Custom vs. Off-The-Shelf Architecture

Robert C. Seacord
Kurt Wallnau
John Robert
Santiago Comella-Dorda
Scott A. Hissam

May 1999

COTS-Based Systems Initiative

Technical Note
CMU/SEI-99-TN-006

Contents

List of Figures

Abstract

Members of the COTS-Based System Initiative at the Software Engineering Institute have developed the Generic Enterprise Ensemble (GEE), a generic approach to building distributed, transaction-based, secure enterprise information systems (EIS). GEE is a tool to help in the selection of technologies and architectural choices when building Enterprise Information Systems. Enterprise JavaBeans^{TM} (EJB) is a specification from Sun Microsystems for an application server based on Java technology. In this paper, a comparison is made between GEE based solutions and off-the-shelf solutions based on the EJB specification.

1 Introduction

The COTS-based systems (CBS) initiative at the Software Engineering Institute has worked with a number of customers to help develop large enterprise systems. This experience enabled the project to recognize common characteristics in these systems. In general, these systems were all mission-critical distributed information systems with transactional and security requirements. Furthermore, these systems consisted of clients requiring remote, concurrent access to multiple, heterogeneous databases. The CBS initiative undertook the development of the Generic Enterprise Ensemble (GEE) to provide a framework and guidelines for developing this class of system from commercial off-the-shelf (COTS) components.

The GEE consists of a collection of *flex points* that represents "flexible" areas in system requirements. For example, confidentiality may be a flex point in a given enterprise application that needs to encrypt data. Each flex point consists of multiple *design options* – each representing possible design approaches for implementing the functionality of the flex point. Figure 1 shows a sample flex point for content delivery in the GEE. The content delivery flex point defines three different design options for delivering system content to an end-user of a system allowing for variation in network configurations, client platforms, and application features. The details of the servlet, applet, and application design options are illustrated in Figure 2.
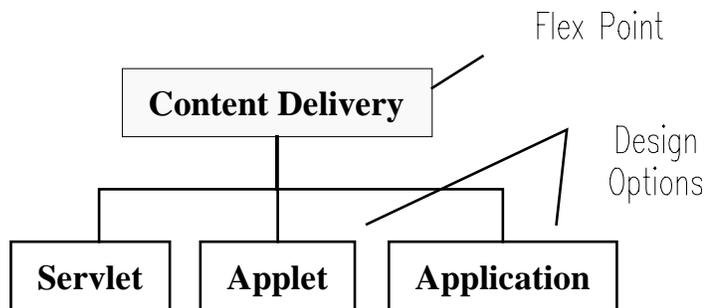
Figure 1 Sample flex point with design options

Each design option has one or more *implementation options*. Implementation options in the GEE are normally represented as COTS-products choices, although at times custom solutions are described. Implementation options for the applet design option, for example, include a collection of products based around Netscape Navigator, Microsoft Internet Explorer, or the Java Plug-in—technology from Sun that allows an applet to use Sun's Java Runtime Environment (JRE) instead of the web browser's default virtual machine.

Servlet | Applet | Application

Applet
JVM

Application

Browser | Browser | JVM

HTTP | HTTP

Servlet

Client HTTP Server | HTTP Server

Business Services IIOP | IIOP | IIOP
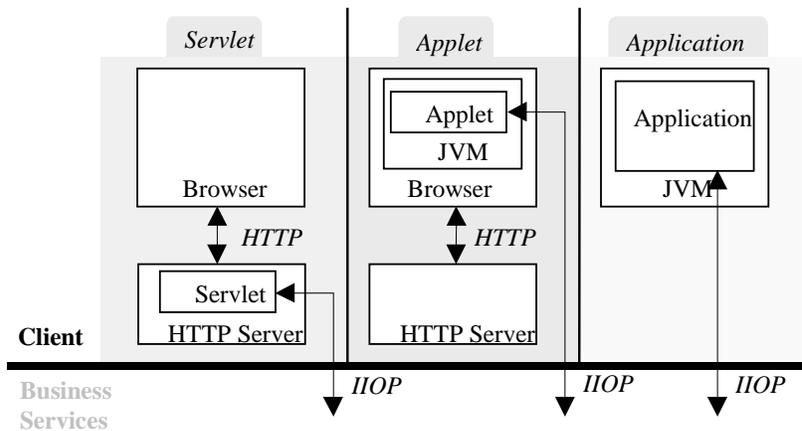
Figure 2 Content delivery design options

Ideally, a system architect could simply select a collection of flex points. Each of these selected flex points provides the architect with design options and COTS products as implementation choices. According to these choices and based on system requirements, the architect can choose an appropriate set of implementation options. Of course, things are never this simple, as relationships and tradeoffs exist between design options and more often between implementation options. For example, it is often the case that products from one vendor integrate poorly with products from another vendor.

