J2EE Development Frameworks

Rod Johnson, Interface21

- Successful open source frameworks are better tested than in-house code.
- Frameworks usually become popular only if they have something to offer. In-house frameworks are often mandated, while a J2EE project is likely to adopt an open source framework only if it delivers clear benefits.

Despite some uncertainty, open source frameworks are increasingly important to J2EE development.

Java 2 Enterprise Edition has excelled at standardizing many important middleware concepts. For example, J2EE provides a standard interface for distributed transaction management, directory services, and messaging. In addition, Java 2 Standard Edition (J2SE), which underpins J2EE, provides a largely successful standard for Java interaction with relational databases.

However, as the "J2EE's Lack of Application Programming Support" sidebar explains, the platform has failed to deliver a satisfactory application programming model.

Sun Microsystems and the large application server vendors have traditionally responded to this problem by advocating development tools as a way to hide J2EE's complexity. However, tools for managing J2EE artifacts aren't nearly as good as tools for working with the Java language, with their sophisticated refactoring capabilities, and J2EE tool support is generally inferior to that of the Microsoft .NET platform. Many J2EE tools are themselves complex, as is the code they generate.

Many in the open source community, especially smaller vendors, have chosen the alternative of developing frameworks designed to simplify the experience of building J2EE applications. Popular frameworks such as Struts, Hibernate, and the Spring Framework play an important role in many of today's J2EE development projects.

WHY USE A FRAMEWORK?

A software framework is a set of classes that make up a reusable design for an application or, more commonly, one tier of an application. Whereas application code calls a class library to perform services, a framework calls application code and thus manages the flow of control. This is often referred to as the Hollywood principle: "Don't call us, we'll call you." The application developer writes code that the framework will then call at runtime.

Designing a framework for use in a wide variety of unknown contexts is challenging. However, the framework approach is well adapted to the complexities of J2EE development because it can provide a simple, easy-to-use model for application programmers.

Using a well-designed open source framework offers many advantages:

- With a good framework, developers write only the code they need to write; they don't get bogged down working directly with low-level infrastructure APIs. This is the key value proposition.
- A well-designed framework can provide structure and consistency to an application. The structure will be clear to additional developers joining the project.
- An easy-to-follow framework can promote best practice through examples and documentation.

J2EE itself defines several frameworks. For example, an Enterprise JavaBeans (EJB) container or Servlet engine relies on the Hollywood principle, with the J2EE runtime instantiating and invoking managed objects. Open source Web application frameworks such as Struts add their own framework over the standard Servlet framework. The emphasis here is on frameworks *above* J2EE, which provide a simpler programming model or other benefits.

OPEN SOURCE FRAMEWORKS EMERGE

Most large J2EE projects have traditionally used in-house frameworks to hide the platform's complexity. Only recently has a consensus emerged about the generic problems that require a generic solution, and around particular frameworks that provide good generic solutions. There is now a clear trend for frameworks to "standardize" more of the infrastructure that formerly was developed on a per-project basis.

One reason for J2EE frameworks' sudden popularity is the platform's increased maturity. Developers now recognize areas in which the standard APIs are deficient and know from experience how difficult it is to write a good framework to fill the gap. In addition, many high-quality frameworks are

J2EE's Lack of Application Programming Support

Despite its success in standardizing interaction with many low-level system services, J2EE has largely failed to provide a satisfactory application programming model. For example, Java Transaction API, Java Naming and Directory Interface, and Java Message Service are designed to accommodate middleware vendors; they impose too much complexity on code to be directly used by application developers.

Similarly, Java Database Connectivity effectively shields J2SE and J2EE application developers from the proprietary wire protocols used to communicate to databases, and it does a fairly good job making Java code portable between databases. However, JDBC provides a cumbersome programming model if used directly, with rich potential for errors—especially in exception handling and managing resources such as connections.

