The Co-Evolution of a Hype and a Software Architecture: Experience of Component-Producing Large-Scale EJB Early Adopters

Lutz Prechelt and Daniel J. Hutzel
abaXX Technology AG, Stuttgart
{lutz.prechelt|daniel.hutzel}@abaxx.com

Abstract

abaXX.components was one of the first API software products fully based on Enterprise JavaBeans™ (EJB) technology. We describe the evolution of its architecture as it moved from simply taking the initial EJB hype for the truth, through several intermediate stages, to using EJB simply as one of several encapsulated implementation techniques. So far, the public perception of how to use EJB properly evolved along a similar path, lagging 6 to 12 months behind.

1. Introduction

abaXX Technology was founded in April 1999 and was possibly the first software company that based all of its work on Enterprise JavaBeans (EJB) and other Java2 EnterpriseEdition (J2EE) Technology.

This report describes the architectural evolution of abaXX's main suite of software products, called abaXX.components, a collection of components and frameworks for building web-based e-business/eCRM applications of all kinds (primarily process-based end-customer portals). The authors are responsible for devising and evolving this architecture and for explaining it to abaXX customers.

2. The software: abaXX.components

abaXX.components is written entirely in Java and uses many technologies of the J2EE and Web world such as Servlets, Java ServerPages (JSP), Enterprise JavaBeans (EJB), JNDI, JDBC, JMS, XML, XSL-T, and others.

abaXX.components comprises frameworks and small-grained and large-grained application components in the following functional areas: web-frontend programming (including portlets), portal-process modeling and programming (workflow), integration of content providers (such as content management systems), email/fax/sms messaging, document generation, user tracking and automatic personalization, e-commerce, and advice/configuration applications.

Most of the product has the form of APIs (separated into client API and extension API), the rest is tools (e.g. the graphical workflow modeler), source code examples, and a complete Web-GUI application called AdministrationCenter for business-view administrators. An application using abaXX.components is always custom software; the implementation project involves a significant part of programming rather than just configuration.

Despite its heavy use of many J2EE technologies including EJB, the product directly supports various application servers such as IBM WebSphere 4.0, BEA Weblogic 5.1 and 6.1, and JBoss 2.6 and 3.0, which are by far less compatible to one another than one might expect.

Our goals for evolving abaXX.components can be summarized as follows:

- continously improving the power, performance, openness, extensibility, and ease-of-use of the components;
- maximizing the flexibility and extensibility of the resulting custom applications;
- exploiting improvements in the application server infrastructure;
- continously adapting newly evolving standards (such as Apache Struts for frontend programming) and new technological trends (such as web services) without breaking backwards compatibility to earlier versions.

3. Enterprise JavaBeans (EJB) quick intro

While this report assumes some technical familiarity with EJB, here is a quick quick overview for all other interested readers.

Enterprise JavaBeans [3] is the name of a technology specification featuring software components (think of each as one logical class) that reside within an EJB container. The EJB container is a major element of a J2EE application server. There are two flavors of enterprise beans: Entity beans represent business/domain objects and session beans represent business services. The EJB specification has seen versions 1.0, 1.1, and 2.0 (2.1

is under way) and starting with EJB 2.0, there is also an extension of session beans not discussed here called MessageDrivenBeans.

Clients access an enterprise bean through a remote proxy interface. The EJB container intercepts each call and transparently inserts additional functionality that provides added value to the caller. This additional functionality comprises life-cycle management, automatic transaction control, automatic persistence management (for entity beans), resource management, access control, remoting, load balancing, and fail-over management (with replication).

The central idea behind EJB and the whole point in using it is the added value from these implicit functions.

4. Enterprise JavaBeans™ (EJB) quick history

The initial hype around EJB claimed that EJB solved essentially all of the problems around all of the above aspects and did so with essentially no disadvantages. Which, of course, is just not true at all.

Rather, the main drawbacks of EJB are:

- The programmer no longer has full control over the above-mentioned characteristics, which sometimes leads to inefficiencies and/or awkward designs.
- Client calls to an EnterpriseBean are always remote calls and hence quite run-time expensive.
- The development of an EnterpriseBean is inconvenient, as it involves many files (home interface, remote interface, implementation class, standard deployment descriptor, application-server-specific deployment descriptor).
- The configuration and use of an EJB container is complex and full of pitfalls.
- Exposing EJB interfaces to application programmers through a Client API can impose dangerous impacts on system performance and robustness. This is because application programmers would then be bothered with the complexity of dealing with remote objects.

In younger times, partial countermeasures against many of these disadvantages have been introduced by the J2EE community:

- The J2EE BluePrints design patterns [4] describe how best to cope with the first, second, and last,
- The notion of local interface, introduced in EJB 2.0 solves the second for the case when the client is another EJB in the same container,
- Modern IDEs and CASE tools cope reasonably well with most of the third.

Since we were early adopters, however, we were usually too early for these developments and had to find our own solutions. The resulting archtectural evolution is described in the following sections.