In reality, there is a need for *roadmaps* that can help to identify collections of mutually compatible design and implementation options. Each of these roadmaps defines a system architecture. The GEE, in fact, constitutes a selection tool used to choose an appropriate architecture, based on system context and COTS product selection. When implementing a solution using the GEE, the system integrator makes an informed selection of technologies and implements the system using those selections. In contrast, Enterprise Java Bean (EJB) can be thought of as a predefined "off-the-shelf" architecture. EJB identifies a fixed collection of pre-integrated technologies that are applicable to a broad range of enterprise systems.

In this paper, we compare our experiences building a model application using the GEE, and building the same application using EJB. The model problem selected was based on a shipping company that developed an n-tier application that allows both employees and customers of that company to track and review information about packages being handled by the company. This model application included web-based access, defined user roles, and secure access to distributed, heterogeneous relational databases. The comparison revealed some of the fundamental tensions between the desire to define and control system architecture, versus the convenience of outsourcing architectural decisions to commercial product vendors.

2   Custom Integration

The GEE is a method for building COTS-based systems that is flexible enough to satisfy the different application needs of mission-critical distributed information systems using best-of-breed technologies and components of interest to government and commercial enterprises.

The GEE consists of *An Illustrated Design & Engineering Handbook* and a collection of working model problems. The handbook describes key design & engineering issues of COTS-integrated systems using the "modern" enterprise as a foil to illustrate design & engineering approaches. Model problems distributed with the GEE can be used as application "starter kits."

As described in the introduction, the GEE delves down to the level of the integration of specific products. As such, the GEE is a wasting asset that must be continually maintained by a development organization. Maintenance of the GEE, however, can be best performed by a single, shared resource within an organization. Lessons learned from implementing organizations can be fed back to this group to improve future versions of the GEE. The model problems distributed with the GEE can be used to establish this technology testbed.

In our work with various customers to help develop large enterprise systems, we found that organizations needs to develop, communicate, and sustain a practitioner's view of the enterprise architecture. In general, we found insufficient prescription in the technical architecture beyond technology selection. In consequence, each project had to develop its own interpretation and design approach and no common "vision" is communicated to designers and engineers. We also found that most enterprises practiced an incomplete approach to technology evaluation. More times than not, this resulted in an initial infatuation followed by a lasting disenchantment with "technology-gizmos." The feasibility of integrating these technologies is tested in live projects, often with fatal consequence. An over emphasis on technology resulted in a corresponding lack of attention to business logic and requirement analysis.

The GEE addresses the lack of prescription in the technical architecture by providing concrete design patterns, sample implementations and use guidelines assist developers in early formative design activities. The incomplete approach to technology evaluation is extended by the introduction of an operational testbed that includes working implementations of GEE variants—encouraging planned and rigorous technology evaluation. Worked-out design alternatives helps the developers to focus on business logic and not infrastructure implementation.

The GEE starts with a conceptual design—an n-tiered system that provides application engineers with high-level design guidelines for example, the role of business objects relative to business services. The real focus of the GEE, however, is on the use of technical and information architectures to build applications.

As discussed in the introduction, the GEE is adaptive through the use of flex points. These are pre-planned to provide controlled adaptation. The GEE supports two kinds of flex point: design flex points that support GEE adaptation to application-specific requirements and implementation flex points that support GEE adaptability to alternative infrastructure technologies

In the implementation of the model problem using the GEE we selected a collection of implementation options that we considered being both representative and best-of-breed. Specific technologies used in the implementation include: Java 2 Platform (JDK 1.2), Java 2 Plug-in, VisiBroker ORB, versions 3.2 and 3.3, VisiBroker Integrated Transaction Service (ITS) , VisiBroker Secure Socket Layer (SSL) Pack, RSA CryptoJ, Netscape Digital Certificates, Netscape Directory Server (NDS), VisiBroker COSNaming service, Java Naming and Directory Interface (JNDI), Oracle 8, and Microsoft and Netscape HTTP browsers and servers.

This product and technology ensemble was reasonable and representative of selections made frequently in practice. Surprisingly, we were unable to integrate this ensemble to the degree that we had initially expected. We anticipated a variety of integration challenges, but we were not prepared for the range of integration "dead ends" that we encountered. Some illustrative examples are included in the following sections.

2.1   Problem #1: Integrating Java 2 Plug-in and VisiBroker

The GEE provides several design options for delivering system content to an end-user of a system: *applets*, *applications,* and *servlets*. Applets are Java classes that run within the JRE integrated into a client's browser. Applications are initially installed on the desktop but can be run any number of times without additional downloads. Servlets are small, platform-independent Java programs that can be used to extend the functionality of a Web server. Servlets differ from applets in that servlets do not run in a Web browser or with a graphical user interface (GUI). Instead, servlets interact with the servlet engine running on the Web server through requests and responses. The request-response paradigm is modeled on HTTP.

A number of the customers with whom we have worked to develop their enterprise systems have made the decision to deploy their clients as applets. The reasons frequently given for this selection include

- Browsers provide a familiar user interface for our clients.

- Applets support the creation and deployment of sophisticated, dynamic GUIs.

- Applets can communicate (securely if necessary) with backend servers using CORBA, RMI, or other remote interfaces.

