Programming .NET Components

Sunday, January 12, 2020 2:38 AM

Programming .NET Components, 2nd Edition
★★★★★ 2 REVIEWS

by Juval Lowy
Publisher: O'Reilly Media, Inc.
Release Date: July 2005
ISBN: 9780596102074
Topic: .NET

I only copied the information which was related to components.

Preface

I've been fortunate in my career to have lived through most generations of Microsoft component technologies. In the mid-1990s, I developed dynamic link libraries and exported their functions, Microsoft Foundation Class (MFC) extension DLLs to expose classes. I experienced firsthand the enormous complexity involved in managing a set of interacting applications comprised of 156 deployed as a single unit, as well as the maintenance and versioning issues raised by their use of ordinal numbers. I helped design COM-like solutions to those problems, and I remember whe about COM and when I generated my first GUID using a command-line utility.

I learned how to write class factories and IDL interfaces long before the release of ATL, and I tried to use RPC before DCOM abstracted it away. I designed component-based applications usin experienced what it takes to share design ideas with other developers who aren't familiar with its requirements. I programmed with MTS and learned the workarounds involved in its use, and I r the elegance and usefulness of COM+ when it came to architecting large-scale enterprise frameworks.

My understanding of component-oriented programming has evolved and grown over that time, just as the component-based technologies themselves have done. I have often asked myself wha fundamental principles of using components are, and in what ways they differ from traditional object-oriented programming. I have tried to learn from my mistakes and to abstract and generaliz ideas and techniques I have encountered or developed on my own. I believe that I have identified some core principles of component-oriented design that transcend any technologies available that result in components that are easier to reuse, extend, and maintain over the long term.

With the advent of the .NET Framework, Windows developers finally have at their disposal a first-class technology that aims at simplifying the task of developing and deploying component-base applications. .NET is the result of much soul-searching by Microsoft, and in my view it improves on the deficiencies of previous technologies—especially COM. It incorporates and enforces a va proven methodologies and approaches, while retaining their core benefits.

To me, .NET is fundamentally a component technology that provides an easy and clean way to generate binary components, in compliance with what I regard as sound design principles. .NET engineered from the ground up to simplify component development and deployment, and to support interoperability between programming languages. It is highly versatile, and .NET componen for building a wide range of component-based applications, from standalone desktop applications to web-based applications and services.

Of course, .NET is more than just a component technology; it's actually a blanket name for a set of technologies.

.NET provides several specialized application frameworks, including Windows Forms for rich Windows clients, ADO.NET for data access, ASP.NET for web applications, and web services for e consuming remote services that use the SOAP and other XML-based protocols. Visual Studio 2005 supports the development of .NET applications in C#, Visual Basic, Managed C++, and J#, use more than a dozen other languages as well. You can host .NET applications in Windows or in SQL Server 2005. Microsoft server products will increasingly support .NET-connected applica coming years, and future versions of Windows will be heavily based on .NET.

Scope of this book

In this book, you'll learn not only about .NET component programming and the related system issues, but also about relevant design options, tips, best practices, and pitfalls. The book avoids n implementation details of .NET and largely confines its coverage to the possibilities and the practical aspects of using .NET as a component technology: how to apply the technology and how to among the available design and programming models. In addition, the book contains many useful utilities, tools, and helper classes I've developed since .NET was introduced five years ago. T aimed at increasing your productivity and the quality of your .NET components. After reading this book, you will be able to start developing .NET components immediately, taking full advantage development infrastructure and application frameworks. The book makes the most of what both .NET 1.1 and .NET 2.0 have to offer.

The term *component* is probably one of the most overloaded and therefore most confusing terms in modern software engineering.

The strategic guide to enterprise-class component and Web development.

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, monol to-maintain code bases. Practitioners have discovered that by breaking down a system into binary components, they can attain much greater reusability, extensibility, and maintainability. These 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 compo oriented programming has caught on in a big way.

Several component technologies, such as DCOM, CORBA, and JavaBeans™ give programmers the means to implement component-oriented applications. However, each technology has its d for example, DCOM is too difficult to master, and the Java™ Virtual Machine (JVM) doesn't support interoperation with other languages.

.NET is the latest entrant to the field, and as you will see later in this chapter and in the rest of this book, it addresses the requirements of component-oriented programming in a way that is both vastly easier to use. These improvements are of little surprise, because the .NET architects were able to learn from both the mistakes and successes of previous technologies.

