.NET Is Coming

Bertrand Meyer, Interactive Software Engineering

Announced in July 2000, .NET, Microsoft's platform for XML-based Web services, is currently undergoing a succession of beta versions for a projected release late this year or early in 2002. .NET has a central role in Microsoft's strategy to integrate the Internet, Web services, building block services, numerous tools for developers, and many other features.

Curiously though, this next generation of software hasn't grabbed the computing world's attention as Java did in its heyday. Yet, in many respects, .NET is a more important phenomenon. The business press, for its part, hasn't missed the technology's significance. Both Business Week (cover story, 30 Oct. 2000) and The Economist (January 2001) have devoted major coverage to .NET.

A FAILURE OR THE FUTURE?

Perhaps it is .NET's breadth of coverage that has puzzled some technical observers and made them wonder whether there was anything beyond the hype. One observer who doesn't think so is John Dvorak, who in his November 2000 *PC Magazine* column wrote that .NET is "surrounded by too many buzzwords and generalities to be understandable. I'm not sure the company knows what .NET is, or whether anybody does. It has the onerous smell of failure about it already."

The basis for this indictment seems to be a Microsoft public relations document describing an exciting future in which—thanks to .NET—from your Web-connected bike at the gym, you can effortlessly change both your evening's restaurant reservation and babysitter while pedaling away without missing a beat. If the aim is to set lofty ideals for the next generation, this example is admittedly underwhelming, but since when are technical journalists supposed to judge new technologies by press releases?

.NET, an open language enterprise and Web development platform, has the potential to dominate the computing industry for years to come.

Others have been more perspicacious. For example, the Patricia Seybold Group wrote that ".NET is a leading example of what we believe will be the dominant architectural model for the third generation of Internet applications" and that it has "ominous implications for a large number of Microsoft competitors."

WHAT .NET ISN'T

In describing .NET, it's useful first to point out what it is not. It's neither an operating system nor a programming language. Microsoft operating systems continue their own evolution—Windows 2000, Me, XP, CE for embedded devices—although you can expect more .NET bits to filter down into the base OS. As for programming languages, .NET has introduced a new one, C# (C-sharp), but it's not the focus of the technology—it's simply the means to an end, the basic notation for programming the .NET runtime.

Technically, C# looks very much like Java, with extensions similar to mechanisms found in Delphi and Microsoft's Visual J++. These extensions include "properties"—an attempt to remedy Java's information-hiding deficiencies—and an event-driven programming model using the notion of "delegates"—object wrappers around functions—that is appropriate for graphical user interface and Web applications.

While it is likely to become a serious competitor to Java, C# is not an attempt to replace all existing languages. In fact, Microsoft's own investment in Visual Basic (recent estimate: 6 million developers) and C++ would make such a goal self-defeating.

Instead, a distinctive characteristic of .NET is its language neutrality. In addition to Microsoft-supported languages, .NET is open to many others including Cobol, Eiffel, Fortran, Perl, Python, Smalltalk, and a host of research languages from ML to Haskell and Oberon. Unlike others in the industry, Microsoft isn't trying to convert the world to a new language.

.NET ARCHITECTURE

So what is .NET? A general definition might be: "An open language platform for enterprise and Web development." The aim is to provide an abstract machine for professional developers, covering both traditional IT—client-server, *n*-tier—and Web-oriented applications. Figure 1 shows the six layers of the platform's overall structure.

Web services. The top layer provides .NET users—persons and companies—with Web services for e-commerce and business-to-business applications.

Frameworks and libraries. A set of frameworks and libraries provides the most immediately attractive aspect for developers. These include ASP.NET, active server pages for developing smart Web sites and services; ADO.NET, an XML-based improvement to ActiveX Data Objects, for databases and object-relational processing; and Windows Forms for graphics. Altogether, .NET contains thousands of reusable components.

Interchange standards. XML-based interchange standards serve as a platform-independent means of exchanging objects. The most important are SOAP (simple object access protocol), an increasingly popular way to encode objects, and WSDL (Web Services Description Language).

