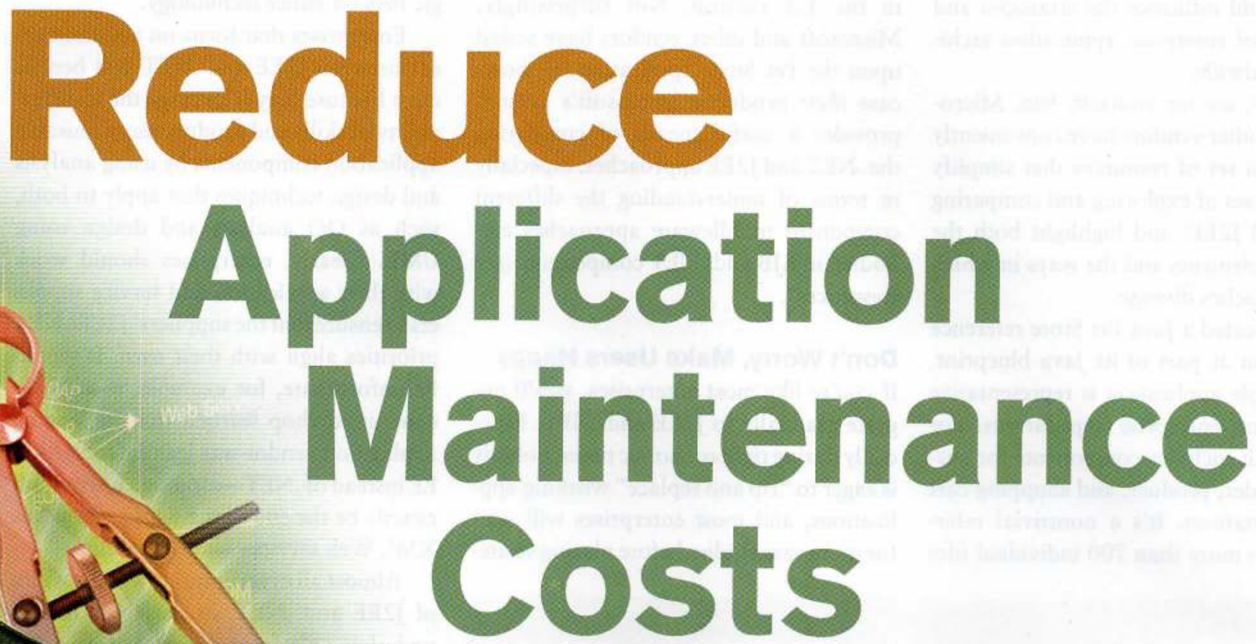
Reduce Application Maintenance Costs



Discover how component-oriented programming and .NET's features can help lower development and maintenance costs and speed time-to-market.

by Juval Löwy

Over the last decade, component-oriented programming has established itself as the predominant software development methodology as the industry has moved away from giant, monolithic, and hard-to-maintain code bases. Practitioners discovered that by breaking a system down into binary components, you can gain reusability, extensibility, and maintainability. Component-oriented programming can bring about major benefits such as faster time-to-market, robust applications, and lower development and long-term maintenance costs.

It's no coincidence that component-oriented programming has caught on in a big way. In this article I'll present the core principles and benefits of component-oriented programming. I'll also contrast it with object-oriented programming, although the two methodologies do have some things in common.

"Component" is probably one of the most overloaded and confusing terms in modern software engineering, and the .NET documentation isn't without its fair share of inconsistencies. The source of the confusion is where to draw the line between a class implementing some logic, its containing physical entity (typically a DLL), and its containing logical deployment, security, and versioning unit (the assembly). For our purposes, a *component* is a .NET class that is responsible for exposing some business logic to clients. A *client* is any party that makes use of the component, typically other classes. An *object* is an instance of a component, similar to the classic object-oriented definition of an object as an instance of a class.