In this chapter, I'll define the basic terms of component-oriented programming and summarize its core principles and their corresponding benefits. These principles apply throughout the book, a to them in later chapters when describing the motivations for particular .NET design patterns.

Component-oriented programming is different from object-oriented programming, although the two methodologies do have things in common. You could say that component-oriented programm sprouted from the well of object-oriented programming methodologies. Therefore, this chapter also contrasts component-oriented programming and object-oriented programming, while briefly c .NET as a component technology.

I'll discuss .NET interface-based programming in detail in Chapter 3. For now, it's important to emphasize that while .NET doesn't force you to do interface-based programming, as you will see should strive to do so 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 m

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 de implementation are encapsulated, the greater is the likelihood that you can change a method or property without affecting the client code. Interfaces maximize encapsulation because the client 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 concept that originated with object-oriented programming is *polymorphism* . Two objects are said to be polymorphic with respect to each other when both derive from a comm type (such as an interface) and implement the exact set of operations defined by that base type. If a client is written to use the operations of a base type, the same client code can interact with that is polymorphic with that 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 whic and object belong.

Component-Oriented Versus Object-Oriented Programming

If every .NET class is a component, and if classes and components share so many qualities, then what is the difference between traditional object-oriented programming and component-orient programming? In a nutshell, object-oriented programming focuses on the relationships between classes that are combined into one large binary executable, while component-oriented program 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 many fine-grained classes, once those classes are compiled, the result is monolithic binary code. All the classes share the same physical deployment unit (typically an EXE), process, address 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 re- entire application and necessitate retesting and redeployment of all the 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).

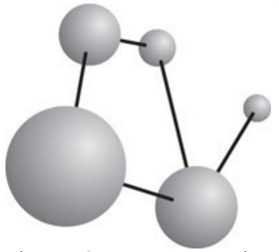

Figure 1-2. A component-oriented application

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 sp 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. Com enabling technologies such as COM, J2EE, CORBA, and .NET provide the "plumbing" or **software infrastructure** needed to connect binary components in a seamless manner, and the main 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 fo in an application into its own file, you loosen the coupling between the classes and the developers responsible for them. If you make a change to one class, although you'll have to re-link the er application, you'll only need to recompile the source file for that class.

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 ca components like LEGO bricks, adding and removing them as you see fit. If you need to modify a component implementation, changes are contained to that component only. No existing client o 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. Improvemen enhancements, and fixes made to a component will immediately be available to all applications that use that component, whether on the same machine or across a 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 comp 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 helps explain the widespread adoption of technologies.

Component-oriented applications usually have a faster time to market, because you can select from a range of available components, either from in-house collections or from third-party compo 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 al 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, applications are often modeled as complex hierarchies of classes, which are designed to approximate as closely as possible the business problem being reuse existing code by inheriting it from an existing base class and specializing its behavior. The problem is that inheritance is a poor way to achieve reuse. When you derive a subclass from a 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 th 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 the base class implementation. White-box reuse simply doesn't allow for 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 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** use contracts between components and clients.

Warning

.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 hierarc 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 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 requirem will see throughout this 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 softw infrastructure (technologies) needed to build the solution.

Principles of Component-Oriented Programming from Book "Programming .NET Components, 2nd Edition by Juval Lowy"

Component-oriented programming requires both systems that support the approach and programmers that adhere to its discipline and its core principles. However, it's often hard to tell the difference between a true principle and a mere component technology being used. **As the supporting technologies become more powerful, no doubt software engineering will extend its understanding of what constitutes component-oriented programming and embrace and the core principles will continue to evolve.** The most important principles of component-oriented programming include:

- **Separation of interface and implementation**
- **Binary compatibility**
- **Language independence**
- **Location transparency**

- **Concurrency management**
- **Version control**
- **Component-based security**

The following subsections discuss these seven important principles. As discussed in the next section, .NET enables complying with all of the core principles, but it does not necessarily enforces these principles.

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 obje principle contrasts with the object-oriented view of the world, which 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 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 principle is known as *separation of interface from implementation*.

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

Interfaces also facilitate 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 a and effort that would have been spent implementing the object. Reuse initiatives have the potential for significant cost reductions and reduced product-development cycle time. One reason why the industry adopted o programming so avidly was its desire to reap the benefits of reuse.