Development environment. The new Visual Studio.Net provides the tool of most direct use to developers: A common software development environment offering facilities for development, compilation, browsing, and debugging shared by many languages...This environment, an outgrowth of Visual Studio extended with an application programming interface, not only supports Microsoft-implemented languages such as Visual C++, Visual Basic, and C# but also allows third-party vendors to plug in tools and compilers for other languages.

Component model. Before .NET there were already three major contenders for leadership in the field of models and standards for component-based development: Corba from the Object Management Group, J2EE from Sun, and Microsoft's COM. .NET brings in one more model, based on object-oriented ideas: With .NET you can build "assemblies," each consisting of a number of classes with well-defined interfaces. The model is quite different from COM, although it provides a transition path; its major attractions are its simplicity and the absence of an IDL (Interface Description Language).

Object model. The object model provides the conceptual basis on which everything else rests, in particular, .NET's OO type system. The common language specification defines restrictions ensuring language operability.

Common language runtime. The common language runtime provides the basic set of mechanisms for executing .NET programs regardless of their language of origin: translation to machine code (judiciously incremental translation, or "jitting"), loading, security mechanisms, memory management (including garbage collection), version control, and interfacing with non-.NET code.

.NET provides these capabilities over a broad spectrum of hardware and software platforms, ranging from very high-end servers and Web farms to PCs, phones, PDAs, other wireless devices, and Internet appliances.

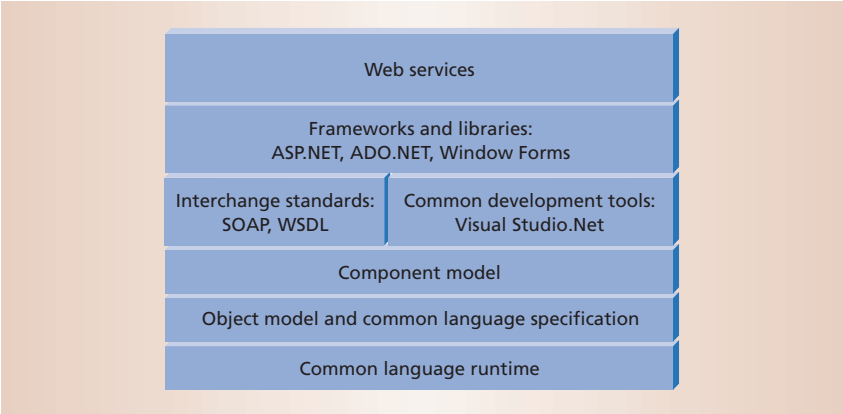

Figure 1. Elements of the .NET architecture. Microsoft's Web development platform consists of six layers from the user-visible Web services to the internal object model and common language runtime.

.NET BENEFITS

Users and developers can expect numerous benefits from the spread of .NET. For many, the most impressive component will be the ASP.NET framework. ASP.NET is not an incremental update of the ASP (active server pages) technology available on Windows. It is a new development that provides tools for building smart Web sites with extensive associated programming facilities. An ASP.NET demo at http://dotnet.eiffel.com, Interactive Software Engineering's Web site devoted to Eiffel under .NET, illustrates some of the framework's most attractive aspects.

- ASP.NET's Web controls provide a user interface similar to what is possible in today's non-Web GUI environments and far beyond what HTML offers as a default. From drag and drop to input validation, Web controls facilitate building Web pages that look like a modern non-Web GUI.
- The Web controls, handled by default on the server side, yield browser- dependent rendering—output that is automatically tailored to the browser. Some operations can be processed on the client side—for example, if the Web site visitor is using a recent version of Internet Explorer or the browser supports dynamic HTML or JavaScript. In the default case, the server handles the interaction and renders everything as plain HTML.
- ASP.NET accomplishes one of the most delicate aspects of Web request processing: maintaining a client's state. HTTP is a stateless protocol, but any realistic Web interface—a shopping basket, for example—must retain client information from one page display to the next. ASP.NET maintains session state without storing client information on the server, thereby freeing developers from using cumbersome manual techniques such as URL encoding, hidden fields, and cookies. It can accomplish this both on a single server and across Web farms.
- Through its connection to ADO.NET, which handles database connections, ASP.NET enables setting up part of a Web page to reflect the contents of a database table directly, without manual intervention. Anyone who has tried to code HTML tables displaying database contents will appreciate this feature.
- Because ASP.NET is directly tied to the .NET object model, compilers, and runtime mechanisms, the code associated with a Web page can be part of an application, however complex, benefiting from mechanisms

such as security, versioning, and jitting, from the efficiency of .NET's compiled approach, and from any .NET-supported languages. .NET's versioning facilities allow on-the-fly updates: Just replace a page with its new version, and it will be automatically compiled the next time around, without the need to stop and restart the server.

