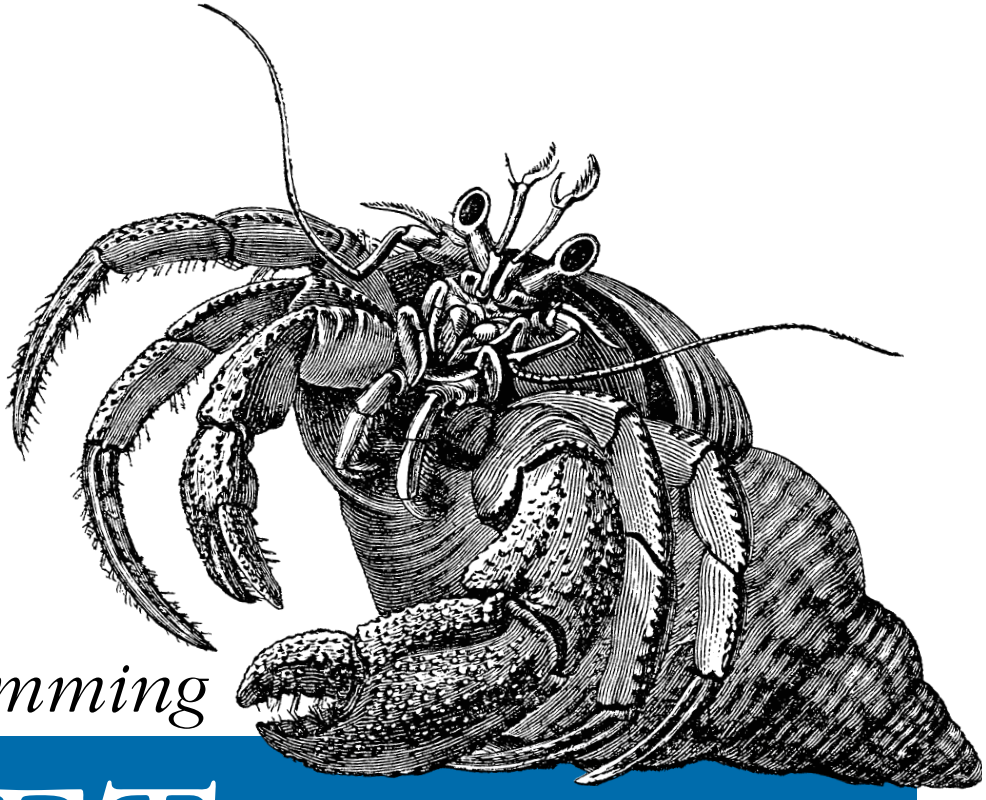
Design and Build Maintainable Systems
Using Component-Oriented Programming



Programming

.NET Components

O'REILLY®

Juval Löwy

Programming .NET Components

Programming .NET Components

Juval Löwy

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

CHAPTER 1

Introducing Component-Oriented Programming

Over the last decade, component-oriented programming has established itself as the predominant software development methodology. The software industry is moving away from giant, monolithic, hard-to-maintain code bases. Practitioners have discovered that by breaking a system down into binary components, they can attain much greater reusability, extensibility, and maintainability. These benefits can, in turn, lead to faster time to market, more robust and highly scalable applications, and lower development and long-term maintenance costs. Consequently, it's no coincidence that component-oriented programming has caught on in a big way.

Several component technologies, such as DCOM, CORBA, and Java Beans now give programmers the means to implement component-oriented applications. However, each technology has its drawbacks; for example, DCOM is too difficult to master, and Java doesn't support interoperation with other languages.

.NET is the newest entrant, and as you will see later in this chapter and in the rest of the book, it addresses the requirements of component-oriented programming in a way that is unique and vastly easier to use. This is little surprise because the .NET architects learned from the mistakes of previous technologies, as well as from their successes.