J2EE's major effort to provide a standard programming model is Enterprise JavaBeans, a component model designed to let developers tap into standard J2EE services such as transaction management, remoting, and thread management. Unfortunately, EJB has proven disappointing in practice—among other problems, it places constraints on the use of inheritance and complicates unit testing. As with other J2EE APIs, productivity and ease of development and testing don't seem to have figured highly as design criteria.

now available that offer outstanding documentation and the support of a focused development team, without imposing licensing fees.

Struts

The trend toward open source frameworks began with Web applications. In 1999-2000, developers realized the deficiencies of the Java Server Pages "Model 1" approach, in which JSP templates handled incoming requests as well as static template data. This meant that JSPs often contained both business logic and complex HTML or other markup.

With no standard framework in place or J2EE specification support, developers responded with their own Front Controller implementations. These moved business logic to Java classes, thereby eliminating the need to maintain such hybrid artifacts.

The Front Controller pattern is often referred to as Web MVC after the classic Model View Controller architecture pattern that is common to GUI development in object-oriented languages. (The name is somewhat misleading in that Web MVC views must *pull* information from the model, whereas in classic MVC, the model *pushes* events to views.)

Initial Front Controller implementations varied greatly in quality. The Apache Software Foundation's release of Struts (http://struts.apache.org) in 2001-2002 changed all this. While not an ideal Web MVC framework, Struts worked well enough to quickly become the de facto standard.

Struts demonstrated all the benefits of open source frameworks such as ease of recruiting personnel familiar with the structure it imposed. By late 2002, it was the natural choice for most J2EE Web applications, and every serious J2EE Web developer was familiar with it.

The near-universal adoption of Struts commoditized an important chunk of the J2EE architectural stack. Even conservative organizations accepted its use in a prominent part of their software infrastructure and agreed to the Apache license's terms.

Hibernate

The next domino to fall was persistence. J2EE "out of the box" provided two means for accessing persistent stores—most often, relational databases: JDBC, the J2SE standard API for relational database management system access; and entity beans, an EJB component type dedicated to modeling a persistent entity.

JDBC's error-prone programming model inhibited object-oriented design by forcing developers to work with relational concepts in Java code. Entity beans, despite hype from Sun and major J2EE vendors, likewise proved to be cumbersome: Initially, the technology was severely underspecified, not even taking into account management of relationships between persistent objects; it made applications difficult to test; and it offered an inadequate query language. By 2003, developers largely ignored entity beans despite enhancements in EJB 2.0 and 2.1.

Early efforts. Solutions to the persistence problem came in the form of object-relational mapping (ORM), which provides *transparent persistence* for plain old Java objects (POJO), a concept described in the sidebar, "The Noninvasive Framework: Power to the POJO." Though not unique to Java, ORM is especially popular in the Java community—compared, for example, to .NET developers, who seem to regard it with suspicion.

Commercial ORM tools such as Oracle's TopLink (www.oracle.com/technology/products/ias/toplink) were well established by the late 1990s, but only a minority of projects used them because they were expensive, complex, and appeared to conflict with the Sun-sanctioned entity bean standard. Nevertheless, they usually achieved better results in practice than JDBC or entity beans, thus proving the case for POJO persistence.

Java Data Objects, which appeared as a Java Community Process (www.jcp.org) specification in 2001, offered generic POJO persistence to any persistent store (although implementations typically provided their best support for relational databases). However, Sun's lukewarm attitude toward JDO, coupled with J2EE vendors' lack of inter-

The Noninvasive Framework: Power to the POJO

Experience shows that developers don't like frameworks that impose excessive constraints on their code. Three novel capabilities of emerging J2EE frameworks that can help developers achieve the goal of a POJO-centric application are transparent persistence, inversion of control, and aspect-oriented programming.

Transparent persistence

This capability refers to the persistence of POJOs to durable stores—typically, relational databases—without the objects needing to make significant object-orientation concessions. It bridges the object-relational impedance mismatch with a framework responsible for mapping persistent objects to rows in a relational database management system, generating all the necessary Structured Query Language code to retrieve and store objects.