Closing the gap

Perhaps ASP.NET's greatest contribution is that it removes the distinction between IT—traditional software development—and Web development. Web-enabling an existing application can be a major effort, and so can equipping a Web page, beyond pretty pictures, with advanced processing. ASP.NET and .NET in general have the potential for merging these two disciplines.

Traditionally, companies have developed software. In recent years, they have produced Web sites, initially little more than marketing brochures, but gradually acquiring more pro- cessing elements—CGI scripts, ASP, JavaScript, and so on. Often, these sites are developed in an ad hoc fashion that does not benefit from the traditional IT experience on the other side of the house. With .NET, a Web page is a program, and a program can easily become a Web page.

As a bonus, it's easy to turn a Web interface into a traditional non-Web Windows client GUI. Visitors to the ASP.NET demo at http://dotnet.eiffel.com can download the source code to see how to do this.

WEB SERVICES

.NET offers mechanisms that make a Web page useful as both a human interface and an application program interface. Because an ASP.NET page is tied to a set of .NET assemblies (program elements), it is already an API. You can use a standard API to receive stock quotes or news updates, schedule a meeting based on your colleagues' Web calendars, or streamline your company's purchasing process.

Through its "Hailstorm" initiative, Microsoft is giving a major push to its "Passport" technology, already built-in in Windows XP, enabling computer users to define a personal profile, make it available as a set of Web services, and specify who can use what parts of it. The aim is to bring major simplifications to the many interactions we pursue with many different companies.

Microsoft, IBM, and other companies developed the SOAP XML-based format, which exports .NET objects to the world at large. Microsoft and IBM are also developing the Web Services Description Language as a new standard for Web services. Sun recently endorsed both SOAP—on which the company had previously sent mixed signals—and WSDL. Although many questions remain about this evolving aspect of .NET technology, there is a growing prospect of achieving true standards, which would provide some of the Internet's biggest potential benefits.

ASP.NET removes the distinction between traditional software development and Web development.

SECURITY

Microsoft's .NET marketing emphasizes the Web services part of the technology. But there is far more to .NET, in particular a set of mechanisms intended for software developers rather than users.

.NET's security policy—a benefit to both users and developers—is a systematic attempt to shed the image of Windows as having a poor security record. It combines four major techniques:

- *Type verification.* When the site administrator turns verification on, all .NET code is subject to verification that it obeys the object model's type system rules. For example, if you assign an expression to a variable, the types must conform as determined by the inheritance hierarchy. This precludes objects pretending to be something other than what they are and rules out a whole class of security violations.
- *Origin verification.* Any .NET assembly can and usually should be signed using 128-bit public key cryptography, which prevents impersonating another software source.
- *A fine-grained permission mechanism.* Each assembly can specify the exact permissions that it requires its callers to have: file read, file read and write, DNS access, and others, including new programmer-defined permissions. In addition, the "stack walk" mechanism ensures that if you require a permission, you enforce it not only on direct callers but also on their direct or indirect callers—so that if A doesn't have permission to call C, it can't circumvent this restriction by calling B, who does have the permission.
- *A notion of "principal."* Software elements can assume various roles during their lifetime, with each role giving access to specific security levels. This notion also includes both predefined variants and programmer-definable ones.

VERSIONING

For developers, rather than end users—although they will indirectly benefit—.NET provides a simple but strong mechanism that lets applications specify precisely what versions they can accept for the modules they use.

Thus far, Windows has offered two extreme, unsatisfactory versioning policies. With dynamic link libraries (DLLs), any version can, in principle, replace any other. This leads to subtle incompatibilities and "DLL hell": An application installs a new DLL version that replaces the previous one, with the sudden and mysterious side effect that some other application doesn't work any more. The other extreme is COM, which avoids this situation, but at the expense of considering any update as a new, incompatible version, which is not flexible enough.