For This Solution

- A basic understanding of component- and object-oriented programming

Go Online

To read this article online at **www.thedotnetmag.com** use this Locator+ code: **NM0205JL_T**

So, if every .NET class is a component, and if both classes and components share so many qualities, you might be wondering what the difference between traditional object-oriented programming and component-oriented programming is. The fundamental difference lies in the way the two methodologies view the final application.

The Component vs. Object Face off

In the traditional object-oriented world, even though the developer or the architect might have factored the business logic into many fine-grained classes, once these classes are compiled the result is one monolithic chunk of binary code. All the classes share the same physical deployment unit (typically an EXE), the same process, the same address space, the same 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 triggers a massive relinking of the entire application, and retesting and redeployment of all other classes.

On the other hand, a component-oriented application is comprised of a collection of interacting binary modules—the components (see Figure 1). The application implements and executes its required business logic by gluing together the functionality offered by the individual components. Component-enabling technologies such as COM, CORBA, and .NET provide the "plumbing" infrastructure to connect binary components together in a seamless manner, and the main distinction between these technologies is the ease with which you can connect components together.

The motivation for breaking down a monolithic application to multiple binary components is analogous in the object-oriented world to the motivation for putting different class' code in different files. By allocating different classes to different files, you promote looser coupling between the classes and between the developers responsible for these classes. A change made to one class may trigger only recompilation of that class' file (although the entire application will have to go through re-linking). But there are more benefits to component-oriented programming than easier project management.

Because the application is a collection of binary building blocks, you can start treating components as Lego blocks—plugging and unplugging components at will. If you need to modify any one of these components, the changes are contained in that component only. No client is affected or requires recompilation and redeployment. You can even update components while the application is running (as long as they aren't used by clients).

In addition, improvements, enhancements, and defect fixes made to one component are immediately available to all applications using that component, on the same machine or across the network. It's also easier to extend a component-oriented application. When you need to implement new requirements, you can provide these in new components, without having to touch existing components that aren't affected by the new requirements.

When you combine these factors, component-oriented programming can reduce the costs involved with long-term maintenance, an issue essential to almost any company, which explains the wide adoption of component technologies. Component-oriented applications also usually enjoy a faster time to market, because developers can select from a wide range of available components (either in-house or from component vendors), and avoid reinventing the wheel. Consider, for example, the rapid development mode of many Visual Basic 6.0 applications, which rely on libraries of ActiveX controls for almost every aspect of the application.

Avoid Complex Class Hierarchies

Object-oriented and component-oriented methods also differ in the emphasis each one places on inheritance and reuse schemas. In object-oriented analysis and design, software engineers often model their application in terms of complex class hierarchies, trying to approximate in their software class hierarchy modeling the business problem being solved. Reusing existing code is promoted by inheriting from an existing base class and specializing its behavior. Unfortunately, inheritance makes a poor reuse mechanism. When developers derive a subclass from a base class, they must be intimately aware of the implementation details of the base class. For example, if you change the value of a member variable it could affect the code in the base class. Or overriding a virtual method in the base class and providing a different behavior could break the code of clients that expect the base behavior. This form of *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. Developers can use an existing component without regard to its internals, as long as the component complies with a 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 allows components to use inheritance of implementation (deriving from a class with functionality and members), and you certainly could develop complex class hierarchies. However, you should keep your class hierarchies as simple and flat as possible, and focus instead on factoring interfaces. Doing so promotes black box reuse of your component instead of white box reuse through inheritance.

