Issue	November 2001
Section	
Main file name	Nu0001comingofaget3.doc
Listing file name	
Sidebar file name	
Table file name	
Screen capture file names	
Infographic/illustration file names	Nu0001comingofagef1.doc
Photos or book scans	
Special instructions for Art dept.	
Editor	
Status	Nu0001comingofage.doc (raw) JBL content edit (t1) author review (t2) final (t3)
EN review	
Character count	
Package length	
ToC blurb	To fully exploit .NET, you should know how the technology evolved, and how it corrects the mistakes and improves on the successes of the past.

Overline:

Byline:
By Juval Lowy

Head:

Software Engineering Comes of Age

.NET finally gets it right.

.NET is the product of 40 years of advances in software engineering. You're probably asking, have they finally got it right? Will the darn thing work? Will I have to spend another weekend lost in the white glare of my computer screen? The answer is yes, yes, that depends on your boss.

To make the most of .NET, you should know where the technology comes from and how the Microsoft created it to correct the mistakes and improve on the successes of the past. To this end, the .NET design team observed how software engineers develop applications and solutions, the hurdles they face, the tradeoffs they make, and the successful design methodologies they use. Understanding .NET's evolution can help you optimize your own application designs for maintainability, extensibility, reusability, and productivity.

Contrary to common knowledge, the past few decades have seen only a handful of software-engineering breakthroughs (see Figure 1). Most of the changes have been incremental. Fifty years ago domain experts, such as scientists or electrical engineers, performed most programming using machine code or manipulating hardware bits. The challenge these domain experts faced was to squeeze every drop of performance and space from enormously expensive commuters, amazingly weak by today's standards.

This situation didn't change significantly even when high-level structured languages came on the scene. Applications still had limited requirements, and the user interface was primitive. When computers became more pervasive, developers realized the real issue was to maintain applications for longer periods. The hardware's cost and even the initial development effort only mounted to a small fraction of the cost spent maintaining an application over its lifetime (normally a few years during the '70s).

Two things became clear at that point: Architects had to design applications for extensibility from day one, and the cost of a downstream change (during the maintenance phase) was sometimes hundreds of time more expensive than making the same change during the application's initial design. Unfortunately, the programming languages developers used during the late '70s and early '80s, such as C, Pascal, and Fortran, landed themselves easily into creating spaghetti-like applications, with tight coupling between different elements of the program. A change made to one part of the system triggered an avalanche of changes all over.

Programs typically used function-data decomposition, meaning that global data structures, such as variables, lists, and arrays, held the data, and global function manipulated the data and executed the business logic. You could write good code with languages like C, but the language itself had no native support for encapsulation, inheritance, and polymorphism. If you wanted those features, you had to manually provide them, and by doing so, you sometimes made the application even more complex.

These problems led to the rise of object-oriented programming (OOP). This approach is based on a simple idea—encapsulate the data and the logic manipulating the data in an object. The object only exposes abstracted entry points implementing some generic behavior contract. The object hides the actual implementation of both the data and the logic. By hiding and encapsulating implementation details, changing these details doesn't affect the client code using them.

With the advent of C++, object-oriented programming became pervasive during the '80s and early '90s. C++ promised to solve the software industry maintenance crisis and lack of reuse. OOP also offered polymorphism between different implementations of the same set of methods, and inheritance of implementation. Software engineers started modeling their applications in terms of complex class hierarchies, trying to approximate as much as possible the business problem their modeling solved.

OOP Fails to Deliver

However, object-oriented programming failed to deliver on its promises of maintainability, extensibility, and reuse, thus becoming probably the great disappointment of software engineering. The reasons have to do with both the nature of OOP itself and how programmers used it. The key to successfully applying object-oriented ideas is careful design and expert experience in system analysis and architecture. C++ and object-oriented analysis and design experts are rare—as few as one in 100 software developers have these skills. Even though every developer used languages like C++, it's really a tool for experts. In the wrong hands, C++ causes more harm than good. On top of that, C++ still allowed for bad practices, including global variables or methods, and "friend" classes. This increased coupling between classes in the application.