In reality, however, objects are rarely reusable. They are often specific to the problems and particular contexts for which they were developed, and unless the objects are "nuts and bolts"—that is, simple and generic— reused even in very similar contexts. This is also true in many engineering disciplines, including mechanical and electrical engineering. For example, consider the computer mouse you use with your workstation. Each pa mouse is designed and manufactured specifically for your make and model. For reasons of branding and electronics specifics, parts such as the body case can't be used in the manufacturing of any other type of mouse 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 US between mouse and computer is well defined, and your mouse can plug into almost any computer adhering to that interface. The basic units of reuse in the computer mouse are the **interfaces** with which the mouse c 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 n 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 (C++) requires all the parties involved—clients and servers—to be monolithic application. During compilation, the compiler inserts the addresses of the server entry points into the client code. Component-oriented programming, in contrast, revolves around packaging code into comp **binary building blocks**). Changes to the component code are contained in the binary unit hosting the component; you don't need to recompile and redeploy the clients. However, the ability to replace and plug in new 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 com 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 s provides binary compatibility.

Language Independence

Unlike in 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 th 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 yo 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. With th web services, components can also now 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 oblivious to the actual locations of the objects it uses. Location transparency means t nothing in the client's code pertaining to where the objects execute. 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 lo 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 s network (e.g. Machine B), or even on different machines across the Internet (e.g., Machine C).

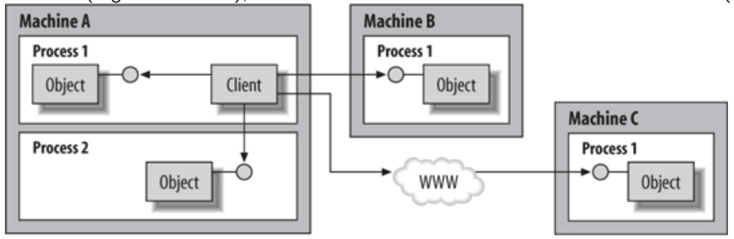

Figure 1-3. With location transparency, the client is oblivious to the actual object location

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 de code base in distributed scenarios. Second, the choice of whether to use the same process for all components or multiple processes for multiple machines has a significant impact on performance and ease of manage scalability, availability, robustness, throughput, and security. Organizations have different priorities and preferences for these trade-offs, yet the same set of components from a particular vendor or team should be able scenarios. Third, the locations of the components tend to change as an application's requirements evolve over time. If the locations are transparent to the client, this movement does not cause problems.
To minimize the cost of long-term maintenance and extensibility, you should avoid having client code make any assumptions regarding the locations of the objects it uses and avoid making explicit calls across process 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 multiple threads will access it concurrently. The safest course is to assume tha component will be used in concurrent situations and to prepare for this eventuality. One option is to provide some mechanism inside the component for synchronizing access. However, this approach has two flaws. Fi 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 reso components in the application to be accessed by the same thread.

To get around these problems, the underlying component technology must provide a *concurrency management* service—that is, a way for components to participate in some application-wide synchronization mechani 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 optimize 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 existir applications. Client developers should be able to deploy new versions of the client application and expect them to work with older versions of components. The underlying component technology should support *versio* allows a component to evolve along different paths and allows different versions of the same component to be deployed on the same machine, or even in the same client process, *side by side*. The component technol also detect incompatibility as soon as possible and alert the client. .NET's solution to version control is discussed in Chapter 5.

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 us so a benign component could well 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 ir 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 damage (even by mistake).
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. Security requirements, policies, ar (such as new users) are among the most volatile aspects of the application lifecycle, and security policies vary between applications and customers. A productive component technology should allow for the componer few security policies as possible and for there to be 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 changes to the code. .NET's rich security infrastructure is the subject of chapter 12.

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 a A typical DLL Hell scenario involves 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 *mydll.dll* in som 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. N *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. Bir 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 Applic 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 H COM is difficult to learn and apply and consequently can be misused or abused, resulting in problems similar to DLL Hell.

Like COM, .NET was designed with DLL Hell in mind. .NET doesn't eliminate all chances of DLL Hell, but does reduce its likelihood substantially. While it's true that the default versioning and d policies of .NET don't allow for DLL Hell, .NET is, after all, an extensible platform. You can choose to override its default behavior to meet some advanced need or to provide your own custom v control policy, but in doing so you risk creating DLL Hell.

