Analyzing Enterprise JavaBeans Systems Using Quality Attribute Design Primitives

Anna Liu
Len Bass
Mark Klein

October 2001

Architecture Tradeoff Analysis Initiative

Technical Note
CMU/SEI-2001-TN-025

Table of Contents

List of Figures

List of Tables

Abstract

Quality attribute requirements such as those for performance, security, modifiability, reliability, and usability have a significant influence on the software architecture of a system. At the Software Engineering Institute, we are studying and codifying the relationship between quality attribute requirements and the architectural design strategies that impact their achievement. In CMU/SEI-2000-TN-017 [Bass 00], we introduced the notion of *quality attribute design primitives*. Quality attribute design primitives (or attribute primitives) are architectural building blocks that target the achievement of one—or sometimes several—quality attribute requirements. Our intent is to codify a fairly comprehensive set of attribute primitives in a manner that articulates how each attribute primitive makes its specific contribution toward the achievement of one or several attribute goals. We believe this will provide a very powerful "language" for constructing or analyzing software architectures in relation to quality attribute requirements. To determine the expressive and explanatory power of these attribute primitives, we will examine various classes of systems. This paper uses attribute primitives to examine the qualities of Enterprise JavaBeans (EJB)-based systems. In particular, we find that attribute primitives hold promise for providing insight into the quality attribute consequences of using various EJB infrastructure features.

1 Introduction

Architects need to understand their designs in terms of quality attributes. For example, they need to understand whether they will achieve deadlines in real-time systems, what kind of modifications are supported by their design, how the system will respond in the event of a failure, and so on. There are large and thriving attribute communities that study various quality attributes, but they each have their own language and sets of concepts. Architects think in terms of architectural patterns [Buschmann 96]. However, what the architect needs is a characterization of architectural patterns in terms of the factors that affect quality attribute behavior so that a software design can be understood in terms of those quality attributes.

What we present in this paper is an initial step toward having such a characterization. We provide a short list of architectural patterns and brief descriptions of how to understand them in terms of three quality attributes: modifiability, performance, and availability. We call these patterns *quality attribute design primitives*. These patterns are, of necessity, system independent. In order to use them, an architect must make them system dependent. This paper demonstrates the application of our list to a portion of the Enterprise JavaBean (EJB) specification. The difference between what we present here and our ultimate goal is both depth and breadth. We expect our final list of architectural patterns to be longer than the list presented here, although we do not expect the list to be unmanageably long. We also expect the final descriptions to exist in more depth than we present here and to include models that motivate the analysis. This paper, however, stands on its own as an example of how EJB features can be understood in terms of more general patterns and reasoning about those patterns.

The document is organized as follows: our initial list of patterns and descriptions is presented in the appendix, but we provide our motivations for it in Section 2. In Section 3, we discuss a portion of the EJB specification and identify the features that we will be discussing. Next, in Section 4, we present the quality attribute interpretation of the portion of the EJB specification that is based on our quality attribute design primitives. We close with some conclusions.

2 Quality Attribute Primitives

To reason about architectural patterns in quality attribute terms, we must be able to characterize quality attributes requirements precisely and give an example of how to reason about architectural patterns. We characterize quality attribute requirements using *general scenarios* and we identify the architectural patterns on which we are focusing as *quality attribute design primitives* (or *attribute primitives*) [Bass 00]. Attribute primitives are an extension of our earlier work on attribute-based architectural styles [Klein 99].

2.1 Quality Attribute General Scenarios

To be able to analyze and evaluate the quality of any system, we first need to characterize the various quality attribute requirements applicable to the system. Quality attribute scenarios serve this purpose [Bass 01]. For the same reason that use cases are essential in determining functional requirements, quality attribute scenarios are used to specify quality attribute requirements. For five important quality attributes (modifiability, performance, availability, security, and usability), we have enumerated a collection of quality attribute *general scenarios* that are intended to encompass all of the generally accepted meanings for these quality attributes. A general scenario is, in effect, a template for generating a specific quality attribute scenario. For example, two (abbreviated) modifiability general scenarios are

- "Changes to the platform occur."
- "Additional distributed users arrive at the system."

These scenarios are "general" in the sense that they are system independent. Collectively, general scenarios provide a system-independent checklist for quality attribute requirements. CMU/SEI-2001-TR-014 presents our initial attempt at a comprehensive list of general scenarios for modifiability, usability, performance, reliability, and security [Bass 01]. Of course, for any particular system or class of systems, not all of the general scenarios for a particular attribute will be relevant, and the analyst must identify those that should be considered and make those system specific.

To use general scenarios for a particular system or class of systems (in this case, EJB systems), they need to be made specific. For example, the above two modifiability general scenarios become

- The platform on which the EJB application runs changes. These platform changes include JVM change, operating system change, hardware change, database driver change, database change, EJB server and container change (across different vendor products, and version upgrades). The system needs to be modified to continue to provide current functionality.

- Additional online users connect to application server, possibly via a Web browser, volume of online requests fluctuates (e.g., increased sales before Christmas or the last few minutes of an online auction). The system should be able to handle the large/varying volumes of client requests with limited modification to the application.

2.2 Quality Attribute Design Primitives

Just as general scenarios provide a template for specifying quality attribute requirements, quality attribute design primitives are templates for “chunks” of architectural designs that target the achievement of specific quality attribute goals [Bass 00].

Attribute primitives provide building blocks for constructing architectures. However, they are building blocks with a focus on achieving quality attribute goals such as performance, reliability, and modifiability goals. Quality attribute design primitives will be described in a manner that illustrates how they contribute to achieving quality attributes. Therefore each attribute primitive will be described not only in terms of its constituent components and connectors, but also in terms of the qualitative and/or quantitative models that can be used to argue how it affects quality attributes. For this document, we are concerned with the following attribute primitives: Naming Server, Client/Server, Separation of Interface from Implementation, Connection Manager, Load Balancing, Replication, Transactions, Logging State Changes.

