

Using J2EE on a Large, Web-Based Project

Eric Altendorf, *Independent Consultant*

Moses Hohman and Roman Zabicki, *ThoughtWorks*

Building large, multitier, Web-based applications requires an array of software services, from dynamic HTML generation to business domain object persistence. Rather than implement these services from scratch, application developers find it more efficient to use standard, publicly available implementations. Sun's popular J2EE framework provides a specification and common set of interfaces for these services in the Java programming language. J2EE lets software vendors offer

competing implementations. This in turn lets application developers build software using their choice of service implementations, which frees them to concentrate on application-specific code. (For pointers to more detailed background information on the J2EE technology family, see the "Further Reading" sidebar.)

The project we describe in this article is a Web-based enterprise software system for a Fortune 100 client, developed using J2EE technologies including Enterprise JavaBeans (EJBs) and Java server pages (JSPs). The system comprises approximately 700 kLOC, 5,600 classes, and 500 JSPs. We have been developing it continuously for three years, and our client has been running it live for two.

We partitioned the system horizontally into three tiers:

- a presentation layer, which drives the user interface

- a session layer, which manages the user workspace and session business logic
- a domain object layer, which handles persistence and entity business logic

The system's front end is a set of JSPs, each with a "display bean" that provides data and defines presentation logic. The display beans access and update the persisted data by means of XML data transfer objects that the session layer loads and manages. The system also integrates several external services accessed over the network.

Here, we discuss our experiences and lessons learned in dealing with five key development topics: J2EE HTML-rendering technologies, JSP internationalization, external system integration, sharing information between tiers using XML data transfer objects, and maintaining complicated domain object hierarchies using code generation.

Java 2 Enterprise Edition is a component-based approach to developing enterprise applications. Here, the authors describe their experiences and lessons learned in using J2EE on a large, Web-based custom development and enterprise-integration project.

Further Reading

For the reader unfamiliar with J2EE and other technologies discussed in this article, we recommend the following sources.

Java 2 Enterprise Edition

- Information about Java 2 Enterprise Edition is available at <http://java.sun.com/j2ee/overview.html>.

Enterprise JavaBeans

- *Mastering Enterprise JavaBeans*, 2nd ed., by Ed Roman et al., John Wiley & Sons, New York, 2001.
- An Enterprise JavaBean overview is available at <http://java.sun.com/products/ejb/index.html>.
- Enterprise JavaBean specifications and document downloads are available at <http://java.sun.com/products/ejb/docs.html>.

Java server pages

- An overview of JSPs is available at <http://java.sun.com/products/jsp/index.html>.
- Specifications for JSP 1.2 and Servlet 2.3 are available at www.jcp.org/aboutJava/communityprocess/final/jsr053.

Other resources

- A discussion of HTML templating and scriptlets is available at Martin Fowler's "Template View ISA Pattern," www.martinfowler.com/isa/serverPage.html.
- Descriptions, tutorials, and downloads of custom tag libraries are available at <http://java.sun.com/products/jsp/taglibraries.html>.

J2EE HTML-rendering technologies

Most Internet applications render the user interface in HTML. Java systems generally use servlets, JSPs, or both to dynamically generate the HTML.

Our solution

We decided to use just JSPs for several reasons. First, there are the oft-quoted benefits of separate development roles: Readily available and relatively inexpensive HTML designers can write static HTML templates (like JSPs), while more expensive enterprise Java programmers provide the back end for dynamic content. However, separation proved to be less important during actual development. An internal business application's graphic design requirements are typically simpler than a public Web site's, so we didn't need a graphic design team. Also, we encourage our consultants to maintain skills and responsibilities at all levels—from HTML to architecture—and thus few specialize in graphic design.

On the other hand, we planned to build the pages so that the client could modify the design once the project was over. So, while the role separation was not so important during

development, we anticipated that later, during maintenance, the client would find it easier to make look and feel and other HTML-only changes that businesses typically require.

Finally, a template-style front end is easier to work with and test. When HTML and dynamic data are mixed in Java code, seeing the flow of both the program and the HTML output is often difficult. It can also be hard to debug the static HTML or make code changes without breaking the HTML.

