

## Chapter 3: Objects, Components, and COM

### Chapter 3: Objects, Components, and COM

Whether you are a seasoned object-oriented developer or someone who is just coming up to speed, it is important to understand some of the basic features that object-oriented programming imparts. Because we will be leveraging these features heavily in our framework in Part II of this book, we are providing a suitable overview in this chapter. To add depth, we will intersperse notes and tips relative to using object-orientation to your advantage in building enterprise applications throughout this chapter.

#### Object Orientation

Object orientation is not an entirely new concept, but it is becoming more prevalent in the underpinnings of modern applications. It has just been within the last ten years or so that object-orientation migrated from academia and experimentation to a true, commercial-grade development methodology. Since then, non-object-oriented development has moved into the minority position.

---

**Note - One important thing to remember is that simply using an object-capable language does not constitute object-oriented development. In addition, simply defining classes within an object-capable language, without taking advantage of the power of object-orientation, does not necessarily make an enterprise application more robust.**

---

To start a definition of object-orientation is to understand that it is rooted in the management of complexity. Modern applications, with their intrinsic business logic and interactions among data elements, can become burdensome to develop and maintain in a traditional procedural environment. Sometimes just the analysis of the business problem domain can become increasingly overwhelming as the system's scope grows from one of simple data management to one that embodies business process knowledge. Object-orientation helps throughout application development by allowing us to use a similar thought process across the analysis, design, and implementation phases. The basic pillars of object-orientation are abstraction, encapsulation, polymorphism, and inheritance. We will discuss these features of object-orientation and how they enable us to build modular, maintainable, and extensible applications.

#### Abstraction and Class Modeling

What is most striking about object-orientation is that it follows the true sense of the business world. In this world, anything that a business deals with, whether it is a widget that a company produces or a financial account that a bank maintains on behalf of a client, is definable in software terms through a class model. This class model defines the information pertinent to the business entity, along with the logic that operates on that information. Additionally, a class definition can contain references to one or more external classes through association or ownership relationships. In the case of a financial account, informational elements might include the account number, the names of the account owners, the current balance, the type of account, and so on. We call these items *properties* (also known as *attributes*) of the class. Similarly, the class can define a function to add a new transaction to the account or modify/delete an existing transaction. We call these items *methods* (also known as *operations*) of the class. What differentiates a class from an object is that a class is a definition, whereas an object is an instance of that definition.

We can also graphically represent our objects using a class diagram. There are many different views on how to represent these diagrams, but the most pervading forms are the Yourdon/Coad and the Rumbaugh methods, named after the individuals who developed them. Many drawing programs have templates predefined for these models, whereas many modeling tools can support some or all of the more popular styles. You can also create your own object modeling technique using simple lines and boxes. We have chosen to use the Rumbaugh model in this book because of the popularity of the Unified Modeling Language (UML), of which it is a component. It also happens to be the model used by the Microsoft Visual Modeler that is bundled with Visual Studio 6.0 Enterprise

Edition. [Figure 3.1](#) shows an example of a graphical depiction for a financial account class.

### **Figure 3.1**

The `CAccount` class using the UML graphical model.

---

**Tip** - It is important to decide on a graphical object representation model early in the project. Make sure that everyone understands how to read it and feels comfortable with it. This model will be your roadmap as you work your way through all phases of the application development process and will be critical as the complexity starts to build.

---

As you can see, we modeled our real-world Account business entity in terms of properties and methods. We call this modeling process *abstraction*, which forms the basis for object orientation. With this in mind, we can further our discussion of other features of object-orientation.

## **Encapsulation**

What should be apparent from [Figure 3.1](#) is that we have bundled everything about the class into a nice, neat package. We formally define everything that the outside world needs to know about this class in terms of these properties and methods. We call the public properties and methods of a class its interface, which represents the concept of encapsulation. In the real-world account example, a customer does not necessarily need to know how the current balance is calculated based on transactions that are added, modified, or deleted. They just need to know their current balance. Similarly, users of the account class do not need to know how the class calculates the current balance either—just that the class properly handles it when the transaction processing methods are called. Thus, we can say that encapsulation has the effect of information hiding and the definition of narrow interfaces into the class. This concept is critical to the development of robust, maintainable applications.