In this chapter, I'll define the basic terms of component-oriented programming and summarize the core principles and corresponding benefits of component-oriented programming. These principles apply throughout the book, and I'll refer to them in later chapters when describing the motivation for a particular .NET design pattern. Component-oriented programming is different from object-oriented programming, although the two methodologies have things in common. You could say that component-oriented programming sprouted from the well of object-oriented programming methodologies. Therefore, this chapter also contrasts component-oriented programming and object-oriented programming, and briefly discusses .NET as a component technology.

Basic Terminology

The term *component* is probably one of the most overloaded and therefore most confusing terms in modern software engineering, and the .NET documentation has its fair share of inconsistency in its handling of this concept. The confusion arises in deciding where to draw the line between a class that implements some logic, the physical entity that contains it (typically a DLL), and the associated logic used to deploy and use it, including type information, security policy, and versioning information (called the *assembly* in .NET). In this book, a component is a .NET class. For example, this is a .NET component:

```
public class MyClass
{
   public string GetMessage( )
   {
      return "Hello";
   }
}
```

Chapter 2 discusses DLLs and assemblies, and explains the rationale behind physical and logical packaging, as well as why it is that every .NET class is a binary component, unlike traditional object-oriented classes.

A component is responsible for exposing business logic to clients. A *client* is any entity that uses the component, although typically, clients are simply other classes. The client's code can be packaged in the same physical unit as the component, in the same logical unit but in a separate physical unit, or in separate physical and logical units altogether. The client code should not have to make any assumptions about such details. An *object* is an instance of a component, a definition that is similar to the classic object-oriented definition of an object as an instance of a class. The object is also sometimes referred to as the *server* because the relationship between client and object, often called the *client-server* model. In this model, the client creates an object and accesses its functionality via a publicly available entry point, traditionally a public method but preferably an interface, as illustrated by Figure 1-1. Note that in the figure an object is an instance of a component; the "lollipop" denotes an interface.

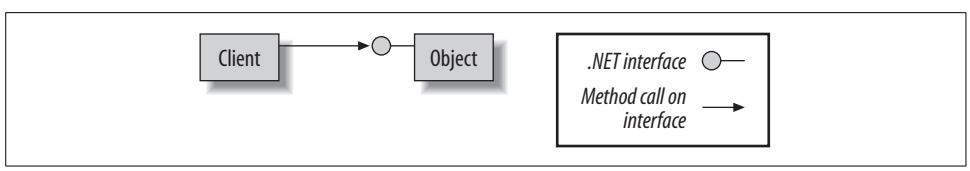


Figure 1-1. A client accessing an object

I'll discuss .NET interface-based programming in detail in Chapter 3. For now, it's important to emphasize that while .NET doesn't enforce interface-based programming, as you will see shortly, you should strive to do so with your own code whenever possible. To emphasize this practice, I represent the entry points of the

components that appear in my design diagrams as interfaces rather than mere public methods.

Although the object depicted in Figure 1-1 is drawn like a COM object with its characteristic lollipop icon, use of this icon isn't restricted to COM, but is accepted as the standard UML symbol for an interface, regardless of the component technology and development platform that implement it.

Interface-based programming promotes *encapsulation*, or the hiding of information from the client. The less a client knows about the way an object is implemented, the better. The more the details of an implementation are encapsulated, the greater the likelihood that you can change a method or property without affecting the client code. Interfaces maximize encapsulation because the client interacts with an abstract service definition instead of an actual object. Encapsulation is *key* to successfully applying both object-oriented and component-oriented methodologies.

Another important term originating from object-oriented programming is *polymorphism*. Two objects are said to be polymorphic with respect to each other when both derive from a common base type (such as an interface) and implement the exact set of operations defined by the base type. If a client is written to use the operations of the base type, the same client code can interact with any object that is polymorphic with the base type. When polymorphism is used properly, changing from one object to another has no effect on the client; it simplifies maintenance of the application to which the client and object belong.

