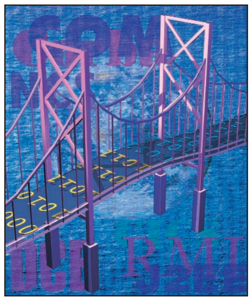

It's Just a Mapping Problem

Steve Vinoski • *IONA Technologies* • *vinoski@ieee.org*

We normally use middleware as the "glue" that connects applications and services. The diverse and heterogeneous nature of many business-computing environments — caused by continuous changes in technology, products, business organizations, and business requirements — means that the need to adapt applications to each other never ceases. In fact, all software performs adaptation at one level or another, but adaptation is middleware's *raison d'être.*

Adapting applications to each other requires mapping between the concepts, functions, or data native to each application, and identifying the abstractions under which each application can perform correctly and efficiently. Determining appropriate abstractions at any level is never easy, but doing so is critical to an application's utility and success. For example, the Web would not have succeeded if users had to place online orders not through the abstraction of filling in a browser-based form, but by writing the SQL necessary to enter transactions directly into a vendor's order database.

Interface vs. Implementation

Middleware often employs declarative languages that help users define abstractions for the services they're implementing or integrating. Interface definition languages (IDLs) are common in RPC systems, such as the Distributed Computing Environment (DCE), and in distributed object environments, such as Corba. In systems like these, a middleware developer is expected to implement — using one or more mainstream programming languages — applications that either supply or consume distributed services. Mappings relate the abstractions provided at the IDL level to features found at the programming language level.

For example, a cornerstone of Corba is its support for multiple programming languages: C, C++, Java, Cobol, PL/I, Smalltalk, and Python. The Corba standard includes mappings from IDL for each supported programming language. Because each separate mapping starts with the same IDL, the mappings all resemble one another to some degree, and yet each mapping is unique because it relies on specific idioms of the target programming language. Using a programming language's idioms lets the mapping be as natural as possible for users of that language.

I know from many years of experience with Corba that users generally dislike the language mappings for IDL. In some cases, they feel the language mapping itself is unnecessarily complicated. For example, this is definitely the case for C++ mapping.[1] In other cases, such as Java, some feel they should be able to use Java directly as their IDL, instead of being forced to use Corba's mapping. This desire is especially prevalent for users familiar with Java-specific middleware such as Java RMI or Enterprise JavaBeans (EJB), and it's even common among Corba users who program only in Java. While Corba includes a "reverse mapping" from Java to IDL (mainly to support interoperability between EJB systems and Corba systems), the IDL that results from applying the reverse mapping to Java class definitions is so abstruse that developers generally avoid it.

What's ironic is that the primary reason developers generally dislike IDL mappings is the same reason the mappings exist in the first place. The goal is to merge middleware abstractions directly into the realm of the programming language, minimizing the "impedance mismatch" between the programming language world and the middleware world. For example, mappings make request invocations on distributed objects and services appear as normal programming-language function calls, and they map distributed system exceptions into native programming language exception-handling mechanisms. In systems like DCE and Corba, in which the primary focus for developers is writing applications that supply or consume requests and replies, language mappings provide the means for programmatic access to the contents of those requests and replies. The language mapping's perceived quality is measured by the transparency with which it makes middleware artifacts appear as natural elements of the programming language and its environment. Unfortunately, the wrong transparencies can incorrectly mask distributed-computing issues, such as those related to concurrency and partial failure.[2] This occurs when multiple levels of abstraction are mixed together inappropriately, thus making them indistinguishable. Many developers find such a mixture of separate concepts and abstractions confusing as well.

Despite their flaws, IDL mappings offer a more pragmatic solution in this space than other alternatives. You might instead consider extending a programming language with new keywords or constructs that support distributed systems, for example, or writing an entire new distributed systems language from the ground up. While such approaches are often technically superior to mapping approaches, they fail from a market-adoption perspective for several reasons.

- First, language extensions and new languages are rarely standardized and are usually supported by only a single vendor. Few users are willing to bet their middleware applications on such a risky approach, given that it could easily fail in the market and leave them with applications based on unsupported technologies.
- Second, such languages are essentially "one size fits all" solutions that try to force homogeneity into a system by requiring everything to be switched over to use them. In many enterprise-computing settings, however, there are systems that simply cannot be replaced without incurring significant cost or downtime; reimplementing them in any language is thus out of the question. Even though Java, for example, is successfully employed in many circles as both a programming language and a middleware system, it's still far from being the universal glue that's applicable in all problem domains. Therefore, it only moves, rather than eliminates, the need for IDLs and their mappings.

You might argue that these approaches are not as different as I claim them to be, because an IDL also attempts to create a certain level of homogeneity within heterogeneous systems. That argument is inaccurate, however, because an IDL introduces homogeneity at the interface level, while a programming language and other "binary standards" such as IBM's System Object Model (SOM), Java, and Microsoft's COM and .NET introduce homogeneity at the implementation level. This difference is significant. For example, the "interface" that we use to drive a car has not changed much over the past 70-80 years, but implementations of the driver–car interface have advanced significantly within that same time period, including innovations such as power steering, automatic transmission, and antilock brakes. Similarly, the separation of interface from implementation provided by IDLs allows them to enable integration without forcing reimplementation of the applications being integrated.

What's ironic is that the primary reason developers generally dislike IDL mappings is the same reason the mappings exist in the first place.

Document Abstractions

Middleware systems oriented around requests and replies tend to promote strongly typed service interfaces that supply or consume strongly typed messages. This isn't surprising, given that we can trace the roots of such middleware back to the programming language level. However, it's commonly understood that the request–reply model isn't applicable to all systems, especially those in which connectivity is occasional, or in which loose coupling is required. In such systems, approaches based on asynchronous messaging are often superior.