Finally, object-oriented programming provides developers next to nothing when it comes to real-time design patterns such as multithreading and concurrency man-

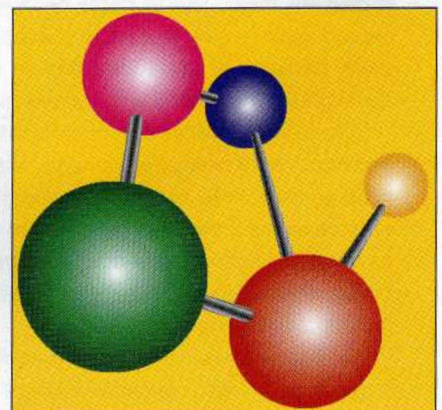


Figure 1. Get the Component-Oriented View. Instead of a traditional monolithic binary entity, a component-oriented application is comprised of multiple interchangeable binary components.

agement, security, distributed application, not to mention application deployment and version control. Object-oriented developers are more or less left to their own devices when it comes to these patterns. As a component technology, .NET supports the developer by providing a superb component development infrastructure, allowing the developers to focus on their business problem at hand, instead of run-time issues. Ultimately, these concepts compliment each other as well, because .NET provides an object-oriented programming paradigm in the context of components.

Take a Component-Oriented Approach

What constitutes component-oriented programming is an ever-evolving set of principles. Often it's hard to tell which aspect is a true principle and which is only a feature of the component technology used. As the supporting technologies become more powerful, software engineering will undoubtedly extend its perception of component-oriented programming and embrace new ideas. Next, I'll outline the core set of principles of component-oriented technologies as they stand today.

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

To use a component, all the client needs to know is the interface definition (the service contract), and have a binary component that implements that interface. This extra level of indirection between the client and the object enables interchangeability between different implementations of the same interface, without affecting the client code. The client doesn't need to recompile its code to use a new version and sometimes doesn't even need to be shut down to do the upgrade. Provided the interface is immutable, the objects implementing the interface are free to evolve and introduce new versions. To implement the interface functionality inside a component, developers still use traditional object-oriented methodologies, but usually the resulting class hierarchies are simple and easy to manage. Unlike COM, .NET doesn't enforce separation of interface from implementation. Developers can work with either interfaces or direct public methods, similar to Java. (For the rationale behind this .NET behavior, see the sidebar, "Bridging the Skill Gap.") So, although from the puritan perspective COM is better, from the practical perspective .NET is better. COM is unnecessarily ugly because it was built on top of Windows, an operating system that is component-agnostic, and when implementing COM you use languages like C++, which are object, not component oriented. .NET on the other hand, is built on top of a fresh, component-oriented runtime, and, therefore, has an easier time providing these core concepts, even though it doesn't enforce them.

Another core principle of component-oriented programming is *binary compatibility* between client and server. Traditional object-oriented programming requires that all the parties involved—clients and servers—be part of one monolithic application. During compilation, the compiler bakes the address of the server entry points into the client code. Component-oriented programming revolves around packaging code into components, also know as *binary building blocks*. Changes to the component code are contained in the binary unit hosting it, and you don't need to recompile and redeploy the clients. But the ability to replace and plug new binary versions of the server implies binary compatibility between the client and the server. This means the client's code must interact

Bridging the Skill Gap

One of the challenges facing the software industry today is the skill gap between what developers should know and be proficient at to meet quality, schedule, and budget goals, and what developers are actually capable of doing. Even developers with a formal computer science background often lack effective component-oriented design skills, which are primarily obtained through experience. Today's aggressive deadlines and a shortage of developers preclude attending dedicated training sessions or providing efficient, on-the-job training. Nowhere is the gap more visible than in adhering to the principles of component-oriented development. Object-oriented concepts are easier to understand and apply than component-oriented principles, so developers are more likely to implement them. This is partly because object-oriented concepts have been around much longer, so a larger portion of developers are familiar with them, and partly because of the added degree of complexity involved with component development compared with monolithic applications.

One of Microsoft's prime goals with the .NET platform is to simplify developing and consuming binary components, and to make component-oriented programming accessible to as many developers as possible. As a result, .NET doesn't enforce some of the 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. Developers that understand only object-oriented concepts will develop .NET objects, but because every .NET class is consumed as a binary component by its clients, these developers will gain many of the benefits of component-oriented programming. Developers who understand and master how to apply these principles will be able to fully maximize the benefit of .NET as a powerful component development technology.

at run time 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 with this contract, the client isn't affected.