Component-Oriented Versus Object-Oriented Programming

If every .NET class is a component, and if both classes and components share so many qualities, then what is the difference between traditional object-oriented programming and component-oriented programming? In a nutshell, object-oriented programming focuses on the relationship between classes that are combined into one large binary executable. Component-oriented programming instead focuses on interchangeable code modules that work independently and don't require you to be familiar with their inner workings to use them.

Building Blocks Versus Monolithic Applications

The fundamental difference between the two methodologies is the way in which they view the final application. In the traditional object-oriented world, even though you may factor the business logic into many fine-grained classes, once these classes are compiled, the result is monolithic binary code. All the classes share the same physical

deployment unit (typically an EXE), process, address space, security privileges, and so on. If multiple developers work on the same code base, they have to share source files. In such an application, a change made to one class can trigger a massive relinking of the entire application and necessitate retesting and redeployment of all other classes.

On the other hand, a component-oriented application comprises a collection of interacting binary application modules—that is, its components and the calls that bind them (see Figure 1-2).

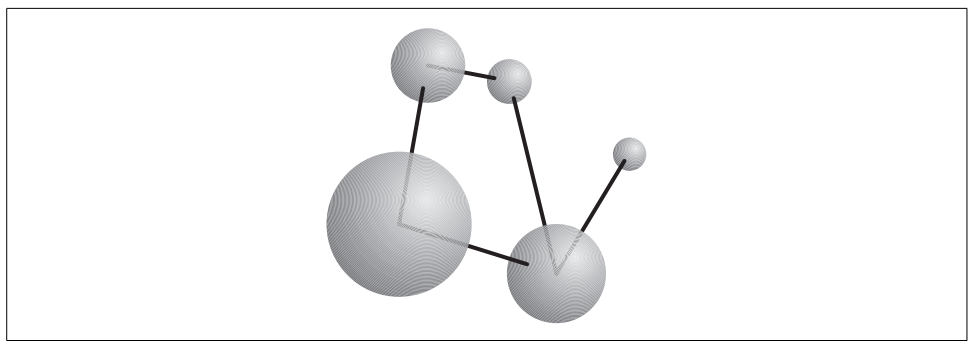


Figure 1-2. A component-oriented application

A particular binary component may not do much on its own. Some may be general-purpose components such as communication wrappers or file-access components. Others may be highly specialized and developed specifically for the application. An application implements and executes its required business logic by gluing together the functionality offered by the individual components. Component-enabling technologies such as COM, J2EE, CORBA, and .NET provide the "plumbing" or infrastructure needed to connect binary components in a seamless manner, and the main distinction between these technologies is the ease with which they allow you to connect those components.

The motivation for breaking down a monolithic application into multiple binary components is analogous to that for placing the code for different classes into different files. By placing the code for each class in an application into its own file, you loosen the coupling between the classes and the developers responsible for them. A change made to one class may require recompilation only of the source file for that class, although the entire application will have to go through relinking.

However, there is more to component-oriented programming than simple software project management. Because a component-based application is a collection of binary building blocks, you can treat its components like Legos, adding and removing them as you see fit. If you need to modify a component, changes are contained to that component only. No existing client of the component requires recompilation or

redeployment. Components can even be updated while a client application is running, as long as the components aren't currently being used.

In addition, improvements, enhancements and fixes made to one component are immediately available to all applications using that component, on the same machine or perhaps across the network.

A component-oriented application is easier to extend as well. When you have new requirements to implement, you can provide them in new components, without having to touch existing components not affected by the new requirements.

These factors enable component-oriented programming to reduce the cost of long-term maintenance, a factor essential to almost any business, which explains the widespread adoption of component technologies.

Component-oriented applications usually have a faster time to market because you can select from a range of available components, either from inhouse collections or from third-party component vendors, and thus avoid repeatedly reinventing the wheel. For example, consider the rapid development enjoyed by many Visual Basic projects, which rely on libraries of ActiveX controls for almost every aspect of the application.

