Microsoft .NET vs. J2EE: How Do They Stack Up?

by Jim Farley, author of Java Distributed Computing and co-author of Java Enterprise in a Nutshell.

November 8, 2000 -- Even if you don't write code dedicated to Microsoft platforms, you have probably heard by now about Microsoft .NET, Microsoft's latest volley in their campaign against all things non-Windows. If you've read the media spin from Microsoft, or browsed through the scant technical material available on the MSDN site, or even if you attended the Microsoft Professional Developers' Conference (where the .NET platform was officially "launched"), you're probably still left with at least two big questions:

- What exactly *is* the .NET platform?
- How does the .NET architecture measure up against J2EE?

What exactly *is* the .NET platform [and] how does the .NET architecture measure up against J2EE?

And, if you think more long-term, you might have a third question rattling around your head:

- What can we learn from the .NET architecture about pushing the envelope of enterprise software development?

The .NET framework is at a very early stage in its lifecycle, and deep details are still being eked out by the Microsoft .NET team. But we can, nevertheless, get fairly decent answers to these questions from the information that's already out there.

What is it?

Java runs on any platform with a Java VM. C# only runs in

Current ruminations about .NET in various forums are reminiscent of the fable of the three blind men attempting to identify an elephant: It's perceived as very different things, depending on your perspective. Some see .NET as Microsoft's next-generation Visual Studio development environment. Some see it as yet another new programming language (C#). Some see it as a new data-exchange and messaging framework, based on XML and SOAP. In reality, .NET wants to be all of these things, and a bit more.

First, let's get some concrete details. Here's one cut at an itemized list of the technical

Windows for the foreseeable future.

components making up the .NET platform:

- **C#**, a "new" language for writing classes and components, that integrates elements of C, C++, and Java, and adds additional features, like metadata tags, related to component development.
- A "**common language runtime**", which runs bytecodes in an Internal Language (IL) format. Code and objects written in one language can, ostensibly, be compiled into the IL runtime, once an IL compiler is developed for the language.

.NET and J2EE offer pretty much the same laundry list of features, albeit in different ways.

- A set of **base components**, accessible from the common language runtime, that provide various functions (networking, containers, etc.).
- **ASP+**, a new version of ASP that supports compilation of ASPs into the common language runtime (and therefore writing ASP scripts using any language with an IL binding).
- **Win Forms and Web Forms**, new UI component frameworks accessible from Visual Studio.
- **ADO+**, a new generation of ADO data access components that use XML and SOAP for data interchange.

By allowing cross-language component interactions, .NET is enfranchising Perl, Eiffel, Cobol, and other programmers.

How do .NET and J2EE compare?

As you can see, the .NET platform has an array of technologies under its umbrella. Microsoft is ostensibly presenting these as alternatives to other existing platforms, like J2EE and CORBA, in order to attract developers to the Windows platform. But how do the comparisons play out item-by-item? One way to lay out the alternatives between .NET and J2EE is shown in the following table:

Microsoft.NET	J2EE	Key differentiators
C# programming language	Java programming language	C# and Java both derive from C and C++. Most significant features (e.g., garbage collection, hierarchical namespaces) are present in both. C# borrows some of the component concepts from JavaBeans (properties/attributes, events, etc.), adds some of its own (like metadata tags), but incorporates these features into the syntax differently. Java runs on any platform with a Java VM. C# only runs in Windows for the foreseeable future. C# is implicitly tied into the IL common language runtime (see below), and is run as just-in-time (JIT) compiled bytecodes or compiled entirely into native code. Java code runs as Java Virtual Machine (VT) bytecodes that are either interpreted in the VM or JIT compiled, or can be compiled entirely into native code.

.NET is a good thing for those of you committed to Microsoft architectures.

.NET will undoubtedly become the default development environment for Microsoft platforms.	.NET common components (aka the ".NET Framework SDK")	Java core API	High-level .NET components will include support for distributed access using XML and SOAP (see ADO+ below).
	Active Server Pages+ (ASP+)	Java ServerPages (JSP)	ASP+ will use Visual Basic, C#, and possibly other languages for code snippets. All get compiled into native code through the common language runtime (as opposed to being interpreted each time, like ASPs). JSPs use Java code (snippets, or JavaBean references), compiled into Java bytecodes (either on-demand or batch-compiled, depending on the JSP implementation).
However, several of the goals of the .NET platform are fairly lofty and not at all guaranteed to fly, at least not in the short term.	IL Common Language Runtime	Java Virtual Machine and CORBA IDL and ORB	.NET common language runtime allows code in multiple languages to use a shared set of components, on Windows. Underlies nearly all of .NET framework (common components, ASP+, etc.). Java's Virtual Machine spec allows Java bytecodes to run on any platform with a compliant JVM. CORBA allows code in multiple languages to use a shared set of objects, on any platform with an ORB available. Not nearly as tightly integrated into J2EE framework.
It would be easy to dismiss .NET as more Microsoft marketing-ware and continue on your merry way. But don't.	Win Forms and Web Forms	Java Swing	Similar web components (e.g., based on JSP) not available in Java standard platform, some proprietary components available through Java IDEs, etc. Win Forms and Web Forms RAD development supported through the MS Visual Studio IDE - no other IDE support announced at this writing. Swing support available in many Java IDEs and tools.
	ADO+ and SOAP-based Web Services	JDBC, EJB, JMS and Java XML Libraries (XML4J, JAXP)	ADO+ is built on the premise of XML data interchange (between remote data objects and layers of multi-tier apps) on top of HTTP (AKA, SOAP). .NET's web services in general assume SOAP messaging models. EJB, JDBC, etc. leave the data interchange protocol at the developer's discretion, and operate on top of either HTTP, RMI/JRMP or IIOP.