The first reason, in particular, is based on the assumption that browsers are a universal paradigm [Seacord 98], are already installed on the client desktop, and are used to interface with a variety of other applications and Web sites. Resultantly, applets are expected to work with the browser the client has previously installed on their desktop. Unfortunately, the JDK release of the JRE environment in the browser can be incompatible with the release used to implement the applet. This is aggravated by the fact that most of the browsers seldom include the latest JRE versions. We found exactly this problem when implementing the GEE version of the model application. The applets in this system were implemented using the Java 2 platform, and this JRE was not integrated into either Navigator or Internet Explorer (IE). Fortunately, this problem could be addressed through use of the Java Plug-in. The Java Plug-in allows Java applets to run using Sun's JRE instead of the browser's default virtual machine.

Unfortunately, the Java 2 Platform also incorporates an object request broker (ORB) that conflicts with the VisiBroker ORB[1]. To execute a Java 2 application (or applet) that uses VisiBroker, it is necessary to use the `Xbootclasspath` flag as an option to the Java Virtual Machine (JVM). The `Xbootclasspath` flag is used to set search path for bootstrap classes and resources. This allows the VisiBroker ORB classes to be searched before the internal JDK libraries, ensuring that CORBA calls are bound correctly to VisiBroker classes. This flag works fine from the command line or from the applet viewer included with the Java 2 platform, but *does not* work from the Java 2 Plug-in. The elimination of this flag effectively prevents the development of applets that use the Java 2 platform to communicate with a VisiBroker ORB. According to Tom Ball, the Java 2 Plug-in development manager, this flag was purposely eliminated as it represented a security hole. There is, of course, no problem using the Java 2 ORB in this scenario, except that it may lack the features or the functionality of the VisiBroker ORB (interceptors, for example) that may be useful elsewhere.

This problem is an example of overlapping functionality and unclear product boundaries. In adding an ORB to the Beta 3 release of JDK 1.2, Sun has taken steps to make CORBA and IIOP ubiquitous on the Web, but has already impaired the customers ability to work with use CORBA implementations from other vendors.

---

[1] An unorthodox solution to this problem, suggested by Inprise, is to remove the ORB classes from Java 2 runtime jar files.

2.2 Problem #2: Integrating COSNaming and JNDI

The GEE uses JNDI to provide a federated name space consisting of the NDS LDAP server and the VisiBroker COSNaming service. Federation of VisiBroker COSNaming was impossible due to a lack of interoperability between the VisiBroker naming service and the JNDI CORBA Service Provider Interface (SPI). The client was unable to get the context for the VisiBroker naming service because VisiBroker did not implement the interoperability proposal submitted to the Object Management Group (OMG) by Sun, IBM, and others [INS 97]. Specifically, JNDI assumed a standard format for identifying the host and port number of the form:

```
iiop://host:port
```

The VisiBroker naming service does not currently support this format, and in fact Inprise, along with IONA, BEA, and DSTC have submitted a competing proposal for an interoperable naming service to the OMG [INS 98]. There is some hope that outstanding issues will be resolved prior to the release of the CORBA 3.0 specification from the OMG.

This problem does not exist using the JavaIDL naming service included with the Java 2 platform since the vendor is the same in both cases (Sun) and both products implement the standard proposed by that vendor.

2.3 Problem #3: Integrating Digital Certificates and JNDI

The NDS is used as a repository of digital certificates for client authentication purposes. As part of the initial client authentication, Netscape provides a single sign-on solution using digital certificates and an optional certificate server. For single sign-on, the Netscape server has a list of authorized clients and their certificates containing public keys. When a client authenticates to the server, the client presents data signed with its private key, and the server verifies this signature using the public key in the certificate list.

However, using JNDI 1.1.1 and LDAP SPI 1.0.2, a client can not use certificates (strong authentication) to authenticate to the Netscape Directory Server 3.1.2. The problem is that LDAP SPI does not support certificates. During a period of several months when this problem was being investigated by the project, several new versions of Netscape, JNDI, and LDAP were released. Although these new versions have been substantially modified, none of these changes has addressed this problem.

It is possible to circumvent JNDI and use LDAP directly as a repository for digital certificates by removing JNDI. This eliminates the benefits derived from its use—a single client interface to a federated name space.

2.4   Problem #4: VisiBroker ITS

A serious limitation exists with the use of the VisiBroker ITS product, which is a pre-integrated solution for providing Java-based transaction services to relational databases from CORBA servers. Resultantly, it suffers from some of the same integration problems that we experienced with our EJB solution. Specifically, the JDBC version 1.x driver used by ITS does not support the interfaces required to perform two-phase commit. [Inprise 98] Failure to support two-phase commit makes it impossible to support transactions to multiple databases, even homogeneous databases. VisiBroker ITS did support two-phase commit in their C++ product line, but only for Oracle v7.3, while GEE was using Oracle v8.0.

On the one hand, these illustrate the same old lament that integrating COTS software is "hard." On the other hand, our less painful experience (to date) with EJB offers a counterexample that is worth noting.