.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 component-oriented design skills, which are primarily acquired through experience. Today's aggressive deadlines, tight budgets, and a continuing shortage of developers precludes, for many, t 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 c development principles. In contrast, object-oriented concepts are easier to understand and apply, partly because they have been around much longer (and hence a larger number of developers with them) and partly because of the added degree of complexity involved with component development as compared to development of monolithic applications.

A primary goal of the **.NET or Spring or JSF** platform is to simplify the development and use of binary components and to make component-oriented programming more accessible. As a result 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 implementa enforces a few of the core concepts and merely enables the rest. Doing so caters to both ends of the skill spectrum. If you understand only object-oriented concepts, you will develop .NET “obj because every .NET class is consumed as a binary component by its clients, you can still gain many of the benefits of component-oriented programming. If, on the other hand, you understand how to apply component-oriented principles, you can fully maximize the benefits 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 nonetheles

The other side of the coin is that amortized over five or seven years of an enterprise application’s lifecycle, the time saved using a RAD(rapid application development) tool is insignificant—the is the cost of long-term maintenance. Applying component-oriented analysis and design methodologies in a large application requires great deal of **skill and time** and will result in unmaintainal done poorly. Moreover, the larger the application, the more likely you are to wish to maintain it for longer period of time, in order to maximize the return on the investment of development. **Main extensibility** have much to do with **proper design**, **architecture**, and **abstractions**, as well as **component** and **interface factoring**. Enterprise developers tend to spend the bulk of their time the code itself, rather than on the **wizards** and the surrounding **tools**. For enterprise developers, Microsoft designates C#, with its focus on **code structure** and **constructs**. Since a typical ent application debug session involves contemplative, time-consuming analysis of the problem, it’s no wonder that the C# team had little regard for edit-and-continue capabilities.

Chapter 2 .NET Component-Oriented-Programming Essentials

Binary Compatibility

As explained in Chapter 1, one of the core principles of component-oriented programming is binary compatibility between client and server. Binary compatibility enables binary components bec enforces both sides to abide by a binary contract (typically, an interface). As long as newer versions of the server abide by the original contract between the two, the client isn’t affected by chan the server. COM was the first component technology to offer true binary compatibility free of DLL Hell (described in the previous chapter), and many developers have come to equate COM’s implementation of binary compatibility (and the resulting restrictions, such as immutable COM interfaces) with the principle itself. The .NET approach to binary compatibility is different from that though, and so are the implications of the programming model. To understand these implications and why they differ from COM’s, this section first briefly describes COM’s binary-compatibility implementation and then discusses .NET’s way of supporting binary compatibility.

COM Binary Compatibility

COM provides binary compatibility by using interface pointers and virtual tables. In COM, the client interacts with the object indirectly via an interface pointer. The interface pointer actually point pointer called the *virtual table pointer*. The virtual table pointer points to a table of function pointers. Each slot in the table points to the location where the corresponding interface’s method code (see Figure 2-9).

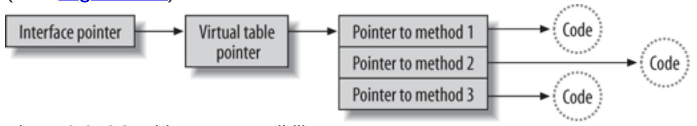

Figure 2-9. COM binary compatibility

When the client uses the interface pointer to call a method (such as the second method of the interface), all the compiler builds into the client’s code is an instruction to jump to the address poir the second entry in the virtual table. In effect, this address is given as an offset from the table’s starting point. At runtime, the **loader** patches this jump command to the actual address, because knows the table memory location. All the COM client records in its code is the offset from the virtual table start address. At runtime, any server that provides the same in-memory table layout (ir table entries point to methods with exactly the same signatures) is considered binary-compatible. This is, by the way, exactly the definition of implementing a COM interface. This model yields t COM prime directive: *Thou Shalt Not Change a Published Interface*. Any change to the virtual table layout will break the existing client’s code. The virtual table layout is isomorphic to the interfa —for example, the second method in the interface is the second entry in the virtual table. The virtual table must have the exact same number of entries as the interface has methods. Any chan interface’s definition must result in the recompiling and redeploying of all clients as well; otherwise, the clients are no longer binary-compatible with the server.