The .NET versioning model defines a standard version numbering policy and lets you specify what is acceptable for the assembly you need: a specific version number ("I want 4.5.2.1 and nothing else will do"), a certain range of versions ("4.5.whatever.whatever"), or the last one that worked. This should satisfy the basic versioning needs of many applications.

COMPONENT MODEL

.NET also offers developers a new component model directly based on OO concepts. By removing the distinction between a program element and a software component, it provides significant benefits over technologies such as Corba and COM. Because an assembly provides a well-defined set of interfaces, other assemblies can use it directly.

To turn a software element into a reusable component, Corba and COM require writing an interface description in a special Interface Description Language (IDL). .NET gets rid of IDL: You can use a .NET assembly directly as a component without any further wrapping because it is already equipped with the necessary information.

The .NET component model is considerably simpler than COM, the previous standard for component-based development on Microsoft platforms. It does away not only with IDL but much of COM's historical baggage, from the HRESULT special type to low-level operations that keep track of references such as AddRef and Release, replacing it with garbage collection. Developers will appreciate this departure from COM, although in the short term the two models must coexist; COM Interop, an interoperability mechanism, should ease the transition.

SELF-DOCUMENTING COMPONENTS

The trick in removing IDL is to use interface information that is already present in the source code, at least in an OO language, where the program text contains the list of classes in an assembly, the list of each class's features (routines/methods and attributes/ field), and essential information for each feature name, number and types of arguments, whether it's a procedure or a function, and type of result, if any.

Compilers for .NET-supported languages retain this information as metadata—the idea of producing self-documenting components—in the generated code. Metadata is not limited to predefined information. You can use custom attributes to specify essentially any information for inclusion in the metadata and keep it with the component after compilation.

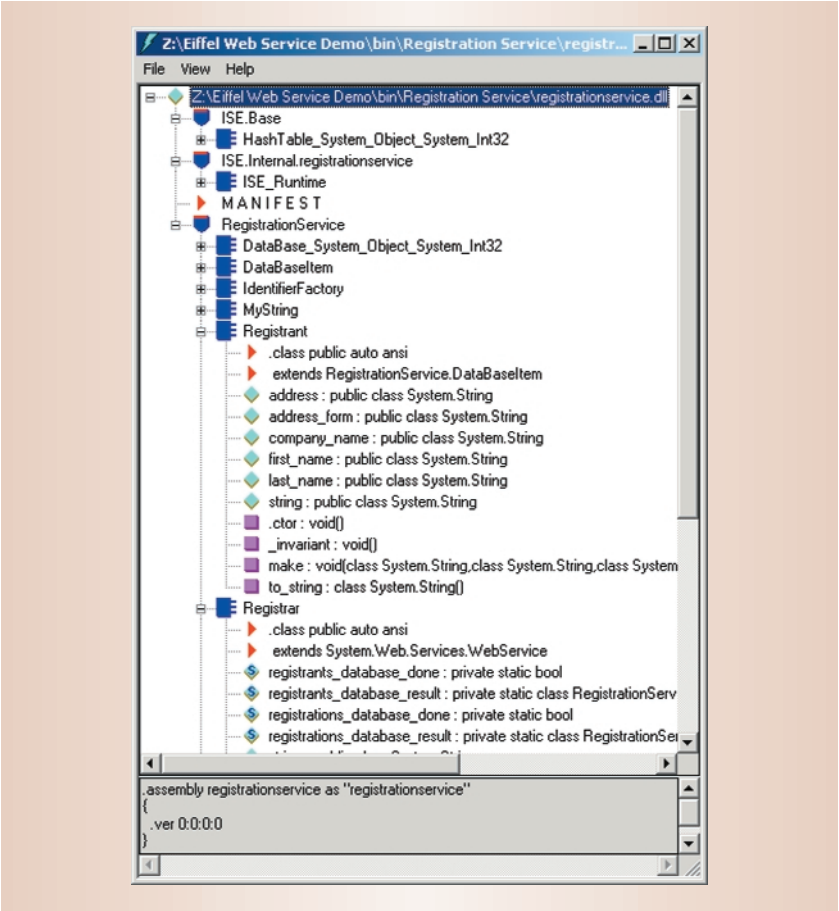

Figure 2. Examining an assembly originating in an Eiffel application with ILDASM, a graphical tool that analyzes metadata.