Consider, for example, the client/server attribute primitive. This is a collaboration between the provider and users of set of services. The attribute primitive separates one collection of responsibilities (the client’s) from another (the server’s). The consequence of this separation is enhanced modifiability; modifying the implementation of the services or modifying the number of servers providing services is invisible (at least in principle) to the clients. Moreover, the addition of new clients has no effect on the server.

The client/server attribute primitive has modifiability as one focus. When we write-up this attribute primitive, we will articulate what we mean by *modifiability* by describing various modifiability general scenarios for which the mechanism is well suited, and we will make qualitative and/or quantitative arguments as to why it is well suited for these scenarios.

In addition, the effect of the client/server on other attributes must also be considered. Separation of computations might improve reliability; and increased network traffic might increase the vulnerability to certain types of security attacks. Each attribute primitive write-up highlights possible side effects on other attributes.

In summary, each attribute primitive write-up addresses one or more quality attributes as characterized by one or more general scenarios. It will offer a description of the components, their relationships, and properties as they are relevant to the general scenario and a rationale for why this ensemble contributes to achieving the general scenario. A thumbnail sketch is a summary of an attribute primitive write-up. The thumbnail sketches of attribute primitives that we use in our application to EJB are in the appendix of this document.

3 Enterprise JavaBeans

As more businesses embrace the electronic marketplace business model, there is an urgent need for business systems to be accessible via the Web. The Java 2 Enterprise Edition (J2EE) specification from Sun Microsystems describes a multi-tiered architecture for constructing enterprise-wide applications that enables systems to be accessible via the Web [Liu 01].

J2EE makes it possible to reuse Java components in a server-side infrastructure. With appropriate component assembly and deployment tools, the aim is to bring the ease of programming associated with GUI-builder tools (like Visual Basic) to building server applications. And by providing a standard framework for J2EE products based upon a single language (Java), J2EE component-based solutions are, in principle, product independent and portable between the J2EE platforms that are provided by various vendors.

The major features that the J2EE platform provides are

- a multi-tiered distributed application model
- a server-side component model, i.e., Enterprise JavaBeans (EJB) model
- a unified security model
- built-in transaction control

A simple depiction of the J2EE multi-tier model is shown in Figure 1. The role of each tier is as follows:

Client Tier: In a Web application, the client tier is composed of an Internet browser that submits HTTP (hypertext transfer protocol) requests and downloads HTML (hypertext markup language) pages from a Web Server. In an application not deployed using a browser, stand-alone Java clients or applets can be used, and these would communicate directly with the Business Component tier, using the Java Remote Method Invocation (RMI) as the underlying protocol.

Web Tier: The Web tier runs a World Wide Web server to handle client requests, and invokes J2EE servlets or Java Server Pages (JSPs). Servlets are invoked by the Web server depending on the type of user request and will query the business logic tier to get the required information to satisfy the request. The servlets then format the information for return to the user via the Web server. JSPs are basically static HTML pages that contain snippets of servlet code. The code is invoked by the JSP mechanism, which also takes responsibility for formatting the dynamic portion of the page.

Business Component Tier: The business components constitute the core business logic for the application. The business components are realized by Enterprise JavaBeans, the software component model supported by J2EE. EJBs receive requests from servlets in the Web tier, or directly from Java clients. EJBs then satisfy the request usually by accessing some data sources, and return the results to the servlet or the Java client. EJB components are hosted by a J2EE environment known as an EJB container. The container supplies a number of services to the EJBs that it hosts. These services include transaction and life-cycle management, state management, security, multi-threading, and resource pooling. EJBs simply specify the type of behavior they require from the container at run time, and then rely on the container to provide the services. This frees the application programmer from cluttering the business logic with code to handle system and environmental issues.

Enterprise Information Systems Tier: This tier typically consists of one or more databases and back-end applications like mainframes and other legacy systems. EJBs must query these data stores to process requests. The Java Database Connectivity (JDBC) drivers are typically used for accessing databases, and the Java Connector Architecture (JCA) standard protocol is used to access packaged applications such as enterprise resource planning (ERP) systems and customer relationship management (CRM) systems, as well as various mainframe-based transaction processing systems.

