents txtSKU As _
   System.Web.UI.WebControls.TextBox
      Protected WithEvents txtDescription As _
   System.Web.UI.WebControls.TextBox
      Protected WithEvents txtUnitPrice As _
   System.Web.UI.WebControls.TextBox
      Protected WithEvents txtStockLoc As _
   System.Web.UI.WebControls.TextBox
      Protected WithEvents CompareValidator1 As_
   System.Web.UI.WebControls.CompareValidator

      Private Sub Page_Load(ByVal sender As System.Object, _
             ByVal e As System.EventArgs) Handles MyBase.Load

      End Sub

      Private Sub cmdAddProduct_Click( _
         ByVal sender As System.Object, _
         ByVal e As System.EventArgs) _
         Handles cmdAddProduct.Click

         Dim objProduct As IProduct
         Dim bln As Boolean

         objProduct = New Product()
         bln = objProduct.Create(txtSKU.Text, _
                  txtDescription.Text, _
                  CDec(txtUnitPrice.Text), _
                  txtStockLoc.Text)

         If bln Then
                ContextUtil.SetComplete()
         Else
                ContextUtil.SetAbort()
         End If

   End Sub
End Class
```

The `Click` event for the `cmdAddProduct` button calls the `Product` component to add a new product to the database. The code creates a new `Product` object and obtains a reference to the `IProduct` interface. It then calls the `Create()` method. If the call was successful (returned `True`), `SetComplete()` is called to indicate to COM+ that this unit of work in the transaction was successful. If not, `SetAbort()`stops the transaction immediately.

`CreateOrder` Component Test Web Form

The code for the `Lcs5-6.aspx` file follows below. Note that again the `Transaction` page attribute is specified and set to `RequiresNew`.

LAB CODE SAMPLE 5-6

```
<%@ Page Language="vb"
   AutoEventWireup="false"
   src="Lcs5-06.aspx.vb"
   Inherits="SupermarketCreateOrder"
   Transaction="RequiresNew"%>
<html>
<head>
<title>Create New Order</title>
</head>
<body>

<form id="Form1"
   method="post"
   runat="server">

<p>Order number:
<asp:TextBox
   id=txtOrderNumber
   runat="server">
</asp:TextBox></p>

<p>Supplier Name:
<asp:TextBox
   id=txtSupplierName
   runat="server">
</asp:TextBox></p>

<p>
<asp:Button
   id=cmdCreateOrder
   runat="server"
   Text="Add">
</asp:Button></p>

</form>
</body>
</html>
```

The code-behind file, `Lcs5-6.aspx.vb`, contains the following code.

```
Imports System.EnterpriseServices
Imports System.Reflection
Imports SupermarketSystem.Supermarket

Public Class SupermarketCreateOrder
   Inherits System.Web.UI.Page
   Protected WithEvents txtOrderNumber As _
      System.Web.UI.WebControls.TextBox
   Protected WithEvents txtSupplierName As _
      System.Web.UI.WebControls.TextBox
   Protected WithEvents cmdCreateOrder As _
      System.Web.UI.WebControls.Button

   Private Sub Page_Load(ByVal sender As System.Object, _
      ByVal e As System.EventArgs) Handles MyBase.Load

   End Sub

   Private Sub cmdCreateOrder_Click( _
      ByVal sender As System.Object, _
      ByVal e As System.EventArgs) _
      Handles cmdCreateOrder.Click

      Dim objCreateOrder As ICreateOrder
      objCreateOrder = New CreateOrder()

      If objCreateOrder.Create(txtOrderNumber.Text, _
                           txtSupplierName.Text) Then
         ContextUtil.SetComplete()
      Else
         ContextUtil.SetAbort()
      End If
   End Sub
End Class
```

Similar to the test Web Form for the `Product` component, this code creates a `CreateOrder` object using `New`. The program calls the `Create()` method and then calls `SetComplete()` or `SetAbort()` upon success or failure, respectively.

Crouch05_393-458.ps 4/23/02 5:59 PM Page 445

`AddToOrder` Test Web Form

The HTML code for the `AddToOrder` Web Form (`Lcs5-7.aspx`) appears below.

LAB CODE SAMPLE 5-7

