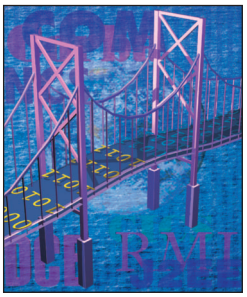
Java Business Integration



Steve Vinoski • *IONA Technologies*

For as long as I can remember, an argument has been making the rounds in distributed computing circles about how best to define interfaces for distributed elements. Some advocate the use of an interface definition language (IDL), whereas others prefer to use programming languages directly. (There are even those who prefer to avoid explicitly defining interfaces altogether, but I'll ignore that position for now.)

Not surprisingly, the Java community has traditionally sided with the programming language approach — arguing, in part, that IDLs are too far removed from the languages that developers use for implementation, and that the resulting mismatch often makes it difficult to map between the two. In practice, this means that either the IDL forces the developer to use an unnatural programming style, or the interface developer is stuck defining cumbersome interfaces to match the programming language. One of the most blatant examples I've personally experienced of this impedance mismatch is the Corba Java-to-IDL reverse mapping,[1] which is intended to let developers define Corba interfaces in pure Java and map the results into Corba IDL. Although the approach produces reasonable results for systems defined entirely in Java, the resulting IDL is strange and overly complex, creating further complications if it must subsequently be mapped to a language other than Java.

The desire to avoid such mismatches has led, in part, to the development of various approaches centered on "plain ol' Java objects" (POJOs). One such example is the Hibernate framework (www.hibernate.org), which provides developers with transparent support for storing Java objects in relational databases. Rather than relying on detailed component interfaces or complex object hierarchies characteristic of older frameworks, the POJO approach lets developers focus on producing normal Java code, while relying on tools and reflective infrastructure to transparently adapt that code for persistence, security, transactions, and other orthogonal qualities and capabilities.

The POJO approach works well if you're a Java programmer developing a pure Java system, but what happens when such purity isn't possible? The world of enterprise integration is often an "impure" place in which heterogeneous networks include everything from mainframes to blades running an amalgam of operating systems and applications. Given that Java is unlikely to be the only language in use in such settings, there's a clear need to step above individual programming languages and define services at a more abstract level that applies equally to various scripting, transformation, and programming languages.

Integration

On the flip side of the coin, traditional enterprise-integration approaches often leave much to be desired. The enterprise application integration (EAI) movement of the late 1990s, for example, was built mostly around proprietary systems with their own canonical protocols and data formats operating in centralized hub configurations. To integrate a system into the hub, administrators had to purchase an adaptor to convert its protocols and data formats to the hub's canonical protocols and data formats. For any message to get from one integrated system to another, it had to go through the hub via at least two protocol and format conversions, which resulted in low overall performance. Support for standards, if any, was often tacked on to EAI products as barely working afterthoughts. I attended a talk at a meeting a few years ago during which an industry analyst stated that, on average, EAI projects required a minimum investment of US$500,000 and 18 calendar months of effort before they realized any results. I don't know for certain whether those figures were accurate, but all in all, it's no secret that much of

the industry regards EAI as a failed integration strategy that served only to lock customers into single-vendor proprietary "money pit" systems.

Because of these shortcomings, the service-oriented architecture (SOA), based mainly on Web services, is largely replacing EAI today.[2] The definitions and distinctions here are admittedly murky, however, because it seems that EAI proponents (apparently having seen the writing on the wall) have recently redefined the term "EAI" to incorporate Web services.

Lessons Learned

Anyone intent on developing a Java-based framework for enterprise integration would do well to keep a few things in mind. One lesson is that today's Java programmers want to write Java, not stylized or otherwise "special" Java that must incorporate framework-specific artifacts. Another lesson is that standardizing a suitable integration-focused framework would be a good idea, given that the alternative is to add yet another proprietary system to a field already rife with stove-piped solutions that don't work together. The most important lesson, however, could well be that when it comes to integration, Java can't solve it all. Making it possible for solutions outside traditional Java programming to work within the framework is thus a necessity, not a nicety.

One standards effort currently targeting the business-integration space is Java Specification Request 208, entitled "Java Business Integration" (JBI).[3] Like other JSRs in the Java Community Process (JCP; www.jcp.org), JBI obviously has to work with the Java 2 platform — in this case, both the standard and enterprise editions (J2SE and J2EE, respectively). It also wisely goes beyond the Java-centric focus in most JSRs and aims to accommodate implementation alternatives outside the pure Java space.

The approach underlying JBI is a bit unusual compared to other JSRs in that it uses Web services at its core. Rather than focusing on how to build Web services using Java, it promotes an architecture that's strongly based on Web services' principles and approaches.

JBI Architecture

Fundamentally, JBI is a pluggable architecture consisting of a container and plug-ins. The container hosts plug-in components that communicate via message routers. Architecturally, components interact via an abstract service model — a messaging model that resides at a level of abstraction above any particular protocol or message-encoding format. JBI is an SOA: it treats its components as service providers and consumers.

JBI is an SOA: it treats its components as service providers and consumers.

Abstractions are quite useful, of course, but concrete instances of them are ultimately necessary to get real work done. JBI uses the Web Services Description Language (WSDL) for both abstract and concrete specification of its component messaging model. The abstract model defines message types that service providers and consumers can exchange, abstract operations comprising those messages, and service types or interfaces that group related operations together. The concrete model defines binding types to specify protocols, endpoints to specify concrete communication details for reaching services, and services that group related endpoints to specify actual service instances.