3 Vendor Integration

The JavaSoft EJB specification defines a component architecture for building distributed, object-oriented business applications in Java. The EJB architecture addresses the development, deployment, and runtime aspects of an enterprise application's life cycle. Figure 3 is Sun's depiction of the EJB architecture at the JavaOne '98 conference.
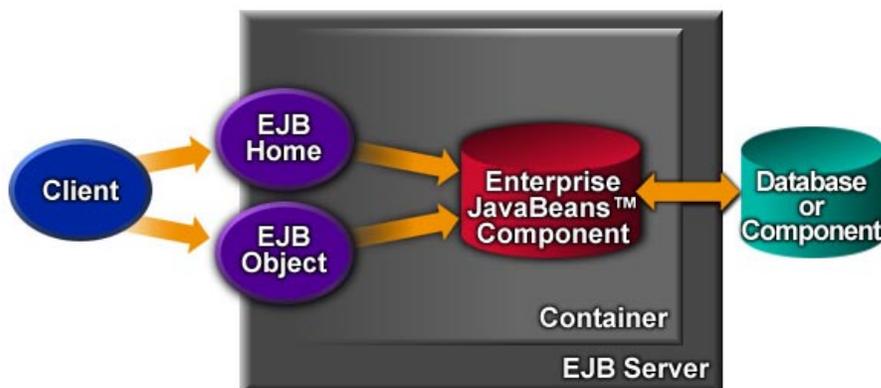
Figure 3 EJB Architecture

An Enterprise JavaBean^{TM} encapsulates business logic. Each enterprise bean is deployed within a component framework, or *container*, that manages the details of security, transactions, connection management, and state management. An EJB container handles low-level details such as multi-threading, resource pooling, clustering, distributed naming, automatic persistence, remote invocation, transaction boundary management, and distributed transaction management. This allows the EJB developer to focus on the business problem to be solved. An Enterprise JavaBean is, in essence, a transactional and secure remote method invocation (RMI) or CORBA object, some of whose runtime properties are specified at deployment using special classes called "deployment descriptors."

The EJB-based version of the model application was implemented using the BEA WebLogic 4.0 EJB server. WebLogic implemented the EJB 1.0 specification, including optional features such as support for entity beans and container-managed persistence. Interestingly, the technologies integrated by WebLogic are very similar to those selected for the GEE-based version of the model application, including JNDI, JDBC, SSL, Access control lists (ACLs) and HTTP servlets.

Most of the integration problems we encountered in our best-of-breed integration approach were already solved by the pre-integration of these technologies by BEA. BEA resolved these integration issues by developing their own versions of key technologies, including RMI, JNDI and SSL. As a result, the problems we encountered in our EJB-based implementation of the model problem were, for the most part, independent from the problems encountered in our custom integration.

3.1 Enterprise JavaBean Types and Persistence

The EJB 1.0 specification [Matena 98] defines two types of enterprise beans: *entity beans* and *session beans*. Support for session beans is mandatory for EJB 1.0 compliant containers. Support for entity beans was optional in EJB 1.0, but has become mandatory in EJB 1.1 [Matena 99].

Session beans exist for the life of a single client and the EJB server. Entity beans are persistent objects that may be used by many users across crashes or shutdowns of the server [Morgan 98].

An entity bean implements an object view of an entity stored in an underlying database or an entity implemented by an existing enterprise application. The protocol for transferring the state of the entity between the instance variables of an enterprise bean and the underlying persistent representation is referred to as *object persistence*.

The EJB specification allows the bean producer to implement the bean's persistence directly in the bean (bean-managed persistence) or delegate the bean's persistence to the container (container-managed persistence). In bean-managed persistence, the bean producer implements object persistence logic directly in the bean, including the creation and finder methods. Creation methods must create records in the persistent store from data passed in as arguments to the method. Finder methods must be able to formulate the proper queries to locate the correct records in the persistent store and return a primary key or keys. In addition to modification of creation and finder methods, the bean producer must also specify methods to refresh the bean from the persistent store, to store the bean in the persistent store, and to remove the bean from persistent store.

In container-managed persistence, the container-provider's tools are used to generate code that moves data between the bean's instance variables and a database or an existing application. The enterprise bean provider must specify the *container-managed fields* property in the deployment descriptor to specify the list of instance fields for which the container-provider tools must generate access calls. Using WebLogic, container-managed fields in the deployment descriptor are specified as follows:

```
containerManagedFields [prefixId prefix]
```

The mapping between instance variables and the underlying database is also specified in the deployment descriptor. The following example shows that the instance variables `prefix` and `prefixId` are mapped to corresponding columns in the `ejbPrefix` table in a JDBC-managed database:

```
(persistentStoreProperties
 persistentStoreType        jdbc
  (jdbc
    tableName                ejbPrefix
    dbIsShared               false
    poolName                 ejbPool
    (attributeMap
       prefix                prefix
       prefixId              prefixid
    ); end attributeMap
  ); end jdbc
); end persistentStoreProperties
```

WebLogic provides a very restricted object to relational database mapping. Instance fields in WebLogic can be mapped to a single table row in a particular table in a database. It is not possible, for example, to create a bean using container-managed persistence that consists of data from two tables with a shared a primary key. The mapping capabilities of EJB servers are completely vendor-dependent, the EJB specification being quiet on the subject. Arbitrarily complex mappings can be achieved using bean-managed persistence. The problem is that, in our model application, implementing bean-managed persistence in a single entity bean required over 600 lines of code, whereas container-managed persistence could be implemented trivially in less than 20 lines. In a large, enterprise system with hundreds or thousands of entity beans, the difference in development and maintenance costs between the two persistence models would be staggering.