```
<%@ Page Language="vb"
   AutoEventWireup="false"
   Codebehind="SupermarketAddToOrder.aspx.vb"
   Inherits="SupermarketWeb.SupermarketAddToOrder"
   Transaction="RequiresNew"%>
<html>
<head>
<title>Add To Order</title>
</head>
<body>

<form id="Form1"
   method="post"
   runat="server">

<p>Order Number:
<asp:TextBox
   id=txtOrderNumber
   runat="server">
</asp:TextBox></p>

<p>SKU:
<asp:TextBox
   id=txtSKU
   runat="server">
</asp:TextBox></p>

<p>Quantity:
<asp:TextBox
   id=txtQuantity
   runat="server">
</asp:TextBox></p>

<p>
<asp:CompareValidator
   id=CompareValidator1
   runat="server"
   ErrorMessage="Quantity must be a whole number!"
   ControlToValidate="txtQuantity"
   Type="Integer"
   Operator="DataTypeCheck">
</asp:CompareValidator></p>

<p>
<asp:Button
   id=cmdAddToOrder
   runat="server"
   Text="Add To Order">
</asp:Button></p>

</form>
</body>
</html>
```

Here is the associated code-behind file (`Lcs5-7.aspx.vb`). `AddItems()` is a method of the `CreateOrder` component, so the code is very similar to the `CreateOrder` Web Form.

```
Imports System.EnterpriseServices
Imports System.Reflection
Imports SupermarketSystem.Supermarket

Public Class SupermarketAddToOrder
   Inherits System.Web.UI.Page
   Protected WithEvents txtOrderNumber As _
      System.Web.UI.WebControls.TextBox
   Protected WithEvents txtSKU As _
      System.Web.UI.WebControls.TextBox
   Protected WithEvents txtQuantity As _
      System.Web.UI.WebControls.TextBox
   Protected WithEvents CompareValidator1 As _
      System.Web.UI.WebControls.CompareValidator
   Protected WithEvents cmdAddToOrder As _
      System.Web.UI.WebControls.Button

   Private Sub Page_Load(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles MyBase.Load

   End Sub

   Private Sub cmdAddToOrder_Click( _
        ByVal sender As System.Object, _
        ByVal e As System.EventArgs) _
        Handles cmdAddToOrder.Click

      Dim objCreateOrder As ICreateOrder
      objCreateOrder = New CreateOrder()

      If objCreateOrder.AddItems(txtOrderNumber.Text, _
         txtSKU.Text, CInt(txtQuantity.Text)) Then

         ContextUtil.SetComplete()
      Else
         ContextUtil.SetAbort()
      End If
   End Sub
End Class
```

`ReceiveOrder` Component Test Web Form

Finally, here is the code (`Lcs5-8.aspx`) for the Web Form that will receive items for an order.

LAB CODE SAMPLE 5-8

```
<%@ Page Language="vb"
   AutoEventWireup="false"
   src="SupermarketReceiveOrder.aspx.vb"
   Inherits="SupermarketReceiveOrder"
   Transaction="RequiresNew"%>
<html>
<head>
<title>Receive Order</title>
</head>
<body>

<form id="Form1"
   method="post"
   runat="server">

<p>Order Number to Receive:

<asp:TextBox
   id=txtOrderToReceive
   runat="server">
</asp:TextBox>
```

```
<asp:Button
   id=cmdGetOrder
   runat="server"
   Text="Get Order">
</asp:Button></p>

<p>
<asp:Panel
   id=Panel1
   runat="server"
   Width="399px"
   Height="144px"
   Enabled="False">

   <p>
   <asp:Label
   id=lblOrderNumber
   runat="server"
   Width="184px"
   Height="19px">
   </asp:Label></p>