Interfaces Versus Inheritance

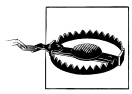
Another important difference between object-oriented and component-oriented applications is the emphasis the two models place on inheritance and reuse models.

In object-oriented analysis and design, you often model applications as complex hierarchies of classes, which are designed to approximate as much as possible the business problem being solved. Existing code is reused by inheriting from an existing base class and specializing its behavior. The problem is that inheritance is a poor way to reuse. When you derive a subclass from a base class, you must be intimately aware of the implementation details of the base class. For example: what is the side effect of changing the value of a member variable? How does it affect the code in the base class? Will overriding a base class method and providing a different behavior break the code of clients that expect the base behavior?

This form of reuse is commonly known as *white box reuse* because you are required to be familiar with the details of its implementation. White box reuse simply doesn't allow for economy of scale in large organizations' reuse programs or easy adoption of third-party frameworks.

Component-oriented programming promotes *black box reuse* instead, which allows you to use an existing component without caring about its internals, as long as the component complies with some predefined set of operations or interfaces. Instead of investing in designing complex class hierarchies, component-oriented developers

spend most of their time factoring out the interfaces used as contracts between components and clients.



.NET does allow components to use inheritance of implementation, and you can certainly use this technique to develop complex class hierarchies. However, you should keep your class hierarchies as simple and as flat as possible, and focus instead on factoring interfaces. Doing so promotes black-box reuse of your component instead of white-box reuse via inheritance.

Finally, object-oriented programming provides few tools or design patterns for dealing with the runtime aspects of the application, such as multithreading and concurrency management, security, distributed applications, deployment, or version control. Object-oriented developers are more or less left to their own devices when it comes to providing infrastructure for handling these common requirements. As you will see throughout the book, .NET supports you by providing a superb component-development infrastructure. Using .NET, you can focus on the business problem at hand instead of the software infrastructure needed to build the solution.

Principles of Component-Oriented Programming

Systems that support component-oriented programming and the programmers that use them adhere to a set of core principles that continues to evolve. The most important of these include:

- Separation of interface and implementation
- Binary compatibility
- Language independence
- Location transparency
- Concurrency management
- Version control
- Component-based security

Often, it's hard to tell the difference between a true principle and a mere feature of the component technology being used. Component programming requires both systems that support the approach and programmers that adhere to its discipline. As the supporting technologies become more powerful, no doubt software engineering will extend its understanding of what constitutes component-oriented programming and embrace new ideas. The following sections discuss these seven important principles of component-oriented programming.

Separation of Interface from Implementation

The fundamental principle of component-oriented programming is that the basic unit in an application is a binary-compatible interface. The interface provides an abstract service definition between a client and the object. This principle contrasts with the object-oriented view of the world that places the object rather than its interface at the center. An *interface* is a logical grouping of method definitions that acts as the *contract* between the client and the service provider. Each provider is free to provide its own interpretation of the interface—that is, its own implementation. The interface is implemented by a black-box binary component that completely encapsulates its interior. This principle is known as *separation of interface from implementation*.

To use a component, the client needs to know only the interface definition (the service *contract*) and be able to access a binary component that implements that interface. This extra level of indirection between the client and the object allows one implementation of an interface to be replaced by another without affecting client code. The client doesn't need to be recompiled to use a new version. Sometimes the client doesn't even need to be shut down to do the upgrade. Provided the interface is immutable, objects implementing the interface are free to evolve, and new versions can be introduced. To implement the functionality promised by an interface inside a component, you use traditional object-oriented methodologies, but the resulting class hierarchies are usually simpler and easier to manage.

Another effect of using interfaces is that they enable reuse. In object oriented-programming, the basic unit of reuse is the object. In theory, different clients should be able to use the same object. Each reuse instance saves the reusing party the amount of time and effort spent implementing the object. Reuse initiatives have the potential for significant cost reduction and reduced product-development cycle time. One reason why the industry adopted object-oriented programming so avidly was its desire to reap the benefits of reuse.