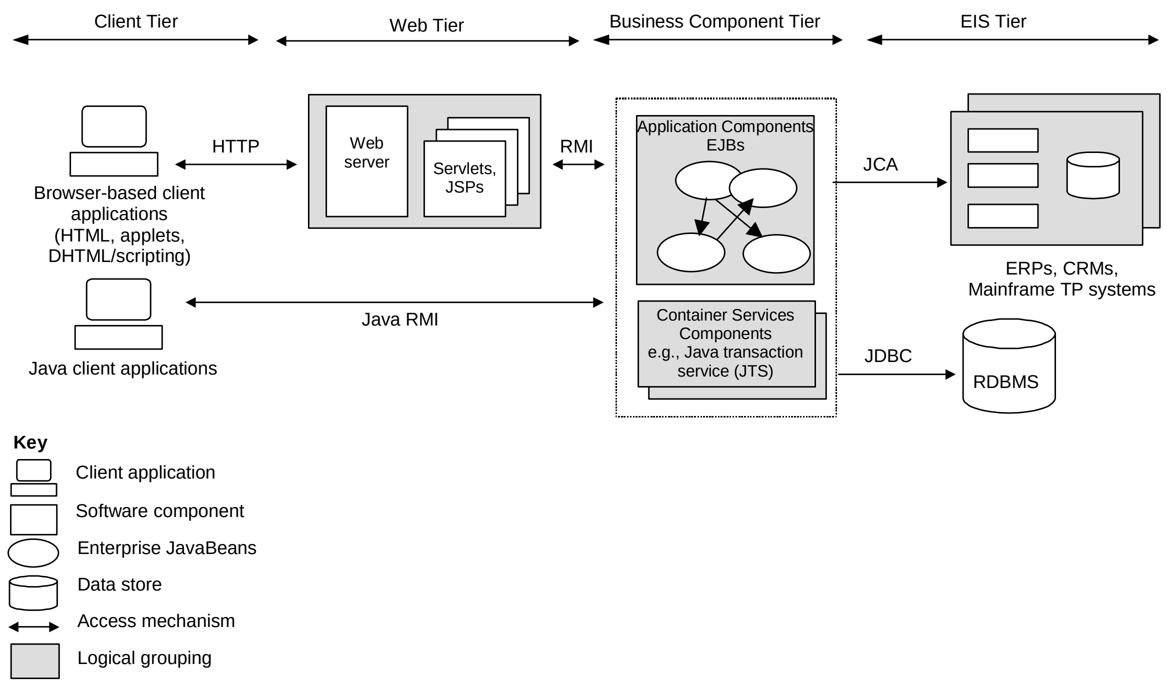

Figure 1: J2EE Multi-Tiered Application Architecture

In the remainder of the document, we will focus on the server-side component model: Enterprise JavaBeans. The EJB specification describes a component-based framework for constructing server-side Java applications. An EJB container provides a run-time environment to host application components and supports these components by providing services such as transaction, persistence, concurrency and security management. When a client invokes a server component, the container automatically allocates a thread and invokes an instance of the component. The container manages all resources on behalf of the component and manages all interactions between the component and the external systems such as database management systems. In the J2EE model, these services are supplied through a set of standard vendor-independent interfaces. The EJB framework thus provides an environment for people to build an enterprise application quickly.

However, ease of development and deployment are not enough; the resultant system must exhibit suitable qualities, that is, suitable performance, reliability, security, usability, and so on. For example, will the resultant EJB system behave responsively when handling a dynamically changing volume of client requests? Web-enabling a business system means opening up the business to potentially thousands and millions of customers in the Internet world. If the current design cannot handle the volume, is the design scalable? Other questions about reliability, security, modifiability, and usability can be asked as well.

In a nutshell we ask: What are the quality attribute ramifications of building systems using the EJB component model? In this paper we use the notion of quality attribute design primitives to explore this question.

4 Analyzing Enterprise JavaBeans Systems Using Quality Attribute Design Primitives

As mentioned earlier, EJB has many features that aid in the development of applications. We will describe a typical configuration of these features and then use quality attribute primitives to draw conclusions about the attribute-related ramifications of using these features. For example, in an EJB context clients access services through a "home interface" via the Java Naming and Directory Interface (JNDI), which obviates the need for the client to know the physical location of the server and thus allows for ease of service relocation. This is a form of a naming service, which is described in one of our thumbnail sketches.

The objective of this section is to

- enumerate a specific set of features of interest to EJB
- recast this specific set of features in terms of attribute primitives
- use the qualitative analysis codified in each attribute primitive to offer reasoning guidance for this style of EJB usage

4.1 A Typical Configuration: A Session Bean-Only Architecture

Figure 2 illustrates a simple EJB application architecture with a session bean (stateless or stateful) that provides all the business functionality in its methods. We choose to use such a simple EJB application architecture to focus on the EJB infrastructure features and the design alternatives that the EJB environment supports. Even with a simple EJB application architecture, many important EJB container services (such as naming and transactions) are exercised.

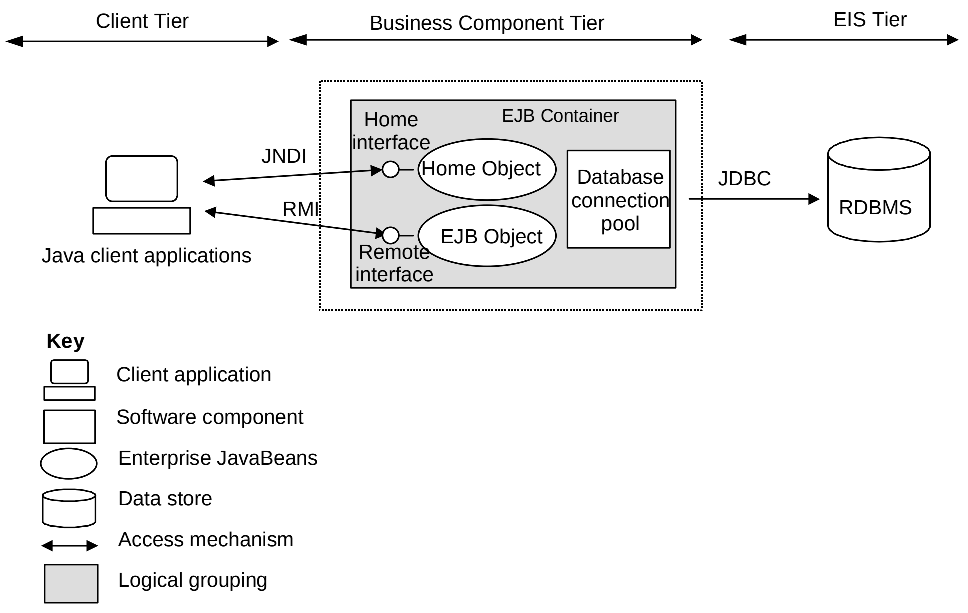

Figure 2: A Simple EJB Application Architecture

All Enterprise JavaBeans are implemented through a *Home Object* and an *EJB Object*. The Home Object implements various EJB life-cycle methods such as the creation and deletion of an EJB instance, as well as the "lookup" of a corresponding EJB Object. The EJB Object is where the actual implementation of the business logic resides. The client first accesses the home object via the home interface using the Java Naming and Directory Interface (JNDI); the client can then obtain a reference to the EJB Object from the home object. The client is now ready to make RMI calls to the EJB Object to request that the business logic be carried out.