Experience

After more than two years of JSP usage, our experiences are mixed. Although JSP technology permits separation of Java from HTML, it does not enforce it. Developers can still embed sections of Java code (called scriptlets) in the JSP or write display-bean methods that generate HTML and call them from the JSPs. In fact, prior to custom tags, many tasks were impossible without resorting to scriptlets or HTML-producing methods. We are still refactoring some of our older JSPs and display beans to separate the HTML and Java.

We did gain the definite advantage of rapid prototyping. Using standard HTML editors, we can quickly create prototype screens, then simply take the generated HTML file, rename it as a JSP, and hook it up to a display bean.

We also learned something about the applicability of JSP technology to different types of pages. The JSP model is excellent for "fill in the blank" pages composed of a static HTML template populated with data drawn from the application (strings, numbers, dates, and so on). However, JSPs are inadequate for highly dynamic pages that require different layouts or designs under different conditions. For pages with moderately varying structure, you must use either HTML and produce display-bean methods or scriptlets that control the page's content display. For greater variations, you often need imports or includes, which create other difficulties, such as maintaining syntactically valid HTML output. They also force you to deal with the implicit dependencies resulting from bean- and variable-sharing or cascading stylesheet usage.

Scriptlets are not necessarily bad if you use them in moderation for basic tasks, such as simple conditional logic. In our experience, however, such simple usage can easily accrete modifications and lead to scriptlet bloat. Worse still, if you need a similar task on sev-

eral pages, independently bloating scriptlets will likely perform that task in slightly different ways, making maintenance difficult.

Custom tags are a good alternative, because they do not accrete modifications in this way. They offer a standardized solution to a common problem with a well-defined interface. This encourages developers to use them carefully, leading to cleaner templates. Our most notable successes with custom tags include tags for conditionally including or hiding country-specific content, which gave us a single JSP that serves all countries, and iterator tags for generating table rows, which we previously created using an HTML-producing method on the display bean.

Internationalization issues

Many business applications must deliver content to users residing in multiple countries, and must thus present data in multiple formats. Sometimes this is as simple as using different currency and date formatting, but in other cases, the logic is so different that it might require what seems like an entirely separate application.

The internationalization required to extend our application to Australia and Canada was only moderate. The presentation-layer functionality we needed was similar to the US version. Most screens had minor differences due to variations in each country's business processes and legal requirements. Nonetheless, the changes surpassed the type of internationalization tasks typically discussed in J2EE tutorials, such as presenting text in different languages or data in different formats.

Our main design decision was whether to

- write each JSP to handle every variation for every country (the monolithic approach) or
- write one version of each JSP for each country (the tailor-made approach)

Our solution

Initially, we picked the tailor-made approach. We had three main reasons for this: tight time constraints, the lead developer's personal preference, and our temporary inability to upgrade to a JSP container that supported custom tags. Without custom tags, a monolithic approach would have forced us to use scriptlets around every localized form input,

table, and hyperlink. As we noted earlier, scriptlets can be messy and quickly render a large JSP unmanageable. If we grew to support five or 10 different countries, the monolithic JSPs would likely be so complex and difficult to read that we would have to split them up into a single JSP for each country anyway. Furthermore, once it became that complex, breaking up the JSP without error would be difficult.

Experience

We quickly realized that the tailor-made approach had serious problems: Screen updates often neglected one country's version of the page; because our system had so many JSPs, keeping them in sync was virtually impossible. Even with only three countries, maintenance was becoming a nightmare.

Once we upgraded to a servlet container that supported custom tags, we found a solution to our problem. Custom tags offer a simple, clean way to define shared behavior over a set of JSPs. This let us unify each screen's localized versions into one easy-to-read, easy-to-modify JSP. Because custom tags can selectively include or exclude sections of a JSP, we did not need lots of increasingly complicated scriptlets.