Even in the hands of experts, object-oriented programming has intrinsic flaws. First, inheritance makes for a poor reuse mechanism. When a developer derives a subclass from a base class, the developer must be intimately aware of the base class's implementation details. For example, what happens when you change a member variable's value? How does this action affect the code in the base class? This form of "white box" reuse simply doesn't allow for economy of scale in large organizations' reuse programs or successful adoption of third-party frameworks.

Second, object-oriented programming provided developers next to nothing when it came to real-time design patterns such as multithreading concurrency management, security, and distributed application—not to mention application deployment and version control. Third, object-oriented methodology assumed the application was one monolithic chunk of code and that reuse was usually source-files based, meaning the reusing party had to have the sources of the objects providing the functionality. When developers in the early '90s started composing applications out of dynamically loaded binary libraries (DLL), that left no easy way of accessing the objects in these modules. The scene was set for the rise of component-oriented programming.

This technology's fundamental principal is that the unit of use is an interface providing abstract service definition, not the object implementing the interface. You implement the interface on a black-box binary component that encapsulates completely its interior. This principal is called 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 the interface. This extra level of indirection allows for "plug and play" between different implementations of the same interface, without affecting the client code at all. The client need not recompile its code to use a new version, or sometimes not even to shut down for an upgrade.

Provided the interfaces are immutable, the objects implementing the interfaces are free to evolve and introduce new versions. Because the client interacts only with an interface, you can introduce a proxy (an object providing the same interface as the real object) between the client and the object. You can do a lot with proxies. You can have them redirect the call to a remote machine or synchronize access to the object by multiple threads; you can also manage transactions, enforce access security, and so on. In essence, you can provide almost any conceivable component service, without having the object developer invest costly development effort.

COM Pushes the Envelope

In component-oriented programming, developers still used traditional object-oriented methodologies inside a component, but usually the resulting object hierarchies were simple and easy to manage. Component technologies, such as COM, were a major breakthrough in software engineering, and they provided additional benefits, including language independence. As long as the client and the object agreed on the interface, and if you gave at run time the right binary signatures, you could use any language to implement the component and its client. Another COM benefit is location transparency. When using proxies, nothing in the client code is pertinent to the object location (such as sockets or pipes calls), and as result, you can change the object location without affecting the client.

But these first attempts at component technologies had their flaws. First, implementation languages like C++ didn't support components natively, so developers had to use intricate and not trivial to learn frameworks such as ATL. Second, the supporting operating system (Windows, for example) provided its services in the form of thousands of inconsistent functional entry points, making the overall programming model complex.

Third, developers still had to manage many aspects of their code, including versioning, memory allocation, and object life cycle. The result was that even though technologies such as COM and CORBA seemed like a good idea, in practice developers spent as much as 80 percent or more of their time on component connectivity and "plumbing" issues, instead of adding business value to their application. Not only that, but you could trace most bugs (and the time spent fixing them) back to connectivity and plumbing defects (for example, memory management and object lifecycle), not to the business problem the application addressed.

In the second half of the '90s, with the Internet boom, many companies (old and new) shifted their engineering focus from design and support of long-term maintenance to rapid development. A software application's ever-shorter shelf life meant that being first to market and beating the competition with successful fast releases became more important than up-front investments in design, maintainability, and developers' skills. A development team could no longer spend the lion's share of its time on "plumbing" issues.

Technologies such as Java tried to remedy this by off-shouldering memory management from the developers and by offering a comprehensive set of base classes for services usually provided by the operating system. However, Java is only incrementally better than C++, and its improvements introduced liabilities, including poor performance and difficulties in sharing expensive objects between clients. The Internet also showed that the component-oriented, client-server model simply didn't scale well from a standalone application (and even a distributed application with a handful of clients) to Internet applications. Suddenly, applications had to withstand huge numbers of clients at peak load, with spiking fluctuating degrees of load, be available all the time, ensure security and system integrity, and maintain overall system consistency.