Metadata is accessible in many ways. ILDASM (intermediate language disassembler) is a graphical tool that lets you visually examine an assembly's interface to obtain information from the metadata. Figure 2 shows an ILDASM screen for an assembly originating in an Eiffel application. In addition, any program can access the metadata via the Reflection library. Because the metadata is also available in XML format, any application, whether it is part of .NET or not, can obtain information about components

One example of a metadata application is our "contract wizard," which programmers can use to equip assemblies compiled from languages other than Eiffel with Eiffel-like contracts. The programmer uses the wizard to explore the classes and their features, interactively deciding to add contract elements—preconditions, postconditions, class invariants. The wizard uses this input to generate a proxy assembly that implements the contracts and call the noncontracted original component. It is not clear how we could have produced such a tool without metadata.

LANGUAGE INTEROPERABILITY

Many companies have a significant software investment in different programming languages. .NET's language interoperability is intended to avoid losing that investment. The level of interoperability goes beyond what has been available in previous efforts. For example, modules written in different languages can both call each other and—in the case of classes—inherit from each other, across languages. Debugging ses-

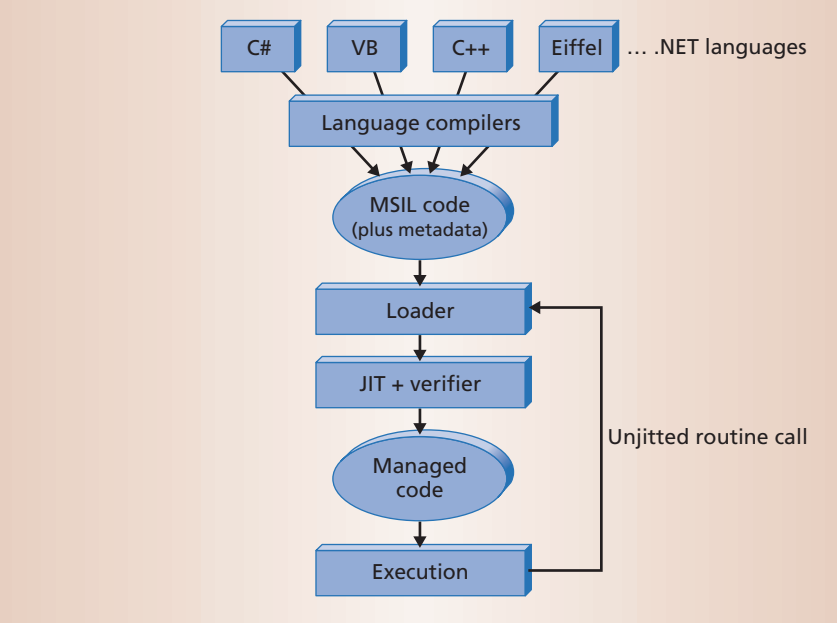

Figure 3. The .NET program execution model. The language compiler produces MSIL code that is translated further to native code.

sions in Visual Studio.Net easily cross language borders. This ability to mix languages without incurring costs for special wrappers or IDL makes it possible for developers to choose the best language for each part of an application, based on the expectation that it will smoothly and automatically interface with the other parts.

The object model

The object model and the common language specification (CLS) are the basis of .NET's language interoperability. The object model is a set of concepts—similar to an OO language, but without the syntax or anything governing the programs' external appearance and presentation—that defines the type system: what is a class, what is an object, how classes can inherit from one another, and so on.

Shortcomings. Some of its choices, obviously influenced by Java, are regrettable. In particular, there is a clear-cut distinction between classes—fully implemented abstractions—and interfaces, which are specified only, with no actual routines or attributes. Some of object technology's major benefits come from the ability to cover the full spectrum from abstract specification and design to the most detailed and technical aspects of implementation through a single notion of class, used as abstractly or concretely as needed at any particular stage.

Also Java-like is the restriction of multiple inheritance to interfaces. The model currently does not support *genericity,* or type-parameterized classes, but there are plans to include a generic mechanism in a later release.