In reality, however, objects are rarely reusable. Objects are often specific to the problem and the particular context they were developed for, and unless the objects are "nuts and bolts," that is, simple and generic, the objects can't be reused even in very similar contexts. This reality is true in many engineering disciplines, including mechanical and electrical engineering. For example, consider the computer mouse you use with your workstation. Each part of this mouse is designed and manufactured specifically for your make and model. For reasons of branding and electronics, parts such as the body case can't be used in the manufacturing of any other type of mouse (even very similar ones), whether made by the same manufacturer or others. However, the interface between mouse and human hand is well defined, and any human (not just yourself) can use the mouse. Similarly, the typical USB interface between mouse and computer is well defined, and your mouse can plug into almost

any computer adhering to the interface. The basic units of reuse in the computer mouse are the interfaces the mouse complies with, not the mouse parts themselves.

In component-oriented programming, the basic unit of reuse is the interface, not a particular component. By separating interfaces from implementation in your application, and using predefined interfaces or defining new interfaces, you enable that application to reuse existing components and enable reuse of your new components in other applications.

Binary Compatibility Between Client and Server

Another core principle of component-oriented programming is *binary compatibility* between client and server. Traditional object-oriented programming requires all the parties involved—clients and servers—to be part of one monolithic application. During compilation, the compiler inserts the address of the server entry points into the client code. Component-oriented programming revolves around packaging code into components, i.e., binary building blocks. Changes to the component code are contained in the binary unit hosting it; you don't need to recompile and redeploy the clients. However, the ability to replace and plug in new binary versions of the server implies binary compatibility between the client and the server, meaning that the client's code must interact at runtime with exactly what it expects as far as the binary layout in memory of the component entry points. This binary compatibility is the basis for the contract between the component and the client. As long as the new version of the component abides by this contract, the client isn't affected. In Chapter 2, you will see how .NET provides binary compatibility.

Language Independence

Unlike traditional object-oriented programming, in component-oriented programming, the server is developed independently of the client. Because the client interacts with the server only at runtime, the only thing that binds the two is binary compatibility. A corollary is that the programming languages that implement the client and server should not affect their ability to interact at runtime. *Language independence* means exactly that: when you develop and deploy components your choice of programming language should be irrelevant. Language independence promotes the interchangeability of components, and their adoption and reuse. .NET achieves language independence through an architecture and implementation called the Common Language Runtime (CLR), which is discussed further in Chapter 2.

Location Transparency

A component-based application contains multiple binary components. These components can all exist in the same process, in different processes on the same machine,

or on different machines on a network. Recently, with the advent of web services, components can also be distributed across the Internet.

The underlying component technology is required to provide a client with *location transparency*, which allows the client code to be independent of the actual location of the object it uses. Location transparency means there is nothing in the client's code pertaining to where the object executes. The same client code must be able to handle all cases of object location (see Figure 1-3), although the client should be able to insist on a specific location as well. Note that in the figure, the object can be in the same process (e.g., Process 1 on Machine A), in different processes on the same machine (e.g., Process 1 and Process 2 on Machine A), on different machines in the same local network, or even across the Internet (e.g., Machines B and C).

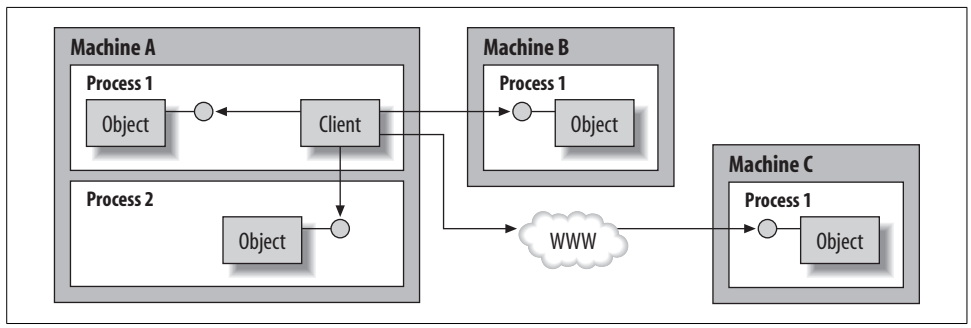


Figure 1-3. Location transparency enables client code to be oblivious of the actual object location