.NET Binary Compatibility

.NET provides binary compatibility using metadata. The high-level client code (such as C# or Visual Basic 2005) is compiled to IL. The client code can’t contain any offsets from memory addres because those offsets depend on the way the JIT compiler structures the native code. At runtime, the JIT compiler compiles and links the IL to native machine code. The original IL contains onl to invoke methods or access fields of an object, based on that type’s metadata—it’s as if instead of a traditional method call in native code, the IL contains only a token identifying the method to type that provides these methods or fields in its metadata is binary-compatible, because the actual binary layout in memory of the type is decided at runtime by the JIT compiler, and there is no pertaining to this layout in the client’s IL.

The main benefit of .NET metadata-based binary compatibility is that every class is a binary component. This, in turn, simplifies developing components tremendously compared with COM. The need for complex frameworks such as ATL, because .NET supports components natively. .NET can bridge the skill gap (mentioned in Chapter 1) between what most developers can do and wh component technologies demand. If you understand only object-oriented programming, you can do that and still gain some component-oriented programming benefits, relying on .NET to mana compatibility and versioning. If you understand the core issues of component-oriented programming, you can maximize your productivity and potential while gaining the full benefits of compone applications. Unlike with COM, with .NET binary compatibility isn't limited to interface-based programming. Any .NET type, be it a class or a struct, is compatible with its clients. Any entry point, instance method, object field, static method, or static field, is binary-compatible. The other main benefit of metadata-based binary compatibility is that it gives you the flexibility of late binding wit of early binding—the code never jumps to the wrong address, and yet you don't bake actual entry-point addresses or offsets into the client's code. For example, consider the following .NET inte

```
public interface IMyInterface
{
   void Method1();
   void Method2();
}
```

The interface is defined and implemented in a server assembly, and its client resides in a separate assembly. The client is compiled against the interface definition, and the compiler provides ty enforcing correct parameter types and return values. However, the same client will function just fine (without recompiling) if you change the order of the methods in the interface:

```
public interface IMyInterface
{
   void Method2();
   void Method1();
}
```

or if you add a new method:

```
public interface IMyInterface
{
   void Method3();
   void Method1();
   void Method2();
}
```

If the client doesn't use one of the methods, you can remove it, and the client will be unaffected. In the past, this level of flexibility was available only with late-binding scripting languages, which the code instead of compiling it. All these changes are forbidden in COM interfaces, because they contradict the prime directive of not changing published interfaces. Adding new methods was reason for defining new interfaces in COM (which in turn complicated the COM programming model). However, .NET lets you remove unused methods, add new methods (or fields), and chang of methods (although you can't change method parameters or remove methods that clients expect to use).

Binary Inheritance

An interesting side effect of metadata-based binary compatibility is that it allows binary inheritance of implementation. In traditional object-oriented programming, the subclass developer had to to a source file describing the base class in order to derive her class from it. In .NET, types are described using metadata, so the subclass developer needs to access the metadata only of the b The compiler reads the metadata from the binary file and knows which methods and fields are available in the base type. In fact, even when both the base class and the subclass are defined in project, the compiler still uses the metadata in the project to compile the subclass. This is why in .NET the order in which classes are defined doesn't matter (unlike in C++, which often requires type declarations or a particular order of listing the included header files). All the non-sealed .NET Framework base classes are available to you to derive and extend using binary inheritance. N inheritance is a double-edged sword, though. Inheritance is a form of white-box reuse: it tends to couple the subclass to the base class, and it requires intimate knowledge of the base class's fu Part of the reason why COM didn't allow binary inheritance of implementation (only of interfaces) was because the COM architects were aware of these liabilities. The .NET architects wanted to inheritance as part of bridging the skill gap. However, I strongly advise against abusing inheritance. Try to keep your class hierarchies as simple and as flat as possible, and use interface-based programming as much as possible.