   <p></p><p>
   <asp:Label
   id=lblReceiveSKU
   runat="server"
   Width="183px"
   Height="19px">
   </asp:Label></p>

   <p>
   <asp:Label
   id=lblLineNumberReceive
   runat="server"
   Width="188px"
   Height="19px">
   </asp:Label></p>

   <p>
   <asp:Label
   id=Label1
   runat="server"
   Width="128px"
```

Crouch05_393-458.ps 4/23/02 5:59 PM Page 449

```
   Height="19px">
   Quantity To Receive:
   </asp:Label>

   <asp:TextBox
   id=txtQuantityToReceive
   runat="server">
   </asp:TextBox>

   <asp:Button
   id=cmdReceive
   runat="server"
   Text="Receive">
   </asp:Button>

</asp:Panel></p>

</form>
</body>
</html>
```

This Web Form wraps page elements in a `Panel` Web Control. The panel is initially disabled to avoid displaying or enabling the order information until a valid order number is keyed into the Web Form. Here's the code-behind file (`Lcs5-8.aspx.vb`).

```
Imports System.EnterpriseServices
Imports System.Reflection
Imports SupermarketSystem.Supermarket

Public Class SupermarketReceiveOrder
   Inherits System.Web.UI.Page
   Protected WithEvents cmdGetOrder As _
      System.Web.UI.WebControls.Button
   Protected WithEvents Panel1 As _
      System.Web.UI.WebControls.Panel
   Protected WithEvents lblOrderNumber As _
      System.Web.UI.WebControls.Label
   Protected WithEvents Label1 As _
      System.Web.UI.WebControls.Label
   Protected WithEvents txtOrderToReceive As _
      System.Web.UI.WebControls.TextBox
   Protected WithEvents cmdReceive As _
      System.Web.UI.WebControls.Button
```

```
Protected WithEvents lblReceiveSKU As _
   System.Web.UI.WebControls.Label
Protected WithEvents lblLineNumberReceive As _
   System.Web.UI.WebControls.Label
Protected WithEvents txtQuantityToReceive As _
   System.Web.UI.WebControls.TextBox

Private Sub Page_Load(ByVal sender As System.Object, _
   ByVal e As System.EventArgs) Handles MyBase.Load

End Sub

Private Sub cmdGetOrder_Click( _
   ByVal sender As System.Object, _
   ByVal e As System.EventArgs) _
   Handles cmdGetOrder.Click

   Dim objReceiveOrder As IReceiveOrder
   Dim intLineNumber As Integer
   Dim strSKUToReceive As String

   objReceiveOrder = New ReceiveOrder()

   intLineNumber = _
        objReceiveOrder.GetNextLineItem( _
        txtOrderToReceive.Text, _
        strSKUToReceive)

   If intLineNumber <> -1 Then
      ViewState("OrderToReceive") = txtOrderToReceive.Text
      ViewState("SKUToReceive") = strSKUToReceive
      ViewState("LineNumber") = intLineNumber
      Panel1.Enabled = True
      lblLineNumberReceive.Text = _
           "Line Number: " & intLineNumber
      lblOrderNumber.Text = _
           "Order Number: " & txtOrderToReceive.Text
      lblReceiveSKU.Text = "SKU: " & strSKUToReceive
   Else
      Panel1.Enabled = False
   End If
End Sub

Private Sub cmdReceive_Click( _
      ByVal sender As System.Object, _
      ByVal e As System.EventArgs) _
      Handles cmdReceive.Click

      Dim objReceiveOrder As IReceiveOrder
      objReceiveOrder = New ReceiveOrder()

      If objReceiveOrder.Receive( _
               ViewState("OrderToReceive"), _
               ViewState("SKUToReceive"), _
               ViewState("LineNumber"), _
               CInt(txtQuantityToReceive.Text)) Then