The .NET object model provides a coherent set of base concepts, based on a strongly typed OO policy. All language compilers generate an intermediate code, MSIL (Microsoft Intermediate Language), a machine language for an abstract stack-oriented, OO machine that knows about objects, inheritance, methods, dynamic binding, and other OO concepts. When a compiler generates MSIL and associated metadata, it produces a .NET assembly that can potentially interact with any other assembly.

Interface rules

Language interoperability comes at a price: the CLS compatibility constraints. Fortunately, those constraints only affect the modules that need to interact with other languages, typically a small subset. All other modules, within a particular language, need not concern themselves with the CLS.

CLS compliance comes at three levels:

- *Supplier* (oddly called "framework"): Others can consume what you produce.
- *Consumer:* Your modules can be clients of others' producer-compliant modules.
- *Extender:* Your classes can inherit from classes in other languages.

Typically, a .NET language compiler provides the option of flagging each module as compliant or noncompliant at the desired level.

EXECUTION MODEL

.NET's common language runtime provides a basic execution schema, shown in Figure 3. The language compiler produces MSIL code, which is not interpreted but is translated again, or jitted, to native code. Thus, the target platform will execute only native code.

Jitting

With the default jitter, individual routines are jitted on demand, as required by the execution, and then kept in their jitted form. With EconoJit, a variant intended for small-memory-footprint devices, you can specify a maximum amount of space and apply a caching policy to remove a jitted routine and make room for a new one. Another option is PreJit, which will perform the translation once for the entire application.

Managed code

Code meant for execution by the .NET runtime is known as "managed"—it benefits from all the runtime's facilities such as garbage collection, exception handling, and security. Garbage collection raises issues for C++, one of the languages .NET offers. C++ has a lax type system that allows casts (conversions) between almost arbitrary types, but it contradicts the requirements of safe GC. Indeed, classic C++ cannot generate managed code on .NET. For the corresponding classes, you have to use "Managed C++", a new variant of the language that imposes

strong restrictions on type mixes and is in fact very close to C#.

Microsoft is sending a strong signal to C++ developers indicating that full compatibility with C, the defining property of classic C++, no longer meets software development demands in the Internet age. The combination of managed and unmanaged classes offers a transition path.

BEYOND WINDOWS

Only some .NET facilities are Windows-specific. The Windows Forms framework is intended to replace the Windows graphical API and the graphical part of Microsoft foundation classes (for C++), and ASP.NET is implemented on top of Microsoft's IIS Web server. Most of the rest of .NET could, in principle, be implemented on top of Linux, Solaris, or other systems.

Will we see .NET on non-Microsoft operating systems? Although you currently need Windows to use .NET, late last year Microsoft submitted key parts of the technology for standardization to ECMA, an international standardization body. The elements still under discussion include the common language runtime, CLS, MSIL, C#, and more than 1,000 components from the basic libraries. Microsoft is actively pushing to complete the work much faster than the usual time it takes for such standards processes.

The recently announced MONO effort (http://www.go-mono.net) is intended to develop an open-source implementation of .NET, based on the ECMA specifications and suitable for running on platforms such as Linux. Such efforts indicate that the transformation of .NET into a multiplatform development environment may happen faster than expected.

An important but underhyped development, .NET has benefited from an investment in several billions of dollars in research. Microsoft's competitors haven't missed the message, as indicated by several recent announcements such as Sun's Open Net Environment. No one in this industry can be guaranteed to hold the lead for long, but Microsoft has the potential to stage a major coup. Whatever happens, .NET will affect nearly everyone involved in any kind of enterprise or Web development.

Bertrand Meyer, *CTO of Interactive Software Engineering, Santa Barbara, Calif. (http://www.dotnetexperts.com), and an adjunct professor at Monash University, Melbourne, is the author of* The .NET Video Course *(Prentice Hall, 2001). Contact him at Bertrand_Meyer@eiffel.com.*

Meyer gratefully acknowledges the assistance of Raphaël Simon of ISE in preparing this article.

Editor Info: Michael J. Lutz, Rochester Institute of Technology, Department of Computer Science, 102 Lomb Memorial Drive, Rochester NY 14623; (phone) +1 716 465-2909; (fax) +1 716 475 7100; mjl@cs.rit.edu