.NET Component Services

.NET provides component services via call interception. To intercept a call, .NET must insert a proxy between the client and the object and do some pre- and post-call processing. Call interception is the key to valuable, productivity-orien services. For example, interception can provide thread safety by trying to acquire a lock before accessing the object and then proceeding to call the object. While the call is in progress, calls coming in from other clients are intercepted as those calls will be blocked when they try to access the lock. When the call returns from the object to the proxy, it unlocks the lock to allow other clients to use the object. Another example of an interception-based component service is *ca* the proxy can verify that the caller has appropriate credentials (such as being a member of a specified role) to call the object, and deny access if it does not. The problem with call interception is that an app domain is too coarse an execu even though cross-app domain calls always go through a proxy, some app domain calls use direct references. To address this problem, app domains are further subdivided into contexts, and objects execute in contexts rather than app d Figure 11-1).

Figure 11-1. App domains and contexts

In .NET, the *context* is the innermost execution scope of an object. Components indicate to .NET which services they require using special context attributes. All objects in the same context are compatible in terms of their component ser requirements, and .NET doesn’t need to perform pre- and post-call processing when these objects access one another. You can, therefore, define a context as a logical grouping of objects that rely on the same set of services. Calls into intercepted to ensure that the objects always get the appropriate runtime environment they require to operate.

Contexts and Object Types

What if a component doesn’t require any services? Why should it pay the interception penalty for cross-context access? To address this point, .NET objects are classified into two categories: those that care about component services an them, and those that don’t. By default, objects are context-agnostic and have no context affinity, which means they always execute in the contexts of their calling clients. Because such context-agnostic objects “jump” from one context to are referred to as *context-agile objects*. The clients of a context-agile object each have a direct reference to it, and no proxies are involved when making intra-app domain calls. Note that objects that derive from MarshalByRefObject are proxy across app domains but are agile inside an app domain. Marshal-by-value objects are also context-agile.

The other type of object is called a *context-bound object*. Context-bound objects always execute in the same context. The designation and affinity of a context-bound object to a context is decided when the object is created and is fixed fo object. To qualify as a context-bound object, the object must derive directly (or have one of its base classes derive) from the abstract class ContextBoundObject:

```
public class MyClass : ContextBoundObject
{...}
```

The client of a context-bound object never has a direct reference to it. Instead, the client always interacts with a context-bound object via a proxy. Because ContextBoundObject is derived from MarshalByRefObject, every context-bound marshaled by reference across app domain boundaries. This makes perfect sense, because the context-bound object can’t leave its context, let alone its app domain.

Component Services Types

.NET provides two kinds of component services: context-bound services and Enterprise Services. The *context-bound services* are available to any context-bound object. .NET offers only one such service: the synchronization domain, de Chapter 8. Using the Synchronization context attribute, a context-bound component gains automatic synchronization and lock sharing:

```
            [Synchronization]
public class MyClass : ContextBoundObject
{
   public MyClass()
   {}
   public void DoSomething()
   {}
}
```

When you add the Synchronization attribute, .NET ensures that only one thread at a time is allowed to access the object, without requiring you to spend the effort implementing this functionality yourself, and you don’t have to worry abou Clients of such objects don’t need to worry about synchronization either. Although this is the only context-bound service provided out-of-the-box, the .NET context is an extensible mechanism, and you can define your own custom contex custom services and extensions. The rest of this chapter examines the .NET context and context-bound objects in detail, and demonstrates how to develop custom context-bound services and attributes.

The second kind of component services are *.NET Enterprise Services* (see the sidebar ".NET Enterprise Services“). Enterprise Services offer more than 20 component services covering an impressive array of domains, from transactions coupled events to web services. Enterprise Services are available only to classes derived from the class ServicedComponent. ServicedComponent is in turn derived from ContextBoundObject, so you can say that Enterprise Services are specialization of context-bound services. If you develop a serviced component, you normally indicate which services the component relies on by using dedicated attributes.

Tip

Enterprise Services attributes aren’t .NET context attributes, but rather COM+ context attributes.