EJB server vendors appear to be deferring support for complex object to relational database mappings to existing companies that specialize in this technology. TOPLink for BEA WebLogic, from The Object People, claims to work with WebLogic and to support 10 different mapping types between entity beans and database tables including: direct-to-field, 1-1, 1-many, and many-many mappings. This technology is not yet generally available, and we have no direct experience to support these vendor claims.

3.2   Security

The Enterprise JavaBeans architecture is designed to shift most of the burden of implementing security management from the bean to the EJB container and server. In particular, an enterprise bean's deployment descriptor includes access control entries that allow the container to perform runtime security management on behalf of the enterprise bean.

WebLogic offers encrypted, authenticated connections using SSL. Authentication takes place at two levels: first, verifying that the communicating parties are who they say they are, and second, verifying that each message exchanged is from the expected sender and has not been altered.

WebLogic also provides ACLs to complement SSL features. ACLs provide a framework for protecting resources. An ACL is a list of rules for access to operations on certain objects by individuals and groups. An ACL allows an organization to define and organize permissions and provides the framework for checking those permissions at runtime. If you are using access control lists, you can write or adapt a realm that handles certificate-based authentication.

To authenticate clients in a servlet or other server-side Java programs, the servlet developer needs to verify that the client presented a certificate, and if that certificate was authorized by a trusted issuer.

Since applets cannot easily access the certificate browser in which they run, authentication from an applet is more difficult. WebLogic provides a special servlet that offers a solution by allowing applets indirect access to the browser's certificate via the WebLogic Server. The servlet captures the browser's certificate and reveals it only when a specific 64-bit token is presented. It then creates an HTML page with an applet tag that passes the token as a parameter.

The goal of the Enterprise JavaBeans architecture, as stated earlier in this section is to shift the burden of implementing security management from the bean to the EJB container and server. It is our opinion that this goal has not been realized. The EJB specification makes no provisions for providing a secure client-side interface other than existing Java programming language security APIs defined in the core package `java.security`. These APIs, by themselves, are insufficient when managing a client side key store, supporting certificate management, and providing a means of establishing a secure authenticated connection to the EJB using SSL or any other security mechanism.

Security in WebLogic depends largely on the use of SSL and the use of browsers. WebLogic does not provide the client side SSL interfaces that can be used to provide a secure, authenticated connection between the client and the enterprise bean.

When using a browser, a user name and password is used for primary authentication after which an SSL connection can be established using certificates managed by the browser. The user name and password are sent in the clear, unless the bean provider provides the code to snoop-proof the transactions. The password is used as a handle to identify a unique user and to determine which system resources that user can access, based on ACLs. These passwords are stored in clear text in the WebLogic property file. No mechanism is provided in the ACL

implementation to allow resources to be accessed using authenticated digital certificates rather than passwords, which are less secure.

Use of SSL and ACL in WebLogic is largely disconnected with no overriding security policy. ACLs are based on user name and password whereas SSL is based on digital certificates. Although EJB has made a promising start (by promising to provide security), significant effort is required by both Sun and EJB vendors to provide a usable, out-of-the-box security solution.

3.3   Distributed Transactions

As described in Section 2.4, JDBC 1.2 does not support XA two phase commit. As a result, it is impossible for an EJB server using JDBC 1.2 to provide direct support for distributed transactions.

It may be possible to implement distributed transactions to a homogenous database by using an underlying XA manager within a particular database. Oracle, for example, allows you to refer to a table on another server. This is transparent to the both WebLogic and VisiBroker ITS so they are unimpaired by the lack of a two-phase commit in either product.

Support for distributed transactions has been added as an extension to the JDBC 2.0 API [White 98]. This feature allows a JDBC driver to support the standard two-phase commit protocol used by the Java Transaction Service (JTS).

Supporting distributed transactions in heterogeneous databases requires a transaction coordinator that understands two-phase commit, but more importantly it requires JTA-compatible transactions resources to support distributed transactions across heterogeneous databases, messaging services, and other data resources.

Adding support for two-phase commit to a database driver is difficult, especially without the cooperation of that vendor. In most cases, developers are relying on the vendor to provide the database driver—Oracle for an Oracle JDBC driver and Sybase for a Sybase JDBC driver. Currently, middleware vendors are far ahead of the database vendors in supporting distributed transactions. For Java-based distributed transactions to succeed, database vendors will need to catch up.