         ContextUtil.SetComplete()
      Else
         ContextUtil.SetAbort()
      End If
   End Sub
End Class
```

This form uses two `Button` Web Controls, `cmdGetOrder` and `cmdReceive`. This makes a two-step process for receiving items for an order. First, the event handler for `cmdGetOrder` calls `GetNextLineItem()`, taking as input the order number the user entered. If there is a line item to receive for the order, the program displays the line-item information in the `Label` Web Control contained within the `Panel` Web Control. The `Panel` Web Control is then enabled, making the information visible to the user. The line-item information is also copied to the `ViewState StateBag` because the program will need this information on the subsequent post-back that will occur when items are received.

The event handler for `cmdReceive` calls the `Receive()` method. The `Receive()` method updates the `OrderDetails` table as well as the `Inventory` table. Using a transaction in this situation helps point out any discrepancies between quantities in inventory and quantities ordered. The `Receive()` method returns `True` on success and `False` on failure, and the program makes an appropriate call to either `SetComplete()` or `SetAbort()` as a result.

d. Now you need to add a reference to the assembly DLL for the components. Right-click the References folder in the Solution Explorer, select **Add Reference**, browse to the compiled DLL, and select it. Click OK to add the reference to the selected assembly.

STEP 5. Run the application.

a. Now you're ready to build the application. Select **Build→Build Solution** from the menu. Select a start page for the application (like `Lcs5-5.aspx`) by right-clicking on a Web Form in the Solution Explorer and selecting **Set As Start Page** from the menu.

b. Run the application by selecting **Debug→Start Without Debugging.**

c. Test the application by entering some products. Then create a new order, add some items to the order, and run an order-receive process. This should complete a full test.

Something important happened when you first called a component's method inside the assembly. The system performed what is known as a *lazy registration*. A lazy registration automatically places the components in the assembly into a new COM+ application in the COM+ Component Services Console. It does this based on the assembly-level attributes specified in the component assembly code. Lab Figure 5–4 shows the Supermarket COM+ application in the COM+ Component Services Console.

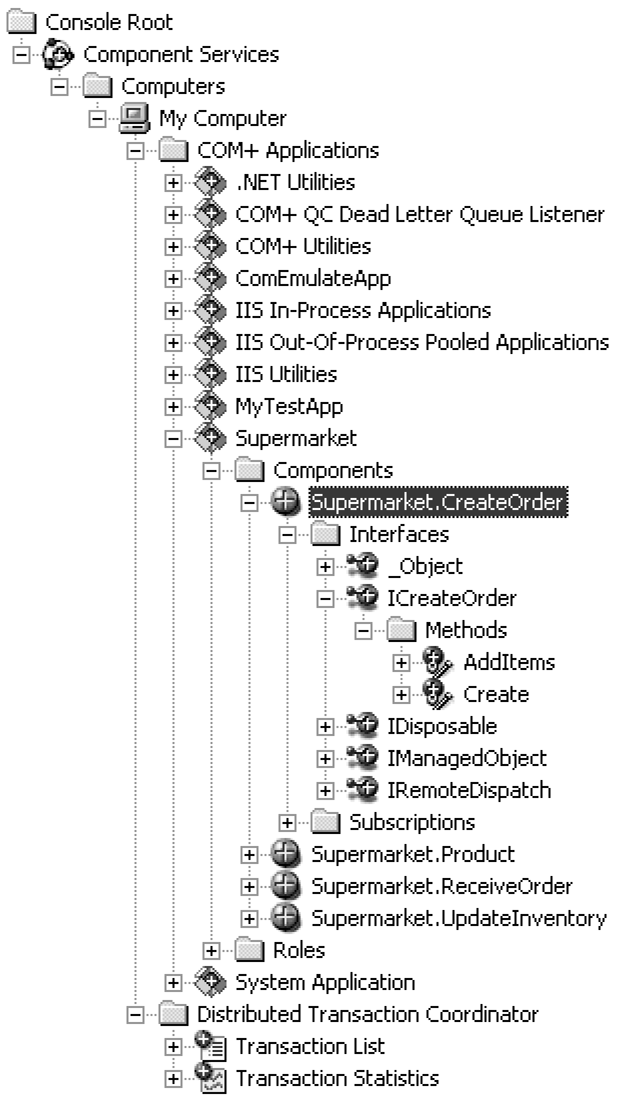
STEP 6. Add role-based security.

a. One of the requirements for the application is that only certain categories of users should be allowed to run certain components. To enable role-based security for the Supermarket COM+ application, right-click on the Supermarket COM+ application icon and select **Properties**. Click the **Security** tab. Check the boxes and radio buttons as shown in Lab Figure 5-5.