A class might implement internal methods and properties but choose not to expose them to the outside world through its interface. Because of this, we are free to change the internal workings of these private items without affecting how the outside world uses our class through its public interface. [Figure 3.2](#) shows how a public method calls a private method to perform a calculation that updates the value of a public property.

### **Figure 3.2**

The interactions between public and private methods and properties.

Suppose, for the sake of argument, we were to expose the internal function (also known as a private method) that calculates current balances. We would do this by defining it to be public versus private. An application using this class, for whatever reason, might deem it acceptable to call this internal method directly and does so in a multitude of places. Now suppose that we must change the calling convention of this method by adding a new parameter to the parameter list, such that we have to modify every piece of software that references this internal method. Assume also that the public transaction methods would not have had to change, only the formerly private method. We have effectively forced ourselves into a potentially large code rewrite, debug, test, and deployment cycle that we could have otherwise handled simply within the object's private methods while leaving the public interface intact. We will see, in the COM model discussion to follow, that we can easily modify only the class and redeploy it across the user base with a minimum of effort. In the corporate world, this translates into time and money.

Because the term *interface* might be a difficult concept to grasp at first, it might be easier to think of as an electrical socket. In the 220-volt parts of the world, there are three-pronged sockets with one of the prongs oriented 90 degrees out from the other two. In the 110-volt parts of the world, there are two- and three-pronged plugs with a different geometry such that you cannot plug a 110-volt oriented plug into a 220-volt socket and vice-versa. Imagine if the 110-volt world suddenly began using 220-volt-style plugs and sockets (assuming voltage will not change). We would have to replace the plug on every electrical device along with all the wall sockets. It would be a huge mess. The same goes for properties and methods. After we define the interfaces of a class and write applications against them, making changes becomes difficult and costly.

---

**Tip** - When defining a class, assume every method is to be defined as private in scope (that is, hidden) unless there is good reason to make it public. When making a method public, take steps to ensure the stability of the calling convention (that is, the parameter list) over the life of the application. Use optional parameters as necessary to cover anticipated future needs.

---

Encapsulation also has the effect of protecting the integrity of objects, which are instantiated using the class definition. We have already touched on this when we stated that a class is responsible for its own inner workings. Outsiders cannot meddle in its internal affairs. Similarly, property definitions can be implemented such that the class rejects invalid property states during the setting process. For example, a date-based property could reject a date literal, such as "June 31, 1997," because it does not constitute a date on any calendar. Again, because the validation logic is contained within the class definition itself, modifying it to meet changing business needs occurs in a single place rather than throughout the application base. This aspect of encapsulation is important, especially for enterprise applications, when we discuss the implementation of validation logic in Chapter 9, "A Two-Part, Distributed Business Object." It further adds to our ability to develop robust, maintainable, and extensible applications.

---

**Note** - One of the common comments that newcomers to object-oriented development make is that it seems like unnecessary effort to package data and functionality together into a unit called a class. It also seems like extra work to define properties and methods, deciding what is to be public and what is to be private. It is much easier to just take a seat behind the keyboard and begin banging out some code. Although it is true that object-oriented development requires a different mindset and a somewhat formal approach to analysis and design, it is this formalization process that leads to less complex development over the long term. The old saying "penny-wise and dollar-foolish" applies here because some time saved up front will lead to potentially huge problems further into the development, and worse yet, the application launch process.

---

Let us switch gears by defining a class with some complexity—with a financial bond—so we can illustrate some other points and begin setting the stage for other features of object-orientation. Let us call it `CBond` (for Class Bond). We define several properties in tabular form in Table 3.1, methods in Table 3.2, and we provide a graphical depiction in Figure 3.3.