3.4   EJB Evolution

EJB is a relatively new technology, but based on our experiences with Java, CORBA, and other technologies, it is possible to anticipate some trends. Current EJB servers implement version 1.0 of the EJB specification. This technology is extremely new and evolving rapidly. For example, during the course of our experiment (a period of several months) several minor revisions of WebLogic were released, the original vendor (WebLogic) was purchased by another company (BEA), the name of the product changed (from Tengah to WebLogic), and

a major version (version 4.0) was released. New releases of WebLogic appear to correct bugs with earlier features and add functionality but also introduce new bugs. Public drafts of version 1.1 of the EJB specification were made available in late spring of 1999, which is significantly expanded from the EJB 1.0 specification. Major changes from the EJB 1.0 specification include:

- support for entity beans has been made mandatory

- tightening and clarification of the specification in some areas to make implementations by different EJB server vendors more consistent

- modifications to the content of the deployment descriptor to better map into the bean provider, application assembler and deployer roles

- replacement of the JavaBeans^{TM} architecture-based deployment descriptor format with an XML technology-based format.

As a result of these and other changes EJB products may change radically – requiring significant revision of existing applications. Major changes will also result from the introduction of the EJB 2.0 specification in 2000 and the migration from the JDK 1.1 base to the Java 2 Enterprise Edition (J2EE) platform.

4 Comparison

In this section, we provide a comparison between building an application using the GEE and using EJB based on our experiences in developing the model application using both technologies. In particular, we examine the impact of both approaches on a number of development and maintenance tasks in an attempt to characterize the overall impact of the paradigm shift.

In comparing the custom integration and EJB solutions, it is interesting to note that our system objectives were not achieved in either case. Both approaches failed to support distributed transactions across homogenous databases, let alone heterogeneous data sources.

4.1 Product Selection

In component-based software engineering, the software development process includes the evaluation and selection of COTS components.

Product selection is a lengthy and arduous process. The first step of the process is to determine what are the best-of-breed components. The next step in the process is to determine if these products can be integrated, either directly, or through wrappers or other "glue" code. Determining if products can be integrated is also a complex process, as vendor claims are not always believable. If these products can not be easily integrated, it is necessary to consider alternate products that may not be best-of-breed but are compatible with other technologies. To make matters worse, the environment is constantly changing with new products and emerging product versions, existing products going away or being refocused, and evolving vendor relationships. Use of EJB simplifies this process by removing the combinatorics involved in the selection process.

Today, a number of vendors exist that have products that support EJB technology or have announced support for EJB in future product releases. Among these vendors are BEA Systems, IBM, Inprise, Netscape, and Sun Microsystems. Each of these products represents a pre-integration of components. These pre-integrated systems can then be evaluated as a whole, simplifying the evaluation and selection process. In effect, the granularity of products being evaluated has increased.

Selection of an EJB approach takes the decision as to what constitutes a best-of-breed solution out of the hands of the system integrator. If, for example, BEA had the best support for distributed transactions while IBM had the best object persistence, it is unlikely that these individual components could separated. An even greater dilemma may arise if one

vendor had the best overall solution, but was lacking in a critical feature (such as strong authentication) that prevented its use. In this case, it might be necessary to use a product that was largely not competitive, but offers this one special feature. This exposes the customer to other risks—that the selected product will not have sufficient market share to survive.

4.2   Changes to Business Logic

For many enterprise systems, business logic tends to evolve over the life of the system. An example of this is a system for preparing tax returns, where changing tax laws require that the business logic be updated each year. More commonly, business logic needs to be re-implemented to reflect evolving business practices.

Changing business logic in EJB is considerably easier than in the best-of-breed solution. This is true because EJB provides a relatively well-defined component model [Brown 98] that promotes the separation of business logic from the underlying services such as transactions, persistence, and security. Although the GEE does implement wrappers to these services, these calls are often interwoven through the business logic. This requires the maintainer to understand the purpose and function of these services to ensure that changes to business logic do not impact their operation.

In EJB, the container automatically provides most of these services. When the container fails to provide these services in a robust and flexible manner, business logic can become intertwined with calls to system services as in the GEE. Container-managed persistence, for example, allows the resulting system to be independent from the underlying data source. If the mapping between an enterprise bean and the underlying data source is too complex to be managed by the container, we have to hand code persistence using bean-managed persistence. In the process, the independence between our system and the data source is lost and it becomes subsequently harder to adapt the enterprise bean to a different data source.

4.3   Upgrading

The one constant aspect of component-based software development is change. Constituent components are constantly evolving–new products are emerging while existing products become dated or obsolete. Keeping existing systems "current" is a difficult but necessary maintenance task. In maintaining a best-of-breed solution, the maintainer needs to continually monitor the marketplace to evaluate new technologies, new products, and new releases of component products and upgrade the system when appropriate. These skills are similar to skills required to initially develop these systems.

In the case of EJB, the server vendor is in reality maintaining a best-of-breed solution, even if some or all of the components are developed in-house. The EJB vendor decides which component versions to include and performs the integration and testing.

Rather than track new releases of all component products, a person maintaining an EJB-based system only needs to be interested in new releases of the EJB server used. EJB servers from other vendors that may overtake the previously selected server in functionality, performance or other "ilities" should also be monitored, in the event that a significant delta in capabilities would justify a migration. Implications of migrating between EJB server implementations are discussed later in this section.