.NET to the Rescue

In my opinion, .NET has learned from the mistakes and lessons of the past and integrated the strengths of existing technologies, including Java, C++, VB, and COM. For example, .NET programming languages (such as, C#) don't have C++'s pitfalls, such as global types and functions, and multiple inheritance of implementation. C# mimics C++ type safety and its object-oriented features. C# also borrows properties from VB and garbage collection from Java.

.NET is a modern and elegant component technology enabling rapid development of interacting binary components. .NET simplifies enormously component development, while maintaining component-oriented programming's core concepts, required for scalable and maintainable applications. .NET gives you fundamental component-oriented development principles, including binary compatibility between client and component, separation of interface from implementation, object location transparency, concurrency management, security, and language independence.

To simplify managing memory allocations and object lifecycle, .NET uses a sophisticated garbage collection that detects when clients no longer use an object and then destroys the object. .NET provides a standard binary way of describing exported types and interfaces. All the developer has to do is put the types definitions as part of the source files, and .NET builds the information automatically as part of code compilation.

.NET maintains zealously version compatibility between object and clients. .NET packs objects into assemblies and gives each assembly an elaborate version number indicating major and minor version numbers, and build number and revision number (provided explicitly by the developer or automatically as part of compilation). At run time, .NET ensures the client always gets a compatible assembly. Developers can also digitally sign the assembly, ensuring authenticity, providing more uniqueness, and allowing the clients to share the assembly.

Even though .NET applications run on operating systems such as Windows, .NET encapsulates and simplifies the interaction with the underlying operating systems services. For example, in Windows the call to create a new window accepts 12 parameters. Even the most trivial Windows program requires at least three files and some 80 lines of code. In .NET, you can achieve that with one file, a few lines of code, and the call to actually create a window requires no parameters. .NET provides operating system services, including threading, network calls, and file I/O, in a consistent and easy to learn manner.

.NET uses heavily object-oriented design patterns in its class libraries and frameworks. These patterns contribute the most to object-oriented programming in NET, offering reusable, proven design solutions to common problems. At the same time, .NET improves some object-oriented concepts. For example, .NET enforces strict inheritance semantics and inheritance conflict resolution. You can derive a class from only one concrete class. You can, however, derive from as many interfaces as you like. When you override a method in a base class, you must declare your intent explicitly, whether you want to override the base class implementation or hide it. .NET allows binary inheritance—a developer doesn't need a base class's source files to subclass and specialize its behavior, he only needs a binary component containing the base class.

.NET Provides a Modern Programming Model

.NET supports many recommended design principles, including separation of user interface from implementation. Application frameworks, such as ASP.NET, provide visual user interface design environment (for controls layout) and a separate class that represents the logic to perform in response to user events.

Instead of having developers code runtime requirements (for example, multithreading concurrency management or object persistence and serialization), developers can use special attributes to declare the class needs. .NET provides numerous attributes, so you can focus on the domain problem at hand. You can also define your own attributes or extend existing ones.

.NET provide a modern, component-oriented, component-security model. Developers don't have to hard code security policy in their applications (making them resilient to policy changes), while maintaining granular control. The classic Windows NT security model is based on what a given user is allowed to do. This model evolved during when component technologies were in their infancy and applications were usually standalone, monolithic chunks of code. In today's highly distributed, component-oriented environment, you need 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.

In addition to the preceding advantages, .NET lets you configure permissions for components and provide evidence to prove that the code has the right credentials to access a resource or perform some sensitive work. Evidence-based security is related closely to the component's origin. System administrators can decide they trust all code 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 doing its work. .NET aims at simplifying component deployment as well. All you need to do is copy the components to the client directory. If you want to share components, you can install them in a known shared location.