Microsoft's first attempt at component technology using DLLs provided binary compatibility utilizing enumerated entry points (called ordinal numbers) to the DLL. However, this approach was fragile and error prone. COM was the first technology to truly provide binary compatibility using virtual tables (interface definitions). .NET takes COM's binary compatibility to a whole new level by using metadata and Just-In-Time compilation, which avoids baking the actual binary layout into the client code until runtime. As a result, changes such as adding new methods don't break the client's code.

Utilize 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, binary compatibility is the only thing that binds the two together. This means the programming language used to implement the client or the server is irrelevant to their ability to interact. Language independence means exactly that: When developing and deploying components, the programming language used shouldn't be taken into account. *Language independence* promotes component interchangeability, adoption, and reuse. By not caring which language was used to develop the component, the client has a wider, richer offering from multiple vendors to choose from.

COM was the first technology to support true language independence, with the canonical example of Visual Basic clients using C++ COM components to do all the things VB can't do. .NET is based on the common language runtime (CLR), a runtime environment where all components execute, regardless of language. So, by its very nature, .NET supports language independence. When you contrast .NET with Java, it significantly lags behind in language independence, because Java clients can only use Java components.

A component-based application contains multiple binary components. These components can be all in the same process, in different processes on the same machine, or on different machines in the same local network. Recently, with the advent of Web services, you can also distribute components across the Internet.

> Component-oriented programming must also allow clients and components to evolve separately.

The underlying component technology is required to provide the client with *location transparency*, allowing the client code to be independent of the actual location of the object it uses. Location transparency means there's nothing in the client's code pertaining to where the object executes. The same client code handles all cases of object location, although the client should be able to insist on a specific location as well (see Figure 2).

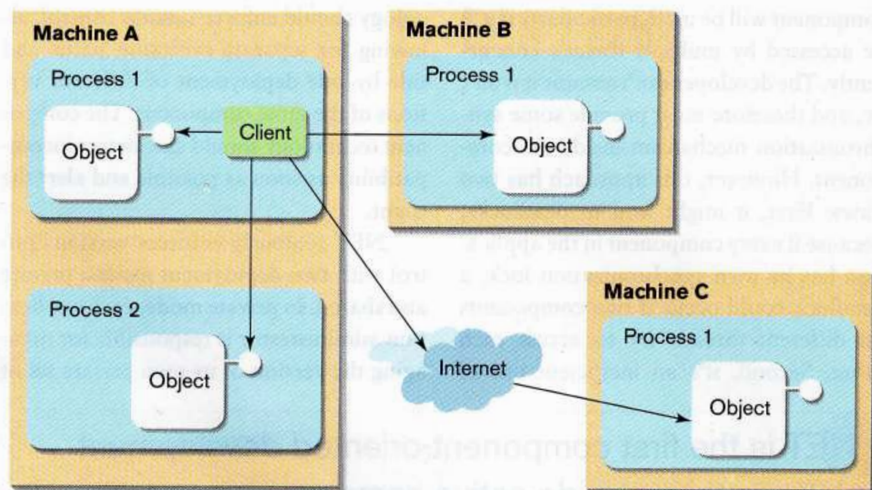


Figure 2. Be Transparent. Location transparency enables the client code to be oblivious of the actual object location. The object can be in the same process, in different processes on the same machine, on different machines in the same local network, or even across the Internet.

Location transparency is crucial to component-oriented programming for a number of reasons. First, it enables developers to develop the client and components locally (providing easier and more productive debugging), yet deploy the same code base in distributed scenarios. Second, using the same process for all components—or multiple processes or even multiple machines—has significant tradeoffs on performance and ease of management vs. scalability, availability, robustness, throughput, and security. Different customers will 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 customers. Third, component location 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 the client code make any assumptions regarding the location of objects it uses, and to avoid making explicit calls across processes or across machines.