**Table 3.1 Properties of a *CBond* Class**

Property	Data Type	Description
Name	String	The descriptive name of the bond.
FaceValue	Single (Currency)	The final redemption value of the bond.
PurchasePrice	Single (Currency)	The price to purchase the bond.
CouponRate	Single (Percent)	The yearly bond coupon payment as a percentage of its face value.
BondTerm	Integer	The length of time to maturity for the bond, expressed in years, in the primary market.
BondType	Integer: (Enumeration)	The bond type used to drive calculation algorithms.

[CouponBond, DiscountBond, ConsolBond])
---

**Table 3.2 Methods of a *CBond* Class**

Method	Description
YieldToMaturity	Calculates the interest rate that equates the present value of the coupon payments over the life of the bond to its value today. Used in the secondary bond market.
BondPrice	Calculates the bond price as the sum of the present values of all the payments for the bond.
CurrentYield	Calculates the current yield as an approximation of the yield to maturity using a simplified formula. Note: Available only on <i>CouponBond</i> types.
DiscountYield	Calculates the discount yield based on the percentage gain on the face value of a bond and the remaining days to maturity.

Each method uses one or more of the public property values to perform the calculation. Some methods require additional information in the form of its parameter list, as can be seen in [Figure 3.3](#). As you might guess, the `BondType` property helps each method determine how to perform the calculation. A sample Visual Basic implementation of the `BondPrice` method might be as follows in [Listing 3.1](#).

**Figure 3.3**

UML representation of a *CBond* class.

**Listing 3.1 The *BondPrice* Method**

```
Public Function BondPrice(IntRate as Single) as Single
    Dim CouponPayment as Single
    Dim j as integer
    Dim p as single
    CouponPayment = CouponRate * FaceValue
    Select Case BondType
    Case btCouponBond
        For j = 1 to BondTerm
            p = p + CouponPayment/(1 + IntRate)^j
        Next j
        p = p + FinalValue/(1 + IntRate)^BondTerm
        BondPrice = p
    Case btDiscountBond
        BondPrice = FaceValue/(1 + IntRate)
    Case btConsolBond
        BondPrice = CouponPayment/IntRate
    End Select
End Sub
```

As you can see, each value of the `BondType` property requires a different use of the properties to perform the correct calculation. The application using the class is not concerned with how the method performs the calculation, but only with the result. Now suppose that you need to modify the calculation algorithm for the `BondPrice` method. Because of encapsulation, you only need to modify the contents of the `BondPrice` method and nothing more. Better yet, because you have not changed the calling convention, the applications using the *CBond* class are none the wiser

that a change occurred.

## Polymorphism

Polymorphism is another standard feature of object-oriented programming. Fundamentally, polymorphism means the capability to define similar properties and methods on dissimilar classes. In essence, we define a common interface on a set of classes such that a calling application can use these classes with a standard set of conventions. Because this sounds complex, let us provide an example.

Suppose you are developing classes that must interact with a relational database. For each of these classes, there can be a standard set of methods to retrieve property values for an object instance from a database. We call this process of storing and retrieving property values *object persistence*, a topic we will discuss in detail in Chapter 5, "Distribution Considerations." We can illustrate an abstract definition of a couple of methods as follows:

```
Public Function RetrieveProperties(ObjectId As Long) As Variant
    ' code to retrieve the property values
End Function

Public Sub SetStateFromVariant(ObjectData As Variant)
    ' code to set the property values from ObjectData
End Sub
```

For each class that is to follow this behavior, it must not only define, but also provide the implementation for these two methods. Suppose you have three such classes—CClassOne, CClassTwo, and CClassThree. An application that creates and loads an object might implement polymorphic code in the following manner (see Listing 3.2).