ORM tools use techniques such as reflection, dynamic byte code generation, or byte code enhancement in a post-processing step to perform this mapping at runtime.

Transparent persistence frees domain objects from the responsibility of managing their persistent representation, enabling them to contain business logic where appropriate, without mixing that with persistence operations. It can also greatly increase productivity by eliminating the need to write verbose and often error-prone persistence code.

Transparent persistence is a goal rather than a reality, but the best ORM solutions come close to achieving it.

Inversion of control

A POJO model can be applied to business services through IoC containers. These let business objects be configured at runtime, and enjoy declarative services such as automatic transaction management. *Inversion of control* is a widely used term that in this case refers to a model in which the framework instantiates application objects and configures them for use.

Dependency injection is a pure Java type of IoC that does not depend on framework APIs and thus can be applied to objects that aren't aware of—or may have been written without knowledge of—the framework.

Configuration is via JavaBean properties (*setter injection*) or constructor arguments (*constructor injection*). This means that application code doesn't implement any framework interfaces; the framework uses reflection to configure it. The framework *injects* dependencies such as collaborating objects or configuration parameters, without application classes needing to perform explicit lookup—as, for example, in the traditional JNDI-based approach to J2EE configuration.

Dependency injection is a simple, but surprisingly powerful, concept. Because the framework is responsible for resolving dependencies on collaborating objects, it can introduce a range of value-adds such as indirection to support hot swapping and codeless generation of proxies that represent remote services.

Aspect-oriented programming

Dependency injection goes a long way toward delivering a POJO application model but fails to address some important requirements, such as the ability to apply declarative transaction management—security checking, custom caching, auditing, and so on—to selected methods.

Traditional solutions to this problem all have substantial disadvantages. Using boilerplate code—for example, to start and commit or roll back a transaction—results in the same code being used in multiple methods. In addition, design patterns such as the Decorator end up with cut-and-paste code. And objects can only benefit special-purpose frameworks such as EJB, which provide a fixed set of services, by conforming to framework APIs and implicit contracts.

The Spring Framework provides a proxy-based AOP solution that complements dependency injection. AspectJ, AspectWerkz, and other AOP technologies are more ambitious, enabling modification of class byte code for more powerful weaving of aspects into an object model.

est in POJO persistence at that time, prevented the technology from achieving popularity.

Hibernate arrives. Radical change came in 2002 for two reasons. First was widespread realization that entity beans had failed in practice, and that developers should ignore that part of the J2EE specifications. By retarding rather than advancing the progress of ORM in Java, entity beans remain a prime example of how poor specifications can stifle development of superior technologies.

The second factor was the arrival of Hibernate (www.hibernate.org), the first popular, fully featured open source ORM solution. Hibernate offered fewer features than TopLink but delivered a robust implementation of the most desirable ones, and its focused development team aggressively sought improvements. Hibernate wasn't particularly innovative, building on the extensive understanding of ORM, but it offered a more intuitive programming model than existing competitors and removed at one stroke the cost and ease-of-use barriers to ORM.

Around the same time, new commercial products offered highly efficient implementations of the JDO specification that targeted relational databases, giving developers a rich choice. Meanwhile, TopLink remained a good option, with its license becoming friendlier to developers.

ORM triumphs. Together, all these factors converged to make ORM the norm rather than the exception by 2003-2004. Although some projects still built their own persistence frameworks, the existence of Hibernate, TopLink, and leading JDO implementations made this extremely difficult undertaking unnecessary and indefensible.

Another part of the application stack was now within the domain of popular frameworks, yet large gaps remained. For example, a typical Web application using Struts and Hibernate still lacked framework support for business logic. Although the J2EE specifications address some of these issues, primarily through EJB, they don't provide an adequate application programming model.