[Microsoft is] fighting Java and open source initiatives on their own terms, putting

The comparisons in this table only scratch the surface. Here's an executive summary of .NET vs. J2EE:

Features: .NET and J2EE offer pretty much the same laundry of list of features, albeit in different ways.

Portability: The .NET core works on Windows only but theoretically supports

their own spin on "open" and attempting to directly address the needs of developers.

development in many languages (once sub-/supersets of these languages have been defined and IL compilers have been created for them). Also, Net's SOAP capabilities will allow components on other platforms to exchange data messages with .NET components. While a few of the elements in .NET, such as SOAP and its discovery and lookup protocols, are provided as public specifications, the core components of the framework (IL runtime environment, ASP+ internals, Win Forms and Web Forms component "contracts", etc.) are kept by Microsoft, and Microsoft will be the only provider of complete .NET development and runtime environments. There has already been some pressure by the development community for Microsoft to open up these specifications, but this would be counter to Microsoft's standard practices.

If you consider yourself an evangelist for Java or open source platforms, then the nature of the war is changing. Be prepared.

Read more on the .NET platform in this in-depth interview by O'Reilly Windows editor John Osborn:

Deep Inside C#: An Interview with Microsoft chief architect Anders Hejlsberg--John gets to the bottom of not only Microsoft's detailed plans for the C# programming language but also the .Net framework.

J2EE, on the other hand, works on any platform with a compliant Java VM and a compliant set of required platform services (EJB container, JMS service, etc., etc.). All of the specifications that define the J2EE platform are published and reviewed publicly, and numerous vendors offer compliant products and development environments. But J2EE is a single-language platform. Calls from/to objects in other languages are possible through CORBA, but CORBA support is not a ubiquitous part of the platform.

Microsoft has put a stake in the ground with SOAP, and they're pushing hard to put something understandable and useful in the hands of developers. J2EE proponents need to do the same with their platform.

The Bigger Picture

These last points highlight some of the key differentiators between .NET and J2EE, and point towards Microsoft's real play here. Microsoft is doing two very notable things with .NET: It is opening up a channel to developers in other programming languages, and it is opening up a channel to non-.NET components by integrating XML and SOAP into their messaging scheme.

By allowing cross-language component interactions, .NET is enfranchising Perl, Eiffel, Cobol, and other programmers by allowing them to play in the Microsoft sandbox. Devotees of these languages are particularly amenable to gestures like this, since for the most part they have felt somewhat disenfranchised and marginalized in the Microsoft/Sun/Open Source wars. And by using XML and SOAP in their component messaging layer, Microsoft is bolstering their diplomatic face and adding an element of openness to their platform, providing ammunition against claims of proprietary behavior.

What's the correct response?

For Microsoft developers:

.NET is a good thing for those of you committed to Microsoft architectures. ASP+ is

better than ASP, ADO+ is better, but different, than ADO and DCOM, C# is better than C and C++. The initial version of .NET won't be real until sometime in 2001, so you have some time to prepare, but this will undoubtedly become the default development environment for Microsoft platforms. And if you're developing within the Microsoft development framework now, you will undoubtedly benefit from adopting elements of the .NET framework into your architectures.

However, several of the goals of the .NET platform are fairly lofty and not at all guaranteed to fly, at least not in the short term. The IL common language runtime, for example, has some fairly significant hurdles to overcome before it has any real payoff for developers. Each language that wants to integrate with the component runtime has to define a subset/superset of the language that maps cleanly into and out of the IL runtime, and has to define constructs that provide the component metadata that IL requires. Then compilers (x-to-IL and IL-to-x) will have to be developed to both compile language structures (objects, components, etc.) into IL component bytecodes, and also generate language-specific interfaces to existing IL components.

There is some historical precedence here. Numerous bridges from non-Java languages to the Java VM have been developed, such as **JPython**, **PERCobol**, **the Tcl/Java project**, and interestingly enough, **Bertrand Meyer** and some other Eiffel folks put together an **Eiffel-to-JavaVM** system a few years back. With the possible exception of JPython, these tools have not been widely adopted, even within their respective language communities, even though they seem to offer a way to write code for the Java environment (albeit not the entire J2EE framework) using your favorite language. Why this lack of enthusiasm? I believe it's because people are hesitant to take on the headaches of adding yet another translation step from their development language to the target framework. If the Java environment is the goal, people will generally choose to learn Java. I predict that the same will be true of .NET: People will generally choose to learn C# and write .NET components in that language.

Another caution: Beware of performance issues with .NET's SOAP-based distributed communications. SOAP essentially means XML over HTTP. HTTP is not a high-performance data protocol, and XML implies an XML parsing layer, which implies more compute overhead. The combination of both could significantly reduce transaction rates relative to alternative messaging/communications channels. XML is a very rich, robust metalanguage for messaging, and HTTP is very portable and avoids many firewall issues. But if transaction rates are a priority for you, keep your options open.

For the Java and Open Source communities:

It would be easy to dismiss .NET as more Microsoft marketing-ware and continue on your merry way. But don't. .NET is a sign of a subtle but significant shift in Microsoft's strategy to evangelize their platforms. They have been fighting alternative frameworks and platforms at the management level pretty well, touting the usual questionable "statistics" about cost of ownership and seamless integration. Now they are fighting Java and open source initiatives on their own terms, putting their own spin on "open" and attempting to directly address the needs of developers, two things that they have been faulted for not doing very well in the past. If you consider yourself an evangelist for Java or open source platforms, then the nature of the war is changing. Be prepared.

Also, Microsoft's IL runtime has at least one notable, if improbable, goal: eliminate the programming language as a barrier to entry to the framework. Java eliminates the platform barrier (within limits, of course: You can't make up for missing hardware resources with software, for example), but in order to work in J2EE, you have to work in Java. .NET wants to let you use the language of your choice to build .NET applications. This is admirable, though there are big questions as to whether and when the IL approach in .NET will actually become broadly useful (see above). Regardless, this points to a weakness in the single-language J2EE approach. The importance of this weakness is questionable, but it exists nonetheless, and deserves some consideration by the Java community. If this is really desired by developers, then maybe the efforts in Java bytecode generators for non- Java languages should be organized and consolidated.

Focusing on J2EE, there are a few issues that should be addressed immediately in order to bolster the advantages of that platform compared to what .NET is shooting for. First, XML support needs to be integrated seamlessly into the framework. I'm not talking about bolting an XML SAX/DOM parser to the set of standard services, or extending the use of XML in configuration files. XML messaging and manipulation need to be there, ready to use. Admittedly, you can use XML payloads on top of JMS messaging, but the platform doesn't facilitate this at all. The XML space is a cluttered mess of standards, de facto standards, APIs and DTDs, which is to be expected when you're dealing with a meta-language.

But Microsoft has put a stake in the ground with SOAP, and they're pushing hard to put something understandable and useful in the hands of developers. J2EE proponents need to do the same with their platform. One possibility that comes to mind is to add an XML messaging "provider" layer on top of JMS, along the lines of the pattern followed by **Java Naming and Directory Interface**, or JNDI, with LDAP, NIS, COS Naming, etc. This in combination with a standard SOAP/BizTalk provider, an ebXML provider, etc. would be an impressive statement.

Clarifications and Corrections

Since the publication of this article in August 2000, 40 readers have responded with their own thoughts about .Net vs. J2EE. (You can find the reader responses on the O'Reilly Web site.) Jim Farley, the author of this article, has sifted through those comments, as well as email he's received, and added the following clarifications and corrections.

Clarifications

The description of C#'s compilation features vs. those of Java seems to have confused some readers. To put it another way: C# code always runs natively. Java code typically runs as interpreted bytecodes, and can run natively. C# is either compiled entirely to native code, or it is compiled into the common language runtime bytecodes and then just-in-time compiled to native code during execution. Java code, on the other hand, typically runs as runtime-interpreted bytecodes (from which its cross-platform abilities spring), and can also run in a just-in-time compiled context. Some Java native-code compilers also exist (Jove, BulletTrain, JET, etc.).

As a side note, Microsoft claims that the default interpretive mode of Java is a liability, in that bytecodes designed for a virtual machine do not lend themselves as well to native optimization. I haven't seen any hard data to prove or disprove that claim, either generally (bytecodes vs. native-compiled languages) or specifically (Java vs. C#).

Several readers, in response to the call to include XML support in J2EE, mentioned the fact that J2EE 1.3 (currently in public draft) requires that any J2EE-compliant product must include Java XML SAX and DOM parsers. But this is just "bolting an XML SAX/DOM parser" to J2EE, as I mentioned. I was calling for it to be taken a step farther, to incorporate XML support directly in the J2EE support APIs. Ideally, J2EE-based components and services would have XML support (for messaging, interface description exports, etc.) automatically built-in, to some extent.

Corrections

I state in the article that C# "borrows some of the component concepts from JavaBeans." This statement can't be proven, and, as several readers pointed out, it's more likely that Microsoft based the component functionality of C# more on their own COM and VB models, with influences from other pre-existing component models.

Post your responses to this article,
or read what others have to say on the O'Reilly Web site!

Jim Farley is a technology consultant, manager and author. His recent activites have included managing an eBusiness program for **GE Research and Development** , and heading up the engineering group at the **Harvard Business School**. He is the co-author of **Java Enterprise in a Nutshell**, and author of **Java Distributed Computing**, both published by O'Reilly.

Further Reading

Developer Madhu Siddalingaiah shares his thoughts on .NET in the **Perspective on Technology** series.