Not surprisingly, mappings in asynchronous messaging systems differ from their request–reply counterparts. In messaging systems, applications tend to deal with untyped messages that flow in only a single direction. An application receives a message, takes action depending on its contents, and may emit one or more messages as a result. To support such applications, messaging middleware focuses on routing, queuing, and store-and-forward capabilities for messages, rather than on transparently transmitting typed requests and replies.

In message-oriented systems, mappings tend to focus at a coarser level of granularity than their request–reply counterparts. This is not surprising because the loose coupling that messaging systems promote minimizes interaction between applications, thus forcing messages to be largely self-contained. Because messages often tend to represent real-world artifacts from business processes (such as purchase orders, invoices, and shipping notices), their mappings are document-oriented, and applications that manipulate them are built around the abstraction of document processing. In such applications, XML continues to rapidly gain favor for representing these documents.

XML's use for document-oriented messaging systems provides for a wide variety of mappings. This is partly because a messaging middleware system does not need to know or care about the contents of the messages it handles unless it's performing content-based routing. Because the middleware avoids type checking message content, such checking is left to the application, which means that the application can choose whatever mapping it prefers. XML documents are not, by default, fully self-describing (contrary to what some believe), but the mechanisms by which they can be "typed" – through Document Type Definitions (DTDs) or XML schemas – are well understood. Like the mapping for the XML document itself, the choice of what kind of validation (if any) to apply to an XML document is left entirely to the application.

The number and variety of available XML mappings seems to grow almost daily. Tools that support XML mappings, such as those based on common approaches like the Document Object Model (DOM), the Simple API for XML (SAX), and XML Stylesheet Language Transformations (XSLT), are available in a variety of programming languages. In fact, the growing use of Web services and SOAP has helped push the envelope on XML parsing techniques, and helped to continually boost the performance of XML parser implementations.

Mapping Transparency

Few mappings are lossless. To illustrate, let's consider the integration of two systems, A and B. Typically, any mapping that adequately represents the abstractions of both systems – thereby allowing them to be integrated – will fail to fully represent all the capabilities of both systems. A mapping between A and B covers the intersection, not the union, of A's and B's capabilities, thus ignoring some features from each.

While it seems obvious that mapping losses are acceptable, I have witnessed numerous cases where people strive for complete mappings. Invariably, they find they can develop 70–80 percent of the mapping with relative ease, but the final 20–30 percent is quite painful, if not practically impossible. The drive for complete mappings is often rooted in a set of theoretical integration needs, rather than a set of actual integration use cases. Without practical use cases, the tendency is to try to map all the features of system A into system B, without regard for how – or even whether – those features are actually used in practice.

This issue is particularly interesting because of the current high level of interest in Web services. Numerous projects are currently mapping Web services onto various technologies, platforms, and formats, including mainframes and J2EE application servers. As I've stated in previous columns, I believe Web services are best used for creating document-oriented, loosely coupled integrations of underlying existing middleware systems. Efforts directed toward supporting that include standardizing flow languages for Web services, such as the Business Process Execution Language for Web Services (BPEL4WS, see www-106.ibm.com/developerworks/webservices/library/ws-bpel/). Other efforts, directed more at mapping existing middleware technologies directly to SOAP or Web Services Definition Language (WSDL), are not as straightforward.

For example, imagine a mapping from Corba IDL to WSDL, which lets Web service applications make use of existing Corba services. On the surface, such a mapping is problematic given that WSDL does not readily support the IDL concept of object references. It's very common for IDL interfaces to have operations that take object references as parameters or return object references as results. For example, the Corba Naming Service operation that resolves an object given its name returns an object reference, as does an object factory in any Corba implementation of the Factory pattern.[3] This is clearly a case where reaching a 100 percent mapping would be difficult, if not impossible.

A pragmatic middle ground is to apply a combination of approaches. First, rather than shooting for a 100 percent mapping, develop a standard that maps an appropriate subset of Corba IDL to WSDL. Integration projects could then use this subset of IDL to wrap any existing IDL that uses nonmappable elements. For instance, the Factory pattern is typically used to create client-specific objects that are used by a single client and then destroyed. You could implement a wrapper Corba object with a WSDL-compatible interface to serve as the Factory pattern client, thus completely shielding applications using the WSDL interface from the object reference returned by the factory object. You can apply this "half map, half wrap" approach in a wide variety of instances, especially when combined with code-generation tools that help create the implementation for you.

Ultimately, your take on mapping's place in middleware comes down to whether you accept the assertion – as I do – that enterprise computing systems tend toward diversity and heterogeneity. Those who understand and accept this assertion readily accept the need for mappings and the limitations they present. Those who continually strive to homogenize their computing systems, on the other hand, are essentially fighting a losing battle because they're swimming upstream against ever-advancing technology and ever-changing business requirements. Change is, of course, inevitable. Part of planning for change means making your middleware applications as loosely coupled and flexible as possible, and good mappings play an important part in achieving those goals.

References

1. D.C. Schmidt and S. Vinoski, "The History of the OMG C++ Mapping," *C/C++ Users J.*, Nov. 2000; www.cuj.com/experts/1811/vinoski.htm.
2. J. Waldo et al., *A Note on Distributed Computing*, tech. report SMLI TR-94-29, Sun Microsystems Laboratories, 1994; www.sunlabs.com/technical-reports/1994/abstract-29.html.
3. E. Gamma et al., *Design Patterns—Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

Steve Vinoski is vice president of platform technologies and chief architect for IONA Technologies. He is coauthor of *Advanced Corba Programming with C++* (Addison Wesley Longman, 1999). Vinoski serves as IONA's alternate representative to the W3C's Web Services Architecture working group. Contact him at vinoski@ieee.org.