Spring

J2EE frameworks have inexorably moved into application frameworks, which aim to provide consistent programming in all tiers and thereby integrate the application stack. The Spring Framework (www.springframework.org) is the dominant product in this space, with adoption comparable to that of Hibernate.

Spring essentially combines *inversion of control* (IoC) and aspect-oriented programming (AOP)—both described in the sidebar, "The Noninvasive Framework: Power to the POJO"—with a service abstraction, to provide a programming model in which application code is implemented in POJOs that are largely decoupled from the J2EE environment (and thus reusable in various environments). Spring also provides an alternative to EJB in many applications—for example, delivering declarative transaction management to any POJO. The Spring approach has proven to deliver excellent results in many kinds of projects, from small Web applications to large enterprise applications.

Other products in the same space include HiveMind (http://jakarta.apache.org/hivemind), which is conceptually similar to Spring but has a somewhat different take on IoC, and NanoContainer (http://nanocontainer.codehaus.org), which combines the PicoContainer IoC container with services. Collectively, these products are referred to as *lightweight containers* to distinguish them from traditional J2EE approaches.

By decoupling a POJO model from J2EE APIs, which are hard to stub at test time, lightweight containers greatly simplify unit testing. It's possible to unit test in a plain JUnit environment, without any need to deploy code to an application server or to simulate an application server environment. Given the increased—and deserved—popularity of test-driven development, this has been a major factor in lightweight frameworks' popularity.

WHAT'S NEXT?

Growing recognition and use of J2EE development frameworks is measurably reducing cost in many projects, as well as delivering better speed to market and higher maintainability. Today's best frameworks offer excellent quality, solid documentation, and numerous books and articles to support their use. Nevertheless, two areas in particular seem set for uncertainty in the J2EE space: the conflict between J2EE "standards" and open source innovation, and the growing importance of AOP.

The open source versus standards conflict looms in two areas. In the presentation tier, JavaServer Faces (JSF), backed by Sun and some of the largest vendors, competes with entrenched open source solutions such as Struts. In the middle tier, EJB 3.0 offers a dependency injection capability reminiscent of a subset of Spring's capabilities, mixed with liberal use of J2SE 5.0 annotations.

In both areas, innovation has traditionally come from open source rather than specifications. However, JSF is somewhat indebted to ASP.NET, while the open source Tapestry project (http://jakarta.apache.org/tapestry)—a mature implementation of many of the same concepts—owes much to Apple's commercial WebObjects.

Likewise, EJB 3.0 seems to be attempting to standardize dependency injection, though it's unclear what benefit this brings—especially if it results in the loss of important features, which seems inevitable. EJB 3.0 also attempts a new departure in entering the application programming space: an area in which the J2EE specifications haven't shone to date.

Meanwhile, AOP's importance is steadily increasing within the J2EE community. While adoption isn't yet widespread, certain uses of AOP, such as declarative transaction management, are already popular. Solutions including Spring and dynaop (http://dynaop.dev.java.net), which offer what might be called "AOP with training wheels," help to increase awareness of AOP. Full-blown AOP technologies such as AspectJ will likely experience wider adoption in the next few years as well.

Significantly, the Java Community Process shows no sign of any move to standardize AOP, although JBoss (www.jboss.com)—which is overtly committed to working through the JCP with the EJB 3.0 specification—is vigorously pursuing proprietary AOP technology.

The next-generation J2EE specifications as a whole are embracing a simpler, POJO programming model, similar to that already offered by combinations such as Spring and Hibernate. J2EE developers are sure to benefit from recent trends, which are driven more by practical experience than by marketing hype. This is a welcome change from the platform's early days, when results often failed to live up to the promise held out by vendors. ■

Rod Johnson *is CEO of Interface21, a J2EE consultancy based in London. Contact him at rod@interface21.com.*

Editor: Richard G. Mathieu, Dept. of Decision Sciences and MIS, St. Louis University, St. Louis, MO; mathieur@slu.edu