Location transparency is crucial to component-oriented programming for a number of reasons. First, it lets you develop the client and components locally (which leads to easier and more productive debugging), yet deploy the same code base in distributed scenarios. Second, the choice of using the same process for all components, or multiple processes for multiple machines, has a significant impact on performance and ease of management versus scalability, availability, robustness, throughput, and security. Organizations have different priorities and preferences for these tradeoffs, yet the same set of components from a particular vendor or team should be able to handle all scenarios. Third, the location of components tends to change as the application's requirements evolve over time.

To minimize the cost of long-term maintenance and extensibility, you should avoid having client code make any assumptions regarding the location of the objects it uses and avoid making explicit calls across processes or across machines. .NET remoting is the name of the technology that enables remote calls in .NET. Chapter 10 is dedicated to .NET remoting and discusses .NET support for location transparency.

Concurrency Management

A component developer can't possibly know in advance all the possible ways in which a component will be used and particularly whether it will be accessed concurrently by multiple threads. The safest course is for you to assume that the component will be used in concurrent situations and to provide some mechanism inside the component for synchronizing access. However, this approach has two flaws. First, it may lead to deadlocks; if every component in the application has its own synchronization lock, a deadlock can occur if two components on different threads try to access each other. Second, it's an inefficient use of system resources for all components in the application to be accessed by the same thread.

The underlying component technology must provide a *concurrency management* service—way for components to participate in some application-wide synchronization mechanism, even when the components are developed separately. In addition, the underlying component technology should allow components and clients to provide their own synchronization solutions for fine-grained control and optimized performance. .NET concurrency management support is discussed in Chapter 8 as part of developing multithreaded .NET applications.

Versioning Support

Component-oriented programming must allow clients and components to evolve separately. Component developers should be able to deploy new versions (or just fixes) of existing components without affecting existing client applications. Client developers should be able to deploy new versions of the client application and expect it to work with older versions of components. The underlying component technology should support *versioning*, which allows a component to evolve along different paths, and for different versions of the same component to be deployed on the same machine, or *side by side*. The component technology should also detect incompatibility as soon as possible and alert the client. .NET's solution to version control is discussed in Chapter 6.

Component-Based Security

In component-oriented programming, components are developed separately from the client applications that use them. Component developers have no way of knowing how a client application or end user will try to use their work. A benign component could be used maliciously to corrupt data or transfer funds between accounts without proper authorization or authentication. Similarly, a client application has no way to know whether it's interacting with a malicious component that will abuse the credentials the client provides. In addition, even if both the client and the component have no ill intent, the end application user can still try to hack into the system or do some other damage (even by mistake).

Version Control and DLL Hell

Historically, the versioning problem has been the source of much aggravation. Early attempts at component technology using DLL and DLL-exported functions created the predicament known as *DLL Hell*. A typical DLL Hell scenario involved two client applications, say A1.0 and B1.0, each using Version C1.0 of a component in the *mydll.dll* file. Both A1.0 and B1.0 install a copy of the *mydll.dll* in some global location such as the System directory. When Version A1.1 is installed, it also installs Version C1.1 of the component, providing new functionality in addition to the functionality defined in C1.0. Note that *mydll.dll* can contain C1.1 and still serve both old and new client application versions because the old clients aren't aware of the new functionality, and the old functionality is still supported. Binary compatibility is maintained via strict management of ordinal numbers for the exported functions (a source for another set of problems associated with DLL Hell). The problem starts when Application B1.0 is reinstalled. As part of installing B1.0, Version C1.0 is reinstalled, overriding C1.1. As a result, A1.1 can't execute.