### Listing 3.2 The *RetrieveObject* Method

```
Public Function RetrieveObject(ClassType As Integer,
    ObjectId As Long) As Object
    Dim OClassAny As Object
    Dim ObjectData as Variant
    Select Case ClassType
    Case CLASS_TYPE_ONE
        Set OClassAny = New CClassOne
    Case CLASS_TYPE_TWO
        Set OClassAny = New CClassTwo
    Case CLASS_TYPE_THREE
        Set OClassAny = New CClassThree
    End Select
    ObjectData = OClassAny.RetrieveProperties(ObjectId)
    Call OClassAny.SetStateFromVariant(ObjectData)
    SetRetrieveObject = OClassAny
End Function
```

In the preceding code example, we use a technique known as *late binding*, wherein Visual Basic performs type checking at runtime rather than at compile time. In this mode, we can declare a generic object (a variable type intrinsic to Visual Basic) to represent the instantiated object based on any of the three class definitions. We must assume that each of these classes defines and implements the `RetrieveProperties` and `SetStateFromVariant` methods as mandated by our polymorphic requirements. If the classes deviate from these conventions, a runtime error will occur. If the classes meet these requirements, we can simplify the coding of the object retrieval process into a single function call on the application. This not only leads to code that is easier to maintain over the life of the application, but also makes extending the application to support new class types much simpler.

The late binding technique of Visual Basic presents us with some concerns. Because late binding performs type checking at runtime, some errors might escape early testing or even propagate into the production application. Furthermore, late binding has a performance penalty because Visual Basic must go through a process known as *runtime discovery* with each object reference to determine the actual methods and properties available on the object. This said, we should scrutinize the use of late-binding approaches in the application wherever possible and choose alternative approaches. We will discuss several approaches to circumvent these issues when we discuss the framework components in Part II of the book.

## Inheritance

The final pillar of object orientation is that of inheritance. Fundamental to this concept is the capability to define the common methods and properties of a related group of classes in a base class. Descendants of this base class can choose to retain the implementation provided by the base class or can override the implementation on its own. In some cases, the base class provides no implementation whatsoever, and it is focused solely on the definition of an interface. We consider these types of base classes abstract because each subclass must provide the complete implementation. Regardless of the mode, the descendent class must maintain the definition of all properties and methods of its base class. Said in another way, the descendent class must define the same interface as its base. This is similar in concept to polymorphism, except that inheritance forces the implementation in a formal manner, such that Visual Basic can perform type checking at compile time.

Looking again at our `CBond` class, we notice that there is a `BondType` property to force certain alternative behaviors by the calculation methods. We can modify our `CBond` class into a single `IBond` base class and three subclasses called `CCouponBond`, `CDiscountBond`, and `CConsolBond`. We use `IBond` here (for Interface Bond) instead of `CBond` to coincide with Microsoft's terminology for interface implementation. Graphically, we represent this as shown in [Figure 3.4](#).

**Figure 3.4**

An inheritance diagram for the `IBond` base class.

If we revisit our bond calculation functions in the context of inheritance, they might look something like [Listing 3.3](#). Disregard the `IBond_` syntax for now because it is a concept that we gain a thorough understanding of in our work in Part II of this book.

### Listing 3.3 The *CalculateBondPrice* Method

```
' From the application
Public Function CalculateBondPrice(BondType as Integer, _
    InRate as Single) As Single
    Dim OBond As IBond
    Select Case BondType
    Case BOND_TYPE_COUPON
        Set OBond = New CCouponBond
    Case BOND_TYPE_DISCOUNT
        Set OBond = New CDiscountBond
    Case BOND_TYPE_CONSOL
        Set OBond = New CConsolBond
    End Select
    CalculateBondPrice = OBond.BondPrice(InRate)
End Function

' From CCouponBond
Implements IBond
Public Function IBond_BondPrice(InRate As Single) As Single
    Dim CouponPayment as Single
    Dim j as integer
    Dim p as single
    CouponPayment = IBond_CouponRate * IBond_FaceValue
    For j = 1 to IBond_BondTerm
        p = p + CouponPayment/(1 + InRate)^j
    Next j
    p = p + IBond_FinalValue/(1 + InRate)^IBond_BondTerm
    IBond_BondPrice = p
End Function

' From CDiscountBond
Implements IBond
Public Function IBond_BondPrice(InRate As Single) As Single
    IBond_BondPrice = FaceValue/(1 + InRate)
End Function

' From CConsolBond
Implements IBond
Public Function IBond_BondPrice(InRate As Single) As Single
    Dim CouponPayment as Single
    CouponPayment = IBond_CouponRate * IBond_FaceValue
    IBond_BondPrice = CouponPayment/InRate
```