.NET Enterprise Services

.NET Enterprise Services are a set of component services designed to ease the development of Enterprise applications. .NET Enterprise Services are the result of integrating COM+ into .NET, and they offer the same range of services e Enterprise application. For the most part, the semantics of the services, the algorithms behind them, the benefits, and the implied programming models remain exactly the same as they were for COM components. .NET components that advantage of Enterprise Services are called *serviced components* because they derive from the class ServicedComponent, defined in the System.EnterpriseServices namespace. You can configure the services by using context attribute For example, if you want .NET to maintain a pool of your objects, and you want there to be at least 3 but no more than 10 objects in the pool, use the ObjectPooling attribute:

```
using System.EnterpriseServices;

[ObjectPooling(MinPoolSize = 3,MaxPoolSize = 10)]
public class MyComponent : ServicedComponent
{...}
```

If you also want transaction support, use the Transaction attribute:

```
using System.EnterpriseServices;

[Transaction]
[ObjectPooling(MinPoolSize = 3,MaxPoolSize = 10)]
public class MyComponent : ServicedComponent
{...}
```

You can also configure the pool parameters (and any other services) after deployment by using the COM+ Component Services Explorer . However, due to .NET’s ability to persist the attributes in the assembly metadata, there is often n the Component Services Explorer for services configuration. You can read about .NET Enterprise Services in my book *COM and .NET Component Services* (O’Reilly).

Chapter 8. Multithreading and Concurrency Management

In a single-threaded application, all operations, regardless of type, duration, or priority, execute on a single thread. Such applications are simple to design and build. All operations are serialized; that is, the operations never run concurrer one at a time. However, there are many situations in which employing multiple threads of execution in your application will increase its performance, throughput, and scalability, as well as improving its responsiveness to users and clients multithreading is one of the most poorly understood and applied concepts of contemporary programming, and many developers tend to misuse the multithreading features and services available on their programming platforms.

This chapter begins with an explanation of .NET threads and shows you how to create and use them. You'll also learn how to spawn and manage multiple threads, develop thread-safe components, and avoid some of the common pitfall multithreaded programming. The chapter then moves on to discuss how to use .NET to synchronize the operations of multiple threads and manage concurrent attempts to access components. The chapter ends by describing several use multithreading services, including thread pool, timers, and thread local storage.

<meta http-equiv="refresh" content="0; url=/library/no-js/" />

Threads and Multithreading

In modern computing terminology, a *thread* is simply a path of execution within a process. Every application runs on at least one thread, which is initialized when the process within which the application runs is started up. The threads of always execute within the context provided by the application process. Typically, you find two kinds of operations in any application: CPU-bound and I/O-bound operations. *CPU-bound* operations use the machine's central processing ur perform intensive or repetitious computations. *I/O-bound* operations are tied to an input or output device such as a user-interface peripheral (keyboard, screen, mouse, or printer), a hard drive (or any non-memory durable storage), a net communication port, or any other hardware device.

It's often useful to create multiple threads within an application, so that operations that are different in nature can be performed in parallel and the machine's CPU (or CPUs) and devices can be used as efficiently as possible. An I/O-bour (such as disk access), for example, can take place concurrently with a CPU-bound operation (such as the processing of an image). As long as two I/O-bound operations don't use the same I/O device (such as disk access and network-s having them run on two different threads will improve your application's ability to efficiently handle these I/O devices and increase the application's throughput and performance. In the case of CPU-bound operations, on a single-CPU ma no performance advantage to allocating two distinct threads to run separate CPU-bound operations. However, you should definitely consider doing so on a multi-CPU machine, because multithreading is the only way to use the extra pro available through each additional CPU.

Almost all modern applications are multithreaded, and many of the features users take for granted would not be possible otherwise. For example, a responsive user interface implies multithreading. The application processes user reques printing or connecting to a remote machine) on a different thread than the one employed for the user interface. If the same thread were being used, the user interface would appear to hang while the other requests were being processed multithreading, because the user interface is on a different thread, it remains responsive while the user's requests are being processed. Multiple threads are also useful in applications that require high throughput. When your application process incoming client requests as fast as it can, it's often advantageous to spin off a number of worker threads to handle requests in parallel.

Warning

Do not create multiple threads just for the sake of having them. You must examine your particular case carefully and evaluate all possible solutions. The decision to use multiple threads can open a Pandora's box of thread-synchronizati component-concurrency issues, as you'll see later in this chapter.

Components and Threads

A component-oriented application doesn't necessarily need to be multithreaded. The way in which an application is divided into components is unrelated to how many execution threads it uses. In any given application, you're as likely to components interacting with one another on a single thread of execution as you are to have multiple threads accessing a single component. What is special about component-oriented applications has to do with the intricate synchronizat inherent in the nature of component development and deployment. You will see later in this chapter how .NET addresses these challenges.