The JBI environment is a collection of components that reside within a single Java virtual machine (JVM). Services can reside outside the JBI environment and connect into it via binding components (BCs), or they can reside within subcontainers, called service engines (SEs), that live directly within the JBI environment. Services connected into a JBI environment, whether through an SE or a BC, communicate with each other via the normalized message router, which is the JBI environment's primary service.

Normalized Message Router

The NMR mediates message exchanges between service consumers and providers and can also act as a kind of discovery service to help consumers locate appropriate service providers. BCs and SEs communicate with the NMR via delivery channels, which are objects that provide methods for sending and accepting messages and for creating message-exchange factories. These factories allow service consumers to create specific message-exchange instances for use with target services.

At a minimum, JBI implementations are required to support message-exchange sequences based on four standard, well-understood WSDL message-exchange patterns (MEPs):

- *In-only*: the service consumer sends a message to the service provider, which provides no response. The exchange is complete when the provider indicates completion or error by invoking an API call to set the status of the exchange. With this MEP, providers have no way to indicate fault details to consumers.
- *Robust in-only*: the consumer sends a message to the provider, which either sets the exchange status (completing the exchange) or returns a fault to the consumer, who must then set the exchange status to complete it.
- *In-out*: the consumer sends a message to the provider, which returns an output message or a fault. The

consumer completes the exchange by setting the exchange status.
- *In-optional-out*: the consumer sends a message to the service provider, which either returns an output message or fault or completes the exchange by setting its status. If the provider returns a fault, the consumer sets the exchange status to complete it; if the provider returns an output, the consumer can complete the exchange by setting its status, or it might respond with a fault, in which case the provider must set the exchange status to complete it.

To perform these MEPs, consumers and providers first create message-exchange instances via message-exchange factories and then invoke send and receive operations on the delivery channels. Message-exchange instances carry not only messages but also metadata and state information for the given exchange.

All the messages exchanged through the NMR are *normalized* messages, which doesn't imply that all messages are converted to a canonical format. As I mentioned, EAI systems already proved that translating messages into canonical formats can negatively impact performance, scalability, and the ability to further integrate systems with other integration infrastructures. The NMR avoids these negative impacts by treating message payloads as opaque data that it simply sends along to the receiver.

The NMR is a classic framework in the sense that it expects BCs and SEs plugging into it to fulfill certain interfaces and contracts. However, the NMR imposes essentially no constraints on how BCs or SEs are internally implemented or how they interface with the elements they're integrating. The fact that developers can implement SEs as subcontainers implies that they can host virtually any type of message processors, including Extensible Stylesheet Language Transformation (XSLT) engines, Business Process Execution Language (BPEL) systems, and even POJOs — whatever can peacefully exist within the JVM hosting the JBI environment. Similarly, BCs are free to adapt virtually any kind of protocol or format employed by third-party applications or middleware systems, which provides an avenue for elements outside the JBI environment, including other JBI environment instances, to participate in JBI message exchanges.

Management

Because the JBI environment deals with pluggable components, it must also deal with lifecycle and deployment issues. For example, administrators need to be able to install components into a JBI environment and to start and stop them for maintenance or debugging purposes. Such needs are fairly standard requirements for integration middleware and distributed computing systems.

To handle these issues, the JBI environment, not surprisingly, makes use of Java Management Extensions. JMX management beans provide functions that deal with installation, deployment, monitoring, and lifecycle concerns for the JBI environment itself, for BC and SE components that plug into JBI environments, and also for subcomponents that might plug into SE subcontainers.

The JBI specification, weighing in at 244 pages, is too rich and detailed to fully cover in this column space. Instead, my intent here is to provide a feel for what the specification is about and explain some of the basics so that you'll have some familiarity when you sit down to read it yourself.

What impresses me most about the JBI specification are its balance and incorporation of lessons from the past. It avoids trying to invent things — something standards should never try to do — and instead relies on existing standards, such as WSDL and JMX, and tried-and-true distributed computing approaches for message processing to bring some order to the world of business integration. I am disappointed, however, that the specification doesn't detail any interfaces for supporting interceptors[4] or filters, which are useful for easily creating message-processing pipelines.

JBI supports the principles of SOA, in part, by being an SOA itself. It also avoids being overly prescriptive, and thus maintains the needed flexibility to extend Java integration's reach.

The specification chooses a reasonable middle ground with respect to the interface-definition issues. JBI's authors also clearly knew well enough to avoid EAI's technical failures, and they paid close attention to lessons learned from the Java Message Service (JMS) and J2EE Connector Architecture (JCA). The end result is that JBI is solid enough to support interoperable, enterprise-capable, and practical integration solutions, which ultimately is what an SOA aims to provide.

References

1. *Java to IDL Language Mapping Specification*, version 1.3, Object Management Group, Sept. 2003; www.omg.org/docs/formal/03-09-04.pdf.
2. S. Vinoski, "Integration with Web Services," *IEEE Internet Computing*, vol. 7, no. 6, 2003, pp. 75–77.
3. R. Ten-Hove and P. Walker, *Java Business Integration* (JBI) 1.0, final release, 24 May 2005; www.jcp.org/en/jsr/detail?id=208.
4. S. Vinoski, "Chain of Responsibility," *IEEE Internet Computing*, vol. 6, no. 6, 2002, pp. 80–83.

Steve Vinoski is chief engineer of product innovation for IONA Technologies. He's been involved in middleware for more than 17 years. Vinoski is coauthor of *Advanced Corba Programming with C++* (Addison-Wesley Longman, 1999), and he has helped develop middleware standards for the Object Management Group (OMG) and the World Wide Web Consortium (W3C). Contact him at vinoski@ieee.org.