A typical EJB container or J2EE application server will provide database connection management for EJB components to use to access business data in the EIS tier. The EJB container also manages the life cycle (i.e., creation, replication, and deletion of EJB Object instances) and routes requests to the appropriate EJB instance. Transaction control and management are also provided by EJB containers.

From this relatively simple EJB application architecture, we can summarize the following EJB features:

- **Java Naming and Directory Interface (JNDI)** – Clients access the home interface via the Java Naming and Directory Interface (JNDI). The home object will in turn pass to the client a reference to the EJB object.
- **EJB Object Remote Interface** – Once a client has access to the remote interface, it can invoke business logic to be carried out by making RMI calls to the EJB Object. The EJB Object basically presents to the client all the available services (or business logic) provided by the session bean (which can be stateless or stateful).

- **Server/Bean instance replication** – The EJB server or container creates and manages multiple instances of the same enterprise server bean.
- **Load balancing for server/bean instances** – Load balancing is a strategy for dynamically routing client requests to a particular server/bean instance for processing.
- **Database connection pooling** – When session beans need to read from and write to a database, they first need to obtain a handle to a database connection. Application server products such as the WebLogic Server provide a pool of ready-to-use database connections for EJB server components to use and re-use.
- **Transactions** – When multiple updates to business data need to be done in an atomic fashion, with consistent intermediate results, and these updates need to allow for concurrent update operations, each with durable states at the end of the operations, the server implementation needs to support transactions. As part of the transaction services, logging is done to record state changes. In the case of failure, the system can be rolled back to the previous consistent state.

We do not claim these are all of the EJB features that contribute to the quality of EJB systems, but these are enough to demonstrate how to use the thumbnail sketches.

4.2 Approach

We will now use attribute primitives to explore the attribute behavior of the EJB application architecture shown in Figure 2. In the next several sections, we will consider the architecture's modifiability, performance, and availability, respectively. For each attribute, our approach is outlined below:

1. We will use general scenarios as the basis for creating EJB-specific scenarios that specify important quality attribute requirements that need to be addressed by EJB.
2. The general scenarios will also lead us to relevant attribute primitive thumbnail sketches (in the appendix).
3. The relevant EJB features are then explained and qualitatively analyzed in terms of the attribute primitives.
4. The qualitative analyses for each of the EJB features are then coalesced.

Note that general scenarios provide the link between EJB-specific scenarios and attribute primitives. General scenarios provide a "bucket" in which EJB-specific scenarios can be placed. General scenarios also provide explicit pointers to attribute primitives. Therefore once an EJB-specific scenario is identified as an instance of a general scenario, one (or possibly several) relevant attribute primitives have also been identified. The remaining challenge is to find an instance of the attribute primitive in the specific architecture that's being analyzed. This is when the person with attribute primitive expertise interacts with someone who has domain (in this case EJB) expertise to map the attribute primitive onto the specific architecture and apply the general analysis codified in the attribute primitive.

We begin with modifiability.

4.3 Modifiability

Table 1 shows several general modifiability scenarios and the EJB-specific modifiability scenarios suggested by them. This implements Step 1 of our approach.

Table 1: Modifiability Scenarios

General Modifiability Scenario	EJB-Specific Modifiability Scenario
The platform on which the system depends is changed. The system must be modified to continue to provide current functionality. The platform change may be a change in hardware including input and output hardware, it could be a change in operating system, or it could be a change in COTS middleware included in the system. Existing functionality of the system should remain unchanged.	The platform on which the EJB application runs changes. These platform changes include changes to the Java Virtual Machine (JVM), operating system, hardware, database driver, database change, EJB server and container (across different vendor products, and version upgrades). The system needs to be modified to continue to provide current functionality.
A request arrives to add additional users, potentially distributed. The system should be modified to enable these additional users to access its services while still maintaining quality of service.	Additional online users connect to an application server, possibly via a Web server and the volume of online requests fluctuates (e.g., increased sales before Christmas or the last few minutes of an online auction). The system should be able to handle the large/varying volumes of client requests with limited modification to the application or, even better, through self re-configuration.

Since each attribute primitive starts by identifying the relevant general scenario, general scenarios can be viewed as an index for the set of attribute primitives. Table 2 maps the two general scenarios in Table 1 to the relevant attribute primitives. This implements Step 2 of our approach.

Table 2: Locating Modifiability General Scenario Within Attribute Primitive Sketch

Portion of General Scenario	Attribute Primitive
Changing the hardware platform on which a service resides.	*Naming Service*
Adding additional users while maintaining other qualities such as performance.	*Client/Server*
Implementation details change without affecting much of the rest of the system.	*Separation of Interface from Implementation*
Addition of functionality without impacting the rest of the system.[1]	*Transactions*

[1] Notice that general scenarios point to attribute primitives in two ways: (1) the attribute primitive directly addresses the general scenario, and (2) the attribute primitive affects the general scenario as a side effect.

We now implement Step 3 of our approach. Each attribute primitive is mapped onto EJB features and the analysis in the thumbnail sketch is used to yield an EJB-specific analysis.

- **JNDI** – JNDI is in part an instance of the *Naming Service* attribute primitive (AP). A naming server places a level of indirection between the client and provider of a service by providing a mapping from a service's logical name to its physical location. The client does not need to know where the server is physically located.