End Function

Although the application portion of this example might look somewhat similar to the polymorphic mechanism from before, there is an important distinction. Because we have defined these subclasses in the context of a base class `IBond`, we have forced the interface implementation of the base class. This, in turn, allows Visual Basic to perform early binding and therefore type checking at compile time. In contrast to late binding, this leads to better application performance, stability, and extensibility.

---

**Tip** - Any class definition that contains a `TYPE` property is a candidate for inheritance-based implementation.

---

Critics have chastised Microsoft for not implementing inheritance properly in Visual Basic in that it does not support a subclass descending from more than one base class, a concept known as multiple-inheritance. Although this lack of implementation technically is a true statement, in reality, multiple inheritance scenarios arise so infrequently that it is not worth the extra complexity that Microsoft would have had to add to Visual Basic to implement it.

Many critics would further argue that Visual Basic and COM, through their interface implementation technique, do not even support single inheritance properly and that the notion of the capability to subclass in this environment is ludicrous. Without taking a side in this debate, we can sufficiently state that interface implementation gives you some of the features afforded by single-inheritance, whether or not you want to formally define them in this manner. The particular side of the debate you might fall into is immaterial for the purposes of our framework development in Part II of this book.

Interface inheritance lends itself to maintainability and extensibility—essential attributes of enterprise applications as discussed in Chapter 1, "An Introduction to the Enterprise." If the implementation of a base method or property must change, we have to make the modifications only to the base class. Each descendent then inherits this new implementation as part of its interface implementation. If the base class physically resides in a different component than its descendants, something we will discuss later in this chapter, we only have to redeploy the component defining the base class.

## Association Relationships

After we have defined the basics of classes with simple property types, we can expand our view to show that classes can have associative relationships with other classes. For example, a class might reference another class in a one-to-one manner, or a class might reference a group of other classes in a one-to-many fashion.

### One-to-One Relationships

We might consider one-to-one relationships as strong or weak in nature. Weak relationships are just simple references to other classes that are shareable across multiple object instances. For example, a `CPerson` class can be referenced by many other classes, with a particular `OPerson` instance being referenced by multiple object instances of disparate classes. Strong relationships, on the other hand, are usually the result of containment relationships, where one object is the sole user of a subordinate object. In an automotive manufacturing application that tracks the serial numbers of finished units, an example might include the `CSerializedEngine` and `CSerializedAutomobile` classes, where each `OSerializedEngine` object can belong to only one `OSerializedAutomobile` object. [Figure 3.5](#) shows a weak reference, whereas [Figure 3.6](#) shows its strong counterpart.

In [Figure 3.5](#), we show a graphical representation of a weak reference. In this example, the `CPerson` class (and thus, object instances based on the class) is referenced by both the `CAccount` and `CLoan` classes. In the real world that forms the basis for this mini-model, the relationship diagram indicates that it is possible for the same person to have both a checking account and a house or car loan at the bank. The same person could have multiple accounts or loans at the same bank.

In **Figure 3.6**, we show the graphical representation of a strong, or containment, reference. In this example, we show how a finished, serialized automobile has an engine and transmission, both of which the manufacturer serializes as well for tracking purposes. Each `OSerializedEngine` and `OSerializedTransmission` instance will reference only one instance of the `CSerializedAutomobile` class.

#### **Figure 3.5**

A weak association relationship.

#### **Figure 3.6**

A strong association relationship.