So, instead of creating a monolithic JSP riddled with scriptlets such as

```
<%if(bean.getLocalization()  
    .equals(Localization.CANADA) ||  
bean.getLocalization()  
    .equals(Localizations.US)) {%>  
    Canada or US-only HTML  
<% } %>
```

we use a simple custom tag:

```
<localize show="CA,US">  
    Canada or US-only HTML  
</localize>
```

This example is a typical tag usage. Not only does the custom tag avoid scriptlet bloat, but we'd hazard a guess that extending this custom tag example to 10 countries would be much easier to understand. So far, the custom tag has proven sufficiently powerful for our screen internationalization needs.

However, we can certainly imagine pages with variations that would be difficult to capture clearly using this kind of in-page conditional logic. In such cases, we could create one JSP per country for the page portions

Custom tags offer a standardized solution to a common problem with a well-defined interface.

Using multiple threads to serve more than one request at a time improved performance.

that varied dramatically and include these JSPs in the surrounding, unvarying template using a custom tag. The problem is that a developer would have to keep track of multiple files' content when editing a single page's static template. Perhaps researchers will develop tools some day that let programmers edit an entire country's template at once (either storing it in multiple files or supplying the necessary conditional inclusion logic behind the scenes), while still keeping invariable content the same across all countries.

Integrating with third-party software

Very few enterprise applications run autonomously; most must connect to existing systems. Given that such systems are often archaic, use proprietary interfaces, or are simply built with a somewhat incompatible technology, this integration rarely comes easily.

Our client's business domain requires intricate calculations. Our application provides this through integration with third-party financial software, which we'll refer to here as TPS (for third-party software). The third-party vendor originally wrote TPS for use as a stand-alone application. The client wanted to integrate TPS within our larger application to ease version management and radically simplify what had been human-powered interfaces between TPS and other business processes.

Our solution

At first, the third-party vendor shipped us a dynamic linked library (DLL) file that exposed a subset of TPS's objects via Microsoft's Component Object Model (COM). To drive the DLL, we wrote an extensive adapter composed of two types of components: a server and a pool of translators it managed.

To perform financial calculations, the main application sends XML describing financial parameters and the type of calculation required to the TPS adapter's server component. The multithreaded server component then forwards this XML request to a translator component. This component translates the XML into a series of COM calls into TPS's library, and returns the result in XML form. The server component maintains a pool of translator components, each of which must run in its own separate Java virtual machine (JVM) process, since the TPS DLL is not thread-safe. We found that using multiple

threads to serve more than one request at a time improved performance.

Experience

Many challenges arose in the process of integrating TPS into an Internet application. The main problem is that TPS has thousands of possible input parameters. Although we don't use all of them, the subset we do use can itself produce numerous parameter combinations. Reproducing this much functionality without error is difficult, especially because we do not use the same interface to the TPS library as the stand-alone TPS.

Because no one had used the COM interface to the complicated TPS library before, our adapter inevitably used the software in unintended ways. One result was that the translator component processes occasionally crashed, sometimes in an unrepeatable manner. To keep these crashes from bringing the TPS application down, we gave the server component the ability to resurrect failed child processes. In such cases, it repeats the request once, returning a service failure only if the child process fails twice on the same request. This provides two kinds of robustness. For unrepeatable crashes, the repeated request returns the desired result. For repeatable crashes, this strategy avoids corrupting the pool with dead translator components.

In addition, our server component restarts translator processes after each successful calculation as well. This eliminates any problems due to memory leaks or variable initialization problems in the third-party DLL (which seldom, but unpredictably, occur). The cost is a small performance hit. However, this choice, coupled with requiring that the server component resurrect child processes, ensures that the TPS financial calculation engine remains available and correct (for all previously tested requests) over long time periods.

Testing the application gave rise to other challenges. Because the COM objects have complex interdependencies—which are fairly common for the problem domain—writing independent, fine-grained unit tests is difficult. We opted instead for a set of regression tests, each composed of XML input and the expected XML output. These tests are fairly simple to automate and natural to generate, because we can retrieve the XML from our application's logs. They are also easy to maintain using Extensible Stylesheet Language

Transformations (XSLTs). However, they do take longer to run than unit tests, so we run them less often—daily instead of hourly, which means we catch regression errors only on a daily basis. However, because only one or two people work on the TPS adapter at a time, we have not found this to be a problem.

Because of the complicated TPS COM object interdependencies, the vendor-supplied DLL requires a lot of quality assurance: The vendors add features, send the results to us, and we use our suite of regression tests to ensure nothing else has changed. Fortunately, we have a good relationship with the vendor, and they’ve always been quick to respond to our bug-fix requests. Nevertheless, this back and forth communication takes a lot of time. Ideally, testing should be collocated with the vendor’s development work so that feedback is more immediate.

To address this problem, we are moving away from the TPS COM interface toward an XML interface. With the XML interface, there is just one COM object with one method that accepts XML input and returns XML output. Now, the vendor’s software translates XML to library function calls. When we find a bug, we simply send them the input XML that exposed it. They can easily process that XML within their debugging environment and track down the cause. Additionally, the vendor can maintain a set of regression tests that reflect how our application uses TPS and incorporate those tests into their QA process. We expect this to dramatically reduce the time it takes to incorporate additional TPS features into our application.

Distribution: XML data transfer objects

To facilitate information flow between tiers, multitiered and distributed enterprise applications commonly use the Data Transfer Object design (www.martinfowler.com/isa/dataTransferObject.html). The DTO encapsulates a set of parameters that one component sends or returns to another component to complete a single remote method call (such as a calculation request from our main application to TPS). In our application, an object called an XML translator constructs an XML-based DTO from data in the domain object layer and ships it to the session or presentation layer, making the data available in a lightweight package. The session and presentation layers

can also construct DTOs to be translated back to the domain object layer, persisting the data. Using DTOs prevents the locking problems, network chattiness, and transactional overhead associated with frequent calls to fine-grained data-access methods on entity beans.

Earlier in our project, DTOs were absolutely necessary because the production EJB and servlet containers were in different cities, and the network latency was far too great to rely on direct access to the entity beans. Even now, with servers running in the same JVM, start-up costs for a new transaction for every method call make the DTOs valuable.

Our solution

For our DTOs, we chose a hybrid XML–Java object approach: An XML document contains the data and is wrapped by a Java class for easy access. We write translators that run in the session layer and perform the DTO–domain object mapping, both by building DTOs from the domain object data and saving modified DTOs back to the domain objects.

We can also construct from scratch “update only” DTOs in the presentation layer with only partially populated element trees. When the translator persists such a DTO, it updates only those fields that contain updated values for domain objects. This lets clients who only need to update a few fields do so easily, rather than have to translate a large domain object graph to XML, modify one or two fields, and then save the entire XML back to the domain objects again.

We generate the Java wrapper classes from a sample XML document. We do not generate the translators, because the data often takes a different form in the presentation layer than in the domain object layer, and we want to control the translation. At the individual-field level, we often use different data types. For example, we represent what might be an integer status ID in the database by its status keyword in XML (“In Progress,” “Expired,” or “Canceled”), and what might be an SQL date in the database could be in ISO 8601 form in the XML. On a structural level, we control traversal of the entity bean graph. For example, if bean Foo is being translated to XML and contains a reference to bean Bar, we might insert into bean Foo’s XML either bean Bar’s XML translation or a `<barId>` element that lets clients look up bean Bar when

Ideally, testing should be collocated with the vendor’s development work so that feedback is more immediate.

The XML objects make debugging much easier, especially in production or in environments where debugging tools are unavailable.

required. Finally, on a semantic level, we occasionally introduce new, presentation- or session-related fields into the XML that do not exist on the domain object layer.

Our translators can also register multiple translators of different “flavors” for a single domain object. This lets us handle the differing needs of various application parts. For example, if one application part requires only subset X of the domain object fields, or requires fields in a certain format, and another part requires subset Y and a different format, it’s no problem. We simply write two flavors of the translator.

Experience

Using XML as the DTO representation was a good choice. Because our application is an intranet application used by relatively few employees, we have not had to worry much about scalability issues, and the XML data structures’ size in the session has not yet been a problem. The XML objects make debugging much easier, especially in production or in environments where debugging tools are unavailable. We can see the current state of the business objects clearly, concisely, and in a human-readable fashion just by looking at the XML. When we have to integrate with other software systems, the XML provides a convenient base for designing the interface.

Our experiences have not been entirely positive, however. We’ve encountered several largely unresolved difficulties with using DTOs.

First, fine-grained access to individual entity-bean fields is still awkward. When only one field is required, it’s inefficient to gather all the data a large DTO requires—which can include a hundred fields from a dozen different domain objects. Also, writing an entire XML translator flavor to return one or two fields is a lot of code overhead. You can write a session-bean method to access that field (in our architecture, all presentation-layer calls must be mediated by a session bean), but to avoid session-bean bloat—and to avoid the original network traffic and transactional issues—we must use such methods in moderation. An alternate solution that we’re currently investigating is to pass the translator one or more Xpaths that specify a subset of the XML DTO’s fields, so that the translator fetches only those fields from the domain object. (For more on Xpaths, see www.w3.org/TR/xpath.)

Second, using DTOs somewhat compli-

cates the modification of domain object classes. When we add a field to a domain object, we must update the sample XML document, regenerate the XML DTO class, and manually update the translators to process the new data. However, this is not usually difficult or time consuming.

Third, and perhaps most significant, sometimes the presentation layer requires domain object business logic, and although DTOs provide a means of sharing data between the two layers, they do not help with sharing behavior. For example, a domain object might have an `isValid()` method that returns `true` when the object’s fields satisfy certain requirements. If the presentation layer must know the underlying domain object’s validity, we must write a session-bean method that accepts a DTO (or ID) as an argument and delegates the call to the existing method on the corresponding domain object. This works, but suffers from the same performance issues as domain object individual-field access. If the presentation layer modifies our DTO, and we need to determine the DTO state’s validity, we have an even more unpleasant situation. In this case, we must either

- write a new method that takes the DTO and duplicates the logic from the domain object method, or
- translate the DTO back to a temporary “dummy” domain object on which we can call the `isValid()` business method

So far, the problem is not that great in our case. A possible future solution is to use method objects¹ that represent rules—such as validity conditions—or other tasks. We could then run these rules against both domain objects and DTOs, keeping the business logic in only one place.

Embracing change: Generated domain objects

Typically, the core of an enterprise application has a domain object layer—a layer of code that manages persistence and business logic for the fundamental objects in the business domain model. Our domain object layer maintains the state of well over 100 domain objects and also provides some of their basic business functionality. Such scale and complexity always provides developers with challenges in keeping systems flexible and maintainable.

Our solution

In our application, most of the domain layer consists of code generated using metadata files that describe the business objects. Several factors motivated our decision to use code generation. First, EJBs are a heavy-weight technology with significant coding overhead: A bean developer must create at least four separate files for each domain object, which involve a fair amount of code or declaration duplication. Making the same change in all these classes slows down development. Because our domain objects were evolving rapidly, especially in early prototyping stages, we had to be able to quickly update the code, regenerating from the changed metadata. Later, when we began to separate persistence and transactional responsibilities (implemented with EJBs) from business responsibilities (implemented with plain Java classes), we required even more classes, further increasing the generators' value.

Second, we'd yet to fully commit to EJB technology—particularly entity beans, because at that time they were not even required by the EJB 1.0 specification (<http://java.sun.com/products/ejb/docs10.html>).

Third, EJB technology was still rather immature; vendor-supplied containers exhibited many bugs (in fact, they still do). Using generated domain objects let us centralize the required hacks and bug workarounds in one place—the generators—yet still apply them across all domain object implementations.

We named our domain object inheritance structure the “martini glass model” for the visual appearance of its UML diagram (see Figure 1). Its complex structure is partly a result of the fact that, unlike many code generation schemes, generated and handwritten customizing code never live in the same file, so merges are never required. Instead, the Java compiler and virtual machine effectively perform merges automatically as a result of the inheritance structure. Our UML model was not designed up front, but rather evolved as we required more features and flexibility. This would have been infeasible without generated code.

We developed the structure in five stages, adding features and complexity on an as-needed basis:

1. We generated the four basic required classes or interfaces: the remote interface, the home interface, the primary key

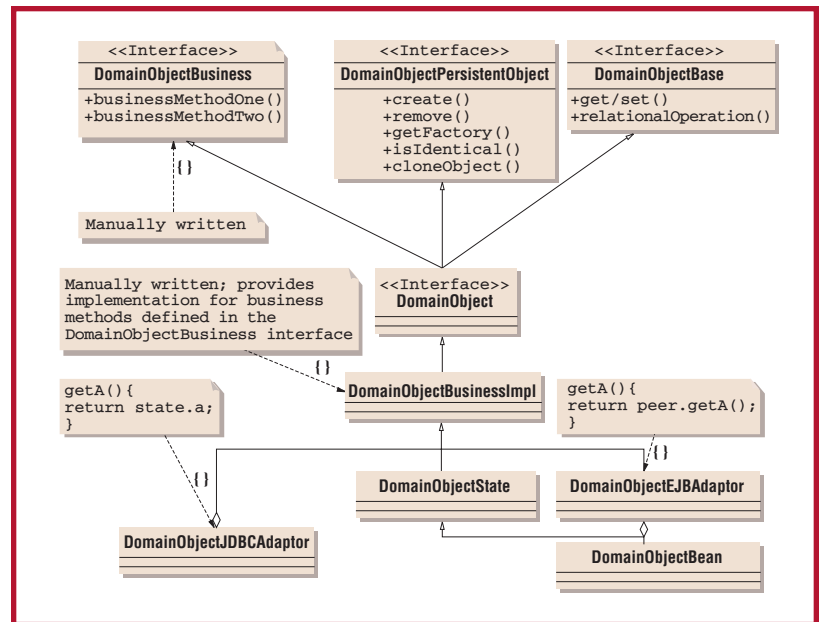


Figure 1. UML diagram of a sample domain object model's core. All classes and interfaces shown are generated from metadata, except for the DomainObjectBusiness interface and the DomainObjectBusinessImpl class, which are hand-written.

1. We generated the four basic required classes, and the entity bean itself.
2. We generated the finder class (the service locator), which encapsulated the Java Naming and Directory Interface (JNDI) calls and provided a caching service for the home interface.
3. We split the entity bean into a generated persistence class (entity bean class) and a handwritten business-logic class. This let us regenerate the entity bean (the field accessors) without obliterating hand-coded business methods.
4. We then generalized the persistence interface (field accessor signatures), allowing for persistence implementations other than entity beans. Because our EJB container did not support read-only access to entity beans, we used our persistence-layer abstraction to permit two implementations: fully protected entity bean read-write access and Java Database Connectivity (JDBC)-based read-only direct database access.
5. We added custom mappers and finders.

Mappers define the mapping between domain objects and the persisted data that represents them, and custom mappers let us implement unusual persistence requirements. Custom finders were useful because our container's query language was insufficiently powerful for the domain object searches we needed. The finders provide an abstraction layer, delegating to either the bean home methods for simple

Custom tags are a powerful, clean way to abstract shared JSP behaviors.

searches, such as “find by primary key,” or custom finders implemented with SQL queries for more complex searches.

Experience

One problem we’ve frequently encountered with our domain objects is deadlocking. Our EJB container uses pessimistic concurrency control, so if one transaction accesses an entity, it will prevent code running in another transaction from accessing that same entity. Deadlock can occur if, for example, an entity calls a session-bean method—thus starting a new transaction—and the called method in turn touches the original entity. To help avoid deadlocking, we carefully set transactional contexts and minimized locking through the use of JDBC-based read-only persistence.

Another problem we faced was performance. By default, every call on an entity-bean remote interface must go through the gauntlet of security context checks, transactional context checks, acquiring locks, and calls through stubs and skeletons—even if all that the caller requires is read-access to a field. In such simple cases, using JDBC-based persistence for read-only access saves a lot of overhead.

One of the challenges in developing Internet applications is that many of the technologies are new and many of the implementations are buggy, incomplete, or not quite written to specification. This was certainly true for our EJB container. Many of its bugs were related to caching and transactional boundaries and were difficult to find, reproduce, and test against because the testing environment must exactly duplicate the running system’s transactional context management. Once we identified the bugs, however, using code generation facilitated workarounds throughout our domain object layer.

One disadvantage with code generation is that to achieve the flexibility of handwritten entity beans, we must use an elaborate structure of supporting interfaces and classes. This creates a complex system that takes longer for new developers to learn. Some of the system’s features are esoteric, and few developers know how to use them. For example, we can set the persistence implementation to JDBC-based read-only for all domain objects accessed in a given transaction. This is useful for avoiding deadlocks in calculations that access many domain objects. Although few developers use or understand such advanced features,

they are consistent throughout the application. This would not be true with handwritten code. Also, the system basics are quite simple. Several other ThoughtWorks projects have successfully used the generators, which let developers get an entire domain object layer up and running rapidly.

The greatest benefit of our domain object-layer generation system is that it lets us maintain our agility and frees us from architectural decisions made years ago. The generators let us easily modify individual domain objects as well as make major, global architectural changes, such as abstracting the persistence interface on every system domain object, or, in the future, migrating to the EJB 2.0 specification.

In the process of developing our system, we learned many lessons.

- JSPs let us rapidly develop a Web application’s user interface and separate Java and HTML development roles. However, developers writing applications with highly dynamic content might find JSPs awkward, making it more difficult to take advantage of the separation benefits.
- Custom tags are a powerful, clean way to abstract shared JSP behaviors, such as iteration or conditional content inclusion.
- Although conventional wisdom advises against maintaining a set of regression tests on third-party code, it is in fact often useful, especially when those tests are written in a portable format such as XML and can thus be transferred to the third-party vendor.
- Data transfer objects are a useful, often necessary part of multitiered applications. However, owing to the resulting dual object representations, developers must carefully consider which business methods belong in the domain object layer and which belong in the presentation layer. Doing so can help avoid excessive code duplication or cross-tier method call delegation.
- It is possible to change even the most fundamental architectural decisions—such as the domain object-layer design—even when you are well into the project. Code generators let developers make sweeping changes easily across the entire

code base and are thus an extremely valuable tool for the agile developer.

At present we are in the process of internationalizing our application for France, our first system-wide language translation. The changes involved have highlighted some unnecessary complexity in our presentation layer. Some of this complexity we can resolve, applying the lessons discussed above uniformly throughout the application. Other issues, such as an awkward algorithm for shuttling data between HTML forms and the XML DTOs, remain problematic. We have experimented with several ideas, from the Apache Struts framework (<http://jakarta.apache.org/struts>) to simple custom-built architectural solutions. We anticipate that more useful lessons will come from these investigations. ☞

Reference

1. K. Beck, *Smalltalk Best Practice Patterns*, Prentice Hall, Upper Saddle River, N.J., 1997.

About the Authors



Eric Altendorf is an independent consultant. He has worked in both academic settings and private industry, and was a software developer and architect at ThoughtWorks on the project described here. His professional interests include agile development, software architecture and design, and generative programming. His research interests include neurally plausible theories of thought and language, reinforcement learning approaches to computer-played Go, and time-series prediction. He received a BS in computer science and mathematics from Oregon State University, and plans to resume study for a doctoral degree in artificial intelligence. Contact him at EricAltendorf@orst.edu.independent.

Moses Hohman is a software developer for ThoughtWorks, a custom e-business software development consulting firm. Currently he works out of the recently opened Bangalore, India, office, where he mentors new software developers and teaches a course on agile software development at the Indian Institute of Information Technology Bangalore. He received an AB in physics from Harvard University and a PhD in physics from the University of Chicago. Contact him at mmhohman@thoughtworks.com.



Roman Zabicki is a software developer for ThoughtWorks. He received a BA in computer science from the University of Chicago. Contact him at rfzabick@thoughtworks.com.



Pullquote