  JNDI also is in part an instance of the *Client/Server* AP. The Client/Server AP is used to manage multiple clients accessing a set of services and allows additional clients to be easily added.
- **EJB Object Remote Interface** – The EJB Object Remote Interface is an instance of the *Separation of Interface from Implementation* AP. The client does not need to know the internal server implementation details; it needs to know only what services the server provides. Details—such as how the server provides its services, the creation and management of EJB server, and session bean instances—are hidden from the client.
- **Stateless session beans**[2] – Stateless session beans with idempotent operations on the server allow the addition of functionality to be an easier task, hence improving the modifiability of the system. However state information between different method calls has to be passed back and forth between the client and server instances. Session management becomes a responsibility for the clients, impacting the modifiability of the clients.
- **Stateful session beans** – The alternative to stateless session beans is stateful session beans. In this case the stateful session beans handle state information on behalf of the client. This creates static bindings between the client and server instance, which makes the system inflexible.
- **Transactions** – When the transactional guarantees are supported by a third-party implementation (e.g., container-managed transaction), decoupling of the EJB components and the resource managers (i.e., Rational Database Management System [RDBMS]) is enabled. The strict adherence to a standard interface such as the XA/Open's Distributed Transaction Processing model will enhance modifiability. By separating application functionality from concerns of consistency, rollback, and recovery, the addition of new functionality is simplified.

Modifiability strategies (Step 4 of our approach): The strategies used to increase modifiability include forms of indirection and separation. These provide the ability to add new server functionality, change server implementations, and change server location.

4.4 Performance

Table 3 shows a general performance scenario and the EJB-specific performance scenario suggested by it.

[2] Note that "stateless" and "stateful" are component properties, not attribute primitives.

Table 3: Performance Scenarios

General Scenario	EJB-Specific Refinement
Events arrive stochastically. On average, the system must complete n responses per unit time.	For a set of random client requests, the system needs to process at least n transactions per second (TPS).

The general scenario in Table 3 points to the following attribute primitives shown in Table 4:

Table 4: Locating Performance General Scenario Within Attribute Primitive

Portion of General Scenario	Attribute Primitive
Events arrive stochastically. On average, the system must complete n responses per unit time.	*Naming Server*
	Client/Server
	Replication
	Load Balancing
	Connection Manager

The attribute primitives in the above table map onto the following EJB features:

- **JNDI** – As pointed out in the *Naming Server* AP, the indirection introduced by JNDI will introduce overhead (that is, additional *execution time*), an important component property that affects performance. However, the JNDI lookup often occurs only once when the client is looking for the session bean. All subsequent service calls are via the EJB Object. The method invocation across a network cannot be avoided even if the EJB programming model is not used. However, the resource usage across the network needs to be accounted for when assessing throughput.
- **Stateless** – Stateless session beans in the server-side can be easily replicated to handle larger volume of requests. Since each bean instance runs in its own thread, bean replicas can take advantage of each other’s input/output (I/O) blocking times. Moreover, as pointed out in the *Client/Server* AP, if different threads can be assigned different priorities, a server will be able to offer clients with varying levels of service. However, since there is no concept of bean priorities, all clients are treated roughly equally.
- **Stateful** – Replication of stateful servers is possible; however, this incurs some performance penalty due to the need to store the state information onto secondary storage (in order to have a failsafe system, an availability consideration). However, the consistency checks here for multiple stateful session beans is a difficult problem. Most existing application server solutions either compromise on correctness, or incur a performance penalty.
- **Server/Instance replication** – Dynamic creation and deletion of EJB instances as supported by the *Replication* AP (depending on volume of requests) can help to ensure a more scalable system that can better handle peak and off-peak loads.

- **Load balancing for server instances** – Load balancing can work on two levels: across different EJB instances, and/or across different EJB server instances in a cluster. As pointed out in the *Load Balancing* AP there are different load-balancing algorithms that can be used to improve throughput.
- **Database connection pooling** – Establishing database connections is a slow operation, expensive in terms of execution time. Hence, the pooling of database connections can enhance performance through the reuse of connections. Application server products such as WebLogic Server provide a pool of ready-to-use database connections for EJB server components to use and re-use. The EJB server component implementation need not deal with issues related to setting up database connections. It can focus on the business functionality implementation. As pointed out in the *Connection Manager* AP, this separation of concerns at the implementation level enhances usability for the EJB programmer and makes the system more modifiable. Also, note that any shared resource can introduce blocking time as pointed out in the *Client/Server* AP.
- **Transactions** – As pointed out in the *Transaction* AP, additional performance overhead is added due to transaction logging.

Performance strategies: There are four basic factors affecting response time: introducing extra computation time due to overhead introduced by indirection and redundancy, reducing the execution time overhead associated with establishing database connections by establishing a shared pool of connections, exploiting physically concurrent processing through replication and load balancing, and introducing blocking time due to mutually exclusive access to database connections and to data via transactions.

4.5 Availability

Table 5 shows several general availability scenarios and the EJB-specific performance scenarios suggested by them.

Table 5: Availability Scenarios

General Scenario	**EJB-Specific refinement**
An internal component fails. The system is able to recognize a failure of an internal component and has strategies to compensate for the fault.	An internal component, such as an enterprise Java Bean instance and a container service, fails. The system should detect this and have strategies to compensate for the fault.
An external component fails. The system has alternative strategies to compensate for the fault.	An external component (such as database, database server hardware, or network connection) fails or times out. The system has alternative strategies to compensate for the fault.
An event arrives at the system for which it was not prepared. Such events could be unknown messages, failure of a component, unexpected timing behavior (too fast or too many), unavailable resources (e.g. disk space), etc. The system has a clear model about what is and is not allowed and has strategies for handling events that are out of scope.	An event arrives at the system for which it was not prepared for. Such events could be unknown messages, failure of a session bean, unexpected request timing and volume behavior (too fast or too many), unavailable resources (e.g. disk space, memory, JVM threads, EJB system threads, database connections in pool, etc). The system needs to have a clear model about what is and is not allowed and has strategies for handling events that are out of scope.