## **One-To-Many Relationships**

One-to-many references occur so often that we have developed a special class, known as a *collection*, to implement this type of relationship, as shown graphically in **Figure 3.7**. In this example, the `CIBonds` class indicates a collection of `IBond` interfaces, each of which can be subclassed as before. This `CIBonds` class has several methods associated with group management, such as `Add`, `Remove`, `Item`, and `Count`. If we defined a `CPortfolio` class, it might have a reference to a `CIBonds` class, as well as `CIStocks` and `CIAssets` classes, each of which are collections of `IBond`, `IStock`, and `IAssset` classes, respectively. Again, each of these final interface classes can be subclassed to provide specific implementations, yet the collection can manage them in their base interface class.

#### **Figure 3.7**

A one-to-many relationship and the collection class.

One-to-many relationships and the collection classes that implement them are synonymous with the master-detail relationships found across many applications. We will be using these collection classes frequently throughout our framework architecture. We will cover collections in detail in Chapter 7, "The ClassManager Library."

## **Class and Object Naming Conventions**

Throughout our discussions in this chapter, we have been alluding to a naming convention for classes and objects without having given any formal definitions. Although the naming convention is arbitrary, it is important to decide on one and adhere to it throughout all phases of the project. This will not only provide a degree of standardization across multiple developers, but also make it easier for developers and maintainers to understand the code without the need for an abundant supply of comments. Standardization is important in classes and objects because the two are often confused. In our examples and throughout the remainder of this book, we will be using an uppercase `c` prefix to denote a class. Similarly, we will be using an uppercase `o` prefix for an object. Furthermore, we will be using the same suffix for both the class and its object instances, as in the case of the `CPerson` class and its `OPerson` instances. For example:

```
Set OPerson = New CPerson
```

## **Component-Based Development**

With some object-orientation fundamentals behind us, we turn our discussion to component-based development (CBD). Many people feel that objects and components are synonymous, when in fact, they are more like siblings. Objects can exist without components, and vice-versa. A component is a reusable, self-contained body of functionality that we can use across a broad application base. Imagine an application suite that has a core piece of functionality contained in an includable source code module. Making a change to this functionality requires that we modify and recompile the source code, testing all applications that are using it. We must then distribute every application that references it. In large applications, this compile time can be extensive. In a component-based model, we can simply modify the component and do the same recompile, test, and distribute just on that component without affecting the applications.

As we alluded in our discussion on layers and tiers in Chapter 2, "Layers and Tiers," a CBD approach has some distinct advantages during the development process. Chief among these is the ability to develop and test

components in isolation before integrated testing.

## Component-Based Development and COM

Object-based CBD allows the packaging of class definitions into a deployable entity. Under the Microsoft Component Object Model (COM) architecture, these packages are special Dynamic Link Libraries (DLLs), a dynamic runtime technology that has been available since the earliest days of Microsoft Windows. Microsoft renamed these COM-style DLLs to ActiveX to indicate that there is a difference. An application gains access to classes in an ActiveX DLL by loading the library containing the class definitions into memory, followed by registration of the classes by the COM engine. Applications can then instantiate objects based on these classes using the COM engine.

The traditional DLL (non-ActiveX) meets the definition for CBD, but it is procedurally based (that is, non-object-based). ActiveX DLLs also meet this definition, being object-based in nature. Because an object-based approach is already rooted in the reusability of functionality, the ActiveX DLL implementation of CBD is widely considered the most powerful and flexible technology when working solely on the Win32 platform.

Although COM is both a component and object engine, it differs from other CBD technologies in that it represents binary reusability of components versus source-code level reusability. Because of its binary basis, we can write COM libraries in any language on the Win32 platform that adheres to the COM specification and its related API. The basic requirement to support the COM API is the capacity of a language to implement an array of function pointers that follow a C-style calling syntax. The COM engine uses this array as a jumping point into the public methods and properties defined on the object. Visual Basic is one of many languages with this capability.