b. Now you can set up the user roles (Receiving Clerks, Supervisors, and Suppliers) inside the COM+ Component Services Console. For each role, right-click the Roles folder under the Supermarket application (see Lab Figure 5-6), select **New→Role** from the menu, and assign a name for the role.

c. Assign users to each role by right-clicking the User folder under the role and selecting **New→User** from the menu. Pick a Windows account name(s) or group(s) to assign to the role. Lab Figure 5-7 shows users assigned to the various roles of the Supermarket application.

d. Now you need to assign each role to a component. Right-click a component in the COM+ application and select **Properties** from the menu. Click the **Security** tab. Check **Enforce component level access checks**, and then check the roles you wish to assign to the component, as shown in Lab Figure 5-8.



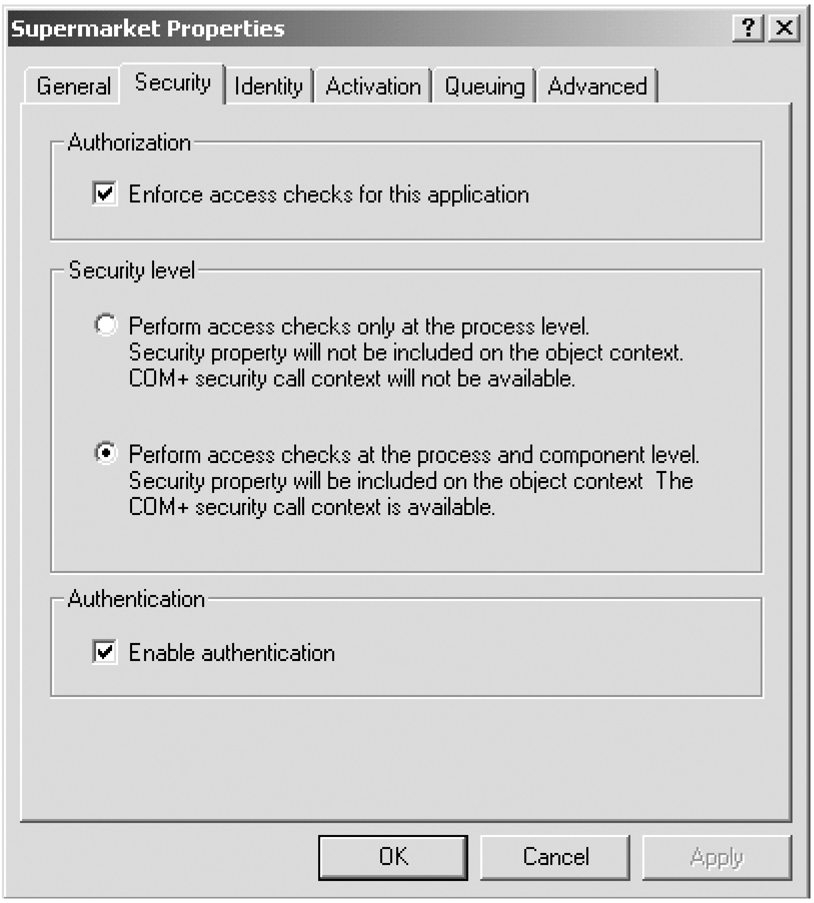
```
Console Root
  Component Services
    Computers
      My Computer
        COM+ Applications
          .NET Utilities
          COM+ QC Dead Letter Queue Listener
          COM+ Utilities
          ComEmulateApp
          IIS In-Process Applications
          IIS Out-Of-Process Pooled Applications
          IIS Utilities
          MyTestApp
          Supermarket
            Components
              Supermarket.CreateOrder
                Interfaces
                  _Object
                  ICreateOrder
                    Methods
                      AddItems
                      Create
                  IDisposable
                  IManagedObject
                  IRemoteDispatch
                Subscriptions
              Supermarket.Product
              Supermarket.ReceiveOrder
              Supermarket.UpdateInventory
            Roles
          System Application
        Distributed Transaction Coordinator
          Transaction List
          Transaction Statistics
```

Lab Figure 5-4 The Supermarket application within the COM+ Component Services Console

STEP 7. Test the role-based security.

a. A convenient way to test the role-based security is to call the components from a console application and run the console application in the security

Crouch05_393-458.ps 4/23/02 5:59 PM Page 454



Authorization

[✓] Enforce access checks for this application

Security level

○ Perform access checks only at the process level.
Security property will not be included on the object context.
COM+ security call context will not be available.

● Perform access checks at the process and component level.
Security property will be included on the object context The
COM+ security call context is available.

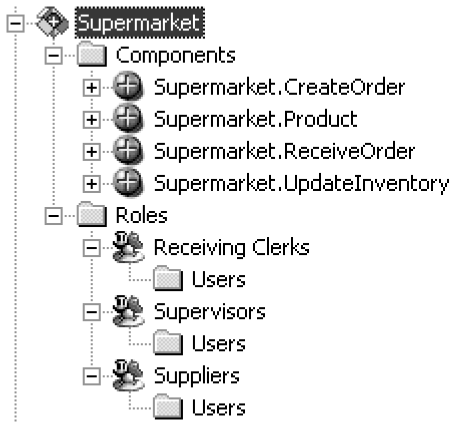
Authentication

[✓] Enable authentication

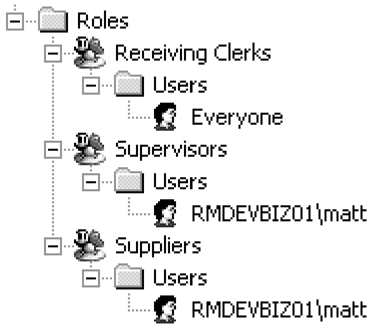
Lab Figure 5-5 Enabling role-based security for the Supermarket application

context of a specific user. Normally, when you run an application from the console, Windows uses the security context of the currently logged-on user. By using the `runas` command, you can run an executable program using any account name (provided, of course, you have the password of that account!). Here's a simple console application you can run to test role-based security.