The general scenarios in Table 5 point to the following attribute primitives shown in Table 6:

Table 6: Locating Availability General Scenario Within Attribute Primitive

Portion of General Scenario	**Attribute Primitive**
An external component fails. The system has alternative strategies to compensate for the fault.	*Logging State Changes*
An event arrives at the system for which it was not prepared.	*Replication*
	Transactions
	Logging State Changes

The attribute primitives in Table 6 map onto the following EJB features:

- **Stateless server** – The stateless server model together with idempotent operations enable easier implementation of fail-over when the stateless server is *replicated*. When a particular stateless server fails, its work can be re-directed to a different server instance without implications for state management.
- **Server/Instance replication** – *Replicated* server instances also make the system more available as if one server instance is down, another instance can take over the work. In an EJB system, reconfiguration could occur at many different levels: e.g., clustering of machines allows the client request to be re-routed to a different machine

when one machine fails, and replication of stateless session beans allows client request to be dynamically re-routed to a good session bean from a dead session bean.

- **Transactions** – As pointed out in the *Transaction* AP, the guaranteed nature (supported by rollback, commit operations via logging) makes transactional systems more robust; hence, it improves availability and reliability by helping to ensure that the system is always in a consistent state and by providing a system-wide strategy for handling certain classes of failures.

Availability strategies: Availability is enhanced by having replicates of some components, by logging information, and by guaranteeing atomicity of database accesses.

5 Insights into EJB from the Analysis

5.1 Design Decision Support

The EJB attribute primitives enumerated in the document can be roughly classified into two categories. One is those attribute primitives that aid in the fundamental understanding of the EJB programming model and the container behavior. This type of EJB attribute primitive is typified by the *Separation of Interface from Implementation* AP. The attribute primitive write-up enhances our understanding of EJB container and server behaviors and supports our reasoning of the total system quality.

The second category of EJB attribute primitives are those that present themselves as a design option; for example, whether to replicate stateful or states servers are alternative design strategies, and the attribute primitive write-up for each of them can assist with the design decision. Architects can analyze and prioritize the various quality requirements, and then choose the appropriate mechanism that has the most positive impact on the more important quality requirement, while having a minimal negative *side effect*.

5.2 Evaluating EJB Paradigm

A valuable by-product of the process for identifying EJB attribute primitives is a critical assessment of the EJB programming model. Here, we summarize a list of observations about EJB features in relation to various quality attributes. Some of these observations follow from the discussion in the previous sections of this paper. Other observations we made about other aspects of EJB while writing this document are summarized below.

Observations based on the body of this document:

- **Stateful servers** – The stateful server model will always be less scalable than a true stateless server model. The stateful session beans model may be useful to maintain session information. However, in the world of e-commerce, where scalability is a primary concern, server side implementations must be kept “truly stateless” to ensure problem-free server replication for system scaling.
- **JNDI** – The initial JNDI lookup can potentially be a bottleneck if an inadequate name server implementation is used.

  The JNDI lookup is often a once-off operation (i.e., when the client is looking for a session bean). All subsequent service calls are via the EJB Object. The method invocation across a network cannot be avoided even if the EJB programming model is not used. Hence, the modifiability benefit here by far outweighs the possible performance degradation.

The delegation from the EJB Object to the actual session bean instance is a relatively cheap local call. Hence, the modifiability benefit here by far outweighs the possible performance degradation.

The benefit arising from availability and performance usually outweighs the complexity of session management on the client side.

- **Server/Instance replication** – There is some overhead involved in the dynamic management of bean instances. However, this is likely to be a relatively small overhead in comparison to the greater gains in responsiveness.
- **Transactions** – The tradeoff here is between the robustness of the system and performance. If the ACID (Atomic, Consistency, Isolation, and Durability) property is necessary, then one needs to investigate whether the throughput supported by the transactional system is "good enough," and whether it's "scalable enough" to handle peak loads.

  Bean-Managed Transaction (BMT) allows for hand-crafted code, which is often optimized for performance. Container-Managed Transaction (CMT) uses the existing transaction services provided by the EJB container, requires less programming effort from the developer, and, as a result, generated code might incur a performance penalty.

  High performance means nothing if the EJB server system cannot provide consistent and accurate business data. However, design consideration is required to make sure that the logging is not a bottleneck.

Additional observations: While we didn't analyze EJB from the points of view of security and usability, we can still make several observations:

- Lack of in-built security on EJB resources – The EJB model assumes that security is taken care of. However, this is often not the case. Hence, what we have is a vast array of third-party security add-on products for handling security issues.
- Lack of usability concerns – The popular Model View Controller (MVC) pattern assists with the usability attribute to some extent. However, the EJB programming paradigm creates the decoupling of presentation model layers, which means that EJB server-side designers often overlook the usability issues. The end result is a complex mesh of presentation logic calling upon a large set of model elements, which incurs a heavy performance penalty and ends up being a maintenance nightmare.

6 Conclusion

Our goal was to assess the utility of mapping between the system-independent attribute primitive write-ups and EJB and to see whether the write-ups provided useful insights into the quality of EJB. We feel that we have successfully demonstrated a means of understanding the quality attribute behavior of EJB. Furthermore this understanding does not depend on having a deep understanding of the quality attributes, but depends only on how to map the general scenarios to EJB-specific scenarios and the attribute primitives to the EJB features. Whether our analysis demonstrated quality aspects of EJB that were previously unknown is not the point. The point is that attribute primitives can be used to make the quality aspects of EJB more generally available to those designing systems using EJB.