In general, most software development tools offer only either flexibility and power, or ease of use and rapid development. For example, VB6 caters for rapid development at the expense of maintainability and scalability, while C++ offers power and capabilities, at the expense of productivity. In contrast, .NET doesn't force a developer to choose one way or the other, and it caters to a wide range of developer skills.

Here are a few examples: Developers can create a window with one line of code, but they can also override and manipulate the underlying message pumping loop. Developers can synchronize manually access to their objects by multiple threads to optimize performance, or they can simply use a synchronization attribute to instruct .NET to do so for them. Software engineers can adhere to strict component-oriented principals, such as separation of interface from implementation, but they can also do plain old object-oriented programming, all in the same code base. Developers can design classic client-server applications, or they can design powerful, high-throughput multi-tier applications, while relying on state-of-the-art enterprise and component services, to address the throughput and scalability requirements.

This freedom results in much smoother programming models, and developers with various degrees of skills and experience can work on the same project, maximizing their skills and productivity.

About the Author:
Juval Lowy is a software architect and the principal of IDesign, a consulting company focused on .NET design and .NET migration. Juval also conducts training classes and conference talks on component-oriented design and development processes. He wrote "COM and .NET Component Services—Mastering COM+" (O'Reilly). Reach him at www.componentware.net.

Captions

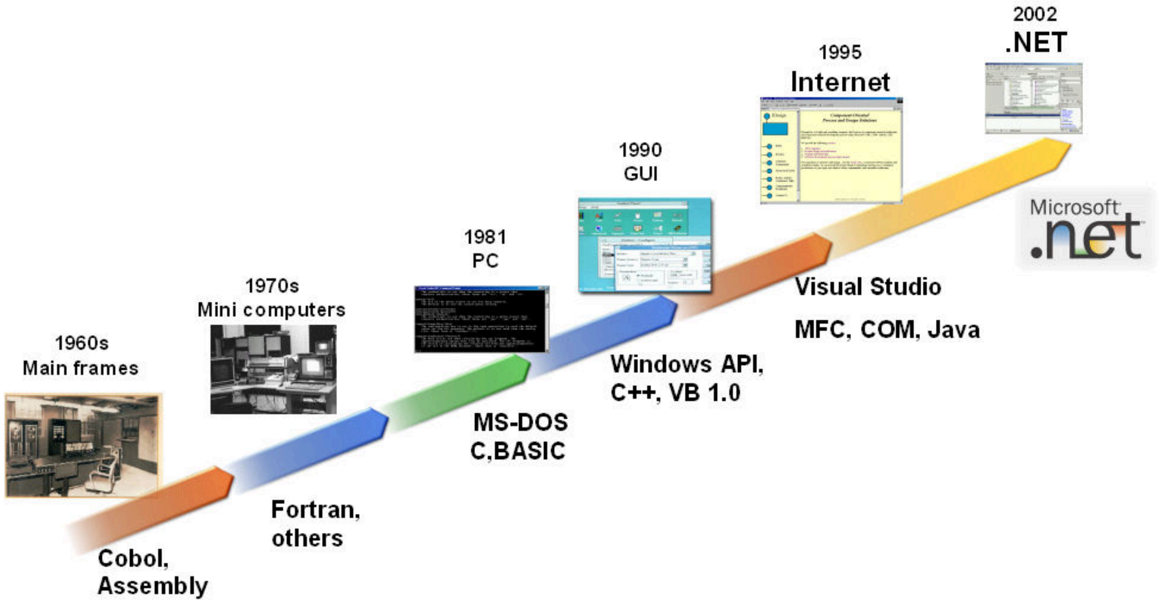

Figure 1: Experience a Brief History of Time
Contrary to common knowledge, the past few decades have seen been only a handful of software engineering breakthroughs. The move to .NET is as significant as the move from DOS to Windows, or from machine code to high-level structured languages.

Pullquotes

When computers became more pervasive, developers realized the real issue was to maintain applications for longer periods.

OOP is based on a simple idea—encapsulate the data and the logic manipulating the data in an object.

Even in the hands of experts, object-oriented programming has intrinsic flaws.