```
Imports System
Imports SupermarketSystem.Supermarket
Imports System.EnterpriseServices

Module Module1
```

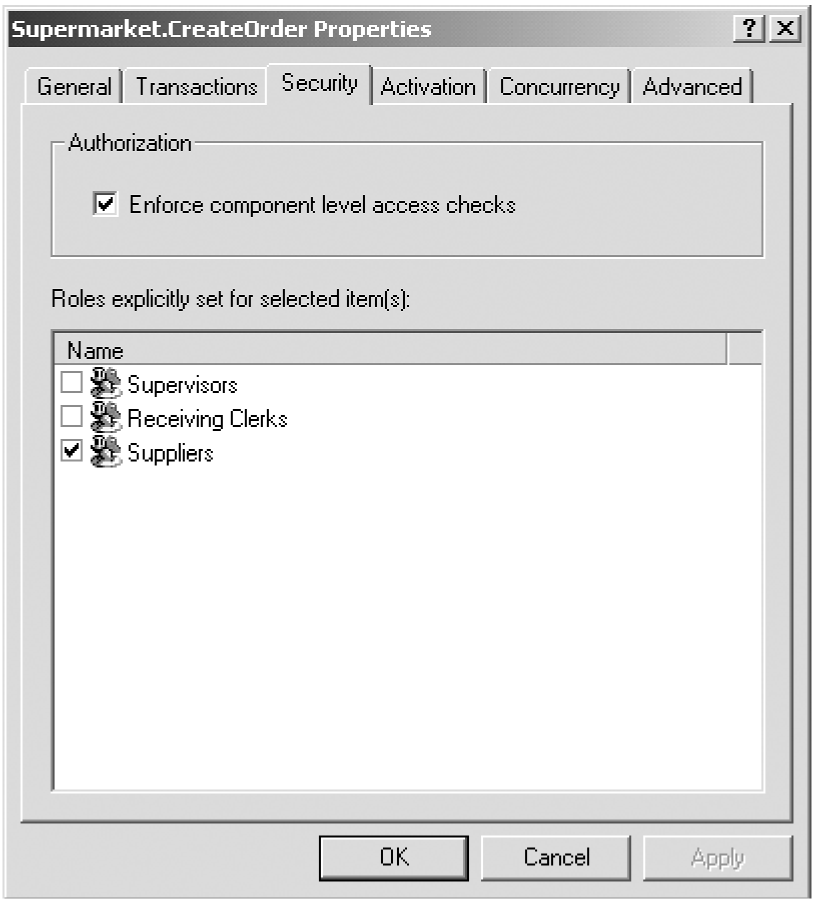


Lab Figure 5-6 Adding roles for the Supermarket application



Lab Figure 5-7 Assigning users to roles