Further, we are encouraged that the codification of these commonly used architectural primitives enables architects to make better informed architectural design decisions through the following concrete ways:

1. General scenarios are a good checklist in compiling quality requirements for a particular system.
2. Candidate architectural primitives can be found that satisfy the corresponding quality requirements identified in 1 above.
3. Tradeoff issues arising from the use of these attribute primitives can be identified through the analysis of “side-effects.”

References

[Bass 00]	Bass, Len; Klein, Mark; & Bachmann, Felix. *Quality Attribute Design Primitives* (CMU/SEI-2000-TN-017, ADA 392284). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. Available WWW <http://www.sei.cmu.edu/publications/documents/00.reports/00tn017.html>.
[Bass 01]	Bass, Len; Klein, Mark; & Moreno, Gabriel. *Applicability of General Scenarios to the Architecture Tradeoff Analysis Method* (CMU/SEI-2001-TR-014). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. Available WWW <http://www.sei.cmu.edu/publications/documents/01.reports/01tr014.html>.
[Buschmann 96]	Buschmann, Frank; Meunier, Regine; Rohnert, Hans; Sommerlad, Peter; & Stal, Michael. *Pattern-Oriented Software Architecture: A System of Patterns*. New York, NY: John Wiley & Sons, 1996.
[Klein 99]	Klein, M.; Kazman, R.; Bass, L.; Carriere, J.; Barbacci, M.; & Lipson, H. “Attribute-Based Architecture Styles, Software Architecture” 225-243. *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*. San Antonio, TX, February 1999.
[Liu 01]	Liu, Anna. “J2EE in 2001.” *Component Development Strategies Newsletter*, Cutter Consortium (September 2001).

Appendix: Attribute Primitives and Attribute Properties

This appendix contains a list of the attribute primitive thumbnail sketches that we used in this paper. The thumbnail sketch is meant to be suggestive of the type of information that will ultimately be contained in the full write-up of an attribute primitive. However we anticipate that complete AP write-ups will have description sections that describe the primitives in terms of components, their properties and their relationships and an analysis section that offers fairly detailed reasoning about how the mechanism contributes to the achievement of its targeted quality attribute.

A.1 Modifiability Attribute Primitives

A.1.1 Naming Server

General Scenario: Modifiability – Change physical location of service with minimal impact on the rest of the system.

Description: The components involved that use a naming server include the client who requests services, the server who encapsulates those services, and the naming server that is an intermediary and provides a mapping from a service to the service's location. The client acquires the server's location and then makes service calls at that location. (A proxy serves a similar purpose, except it makes the service calls on the client's behalf, obviating the need for the client to be concerned with location at all.)

Analysis Principles: Indirection is a fundamental strategy for achieving some aspects of modifiability. Indirection is accomplished by using an intermediary that hides information. By using an intermediary, it hides service location, thus allowing for location independence. Clients of a service can acquire a service's location without having it "hard-coded" into the service call. The client does not need to know where the server is physically located at compile time; rather, the location is acquired at runtime, thus allowing the location of the service to be easily changed.

Side Effects: Indirection introduces execution overhead, possibly degrading response time. The degree of overhead depends on how often the location must be determined relative to the frequency of service calls and how long it takes to resolve the location.

A.1.2 Client/Server[3]

General Scenario(s): Modifiability – Number of users changes while maintaining other qualities.

Description: The Client/Server attribute primitive uses the strategy of separation as a means for achieving several different attribute goals. Naturally, the client and the server are the main components involved. The client is active and initiates action; the server is passive and encapsulates common services needed by all of the clients. Servers can be implemented as stateless or stateful on the same processor as the clients or on different processors, in a single process or in multiple processes, and on a single processor or as a cluster on multiple processor, just to name a few of the alternatives.

Analysis Principles: Multi-threaded or reentrant servers facilitate adding more clients since each client will exist independently without affecting existing clients.

Side Effects: If session management is performed at the server side, then the server is a single point of failure. If session management is performed at the client side, then servers can be replicated but each communication between the client and the server must include the current state of the session. Deploying the server on one machine and the clients on others improves performance since it provides dedicated computation power for the clients and eliminates the network traffic between the user's computer and the combined client/server computer.

A.1.3 Separation of Interface from Implementation

General Scenario: Modifiability – Change the implementation of a service.

Description: A module serves as an interface to another module or an interface is written in an interface definition language (IDL). In addition to implementation details being hidden, the specific language and syntax are hidden. The only requirement is for the client and the server body to conform to the IDL specification.

Analysis Principles: The client does not need to know the internal server implementation details including the implementation language. It only needs to know what services the server provides. This offers flexibility in changing the implementation.

Side Effects: Extra levels of indirection may degrade performance. Also it is necessary to understand how other quality attributes change when the implementation changes.

[3] You will notice that the Client/Server AP appears under several attributes. The description section is the same for each, but the analysis section will focus on the specific attribute.

A.2 Performance Attribute Primitives

A.2.1 Connection Manager

General Scenario: Performance – Require bounded response time and/or certain system throughput.

Description: When consumers need to use a resource, they first need to obtain a handle to connect to that resource. This attribute primitive concerns the creation of a pool of connections that are created ahead of time and shared by all clients.

Quality Attribute Analysis: Establishing connections is a slow operation. Hence, the pooling of connections enhances performance. During consumer start-up time, initial connection setup takes some time. However, initial connection time is a one-time event. Improvement of run-time performance due to connection pooling will overcome this start-up overhead as the number of consumers per connection increases.

Side Effects: The consumer implementation need not deal with connection setup issues. It can focus on its implementation. This separation of concerns at the implementation level makes the system more modifiable.

A.2.2 Load Balancing

General Scenario: Performance – Require bounded response time and/or certain system throughput.