DCOM offers elegant location transparency to COM components, and .NET also offers location transparency using the infrastructure of .NET Remoting. .NET also has native XML Web services support that completely takes care of the details (both on the server and client sides) of invoking a call over the Internet. Java uses Remote Method Invocation (RMI) for remote calls, and, as such, doesn't provide true location transparency.

Provide Concurrency Management

A component developer can't possibly know in advance all the possible ways the

component will be used, particularly if it'll be accessed by multiple threads concurrently. The developer can't assume it won't be, and therefore must provide some synchronization mechanism inside the component. However, this approach has two flaws: First, it might lead to deadlocks, because if every component in the application has its own synchronization lock, a deadlock could occur if two components on different threads try to access each other. Second, it's an inefficient use of system resources when all the components in the application are always accessed by the same thread.

.NET is the first component-oriented development platform that provide native component-oriented aspects, all the way from the runtime to the development tools.

To resolve this issue, the underlying component technology must provide some *concurrency management* service: a way for components to participate in some application-wide synchronization mechanism, even though the components were developed separately. In addition, the underlying component technology should enable components and clients to provide their own synchronization solutions for fine-grained control and optimized performance. COM provides concurrency management through an apartment, a cumbersome mechanism that relies on thread affinity. .NET concurrency management support is elegant and granular. Developers can either provide manual synchronization using locks, or they can ask .NET to automatically synchronize access to their components using attributes. .NET's automatic synchronization mechanism allows multiple components, from different vendors, to share a lock.

Component-oriented programming must also 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 component versions. The underlying component technology should enforce *version control*, allowing for separate evolution paths and side-by-side deployment of different versions of the same component. The component technology should also detect incompatibility as soon as possible and alert the client.

.NET zealously enforces version control with two deployment modes: private and shared. In private mode, each application administrator is responsible for managing the version of its own private set of components. Changes made to one application and its private set don't affect other applications. Shared mode allows all applications to share a global components repository. The global repository can contain multiple versions of the same component. .NET automatically loads a compatible component version for the client, either from its private set or from the global, shared location.

Ensure Security

In component-oriented programming, components are developed separately from the client applications using them. Component developers have no way of knowing anything about the intent of the client application and the end user. A benign component could be used maliciously to cause damage such as data corruption or perform illegal operations, for example, transferring funds without proper authorization and authentication. Similarly, a client application has no way of knowing whether it's interacting with a malicious component that will abuse the credentials of the calling client. In addition, even if both the client and the component have no ill intent, the end user can still try to hack into the system or do some other damage (even by mistake).

The underlying component technology must provide the security infrastructure to deal with these scenarios, without coupling components and client applications to each other by the nature of the security mechanism. In addition, security requirements, policies and events (such as new users) are one of the most volatile, ever changing aspects of the application life cycle, 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 incorporate security awareness into the code itself, and allow for system administrators to customize and manage the application security policy without escalating code changes back to the developer. .NET offers rich security infrastructure to address these concerns. It verifies at runtime that the calling client (in fact, the entire clients calling chain) has the right permissions to access the component and that the component has permissions to carry out what the component is asked to do. .NET allows system administrators to configure what components are allowed to do, and what evidence these components should provide to verify their origin and authenticity.

.NET is the first component-oriented development platform that provide native component-oriented aspects, all the way from the runtime to the development tools. To make the most of .NET, and come to terms with it, you need to understand the rationale behind it. Adhering to the abstract principles of component-oriented programming, that were the driving forces behind .NET, will provide robust and extensible applications for years to come. .net

About the Author

Juval Löwy is a software architect and the principal of IDesign, a consulting and training company focused on .NET design and migration. Juval is the author of *COM and .NET Component Services* (2001, O'Reilly & Associates). This article is based on excerpts from his upcoming book on programming .NET components, also from O'Reilly & Associates. Juval is a frequent speaker at international software development conferences, and he chairs the program committee of the .NET California Bay Area User Group. Contact him at www.idesign.net.