Interestingly enough, addressing the issue of DLL Hell was one of the driving forces behind COM. Even though COM makes wide use of objects in DLLs, COM can completely eliminate DLL Hell. However, COM is difficult to learn and apply, and consequently can be misused or abused, resulting in problems similar to DLL Hell.

Like COM in its time, .NET was designed with DLL Hell in mind. .NET doesn't eliminate all chances of DLL Hell but reduces its likelihood substantially. The default .NET versioning and deployment policies don't allow for DLL Hell. However, .NET is an extensible platform. You can choose to override the default behavior for some advanced need or to provide your own custom version control policy, but you risk DLL Hell.

To lessen the danger, a component technology must provide a security infrastructure to deal with these scenarios, without coupling components and client applications to each other. In addition, security requirements, policies, and events (such as new users) are among the most volatile aspects of the application lifecycle, not to mention the fact that security policies vary between applications and customers. A productive component technology should allow for the components to have as few security policies and as little security awareness as possible in the code itself. It should also allow system administrators to customize and manage the application security policy without requiring you to make changes to the code. .NET's rich security infrastructure is the subject of Chapter 12.

.NET Adherence to Component Principles

One challenge facing the software industry today is the skill gap between what developers should know and what they do know. Even if you have formal training in computer science, you may lack effective component-oriented design skills, which are

.NET Versus COM

If you're a seasoned COM developer, .NET might seem to be missing many of the elements you have taken for granted as part of your component development environment. If you have no COM background, you can skip this section. If you are still reading, you should know that the seemingly missing elements remain in .NET, although they are expressed differently:

- There is no base interface such as `IUnknown` that all components derive from. Instead, all components derive from the class `System.Object`. Every .NET object is therefore polymorphic with `System.Object`.
- There are no class factories. In .NET, the runtime resolves a type declaration to the assembly containing it and the exact class or struct within the assembly. Chapter 2 discusses this mechanism.
- There is no reference counting of objects. .NET has a sophisticated garbage collection mechanism that detects when an object is no longer used by clients and then destroys it. Chapter 4 describes the .NET garbage collection mechanism and the various ways you can manage resources held by objects.
- There are no IDL files or type libraries to describe your interfaces and custom types. Instead, you put those definitions in your source code. The compiler is responsible for embedding the type definitions in a special format in your assembly, called *metadata*. Metadata is described in Chapter 2.
- Component dependencies are captured by the compiler during compilation and persisted in a special format in your assembly, called a *manifest*. The manifest is described in Chapter 2.
- Identification isn't based on globally unique identifiers (GUIDs). Uniqueness of type (class or interface) is provided by scoping the types with the namespace and assembly name. When an assembly is shared between clients, the assembly must contain a *strong name*—i.e., a unique digital signature generated by using an encryption key. The strong name also guarantees component authenticity, and .NET refuses to execute a mismatch. In essence, these are GUIDs, but you don't have to manage them any more. Chapter 6 discusses shared assemblies and strong names.
- There are no apartments. By default, every .NET component executes in a free-threaded environment, and it's up to you to synchronize access. Synchronization can be done either by using manual synchronization locks or by relying on automatic .NET synchronization domains. .NET multithreading and synchronization are discussed in Chapter 8.

primarily acquired through experience. Today's aggressive deadlines, tight budgets, and a continuing shortage of developers precludes, for many, the opportunity to attend formal training sessions or to receive effective on-the-job training. Nowhere is

the skill gap more apparent than among developers at companies who attempt to adhere to component development principles. In contrast, object-oriented concepts are easier to understand and apply, partly because they have been around much longer, so a larger number of developers are familiar with them, and partly because of the added degree of complexity involved with component development compared to monolithic applications.