Description: Load balancing dynamically routes client requests to a particular server instance for processing.

Analysis Principles: There may be different load-balancing algorithms used: round-robin, least loaded server, random, etc. The aim is to distribute client requests as evenly as possible over different server components to ensure the highest possible system throughput. The benefits gained depend on whether component instances reside on the same server or on different servers and the degree to which they suspend for I/O.

Side Effects: If the load-balancing algorithm is implemented in a central request router, than this is potentially a single point of failure and a performance bottleneck. If the load-balancing algorithm is distributed, then additional network traffic is required to keep the different portions of the algorithm synchronized.

A.2.3 Client/Server

General Scenario: Performance – Require bounded response time and/or certain system throughput.

Description: The client/server attribute primitive uses the strategy of separation as a means for achieving several different attribute goals. Naturally, the client and the server are the main components involved. The client is active and initiates action; the server is passive and encapsulates common services needed by all of the clients. Servers can be implemented as stateless or stateful, on the same processor as the clients or on different processors, in a single process or in multiple processes, and on a single processor or as a cluster on multiple processor, just to name a few of the alternatives.

Analysis Principles: Multi-threaded or reentrant servers facilitate adding more clients thus enabling each thread to exploit the I/O blocking time of other threads and also enabling levels of service by assigning different priorities to different threads. This allows for increasing throughput in general and provides the ability to manage throughput for various priority levels. Furthermore, performing computations of the client on the client's computer increases the computation power available for that computation and reduces the network traffic if the computation was performed on the server computer. On the other hand if services require mutually exclusive access, while one client is being served other clients can be blocked.

Side Effects: Changes to the server are easily deployed without affecting the clients. If session management is performed at the server side, then the server is a single point of failure. If session management is performed at the client side, then servers can be replicated, but each communication between the client and the server must include the current state of the session.

A.3 Availability Attribute Primitives

A.3.1 Replication

General Scenario: Availability – An internal component fails and the system is able to recognize the failure and has strategies to compensate for the fault.

Description: Multiple instances of the same component. Reconfiguration enables recovery from error by switching to redundant components.

Analysis Principles: Replicated component instances are a building block for increasing reliability. If one instance is down, another instance can take over the work. One must be aware of common mode failures that can simultaneously affect all replicas and thus prevent availability benefits from accruing. Also the failure detection and voting scheme can dramatically impact availability.

Side Effects: Naturally there is some overhead involved in the dynamic management of component instances. Dynamic creation and deletion of component instances (depending on volume of requests) can help a system scale to handle peak and off-peak times. The question tends not to be whether or not to replicate, but rather how to fine tune the amount of replication given fixed system resources (e.g., memory, optimal thread number, database connection numbers). Provided that the redundant components are not idle, the redundant/replicated component can share the workload, hence enhancing performance.

A.3.2 Transactions

General Scenario: Availability – An internal component fails, and the system is able to recognize the failure and has strategies to compensate the fault.

Description: Transactions allow multiple updates to business data in an atomic fashion, with consistent intermediate results, and they enable concurrent update operations, each with durable states at the end of the operations.

Analysis Principles: Guaranteed-nature atomicity plus rollback and logging enable transactional systems to contribute to availability and reliability.

Side Effects: Since container-managed transactions are simple to use and enable decoupling between the server components and the resource managers (i.e., RDBMS), they also facilitate modifiability.

Additional performance overhead due to transaction logging; serialization of operations (requiring one customer to wait until completion of the current customer’s request) also slows down the system.

A.3.3 Logging State Changes

General Scenario: An internal or external component fails. The system is able to recognize the failure and has strategies to compensate for the fault.

Description: As part of the transaction services, logging is done to record state changes. In the case of failure, the system can then be rolled back to its previous consistent state.

Quality Attribute Analysis: Availability and Reliability. In order to recover in the event of a failure, the current state must be available to the component with current control. One technique for ensuring this is to record state changes so that the current state can be recovered.

Side Effects: Logging (especially persistent logging) incurs a performance penalty.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
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 October 2001		3. REPORT TYPE AND DATES COVERED Final
4. TITLE AND SUBTITLE Analyzing Enterprise JavaBeans Systems Using Quality Attribute Design Primitives			5. FUNDING NUMBERS F19628-00-C-0003
6. AUTHOR(S) Anna Liu, Len Bass, Mark Klein			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213			8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2001-TN-025
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116			10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES			
12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS			12B DISTRIBUTION CODE
13. ABSTRACT (MAXIMUM 200 WORDS) Quality attribute requirements such as those for performance, security, modifiability, reliability, and usability have a significant influence on the software architecture of a system. At the Software Engineering Institute, we are studying and codifying the relationship between quality attribute requirements and the architectural design strategies that impact their achievement. In CMU/SEI-2000-TN-017 [Bass 00], we introduced the notion of *quality attribute design primitives*. Quality attribute design primitives (or attribute primitives) are architectural building blocks that target the achievement of one—or sometimes several—quality attribute requirements. Our intent is to codify a fairly comprehensive set of attribute primitives in a manner that articulates how each attribute primitive makes its specific contribution toward the achievement of one or several attribute goals. We believe this will provide a very powerful “language” for constructing or analyzing software architectures in relation to quality attribute requirements. To determine the expressive and explanatory power of these attribute primitives, we will examine various classes of systems. This paper uses attribute primitives to examine the qualities of Enterprise JavaBeans (EJB)-based systems. In particular, we find that attribute primitives hold promise for providing insight into the quality attribute consequences of using various EJB infrastructure features.			
14. SUBJECT TERMS Enterprise JavaBeans (EJB), quality attribute design primitive, quality attribute requirements, software architecture			15. NUMBER OF PAGES 33
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