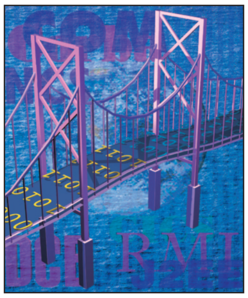

# The Language Divide

**Steve Vinoski** • *IONA Technologies*

One of my favorite movies is 1980's *The Blues Brothers* — the humorous story of two shady brothers trying to get their band back together to raise money to save the orphanage where they grew up. In one classic scene, the group pretends to be another band, called "The Good Ol' Boys," so that they can play at a music establishment they find while driving around looking for work. When brother Elwood asks the woman behind the bar what kind of music they usually feature, she responds, "Oh, we got both kinds. We got country and western!"

I often get the impression that a similarly myopic view pervades the enterprise middleware space as far as programming languages are concerned. If I were to ask a group of typical developers of middleware-based enterprise applications what programming languages they used, I wouldn't be surprised to hear, "Oh, we use both kinds — Java and C++!"

When it comes to Web development, scripting languages such as Perl (www.perl.org), Python (www.python.org), Ruby (www.ruby-lang.org), PHP (www.php.net), and Javascript seem to rule the roost. Java has some presence there as well, of course, in the form of servlets, JavaServer Pages and JavaServer Faces, and frameworks such as Struts and Tapestry. Yet, the converse doesn't seem to be true: the intersection between enterprise middleware and scripting languages appears practically empty.

## Looking Back

A significant predecessor of today's middleware was the transaction processing (TP) monitor. The middleware-level TP monitor evolved from early systems such as SABRE (jointly developed by American Airlines and IBM to handle airline reservations) and IBM's Customer Information Control System (CICS) and Information Management System (IMS). These early systems, developed in the 1960s and 1970s, were more operating system than middleware.

As the 1970s gave way to the 1980s, mainframes came under increasing market pressure from vendors of minicomputers and Unix workstation servers who found that customer scenarios involving mainframe displacement often required them to provide TP monitor functionality. Advances in operating systems and networking technologies allowed those vendors to build their TP monitors as true middleware, layered above the OS rather than as part of it.

Such TP monitor middleware systems were generally written in C or in proprietary languages specific to the underlying OS, with critical sections coded in assembly language. This allowed for the high performance and transaction rates customers expected from these systems. Furthermore, these systems required access to OS networking interfaces, persistent storage interfaces, and process creation and monitoring facilities that were typically accessible only through C or proprietary language APIs.

Around the same time, significant shifts occurred in software development in general. As software systems grew larger and lived longer, worries over maintenance and quality issues grew. Concerns in these areas fed directly into the creation of software development approaches such as structured programming and object-oriented programming (OOP). The benefits of modularity and data hiding were cornerstones of both.

As software development methodologies helped turn the practice from sheer wizardry into more of a science-and-engineering endeavor, instructors in the late 1970s and the 1980s pushed students to use languages that directly supported structured programming or OOP. For practical use, such languages also had to compile into good code that ran well on the era's hardware. Imperative languages such as Pascal, C, and C++ fit these requirements perfectly, letting developers focus on stepwise refinement of functions and using those functions through programming conventions to hide data. C++ also added powerful data encapsulation via objects along with compiler enforcement of data visibility. Programmers largely avoided languages that treated code and data as equals, or that didn't run well on the processors of the day.

The 1980s also saw R&D increasingly steered toward systems based on remote procedure calls (RPCs) and distributed objects. Such research was often based around specialized programming languages. In fact, "language" might be the wrong term here, given that these research efforts typically generated products, such as Emerald (perhaps the most famous and influential),[1] which served as complete environments that supported object descriptions, object implementations, communication runtimes, and directory services. Typically, such systems were implemented in C under the covers.

By the late 1980s, all these influences were starting to make their mark on the burgeoning middleware industry. As a rule, serious middleware of the day — even for distributed object systems — was written in C using data-hiding techniques and object-based approaches to ensure maintainability and code quality. C++ systems were just beginning to appear by the end of the decade.

The 1990s saw significant growth in middleware and middleware-based applications, as the research and early prototypes of the 1980s finally turned into commercially viable software that was ready for production deployment. Unsurprisingly, these systems were largely based on distributed object concepts and were implemented in C++ or C. As the 1990s wore on, message-oriented middleware — another descendant from the mainframe world — gained popularity (again, typically implemented in C++ or C). Java entered the picture in the mid-1990s and quickly became a popular implementation language for middleware systems before the decade was out.

One significant area of focus for 1990s middleware was performance. Building layers of abstraction as middleware helped isolate applications from variations and limitations in the underlying OS, but it also added potentially significant runtime overhead. Developers expended much effort in identifying the sources of this overhead and devising patterns for eliminating them.[2,3] In general, middleware's heritage is such that developers simply expect it to perform well, and so it's not inconceivable that middleware wouldn't have succeeded without the performance focus. Unfortunately, however, other important areas, such as system features and API comprehensiveness, sometimes remain underdeveloped and overlooked as a result.[4]

**By and large, the enterprise middleware space ignored approaches that didn't follow the unwritten rules.**

As we entered the 21st century, the unwritten rules for building middleware systems were well established: you wrote the system in C++ or Java, and you focused heavily on performance. By and large, the enterprise middleware space ignored approaches that didn't follow these rules.

## Across the Divide

Dynamic languages and functional programming languages weren't standing still while all this was going on. LISP, the granddaddy of them all, served as the foundation for research into artificial intelligence (AI) for several decades. Still, developers have long viewed this incredibly simple yet powerful language as a poor performer because it's an interpreted language. Part of the group I worked in at Texas Instruments in the mid-1980s developed the "LISP chip" — a 32-bit microprocessor designed specifically to enhance LISP applications' performance. First released in 1987, the chip was enormous for its time, packing more than 550,000 transistors into a square centimeter. Unfortunately, even such specialized hardware couldn't help LISP shake its reputation for poor performance.

The late 1980s and the 1990s saw the development of interpreted scripting languages such as Perl, the Tool Command Language (Tcl; www.tcl.tk), and Python. Perl grew out of the system administration world, offering the capabilities of various Unix tools in a single, highly portable language. The University of California, Berkeley's John Ousterhout developed Tcl to provide an embeddable command language for software tools. Python came along a couple of years later, offering many of the same capabilities as Perl but with a much cleaner syntax and style.

Such languages might have been forever relegated to niches if the World Wide Web hadn't come along. The first Web servers simply returned static files for each request, but it wasn't long before Web site developers realized that they could use small programs executed by Web servers to dynamically generate content. The common gateway interface (CGI) evolved as a standard way to let Web servers launch programs to serve requests. Because handling such requests typically involved file access and substantial string manipulation to create HTML responses, scripting languages were a natural fit for solving the problem. Not only did such languages provide highly portable and easy-to-use file and database access and string-handling facilities, they also allowed for rapid development approaches that fit well with Web site developers' "need to tweak." These languages let programmers avoid compiling and redeploying programs just to make small changes to site appearance and content. With the Web's incredible growth in the 1990s, this constant tweaking was vital to enabling Web sites to change frequently to always appear fresh. Scripting languages' popularity grew immensely as a result.

## The Same, but Different

Interestingly, much similarity exists between server-side invocations in object-oriented or service-oriented middleware and Web server invocations via CGI or similar approaches. The middleware or Web server receives a request, dispatches it to whatever entity is registered to handle it, and then returns any generated response. Why, then, do these two camps remain so far apart?

I can think of at least two reasons: data and performance. Web servers deal mostly with textual data in the form of HTML or, increasingly, XML. Middleware servers, on the other hand, have traditionally operated on binary data representing programming language data structures, evolving from early RPC systems that marshaled requests and responses consisting of C structs directly to and from memory. Performance rears its head here because representing information as text results in far more data than representing the same information in binary format. Nevertheless, with the advent of SOAP and Web services, middleware systems now also deal heavily with XML messages.

Of greater concern in this picture regarding performance is the CGI approach's process-startup overhead. Starting a new OS process to handle each incoming request immediately raises red flags for any respectable middleware developer. Not surprisingly, these issues also quickly became problems in the Web world as sites started scaling up. Developers thus devised new approaches, such as building scripting language interpreters into plug-ins that could be hosted directly in-process by the Web server. These approaches not only avoided the process-per-request startup overhead but also brought even greater resemblance to request handling in the middleware and Web worlds.

## The Java Bridge

Sitting somewhere in the middle of all this is Java. For some reason, it doesn't suffer from the same performance stigma as previous interpreted languages. Maybe that's because hardware was fast enough by the time it came along to make Java applications run well. Perhaps Sun's massive marketing push in the '90s made the difference, or maybe Java's ability to work well in the Web context helped it ride the Web's growth. Whatever the reason, Java's popularity has helped restore respectability to interpreted programming languages in general.

Java sits in the middle of this picture more than just figuratively. The Java servlet architecture, for example, has its feet planted firmly in both the middleware and Web camps. Servlets handle HTTP requests, thus integrating cleanly with the Web, and yet their Java implementations allow them to easily integrate with other back-end Java and non-Java middleware and frameworks. Not surprisingly, servlets' utility and flexibility have led developers to often cite them as the single best part of Java's foray into the enterprise middleware space.

Java, and now C#, might also provide a more direct answer to bridging the language divide between middleware and the Web. If you separate both Java and C# into two parts — the programming language and the virtual machine (VM) — it's readily apparent that other languages, including the widely popular scripting languages, can also sit on top of the VMs. Several projects have already been doing precisely that. For example, Jython (www.jython.org) is a Python implementation that runs on the Java VM, whereas IronPython (www.ironpython.com) is a Python implementation for the Common Language Runtime that sits under C#. Furthermore, Java Specification Request (JSR) 223 has opened the door to integrating other scripting languages into the Java platform (see www.jcp.org/en/jsr/detail?id=223). For example, the next release of Java, codenamed Mustang, uses JSR 223 to integrate the Mozilla Rhino Javascript interpreter (www.mozilla.org/rhino/) into the platform.

The days of C++-only or Java-only middleware systems are numbered, and dynamic scripting languages will soon become an important part of middleware development. Perhaps when asked what languages they use, middleware developers of the near future will respond, "Oh, we use both kinds — compiled and interpreted!"

### References

1. A. Black et al., "Distribution and Abstract Types in Emerald," *IEEE Trans. Software Engineering*, vol. 13, no. 1, 1987, pp. 65–76.
2. D.C. Schmidt et al., *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*, John Wiley & Sons, 2000.
3. M. Voelter, M., Kircher, and U. Zdun, *Remoting Patterns: Foundations of Enterprise, Internet, and Realtime Distributed Object Middleware*, John Wiley & Sons, 2004.
4. S. Vinoski. "The Performance Presumption," *IEEE Internet Computing*, vol. 7, no. 2, 2003, pp. 88–90.

**Steve Vinoski** is chief engineer for IONA Technologies. He's been involved in middleware for more than 17 years. Vinoski has helped develop middleware standards for the Object Management Group (OMG) and the World Wide Web Consortium (W3C). Contact him at vinoski@ieee.org.