```
    Sub Main()
       Dim objCreateOrder As ICreateOrder

       objCreateOrder = New CreateOrder()

       objCreateOrder.Create("834957239-1", "My Supplier")

    End Sub

End Module
```



Authorization

[✓] Enforce component level access checks

Roles explicitly set for selected item(s):

Name
☐ Supervisors
☐ Receiving Clerks
✓ Suppliers

Lab Figure 5-8 Assigning roles to components

Compile this small testing console application and run it using the `runas` command below:

```
runas /user:rmdevbiz01\matt roletester.exe
```

Given the role configuration of the `CreateOrder` component, the username “matt” should be the only user allowed to run the application. Assuming that the user account “matt” has appropriate Windows permissions set to run the program, it should execute without errors.

b. You can also perform a negative test by rerunning the program with an account name not included in the Supplier role.

5.7 Summary

Let's recap what you've learned in this chapter about writing .NET managed components.

- Managed code is application code that executes within the .NET Framework.
- Runtime systems enable application programs you write to access operating system–level services. Several different runtime environments are available for various systems. A runtime environment is usually specific to a programming language.
- Intermediate languages provide an additional layer of abstraction over another existing runtime environment. The goal of intermediate languages is to provide portability of application code across operating systems. The intermediate language for the .NET Framework is called the *Microsoft Intermediate Language* (MSIL).
- The runtime environment for the .NET Framework is called the *Common Language Runtime* (CLR). It supports self-describing code through metadata, a specification for the Common Type System (CTS), just-in-time compilation of code, and the concept of assembly manifests, which describe a packaged set of components, programs, files, and other dependencies an application needs in order to function properly.
- COM is a programming model invented by Microsoft that allows for binary-level software component reuse.
- Transactions provide consistency and data integrity to a software system. By checking the success or failure of smaller atomic tasks that make up the transaction, you can decide whether the transaction should succeed as a whole.
- Windows provides programmers with easy transaction support for their applications through COM+ Component Services. COM+ Component Services provides transaction support, object pooling, and component-level runtime security.
- Components are simply classes at their most basic level. Components have properties and methods, which roughly correspond to member functions and member variables, respectively.
- Serviced components are classes that take advantage of COM+ Component Services. Security is provided by grouping users into roles and assigning those roles to components.

Crouch05_393-458.ps 4/23/02 5:59 PM Page 458

5.8 What's Ahead

In Chapter 6 we'll explore Web Services, the most important part of the .NET Framework. Chapter 7 covers data access with ADO.NET, and we'll wrap up with .NET Web application security topics in Chapter 8.