COM actually has two modes of operation: local and remote invocation. The distinction between these two will become important as we discuss distribution in Chapter 6, "Understanding Development Fundamentals and Design Goals of an Enterprise Application."

In local invocation, a component is loaded into the memory space of a single computer. This component can load directly into an application's process space, or it can be loaded in a separate process space with an interprocess communication mechanism. In this latter approach, we must establish a communication channel between the process spaces. In the case of distributed computing, these processes reside on physically different machines, and the communication channel must occur over a network connection. We call the local invocation method an *in-process* invocation, and we call the remote invocation method *out-of-process*. We can actually make a local, out-of-process reference as well, which effectively removes the network portion of the communication channel. Microsoft developed a local, out-of-process mode of invocation for application automation, for example, when a Microsoft Word document activates an embedded Microsoft Excel worksheet.

With in-process servers, an application can reference an object, its methods, and its properties using memory pointers as it shares a memory space with the component. [Figure 3.8](#) depicts the local, in-process invocation.

### Figure 3.8

The local, in-process invocation mode of COM.

In the out-of-process server mode, all data must be serialized (that is, made suitable for transport), sent over the interprocess boundary, and then deserialized. We call this serialization process *marshalling*, a topic that we will cover in detail in Chapter 6. Additionally, the out-of-process mode must set up a "proxy" structure on the application (or client) side, and a "stub" structure on the component (or server) side. [Figure 3.9](#) depicts the local, out-of-process mode.

### Figure 3.9

The local, out-of-process invocation mode of COM.

The reason for this proxy/stub setup is to allow the client and server sides of the boundary to maintain their generic COM programming view, without having to be concerned about the details of crossing a process boundary. In this mode, neither side is aware that a process boundary is in place. The client thinks that it is invoking a local, in-process server. The server thinks that we have called it in an in-process manner. The in-

process mode of COM is fast and efficient, whereas the out-of-process mode adds extra steps and overhead to accomplish the same tasks.

---

**Tip - We should not use an out-of-process approach in speed-critical areas of an application. Examples of where not to use an out-of-process approach would include graphic rendering or genetic algorithm processing.**

---

If the processes reside on different machines, we must add a pair of network interface cards (NICs) to the diagram. Additionally, we must use the remote procedure call (RPC) mechanism to allow the proxy/stub pair to communicate. We refer to the remote, out-of-process mode of COM as Distributed COM (DCOM). [Figure 3.10](#) depicts DCOM. As we might imagine, DCOM is expensive from an overall performance standpoint relative to standard COM.

## COM-Definable Entities

A COM library not only enables us to define classes in terms of properties and methods, but also to define enumerations, events, and interfaces used in inheritance relationships. We already have talked about properties, methods, and interfaces, so let us complete the definition by talking about enumerations and events.

### Figure 3.10

The remote, out-of-process invocation mode of COM.

Enumerations are nothing more than a list of named integral values, no different from global constants. What differentiates them is that they become a part of the COM component. In essence, the COM component predefines the constants needed by the application in the form of these enumerations. By bundling them with the classes that rely on them and giving them human-readable names, we can ensure a certain level of robustness and ease of code development throughout the overall application.

---

**Tip - Use public enumerations in place of constants when they tie intrinsically to the operation of a class. This will keep you from having to redefine the constants for each application that uses the class, because they become part of the COM component itself. Where goes the class, so go its enumerations.**

---

Events defined for a class are formal messages sent from an object instance to its application. The application can implement an event handler to respond to these messages in whatever manner deemed necessary.

---

**Note - Visual Basic and COM define events as part of a class, alongside properties and methods. One might assume then that we can define events on an interface, thereby making them available to classes implementing the interface. Although this is a reasonable assumption and a desirable feature, Visual Basic and COM do not support this. As such, do not plan to use events in conjunction with interface implementation.**

---

## Component Coupling

With the flexibility to place COM classes into components and then have these components reference each other, it can become easy to create an environment of high coupling. *Coupling* occurs when we create a reference from a COM class in one component to the interface of a COM class in another component. Because components are different physical entities, this has the effect of hooking the two components together relative to distribution.