A primary goal of the .NET platform is to simplify the development and use of binary components and to make component-oriented programming accessible. As a result, .NET doesn't enforce some core principles of component-oriented programming, such as separation of interface from implementation, and unlike COM, .NET allows binary inheritance of implementation. Instead, .NET merely enforces a few of the concepts and enables the rest. Doing so caters to both ends of the skill spectrum. If you understand only object-oriented concepts, you will develop .NET "objects," but because every .NET class is consumed as a binary component by its clients, you can gain many of the benefits of component-oriented programming. If you understand and master how to apply component-oriented principles, you can fully maximize the benefit of .NET as a powerful component-development technology.

This duality can be confusing. Throughout the book, whenever applicable, I will point out the places where .NET doesn't enforce a core principle and suggest methods to stick with it nonetheless.

Developing .NET Components

A component technology is more than just a set of rules and guidelines on how to build components. A successful component technology must provide a development environment and tools that will allow you to rapidly develop components. .NET offers a superb development environment and semantics that are the product of years of observing the way you use COM and the hurdles you face. All .NET programming languages are component-oriented in their very nature, and the primary development environment (Visual Studio.NET) provides views, wizards, and tools that are oriented toward developing components. .NET shields you from the underlying raw operating services and provides instead operating system-like services (such as filesystem access or threading) in a component-oriented manner. The services are factored to various components in a logical and consistent fashion, resulting in a uniform programming model. You will see numerous examples of these services throughout this book. The following sections detail key factors that enable .NET to significantly simplify component development.

The .NET Base Classes

When you develop .NET components, there is no need for a hard-to-learn component development framework such as the Active Template Library (ATL), which was used to develop COM components in C++. .NET takes care of all the underlying plumbing. In addition, to help you develop your business logic faster, .NET provides you with more than 8,000 base classes (from message boxes to security permissions), available through a common library available to all .NET languages. The base classes are easy to learn and apply. You can use the base classes as-is or derive from them to extend and specialize their behavior. You will see examples of how to use these base classes throughout the book.

Attribute-Based Programming

When developing components, you can use attributes to declare their special runtime and other needs, rather than coding them. This is analogous to the way COM developers declare the threading model attribute of their components. .NET offers numerous attributes, allowing you to focus on the domain problem at hand. You can also define your own attributes or extend existing ones. Appendix C discusses reflection and custom attributes.

Component-Oriented Security

The classic Windows NT security model is based on what a given user is allowed to do. This model emerged at a time when COM was in its infancy, and applications were usually standalone and monolithic. In today's highly distributed, component-oriented environment, there is a need for a security model based on what a given piece of code, a component, is allowed to do, not only on what its caller is allowed to do.

.NET allows you to configure permissions for a piece of code and to provide evidence proving the code has the right credentials to access a resource or perform sensitive work. Evidence is tightly related to the component's origin. System administrators can decide that they trust all code that came from a particular vendor but distrust everything else, from downloaded components to malicious attacks. A component can also demand that a permission check be performed to verify that all callers in its call chain have the right permissions before it proceeds to do its work. Chapter 12 is dedicated to .NET's rich security infrastructure.

Simplified Deployment

Installing a .NET component can be as simple as copying it to the directory of the application using it. This is in contrast to COM, which relies on the Registry for component deployment to let it know where to look for the component file and how to treat it. .NET maintains tight version control, enabling side-by-side execution of new

and old versions of a shared component on the same machine. The net result is a zero-impact install; by default, you can't harm another application by installing yours, thus ending DLL Hell. The .NET motto is: it just works. If you want to install components to be shared by multiple applications, you can install them in a storage area called the *Global Assembly Cache* (GAC). If the GAC already contains a previous version of your assembly, it keeps it, for use by clients that were built against the old version. You can purge old versions as well, but that isn't the default. .NET shared deployment and version control is discussed in Chapter 2.