While an EJB solution simplifies the problem of tracking the evolution of multiple component products (and their competitors), it also reduces the flexibility system maintainers have in selectively upgrading component versions to incorporate new functionality.

4.4   Changing Vendors

The promise of open systems is that a system integrator or maintainer can switch to an alternate vendor if the initial vendor fails to support, maintain, and enhance a product in a satisfactory manner. Open systems are based on standard interfaces–allowing an application to be quickly ported to an alternate implementation. Both the GEE and EJB solutions are based on a collection of open standards, including CORBA, RMI, Internet Inter-ORB Protocol (IIOP), and SSL. But, how easy is it to port these systems to alternate implementations of these standard interfaces?

During the development of GEE, we found that it is often difficult to intermix technology from different vendors. For example, the two principal browsers in use today, Microsoft Internet Explorer and Netscape Navigator both support SSL, but differ significantly in the manner in which they support digital certificates [Seacord 98]. Changing one product can often lead to a cascading effect where numerous other products are impacted. This can result from something as simple as a new product requiring a later release of the JDK, in turn requiring that other products be upgraded to work in the same environment. Isolation and wrapping of components can be used to reduce the impact of these changes, but this is not always feasible.

As EJB is a standard, one might expect that portability between EJB implementations would not be an issue. Unfortunately, the EJB specification is extremely porous in many areas and deliberately vague in others. This may have resulted from an attempt to accommodate existing differences in application servers or to provide an opportunity for EJB server vendors to differentiate them in the market. Regardless of the cause, holes in the specification provide the rocks upon which source code portability crashes and sinks. In the EJB specification, for example, there is no mention of what virtual machine should be used. As a result, existing EJB servers are based on both JDK 1.1 and JDK 1.2. EJB is defined largely by underlying JDK APIs such as JTS, JDBC, RMI, JIDL, JNDI and JMAPI[2]. These

---

[2] See Appendix- Acronym List.

APIs have changed significantly in JDK 1.2, and as a result, portability between EJB servers based on different JDKs is seriously degraded.

Problems porting today from EJB servers based on JDK 1.1 to EJB servers based on JDK 1.2 should be indicative of problems upgrading to new JDK 1.2 versions of EJB servers in the future.

We have also found more subtle differences in other interfaces. For example, the RMI API can use different middleware protocols, like the native Java Remote Method Protocol (JRMP) or IIOP. Unfortunately, different capabilities in IIOP and JRMP make it difficult to hide the underlying protocol from users of the RMI API.

While many of these problems are easily discovered and remedied, semantic differences also exist between EBJ servers, particularly in what can be expressed in the deployment descriptors. For example, some servers support user control over object pools and some do not. The deployment descriptor for servers that support pooling control have a property for the number of instances of a bean in the pool, while this property is absent from deployment descriptors of servers without pooling control. The only solution, in this case, is to move functionality from the deployment descriptor to the source code—which could be a rather unpleasant task.

Portability for both the GEE and EJB solutions can be problematic, depending on the extent to which non-standard extensions to products are used, and differences in vendor interpretations of specifications. It may be easier in GEE to replace a given component, such as a browser than to make a wholesale port from one EJB vendor to another since all aspects of the system may be effected.

5 Conclusions

In general, we have had greater success using technologies that have been pre-integrated by a single vendor than we have had integrating best-of-breed products ourselves. While this could be simply explained as a lack of skill on the part of our integration team, we would like to believe that there are other factors involved.

Integration vendors have options available to them that are not normally available to application developers. An established middleware vendor such as BEA can leverage other vendors to resolve integration issues prior to releasing an integration framework.

BEA took a different approach to integration in that they developed their own versions of key technologies, including RMI, JNDI, and SSL. Although this approach allows BEA to seamlessly integrate their technology, this option is not normally available to an application development team due to cost and other factors. Interestingly, the BEA WebLogic product also fails to provide a 2-phase commit protocol because of their dependency on JDBC 1.2 [Tengah 99].

Given time and coinciding vendor interests, the problems we experienced in the custom integration of best-of-breed technologies can be resolved. However, the best-of-breed integration effort is effectively held hostage to the whims of vendor priorities—a state of affairs that is not always conducive to the use of these technologies in enterprise applications. Problems resolved for a particular set of product versions often resurface in different forms in later versions of these same products.

If an enterprise does not have a large number of product selection constraints that must be satisfied and can accept an integrating vendor's product choices, technologies such as EJB can be a good solution. In this case, the value added in integration can outweigh the constraints imposed by a fixed architecture or product constraint.

However, there are situations where best-of-breed or "latest and greatest" versions of products are essential to satisfy system requirements. In these cases it is important to maintain control over application architecture and technology selection decisions—regardless of increased integration costs.

Another dimension of the question of building or buying an architecture concerns the types of technology competencies an organization must build to define and maintain an enterprise information system. The technology competency required to integrate HTTP

browsers/servers distributed transactions, distributed objects (Java and CORBA), naming and directory services, system management, and database can be daunting. Moreover, this competency is a wasting asset—technologies change and force us to continually reacquire technology competency. If there is no overwhelming need to develop competencies in these areas, an EJB solution may allow an organization to focus on its key business processes.