Wherever we distribute a component that references other components, we also must distribute all the referenced components, all their referenced components, and so on. One reason for coupling is that we might not properly group functionality into common components. Functionality that represents a single subpart of the overall business application might be a good candidate for a single component. Alternatively, functionality that represents similar design patterns might belong in a single component.

---

**Tip - It is important during the analysis and design phases to group components based on similar functionality. Although we invariably need to create system-level classes for use by other classes, we should try to minimize the creation of a chain of component references. These chains lead to administration and maintenance issues after the application is in production.**

---

Another issue that leads to coupling is that we try to over-modularize the application by placing small snippets of subparts into components. Beyond the coupling aspects, each ActiveX DLL has a certain amount of overhead to load and retain in memory. Placing functionality in ten components when two would suffice adds unnecessary performance overhead and complexity to your application.

From a performance perspective, we can look at the time necessary to initialize the two scenarios. There are two initialization times to look at: the first is the time required to initialize the component, and the second is the time required to initialize the object. Remembering that a component in the COM world is a specialized DLL, we can infer that some initialization time is associated with the DLL. When Visual Basic must load an ActiveX DLL, it must go through a process of "learning" what objects are defined in the component in terms of properties, methods, and events. In the two scenarios, the 10-DLL case will have five times the load time of the 2-DLL case, assuming negligible differences in the aggregate learning time of the objects within the components.

From a complexity perspective, the more components created means more work on the development team. One of the problematic issues with any object-oriented or interface implementation project is that of recompilation and distribution when something changes, especially in the early development phases of the application. For example, if the definition of a core class referenced throughout the project changes, it is much easier to recompile the two components versus the ten. As you might already know from multitiered development in the DCOM environment, propagating such seemingly simple changes across tiers can be very difficult. Thus, appropriate minimization of the number of components up front is desirable.

We are not trying to say that you should place all your functionality into one component—this leads to its own set of problems. The moral of the story is that one should not force modularity purely for the sake of doing so. You should find an appropriate balance that can come only from experience in developing these sorts of systems. The framework presented in Part II is a good starting point for understanding where these lines of balance should be drawn.

When we need to provide a superset of functionality based on classes in separate components, there is a tendency to have one class directly reference the other to do this. In this case, we can put the new functionality on an existing class or we can implement a new class within one of the components to handle this. Remember that the tenant of CBD is ultimately a high level of modularity. If we design our components well, there might be other applications that need the base functionality afforded by one component, but not that of the secondary component or the bridging functionality binding them together. If we design our components in the manner just discussed, we must distribute both components just to get to the little bit of functionality that we need in one.

---

**Tip - To minimize coupling between loosely related components, it is always better to build a third component to provide the bridge between the two components. In this manner, each can be distributed independent of the other.**

---

**Figure 3.11** shows tight coupling, whereas **Figure 3.12** shows its bridged counterpart.

**Figure 3.11**

A graphical representation of tight coupling.

In **Figure 3.11**, it should be clear that components A and B must travel together wherever they go. An application that only needs component A must bring along component B as well. An application that uses component A might go through test, debug, and redistribution whenever component B changes, although it is not using it.

In **Figure 3.12**, we show components A and B bridged together by component C. In this implementation, both A and B can be used singularly in applications, whereas applications that need the bridged functionality can use component C to provide this.

**Figure 3.12**

A graphical representation of bridged coupling.

**Summary**

We have learned some of the important concepts of object orientation and component-based development in this chapter. We have also learned how Microsoft's Visual Basic and the Component Object Model implement these concepts and how we can begin to use them to build modular, flexible applications. In the next chapter, we turn our attention to understanding the Relational Database Management system because it is the foundation for the information storage and retrieval component of our application. We will also begin laying the groundwork for good database design techniques, specifically as they pertain to our framework.

© Copyright Macmillan USA. All rights reserved.