5. Evolution phase 1: Naïveté

"Is it important? Then it's an EJB."

When EJB appeared, the hype and public perception essentially said "Just do it. EJB will solve your problems." When we started developing with EJB technology, we decided to follow just this approach, in order to see where it would work and where not.

Roughly speaking, in the first version of abaXX.components, each business service was a session bean and each business object – even any persistent object – was an entity bean.

The resulting software worked, but the runtime performance was rather poor, due to EJB-born brute-force remoting, transaction management, and locking overhead.

6. Evolution phase 2: Repair

"If we don't fix this, we're dead."

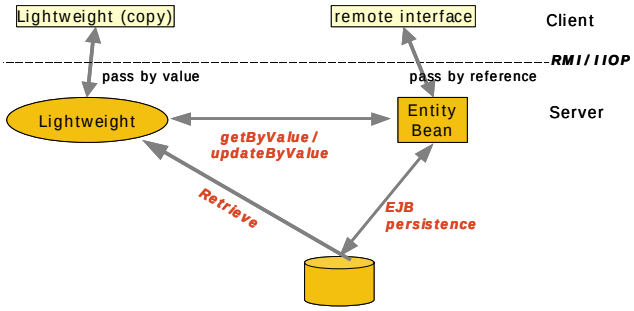
It turned out that in our typical web applications, say, for e-commerce, most of the operations were read-only accesses to many different entity beans during catalog browsing. But for retrieving, say, 20 entity beans with an EJB finder method, the early application servers would execute as many as 21 SQL queries in the database: one for obtaining the keys and for each bean. And instantiating and returning an entity bean is also a costly operation.

So we had to invent a remedy. It consisted of two pieces:

For every entity bean *Mybean*, we would introduce a corresponding normal (i.e. non-remote) Java class *MybeanData* with the same set of attributes. We called these objects *Lightweights*. The J2EE design patterns today call this idea *Value Objects* or *Transfer Objects*. Each EntityBean *Mybean* would have a method *getByValue()* for obtaining a *MybeanData* instance and a method *updateByValue(MybeanData)* for setting all its attributes from a single *MybeanData* with just one remote call.

For reading multiple records from the database in a single step for browsing (read-only) purposes, we created a simple and very efficient object-relational mapping framework called *Retrieve*. Based on a SQL select clause declared in each Lightweight class, Retrieve can obtain large sets of *MybeanData* objects from the database quickly. Selection and ordering criteria are encapsulated in *Filter* and *Sorter* classes, respectively, so that the business logic is kept completely free from SQL code.

Retrieve is similar to what the J2EE design patterns today call a *Fast Lane Reader*.



With these additions, it was possible to create reasonably efficient applications.

Meanwhile the hype around EJB split into two groups of people: Those who had not yet tried out the technology were still mesmerized. The others were somewhat sobered, invented Value Objects themselves or dropped using EntityBeans altogether.

With our two additions (and a few extensions such as a *Service Locator*, called *IContext*), we were sufficiently satisfied for a while and just took EJB for granted: There were plenty of EJBs in our software and it worked.

7. Evolution phase 3: Emancipation

"EJBs shouldn't be central at all."

However after a while it became more and more clear that the Value Objects were much more important for most of the application programmers most of the time compared to the EntityBeans themselves. So we turned the previous view of EJB use upside down and made the following changes to our architecture and APIs:

- We now considered the Lightweight to be the business object, rather than the EntityBean.
- Consequently, we now named the Lightweight *Mybean* (rather than *MybeanData*) and the EntityBean *MybeanEJB* (rather than *Mybean*) and stowed the EntityBeans away out of sight in separate subpackages.
- All business logic used Lightweight objects only, never EntityBeans.
- Access to EntityBeans was restricted to *Manager* classes. A Manager class provides explicit create-read-update-delete (CRUD) functionality for one or a few types of closely related business objects. Its implementation uses a SessionBean; for representing the business objects, its API reflects the respective Lightweight classes only; its implementation uses EntityBeans.
- The Manager class is not a SessionBean itself, it only uses a SessionBean for its core functionality.

This kind of Manager services and their implementation is a combination of both, *Session Facade* and *Business Delegate* patterns in the J2EE patterns catalog: a mechanism that decouples the EJB tier from its clients. The architecture has now effectively downgraded enterprise beans from the heart and soul of the architecture to a mere implementation mechanism.

By this time, the EJB hype had largely worn off. Some people now viewed EJB as an unnecessary and expensive luxury, others understood that it is well-suited for situations where strong scalability and availability is required and where transaction security is critical, such as in financial services. The latter group had begun to search for ways of using EJB in a selective manner.

8. Evolution phase 4: Independence

"Use EJB only if you really want to."

We searched, too, and found that given the previous changes, the rest of the way was not very long.

So far, we already had provided an EJB-free client API. Only those few programmers who needed to modify or extend one of our components would still see the EJBs (in the separate extension API).

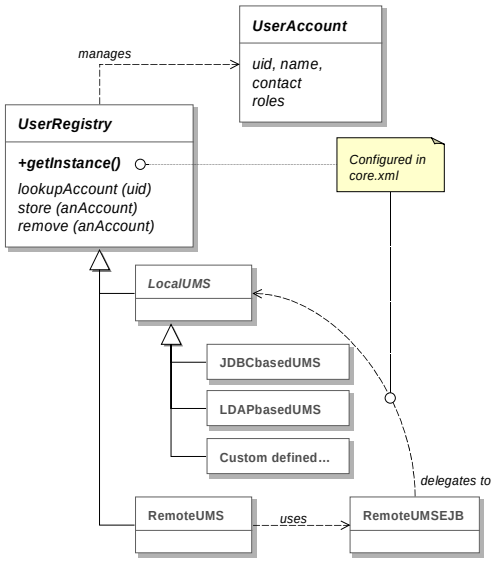
The remaining step was to introduce what we now call EJB-on-demand[1]: Simply by setting a load-time configuration option, programmers can (at least for many of our components) decide on a service-by-service basis where EJBs are to be used and where a plain Java solution should be used instead. The latter does not have the added value gained from the management features of the EJB container, but also requires fewer resources and, if no EJB services are used at all, simplifies the infrastructure and its operation enormously.

In terms of the product implementation, this is relatively easy to achieve for the services: In EJB mode, the client API *Myservice* is implemented by a simple wrapper *RemoteMyservice* that delegates to the SessionBean *MyserviceEJB*, which in turn delegates to the core plain Java solution *LocalMyservice* (which does the actual work). In non-EJB mode, *Myservice* is directly implemented by *LocalMyservice*.

Which of these two options is active is hidden by the Service Locator *IContext*, which is called to instantiate a *Myservice* object and will return either a *LocalMyservice* or a *RemoteMyservice*. Obviously, this scheme can be

[1] This is a generalization of a notion introduced by Gartner Group as *dual topology* [1]: Application Servers should be offered in a variant for lightweight applications not using EJB as well as one for full-blown J2EE applications including EJBs. *EJB-on-demand* extends this idea to application components.

extended to cover still other possibilities than just an EJB versus a non-EJB implementation, as is exemplified by the figure shown below about our User Management Service (UMS) that implements the UserRegistry interface.



For Manager services, that is, if EntityBeans are involved, the EJB-less implementation requires a little more work: MyserviceEJB will then directly implement the service using EntityBeans, whereas LocalMyservice is a separate implementation using JDBC or whatever other persistence mechanism is to be used.

And the EJB hype? As of today, where the pros and cons of EJB are much better understood, many projects think quite hard about whether they should employ EJB or rather try to get by without it.

EJB-on-demand is a nice way out of this dilemma: One can build an application in such a way that it can start without EJB and be upgraded later, but largely without rework of any client code. And the external application components (if it's abaXX.components) can just be switched to EJB mode without any additional implementation work at all.

9. Conclusion

Here is a summary of the lessons we learned from the above during over 3 years of developing abaXX.components.

About following a hype: Following a technological hype (either with the masses or even ahead of the masses) may be viable, but not necessarily efficient. If we had taken the time to evaluate EJB more thoroughly before building the large number of components that we did, we could have gotten to where we are with less effort.

About developing an API-based software product: The most dangerous aspect of starting with a less-than-perfect architecture for a high-level API-based software product is backwards compatibility. There were 8 releases of abaXX.components so far, two of them not being backwards compatible to the one before. The first of these steps was relatively easy; we made it after just half a year, with only two customers involved. The recent second one was quite difficult, as our customers now have systems in operation that represent over 80 Million US$ of investments. We have managed to contain incompatible changes to a small number of places in the API and are providing a backwards compatibility add-on package, but we don't yet know how costly the migration of a large project solution really will be and whether our customers will do it.

About the EJB technology: After all we have done, we are still unsure whether we love or hate EJB technology. Some of the promises hold true: EJB and J2EE application servers are now a rock-solid foundation for business-critical, large-scale, transaction-intensive systems. On the other hand there are so many design flaws, omissions, oddities, and pitfalls in EJB that it is really hard to be enthusiastic about it [2]. One thing, however, we are sure about: If one is to build a suite of reusable application components based on EJB, it is a good idea to hide the EJB aspects of the implementation behind simple, non-EJB interfaces.

References

[1] Yefim Natis: *Application Server Scenarios: From Stovepipes to Services*, Gartner Research AV-14-5983, October 2001.

[2] Robin Sharp and Dinu Fancellu: *101 Damnations of EJB*, http://www.softwarereality.com/programming/ejb, February 2002.

[3] Sun Microsystems: *Enterprise JavaBeans 2.0 Final Specification*, September 2001.

[4] Sun Microsystems: *J2EE BluePrints Design Patterns*, http://java.sun.com/blueprints/patterns.