Lastly, there are business implications associated with so-called "vendor lock." On one hand, EJB represents an "open" interface specification (one of the goals of EJB is to allow Enterprise JavaBeans to be deployable across all vendor containers). On the other hand, there is a slippery slope in using vendor specific enhancements that can effectively lock the evolution of the product to a particular product-line. Still, integration frameworks based on open system interfaces holds tremendous promise.

Appendix A: Acronym List

Acronym	Definition
CBS	COTS-based systems
COTS	commercial off-the-shelf
CORBA	common object request broker architecture
EJB	enterprise Java Bean
GEE	generic enterprise ensemble
IIOP	internet inter-ORB protocol
IOR	interoperable object reference
ITS	integrated transaction service
JDBC	Java database connectivity
JIDL	Java interface definition language
JMAPI	Java management API
JNDI	Java naming and directory interface
JRE	Java runtime environment
JRMP	Java remote method protocol
JVM	Java virtual machine
LDAP	lightweight directory access protocol
NDS	Netscape directory server
ORB	object request broker
RMI	remote method indicator
SSL	secure socked layer

Acknowledgements

The authors acknowledge the work of the GEE project team members: Tom Boyea and Fred Long.

References

Brown 98   Brown, Alan W. & Wallnau, Kurt C. "The current state CBSE." *IEEE Software*, (September/October 1998).

Inprise 98   VisiBroker Integrated Transaction Service Programmer's Guide, Inprise Corporation, Scotts Valley, CA. 1998

INS 97   Interoperable Naming Service, Version 1.5, June 6, 1997.

INS 98   Interoperable Naming Services, BEA Systems, Cooperative Research Centre for Distributed Systems Technology, Inprise Corporation, IONA Technologies, PLC, October 19, 1998.

Karpinski 98   Karpinski, Richard. "Sun to Fix Flaws in Java Beans." *Internet Week*, 736 (October 1998).

Matena 98   Matena, Vlada & Hapner, Mark. Sun Microsystems Enterprise JavaBeans, Version 1.0. March 21, 1998.

Matena 99   Matena, Vlada & Hapner, Mark. Sun Microsystems Enterprise JavaBeans, Version 1.1 Public Draft. May 7, 1999.

Morgan 98   Morgan, Bryan. *Enterprise Java Beans: Coming soon to a server near you.* Java World. Available WWW <URL: http://www.javaworld.com/javaworld/jw-06-1998/jw-06-distributed.html> (June 1998).

Seacord 98   Seacord, Robert C. & Hissam, Scott A. "Browsers for distributed systems: Universal paradigm or siren's song?" *World Wide Web Journal*, *Volume 1* (1998), Baltzer Science Publishers BV: 181-191.

Stehlo 98   Strehlo, Kevin. "Let Java Middleware Juggle Your Tiers." *JavaPro, (February/March 1998).*

Tengah 99   Using Tengah Enterprise JavaBeans, WebLogic. Available WWW <URL http://www.weblogic.com/docs/classdocs/API_ejb.html> (June 99)

White 98   White, Seth & Hapner, Mark. JDBC TM 2.0 API, Sun Microsystems Inc.,
                 May 1998.

REPORT DOCUMENTATION PAGE

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (LEAVE BLANK)	2. REPORT DATE July 1999	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE Custom vs. Off-The-Shelf Architecture	5. FUNDING NUMBERS C — F19628-95-C-0003

6. author(s)

ROBERT C. SEACORD, KURT WALLNAU, JOHN ROBERT, SANTIAGO COMELLA-DORDA, SCOTT A. HISSAM

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-99-TN-006

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/DIB 5 Eglin Street Hanscom AFB, MA 01731-2116	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12.A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS	12.B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

MEMBERS OF THE COTS-BASED SYSTEM INITIATIVE AT THE SOFTWARE ENGINEERING INSTITUTE HAVE DEVELOPED THE GENERIC ENTERPRISE ENSEMBLE (GEE), A GENERIC APPROACH TO BUILDING DISTRIBUTED, TRANSACTION-BASED, SECURE ENTERPRISE INFORMATION SYSTEMS (EIS). GEE IS A TOOL TO HELP IN THE SELECTION OF TECHNOLOGIES AND ARCHITECTURAL CHOICES WHEN BUILDING ENTERPRISE INFORMATION SYSTEMS. ENTERPRISE JAVABEANS™ (EJB) IS A SPECIFICATION FROM SUN MICROSYSTEMS FOR AN APPLICATION SERVER BASED ON JAVA TECHNOLOGY. IN THIS PAPER, A COMPARISON IS MADE BETWEEN GEE BASED SOLUTIONS AND OFF-THE-SHELF SOLUTIONS BASED ON THE EJB SPECIFICATION.

14. SUBJECT TERMS COTS, Generic Enterprise Ensemble (GEE), Enterprise Java Beans (EJB), Java	15. NUMBER OF PAGES 20
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL