Enterprise JavaBeans 2.1

STEFAN DENNINGER and INGO PETERS with ROB CASTANEDA
translated by David Kramer

Apress™

Enterprise JavaBeans 2.1

ISBN (pbk): 1-59059- 088-0

Printed and bound in the United States of America 12345678910

Translator, Editor, Compositor: David Kramer

Technical Reviewer: Mary Schladenhauffen

Editorial Directors: Dan Appleman, Gary Cornell, Simon Hayes, Martin Streicher, Karen Watterson, John Zukowski

Managing and Production Editor: Grace Wong

Proofreader: Lori Bring

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, Inc.,175 Fifth Avenue, New York, NY, 10010 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany

In the United States, phone 1-800-SPRINGER, email orders@springer-ny.com, or visit http://www.springer-ny.com

Outside the United States, fax +49 6221 345229 email orders@springer.de, or visit http://www.springer.de,

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax: 510-549-5939, email info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

CHAPTER 3

The Architecture of Enterprise JavaBeans

IN CHAPTER 2 WE INDICATED that Enterprise JavaBeans (EJB) is a component of the Java-2 platform, Enterprise Edition (for details see [26]). In this model EJB takes over the part of the server-side application logic that is available in the form of components: the Enterprise Beans. This chapter introduces the architecture of Enterprise JavaBeans. Figure 3-1 shows *Enterprise Beans* (in their incarnations as *entity*, *message-driven*, and *session* beans) as the central elements.

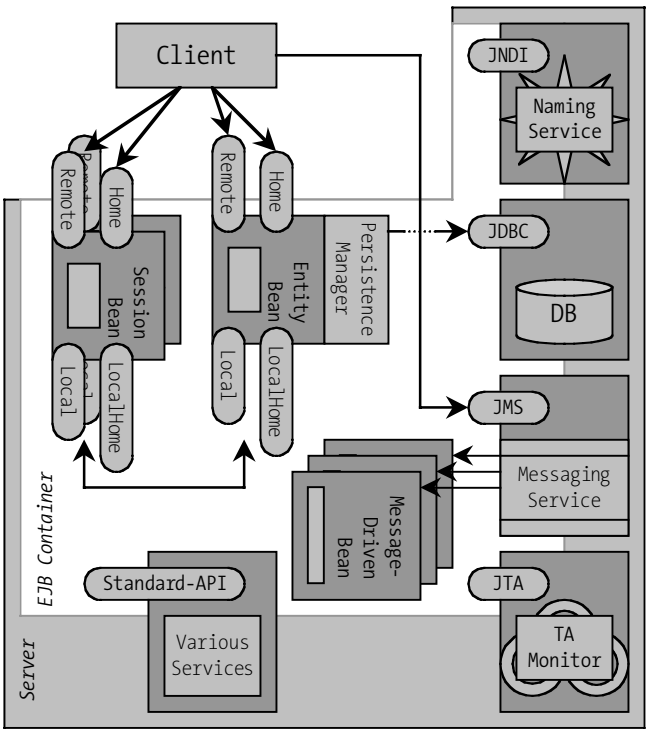

Figure 3-1. Overview of the EJB architecture.

They contain the application logic used by the *client* programs. Enterprise Beans reside in an *EJB container*, which makes a run-time environment available to them (so that, for example, they can be addressed by client programs via the *home* and *remote* interfaces and have the possibility of communication among one another via the *local home* and *local* interfaces, so that life-cycle management can be provided). The EJB container is linked to *services* via the standard programming interface, services that are available to the bean (for example, access to databases via JDBC, access to a transaction service via JTA, and access to a message service via JMS). The EJB container is installed (possibly in addition to other containers) in an *application server*.

We shall now go into the details of the individual components of the architecture and their interrelationships.

The Server

The server is the fundamental component of the EJB architecture. Here we are deliberately not speaking of an *EJB server*. Actually, it should be called a *J2EE server*. Sun Microsystems' strategy in relationship to enterprise applications in the framework of the J2EE platform involves Enterprise JavaBeans to a considerably greater extent in the full portfolio of Java-based programming interfaces and products than was the case in version 1.0 of the Enterprise JavaBeans specification.

The specification of Enterprise JavaBeans in version 2.1 does not define any sort of requirement on the server (just as in versions 1.0 and 1.1). The reason for this is presumably their stronger integration into the Java 2 platform, Enterprise Edition (see Figure 3-2).

A J2EE-conforming server is a run-time environment for various containers (of which one or more can be EJB containers). Each container, in turn, makes a run-time environment available for a particular type of component. Creators of Java application servers tend more and more to support the J2EE platform. There is scarcely a producer who offers a pure EJB server. Many suppliers of databases, transaction monitors, or CORBA ORBs have meanwhile begun to support Enterprise JavaBeans.

In the environment of the J2EE platform (and thus indirectly in the EJB architecture) the server component has the responsibility of providing basic functionality. This, includes, for example:

- Thread and process management (so that several containers can offer their services to the server in parallel);
- Support of clustering and load sharing (that is, the ability to run several servers cooperatively and to distribute client requests according to the load on each server to obtain the best-possible response times);

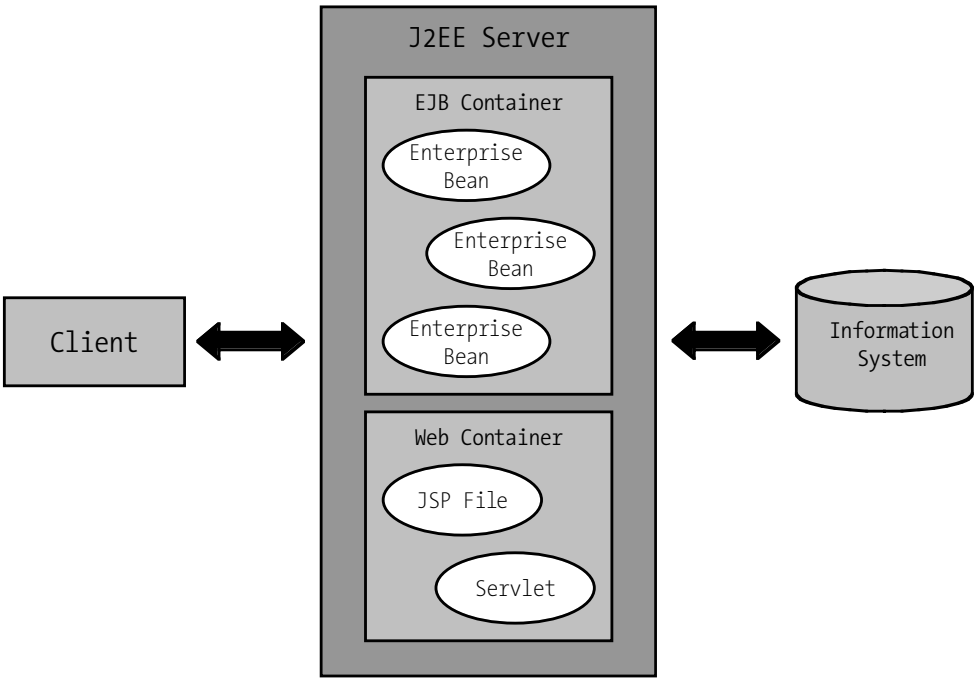

Figure 3-2. EJB in the context of Java2, Enterprise Edition.

- Security against breakdown (fail-safe);
- A naming and directory service (for locating components);
- Access to and pooling of operating system resources (for example, network sockets for the operation of a web container).

The interface between the server and containers is highly dependent on the producer. Neither the specification of Enterprise JavaBeans nor that of the Java 2 platform, Enterprise Edition, defines the protocol for this. The specification of Enterprise JavaBeans in version 2.1 assumes that the producer of the server and that of the container are one and the same.

The EJB Container

The EJB container is a run-time environment for Enterprise Bean components. Just as an EJB container is assigned to the server as a run-time environment and service provider, a bean is dependent on its EJB container, which provides it a run-time environment and services. Such services are provided to the bean via standard programming interfaces. The specification in version 2.1 obliges the EJB container to provide *at least* the following programming interfaces:

- The API (Application Programming Interface) of the Java 2 platform, Standard Edition, in version 1.4;
- The API of the specification of Enterprise JavaBeans 2.1;
- The API of JNDI 1.2 (Java Naming and Directory Interface);
- The UserTransaction API from JTA 1.0.1 (Java Transaction API);
- The API of the JDBC 2.0 extension (Java Database Connectivity);
- The API of JMS 1.1 (Java Message Service);
- The API of Java Mail 1.3 (for sending e-mail);
- The API of JAXP 1.1, 1.2 (Java XML Parser);
- The API of JAXR and JAX-RPC 1.0 (Java XML Remote Procedure Calls);
- The API of Java Connectors 1.5;
- The API of Java Web Services 1.0.

A provider of a Java application server may offer additional services via the standard interface. Some producers offer, for example, a generic service interface particular to the producer by means of which specially developed services (such as a logging service or user management) can be offered. If an Enterprise Bean uses such proprietary services, then it cannot simply be placed in any available container.

The EJB container provides Enterprise Beans a run-time environment, and also offers the Enterprise Beans particular services at run-time, via the above-mentioned (static) programming interfaces. We now would like to examine the most important aspects of both domains (that of the run-time environment as well as that of provided services).

Control of the Life Cycle of a Bean (Run-Time Environment)

The EJB container is responsible for the life cycle of a bean. The EJB container generates bean instances (for example, when a client requests one), deploys them in various states via callback methods, and manages them in pools in case they will not be needed until a later time. The bean instances are also deleted by the EJB container. The states and respective methods that lead to a state transition depend on the type of bean. They are discussed in detail in Chapter 4, “Session Beans”; Chapter 5, “Entity Beans”; and Chapter 6, “Message-Driven Beans.”

Instance Pooling (Run-Time Environment); Activation and Passivation

A system that supports mission-critical applications must be capable of dealing with great demands. It must be capable of serving a large number of concurrent clients without them having to put up with long response times. The greater the number of clients, the greater generally the number of objects generated in the server.

In order to keep the number of bean instances from growing without bound and to prevent bean instances from continually being created and then destroyed, an EJB container maintains a certain number of bean instances in a pool. As long as a bean is in the pool, it is in a `Pooled` state and is to a certain extent deactivated. On demand (as a rule, as a result of a client request) the first available bean instance of the requisite type is taken from the pool. It is reactivated and placed in a `Ready` state.

Such instance pooling is common practice in database applications. This practice has also proved itself in the management of thread objects (such as in server development). With pooling, fewer bean instances than the number of client connections are needed, as a rule. By avoiding permanent object generation and destruction, system performance is enhanced. Instance pooling is closely connected with the life-cycle management of a bean. Finally, we note that the specification does not obligate an EJB container to support instance pooling.

A further possibility for the EJB container to save system resources is to make Enterprise Bean instances that are currently not needed (for example, from object serialization) persistent and to remove them from memory (*passivation*). Upon demand these instances can again be deserialized and made available in memory (*activation*). Such a temporary storage of bean instances (those, for example, for which a certain time period has expired) on secondary media can reduce the burden on system memory resources. This behavior is similar to that of an operating system page file.

Whether bean instances can be pooled, activated, or passivated depends on their type. In the description of bean types in what follows we shall make the necessary distinctions.

Distribution (Run-Time Environment)

The EJB container ensures that Enterprise Beans can be used by client programs, which as a rule are not running in the same process. The client does not know on what server the Enterprise Bean that it is currently using is located. Even when an Enterprise Bean uses another Enterprise Bean, it is a client. The place where the Enterprise Bean exists is transparent to the client. The use of an Enterprise

Bean on another computer is not essentially different for the client from the use of objects that are located in the same address space.

For distributed communication between the various parties, Java Remote Method Invocation (Java RMI) is used. To achieve interoperability among application servers supplied by different manufacturers the EJB specification prescribes support for the communications protocol of the CORBA specification, IIOP, for an EJB container that conforms to specifications. Distributed communication thus takes place over the protocol RMI-IIOP (RMI over IIOP). In no case does the developer of an Enterprise Bean have to worry about the bean being able to be accessed from outside. This is the sole task of the EJB container.

This book assumes that the reader is already familiar with the techniques of remote method invocation (or else see [13] and [4] for information on RMI). In particular, the reader should understand the notions of *stub* and *skeleton*.

Since version 2.0 of the Enterprise JavaBeans specification the client has had the ability to communicate with an Enterprise Bean via the *Local* interface. In many applications it is necessary that Enterprise Beans, all of which are installed in the same EJB container (aggregation of components), communicate with one another. In version 1.1 of the EJB specification this means a remote call to a component located in the same address space. The overhead of the RMI protocol is unnecessary, and it leads to a degradation of performance. The advantage of using local interfaces is that RMI is thereby completely ignored. Local interfaces can then be sensibly used only if client and Enterprise Bean are located in the same address space of a Java virtual machine. The location of the Enterprise Bean is thus no longer transparent to the client. Transparency of location applies only to EJB components that are called over the remote interface. Furthermore, the semantics of a method call to a component are altered when a local interface is used. With calls to the local interface, parameters are passed via *call by reference*, while calls to the remote interface use *call by value*.

The various chapters on various types of beans will deal extensively with the difference between the remote and local interfaces and will delve deeply into the viewpoint of the client with respect to an Enterprise Bean.

Naming and Directory Service (Service)

If a client wishes to find a bean, it is directed to a naming interface. A naming service offers the possibility of associating references to removed objects under a particular name, which may be assigned arbitrarily, at a definite place (binding). It also offers the possibility of looking up the object bound to the naming service under that name (lookup). This is similar to the white pages, whereby one looks up an object by name.

A directory service is more powerful than a naming service. It supports not only binding of references to a name, it can also manage distributed objects and other resources (such as printers, files, application servers) in hierarchical structures, and it offers wide-ranging possibilities for administration. With a directory service a client can be provided with additional descriptive information about a reference to a remote object. This is similar to the yellow pages, whereby one looks up an object by its attributes.

The interface over which the naming and directory services are accessed is JNDI (Java Naming and Directory Interface). The bean, too, can receive information via the naming and directory service. The EJB container provides the bean instance information, for example, that was established at the time of installation of the component (so-called environment entries). In this way it is possible to influence the behavior of Enterprise Beans by external parameterization. The bean also has the possibility of accessing particular resources, such as database connections or a message service, via the naming and directory services.

Persistence (Service)

The EJB container provides the beans, via the naming and directory services, the possibility of accessing database connections. Beans can thereby themselves ensure that their state is made persistent. However, the specification of Enterprise JavaBeans provides for a mechanism by which the state of certain Enterprise Bean types can be made persistent automatically. (We shall return to this mechanism in Chapter 5, "Entity Beans.")

As a rule, in the case of automatic storage of the state of Enterprise Beans by the EJB container the data are made persistent in a database. One can imagine other EJB containers that provide persistence mechanisms that place the data in other storage systems (for example, in the file system or electronic archives). There is also the possibility of developing EJB containers that make use of the interfaces of other application systems in order to read or write data from or to that location. Thus, for example, the data in an old main-frame system can be bound by means of special EJB containers to component-oriented systems (the container then acts to a certain extent as a wrapper for the old systems).

However, what is really at issue is that in the case of automatic persistence of an Enterprise Bean, the bean couldn't care less where the data are stored. The EJB container takes on the task of ensuring that the data are stored and remain in a consistent state. Thus a particular bean can be installed in various EJB containers that support varying storage systems as persistence medium. For the Enterprise Bean the persistence remains transparent. It doesn't know where its data are stored or where the data come from that the EJB container uses to initialize it.

Transactions (Service and Run-Time Environment)

Transactions are a proven technique for simplifying the development of distributed applications. Among other things, transactions support the applications developer in dealing with error situations that can occur from simultaneous access to particular data by multiple users.

A developer who uses transactions splits up the actions to be executed into atomic units (transactions). The transaction monitor (the EJB container) ensures that the individual actions of a transaction are all executed successfully. If an action fails, the successful actions thus far executed are canceled.

The support of transactions is a fundamental part of the specification of Enterprise JavaBeans. In distributed systems in which several users work simultaneously on many separate actions with the same data (which may be distributed among several back-end systems), a transaction service on the level of the application server is unavoidable. It is the task of the EJB container to ensure that the necessary protocols (for example, the two-phase commit protocol between a transaction monitor and a database system, context propagation, and a distributed two-phase commit) for handling transactions are available. The specification of Enterprise JavaBeans supports flat transactions; that is, transactions cannot be nested.

The developer of an Enterprise Bean can choose how he or she wishes to use transactions. On the one hand, transactions can be used explicitly, by communicating over JTA directly with the transaction service of the EJB container. Alternatively, the developer can opt for declarative (or implicit) transactions. In this case, at the time of installation of an Enterprise Bean in the EJB container it is specified what methods should run within which transactions. The EJB container intervenes when these methods are called, and it ensures that they are called within the appropriate transaction context.

In the case of declarative transactions the bean developer does not need to be concerned with manipulating the transactions. During installation in EJB container $\mathcal{A}$, a bean can be installed with a completely different transactional behavior from that during installation in EJB container $\mathcal{B}$. The bean itself remains untouched in each case, since in every case the container remains responsible for guaranteeing the desired transactional behavior. Much more detail on transactions in the EJB context is to be found in Chapter 7, “Transactions.”

Message (Service)

With version 2.0 of the specification of Enterprise JavaBeans the EJB container is obligated to integrate a message service via JMS-API (Java Message Service). With the definition of a new bean type—the message-driven bean—the message service is integrated into the EJB container in a significant way. The development

of applications based on Enterprise JavaBeans thus gains two additional dimensions: asynchronicity and parallel processing.

Basically, a message system enables the asynchronous exchange of messages among two or more clients. In contrast to the case of a client–server system, here the architecture of a message system is designed around a loose coupling of equal partners. Each client of the message system can send and receive messages asynchronously. The sender of a message remains largely anonymous, and the same holds for the recipient. Message systems are also known under the name *message-oriented middleware* (MOM).

In addition to the use of message-driven beans, the message service can, of course, also be used for asynchronous message exchange between any two parties. Enterprise Beans, like clients, can also send messages. Through the use of messaging, processes can, for example, decouple from each other or even create interfaces to other systems, making it an excellent component for enterprise integration.

Chapter 6 offers a fuller discussion of Java Message Service message-driven beans.

Security (Run-Time Environment)

The specification obligates the EJB container to provide Enterprise JavaBeans an infrastructure for security management as a part of the run-time environment. It is the task of the system administrator and of the installer of the Enterprise Beans to establish security policy. Once again, the EJB container is responsible for the implementation of this security policy. The goal here is (as with container-managed automatic persistence and declarative transactions) to make the security mechanisms transparent to the Enterprise JavaBeans, so that they can be deployed in as many systems as possible.

If the security strategy were implemented in the bean, it would be problematic to employ the same bean under both more- and less-strict security requirements. On the other hand, it makes much more sense to place the security mechanisms in the run-time environment of the components. They are thereby reusable to a great extent, and the security policy can be adapted from the outside as the situation warrants. The specification of Enterprise JavaBeans specifically mentions that it is preferable that no logic relating to security be present in the code of a bean.

It is possible to define *user roles*. In every Enterprise Bean particular *permissions* can be assigned to a particular role. The assignment takes place, as with the establishment of the user roles, at one of the following times:

- The time of bean installation;
- The time at which several beans are combined into an *aggregate*.

Permissions are focused essentially on whether the user is permitted to call particular methods of an Enterprise Bean. At run time the EJB container determines whether a client call to a bean method should be allowed to be executed. To this end it compares the role of the client with the permissions of the respective Enterprise Bean method.

As a rule, in addition to the security mechanisms that we have described, an EJB container offers the following security attributes:

- Authentication of the user by a user ID and password;
- Secure communication (for example, via the use of secure socket layers).

Chapter 8 contains a more detailed discussion of security in relation to Enterprise JavaBeans.

Finally, we note that the EJB container is the central instance in the component model of Enterprise JavaBeans. It provides the Enterprise Beans (the components) a convenient run-time environment at a very high level of abstraction and makes a variety of services available by way of standard interfaces.

The Persistence Manager

The persistence manager is the building block in the architecture of Enterprise JavaBeans that enables the automatic persistence of particular components. It was introduced with version 2.0 of the EJB specification to achieve a better separation of the physical data storage from the object model. The goal was to improve the portability of persistent EJB components to application servers of other manufacturers. Moreover, improvements were introduced for the mapping of a persistent component onto the storage medium, as well as the possibility of constructing declarative relations between persistent components and an abstract query language. In version 1.1 one was often forced, in the situation of automatic container-governed persistence, to rely on the use of proprietary tools (object-relational "OR" mapping tools) or the use of proprietary extensions of the EJB container, with the result that the portability of the components was greatly compromised.

As always, persistence is managed by the EJB container; that is, it determines *when* the data of a component are loaded or stored. The EJB container also determines whether in the case of an action's failure a successfully executed saving operation should be undone (transaction). The persistence manager, on the other hand, is responsible for *where* and *how* the persistent data are stored. It takes over communication with the storage medium (for example, a database). The mapping of the persistent data of an Enterprise Bean onto the storage medium (for example, the mapping onto one or more database tables) is determined at the installation of a component. The persistence manager

plays no role when the Enterprise Bean itself looks after persistence or when the components possess no persistent data.

In most cases a database is used for storing data. In spite of the ANSI SQL standard the databases of different producers are not one hundred percent compatible with one another. For example, they use different key words in the syntax of their query languages. It is usual that particular database functions that distinguish one database from those of other producers are usable only with proprietary extensions of the standard query language SQL. The persistence manager is supposed to catch these difficulties as well. Depending on the implemented database, a specialized persistence manager can be used that is able to deal with the peculiarities of the database system.

A further responsibility of the persistence manager is the formulation of search queries. With knowledge of the mapping of the data and of the peculiarities of the storage medium in use it can translate abstract search queries into concrete search queries. For the formulation of abstract search queries for finding EJB components the specification of Enterprise JavaBeans offers a query language called EJB-QL (Enterprise JavaBeans Query Language). EJB-QL was introduced in version 2.0 of the EJB specification and is further enhanced in version 2.1.

The persistence manager and the query language EJB-QL are dealt with extensively in Chapter 5.

Enterprise Beans

Enterprise Beans are the server-side components used in the component architecture of Enterprise JavaBeans. They implement the application logic on which the client programs rely. The functionality of the server and the EJB container ensures only that beans can be used. Enterprise Beans are installed in an EJB container, which offers them an environment at run time in which they can exist. Enterprise Beans rely implicitly or explicitly on the services that the EJB container offers:

Implicitly in the case of

- container-managed persistence (CMP);
- declarative transactions;
- security.

Explicitly in the case of

- The use of explicit transactions;
- bean-managed persistence (BMP);
- the sending of asynchronous messages.

Types of Enterprise Beans

There are three different forms of Enterprise Beans, which differ more or less sharply one from the other: entity beans, message-driven beans, and session beans. Table 3-1 describes the basic differences among these three types of Enterprise Beans.

Table 3-1. Defining characteristics distinguishing session, message-driven, and entity Beans (see [25]).

	Session Bean	Message-Driven Bean	Entity Bean
Task of the bean	Represents a server-side service that executes tasks for a client.	Represents server-side enterprise logic for the processing of asynchronous messages.	Represents an enterprise object whose data are located in permanent storage.
Access to the bean	A session bean is a private resource for the client, available to the client exclusively.	A message-driven bean is not directly accessible to the client. Communication is effected exclusively via sending messages over a particular channel of the message service.	An entity bean is a central resource; the bean instance is used simultaneously by several clients, and its data are available to all clients.
Persistence of the bean	Not persistent. When the bound client or server is terminated, the bean is no longer accessible.	Not persistent. When the server is terminated, the bean is no longer accessible. The messages that have not yet been delivered to the bean are persistent as required. (More on this in Chapter 6.)	Persistent. When bound clients or server is terminated, the state of the entity bean is located in a persistent storage medium. The bean can be recreated at a later time.

Session beans model ordinary processes or events. For example, this could be the entering of a new customer in an enterprise resource planning system (ERP), the execution of a booking in a booking system, or setting a production plan based on open orders. Session beans can be viewed as an extension of the client's arm toward the server. This point of view is supported by the fact that a session bean is a private resource of a particular client.

Entity beans, on the other hand, represent objects in the real world that are associated with particular data, such as a customer, a booking account, or a product. An instance of a particular entity bean type can be used simultaneously by several clients. Session beans usually operate on data represented by entity beans.

Message-driven beans are recipients of asynchronous messages. A message service acts as a mediator between the sender of a message and the message-driven bean. Entity and session beans are addressed via the *remote* or *local* interface. Calls to entity or session beans are synchronous; that is, the execution of the client is blocked until the method of the Enterprise Bean has been processed. After the method call has returned, the client can continue its processing. Message-driven beans can be addressed by the client only (indirectly) by sending a message over a particular channel of the message service. A particular type of message-driven bean receives all messages that are sent over a particular channel of the message service. Communication over a message service is asynchronous. That is, the execution of the client can proceed directly after a message is sent. It does not remain blocked until the message has been delivered and processed. The container can deploy several instances of a particular message-driven bean type for the processing of messages. Thus in this case parallel processing is possible. Message-driven beans have no state between the processing of several messages. Furthermore, they have no identity vis-à-vis the client. In a certain sense the are similar to *stateless* session beans (see the following paragraph). For the processing of a message, message-driven beans can use session or entity beans as well as all services that the container offers.

There is another distinction to be made with regard to session beans, namely, whether a session bean is *stateless* or *stateful*. Stateless session beans do not store any data from one method call to the next. The methods of a stateless session bean operate only with the data that are passed to it as parameters. Stateless session beans of the same type all possess the same identity. Since they have no state, there is neither the necessity nor the possibility of distinguishing one from the other.

Stateful session beans, on the other hand, store data over many method calls. Calls by methods to stateful session beans can change the state of the bean. The state is lost when the client is no longer using the bean or when the server is taken down. Stateful session beans of the same type have differing identities at run time. The EJB container must be able to distinguish them, since they have differing states for their clients.

A session bean receives its identity from the EJB container. In contrast to entity beans, the identity of a session bean is not externally visible. Since clients always work with a session bean that for them is an exclusive instance, there is no need for such visibility.

Entity beans can be distinguished by whether they themselves are responsible for making their data persistent or whether the EJB container takes over this task. In the first case one speaks of *bean-managed persistence*, while in the second the talk is of *container-managed persistence*.

Entity beans of the same type have differing identities at run time. An entity bean of a particular type is identified at run time by its primary key, which is

allocated by the EJB container. It is thereby bound to particular data, which it represents in its activation phase. The identity of an entity bean is outwardly visible.

The bean types play a role in resource management of the EJB container. With entity beans, message-driven beans, and stateless session beans the container can instigate pooling, while with stateful session beans it can instigate passivation and activation (serialization and deserialization onto a secondary storage medium).

The interface between an entity bean and the EJB container is called the *context* (`javax.ejb.EJBContext`). This interface is again specialized for the three bean types (to `javax.ejb.EntityContext`, `javax.ejb.MessageDrivenContext`, and `javax.ejb.SessionContext`). The bean can communicate with the container using the context that is passed by the EJB container to the bean. The context remains bound to a bean during its entire life span. By means of the context the EJB container manages the identity of an Enterprise Bean. With a change in the context the EJB container can change the identity of a bean.

Chapters 4, 5, and 6 provide an extensive discussion of the technical details of session, message-driven, and entity beans. The second section of Chapter 9 deals with the differing semantics of the various bean types.

Components of an Enterprise Bean

An Enterprise Bean possesses the following components:

- The remote interface and the (remote) home interface *or* the local and local home interface (for entity and session beans);
- The bean class (for entity, message-driven, and session beans);
- The primary key or primary key class (for entity beans);
- The deployment descriptor (for entity, message-driven, and session beans).

One speaks of the remote client view when an Enterprise Bean is addressable over the remote interface. If an Enterprise Bean uses the local interface, one speaks of the local client view. Basically, an Enterprise Bean can support both the local and remote client views. However, the specification advises that one choose one of the two cases.

Let us describe the individual components of a bean by way of an example. We would like to develop an entity bean that represents a bank account. The components should make it possible to ascertain the account number, a description of the account, and the current balance of the account. Furthermore, the balance of the account should be capable of being raised or lowered by an arbitrary amount. The bank-account bean should be able to be addressed by the remote client, that is, from a client that is located outside the address space of the bank-account bean.

This chapter concentrates on the representation of the special features determined by the architecture. In this example we shall not go into the special features of a particular bean type (that will be done extensively in Chapters 4, 5, and 6). Since an entity bean exhibits all the components mentioned above, it is best suited for this introductory example. Moreover, we shall not use the local interface in this example. Since EJB is a distributed component architecture, the use of the remote interface is standard. The use of the local interface is analogous to that of the remote interface, and it will be dealt with in Chapters 4 and 5.

Remote Interface

The remote interface defines those methods that are not offered externally by a bean. The methods of the remote interface thus reflect the functionality that is expected or demanded by the components. The remote interface must be derived from `javax.ejb.EJBObject`, which in turn is derived from `java.rmi.Remote`. All methods of the remote interface must declare the exception `java.rmi.RemoteException`. See Listing 3-1.

Listing 3-1. Remote interface of BankAccount.

```
package ejb.bankaccount;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;

public interface BankAccount extends EJBObject
{
  //ascertain account number
  public String getAccNumber()
    throws RemoteException;
  //ascertain account description
  public String getAccDescription()
    throws RemoteException;
  //ascertain account balance
  public float getBalance()
    throws RemoteException;
  //increase account balance
  public void increaseBalance(float amount)
    throws RemoteException;
  //reduce account balance
  public void decreaseBalance(float amount)
    throws RemoteException;
}
```

Home Interface

The home interface must be derived from `javax.ejb.EJBHome` (in this interface is to be found the method for deleting a bean; it does not need to be separately declared). `EJBHome`, for its part, is likewise derived from `javax.rmi.Remote`. In the home interface as well all methods declare the triggering of an exception of type `java.rmi.RemoteExeption`. As in the case of the remote interface, everything points to the distributed character and the embedding in the EJB framework. See Listing 3-2.

Listing 3-2. The home interface of BankAccount.

```
package ejb.bankaccount;

import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import javax.ejb.FinderException;

public interface BankAccountHome extends EJBHome
{

  //generate an account
  public BankAccount create(String accNo,
                                  String accDescription,
                                  float initialBalance)
    throws CreateException, RemoteException;

  //find a particular account
  public BankAccount findByPrimaryKey(String accPK)
    throws FinderException, RemoteException;

}
```

Bean Classes

Bean classes implement the methods that have been declared in the home and remote interfaces (with the exception of the `findByPrimaryKey` method), without actually implementing these two interfaces. The signatures of the methods of the remote and home interfaces must agree with the corresponding methods in the bean class. The bean class must implement an interface that depends on its type, and indeed, it must be `javax.ejb.EntityBean`, `javax.ejb.MessageDrivenBean`, or `javax.ejb.SessionBean`. The bean implements neither its home nor its remote

interface. Only in the case of an entity bean with container-managed automatic persistence is the class *abstract*. The classes of session, message-driven, and entity beans, which manage their own persistence, are *concrete* classes. See Listing 3-3.

Listing 3-3. Bean class of `BankAccount`.

```
package ejb.bankaccount;

import javax.ejb.CreateException;
import javax.ejb.EntityBean;
import javax.ejb.EntityContext;
import javax.ejb.RemoveException;

public abstract class BankAccountBean implements EntityBean {
  private EntityContext theContext;

  public BankAccountBean() {
  }

  //the create method of the home interface

  public String ejbCreate(String accNo,
                                  String accDescription,
                                  float initialBalance)
    throws CreateException
  {
    setAccountNumber(accNo);
    setAccountDescription(accDescription);
    setAccountBalance(initialBalance);
    return null;
  }

  public void ejbPostCreate(String accNo,
                                  String accDescription,
                                  float initialBalance)
    throws CreateException
  {
  }

  //abstract getter/setter methods

  public abstract String getAccountNumber();
  public abstract void setAccountNumber(String acn);

  public abstract String getAccountDescription();
  public abstract void setAccountDescription(String acd);
  public abstract float getAccountBalance();
  public abstract void setAccountBalance(float acb);
```

```
    //the methods of the remote interface
    public String getAccNumber() {
      return getAccountNumber();
    }
    public String getAccDescription() {
      return getAccountDescription();
    }
    public float getBalance() {
      return getAccountBalance();
    }
    public void increaseBalance(float amount) {
      float acb = getAccountBalance();
      acb += amount;
      setAccountBalance(acb);
    }
    public void decreaseBalance(float amount) {
      float acb = getAccountBalance();
      acb -= amount;
      setAccountBalance(acb);
    }
    //the methods of the javax.ejb.EntityBean interface
    public void setEntityContext(EntityContext ctx) {
      theContext = ctx;
    }
    public void unsetEntityContext() {
      theContext = null;
    }
    public void ejbRemove()
      throws RemoveException
    {
    }
    public void ejbActivate() {
    }
    public void ejbPassivate() {
    }
    public void ejbLoad() {
    }
    public void ejbStore() {
    }
}
```

Primary Key (Primary Key Class)

The primary key is relevant only for entity beans. Its purpose is to identify an entity of a particular type uniquely. As in the case of the primary key of a database table, it contains those attributes that are necessary for unique identification. With the primary key a particular entity can be found, which then is associated by the EJB container with an entity bean instance of the correct type. With the primary key the identity of an entity bean is externally visible. The primary key class is irrelevant for session and message-driven beans, since their identity is never externally visible. The specification distinguishes two types of primary keys:

- primary keys that refer to a field of the entity bean class;
- primary keys that refer to several fields of the entity bean class.

A primary key that refers to only a single field of the entity bean class can be represented by a standard Java class (for example, `java.lang.String`, `java.lang.Integer`). In our example the class `java.lang.String` is the primary key class, since the unique identification of an account is possible via its (alphanumeric) account number.

A primary key that refers to several fields of the entity bean class is represented, as a rule, by a class specially developed for that purpose. Such a class must be a public class, and it must have a public constructor without arguments. The fields of the primary key class that represent the primary key of the entity bean must address those of the entity bean class by name. Furthermore, these fields must also be public. The class must be RMI-IIOP compatible (serializable), and it must implement the methods `equals( )` and `hashCode( )`. Listing 3-4 shows an example of such a primary key class for an account bean that requires for its unique identification the number of the client as well as the account number (client-capable system).

Listing 3-4. Example of a primary key class for multipart keys.

```
package ejb.custom;
public class CustomAccountPK implements java.io.Serializable
{
  public String clientNumber;
  public String accountNumber;
  public CustomAccountPK() {
  }
  public int hashCode() {
    return clientNumber.hashCode() ^
       accountNumber.hashCode();
  }
```

```
  public boolean equals(Object obj) {
    if(!(obj instanceof CustomAccountPK)) {
      return false;
    }
    CustomAccountPK pk = (CustomAccountPK)obj;
    return (clientNumber.equals(pk.clientNumber)
      && accountNumber.equals(pk.accountNumber));
  }
  public String toString() {
    return clientNumber + ":" + accountNumber;
  }
}
```

The Deployment Descriptor

The deployment descriptor is a file in XML format (details on XML can be found in [3]) that describes one or more beans or how several beans can be collected into an aggregate. All such information that is not to be found in the code of a bean is placed in the deployment descriptor. Essentially, this is declarative information. This information is of particular importance for those collecting several Enterprise Beans into an application or installing Enterprise Beans in one EJB container. The EJB container is informed via the deployment descriptor how it is to handle the component(s) at run time.

The deployment descriptor contains information on the structure of an Enterprise Bean and its external dependencies (for example, to other beans or to particular resources such as connections to a database). Furthermore, it contains information about how the components should behave at run time or how they can be combined with other components into more complex building blocks. We shall illustrate this for our example bean by showing in Listing 3-5 a suitable complete deployment descriptor (for a full description of the deployment descriptor see [21]).

Listing 3-5. Deployment descriptor of `BankAccount`.

```
<?xml version="1.0" ?>
<ejb-jar version="2.1" xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd">
  <description>
    This deployment descriptor contains information on
    the entity bean BankAccount.
  </description>
```

```
<enterprise-beans>
  <entity>
    <!-- Name of the Enterprise Bean -->
    <ejb-name>BankAccount</ejb-name>
    <!-- class of the Home Interface -->
    <home>ejb.bankaccount.BankAccountHome</home>
    <!-- class of the Remote Interface -->
    <remote>ejb.bankaccount.BankAccount</remote>
    <!-- class of the Enterprise Bean -->
    <ejb-class>ejb.bankaccount.BankAccountBean</ejb-class>
    <!-- type of persistence -->
    <persistence-type>Container</persistence-type>
    <!-- class of the primary key -->
    <prim-key-class>java.lang.String</prim-key-class>
    <!-- specifies whether the implementation of the
     Enterprise Bean is reentrant -->
    <reentrant>False</reentrant>
    <!-- the EJB version for which this Enterprise
     Bean was developed -->
    <cmp-version>2.x</cmp-version>
    <!-- Name of the persistence mechanism -->
    <abstract-schema-name>AccountBean
    </abstract-schema-name>
    <!-- list of the persistent attributes
     of the Enterprise Bean -->
    <cmp-field>
      <description>account number</description>
      <field-name>accountNumber</field-name>
    </cmp-field>
    <cmp-field>
      <description>account description</description>
      <field-name>accountDescription</field-name>
    </cmp-field>
    <cmp-field>
      <description>account balance</description>
      <field-name>accountBalance</field-name>
    </cmp-field>
    <!-- primary key field -->
    <primkey-field>accountNumber</primkey-field>
  </entity>
</enterprise-beans>
<assembly-descriptor>
  <!-- Definition of the user role 'Banker'-->
  <security-role>
```

```
      <description> the role of the banker
      </description>
      <role-name>Banker</role-name>
    </security-role>
    <!-- Definition of access rights at the method level-->
    <method-permission>
      <role-name>banker</role-name>
      <method>
        <ejb-name>BankAccount</ejb-name>
        <method-name>*</method-name>
      </method>
    </method-permission>
    <!-- Definition of transactional behavior
      per method of the Enterprise Bean -->
    <container-transaction>
      <method>
        <ejb-name>BankAccount</ejb-name>
        <method-name>increaseBalance</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
    <container-transaction>
      <method>
        <ejb-name>BankAccount</ejb-name>
        <method-name>decreaseBalance</method-name>
      </method>
      <trans-attribute>Required</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

Since our example deals with an entity bean with container-managed persistence, instructions for the persistence manager must also be provided. It must know which fields of the bean are to be mapped to which columns in which table or tables. Moreover, it requires instructions about the database in which the data are to be stored. The specification does not specify what form these instructions are to take. It prescribes only that the creator must provide tools by means of which such instructions can be supplied. It is thus clear that these tools will be different depending on the creator, as also will be the format in which the instructions are provided. In the simplest case these instructions can be given in a file in XML format. Listing 3-6 shows an imaginary example. Later in this chapter we shall call these instructions the *persistence descriptor*.

Listing 3-6. Example of mapping instructions for the persistence manager.

```
<abstract-schema>
 <name>AccountBean</name>
 <data-source>
  <name>test-db</name>
  <type>oracle</type>
 </data-source>
 <table-name>account</table-name>
 <field-mapping>
  <bean-field>accountNumber</bean-field>
  <column>acno</column>
 </field-mapping>
 <field-mapping>
  <bean-field>accountDescription</bean-field>
  <column>acdesc</column>
 </field-mapping>
 <field-mapping>
  <bean-field>accountBalance</bean-field>
  <column>acbal</column>
 </field-mapping>
</abstract-schema>
```

If all components are present, then according to bean type, the home and remote interfaces, the bean class(es), the primary key class(es), and the deployment descriptor (which can contain a description of several Enterprise Bean components) are packed in JAR format into a file. The acronym JAR stands for "Java archive" and corresponds to the popular ZIP format. The components of an Enterprise Bean are then complete and are packed as component(s) in a jar file (further details on packing a component in a jar file can be found in [21]). Table 3-2 shows once more which components are relevant to which bean type.

Table 3-2. Overview of the components of various bean types.

	Entity		**Session**	**Message-Driven**
	Container Managed	**Bean Managed**		
Remote, local interface	X	X	X	
Local & remote home interface	X	X	X	
Concrete bean class		X	X	X
Abstract bean class	X			
Deployment descriptor	X	X	X	X
Persistence descriptor	X			

How Everything Works Together

Let us assume that the component `BankAccount` is not included in an aggregate, but is installed directly in an EJB container, in order to be used by client programs. The installation takes place with the help of the relevant tools. The specification obligates the creator of an EJB container and the persistence manager to provide these tools. It is left to the creator to determine what these tools look like and how they are to be used. As a rule, they support the capturing of all relevant data for the installation of a component via a graphical user interface. However, of primary importance is the result of the tool-supported installation procedure. It provides the missing links in the EJB architecture: the implementation of the home and remote interfaces (respectively the local home and local interfaces) of the Enterprise Bean and in the case of a container-managed entity bean the implementation of the concrete bean class. Figure 3-3 shows these relationships.

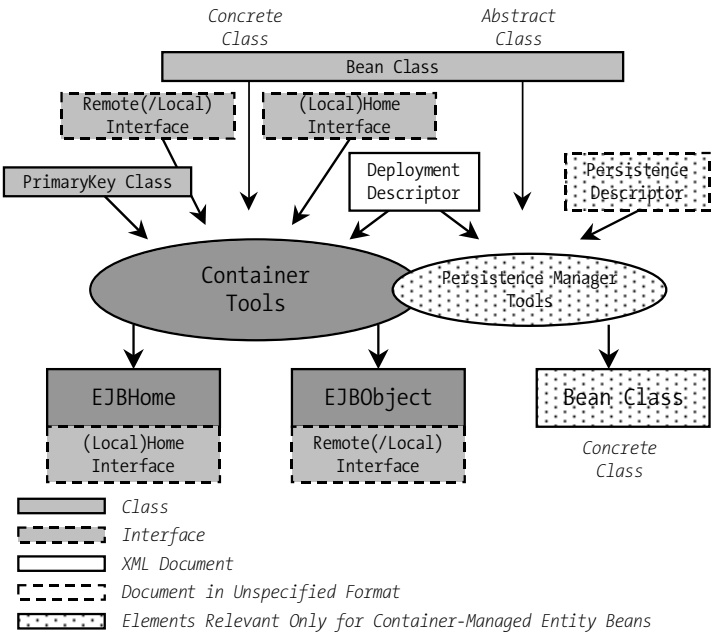

Figure 3-3. Generation of the missing architectural components.

As we have already mentioned, an Enterprise Bean uses either local or remote interfaces. The implementation class of the home or local home interface is usually called `EJBHome`, while that of the local or remote interface is denoted by `EJBHome`, while the remote interface is usually known as `EJBObject`. The classes `EJBHome` and `EJBObject` are generated from the components of a bean by the tools of the container's creator.

If we are dealing with an entity bean with container-managed persistence, then it is the job of the persistence manager to generate a concrete class. This is derived from the abstract bean class and represents the necessary code for persistence. For this generation, the additionally created instructions regarding the database and table(s) from the persistence descriptor as well as the persistence fields of the components are employed (see in this respect the example in Listing 3-6).

The `EJBHome` object serves at run time as a sort of object factory, and the `EJBObject` as a sort of functional wrapper for the Enterprise Bean in question. (See Figure 3-4.) They are the extension of the run-time environment of the EJB container for particular bean types.

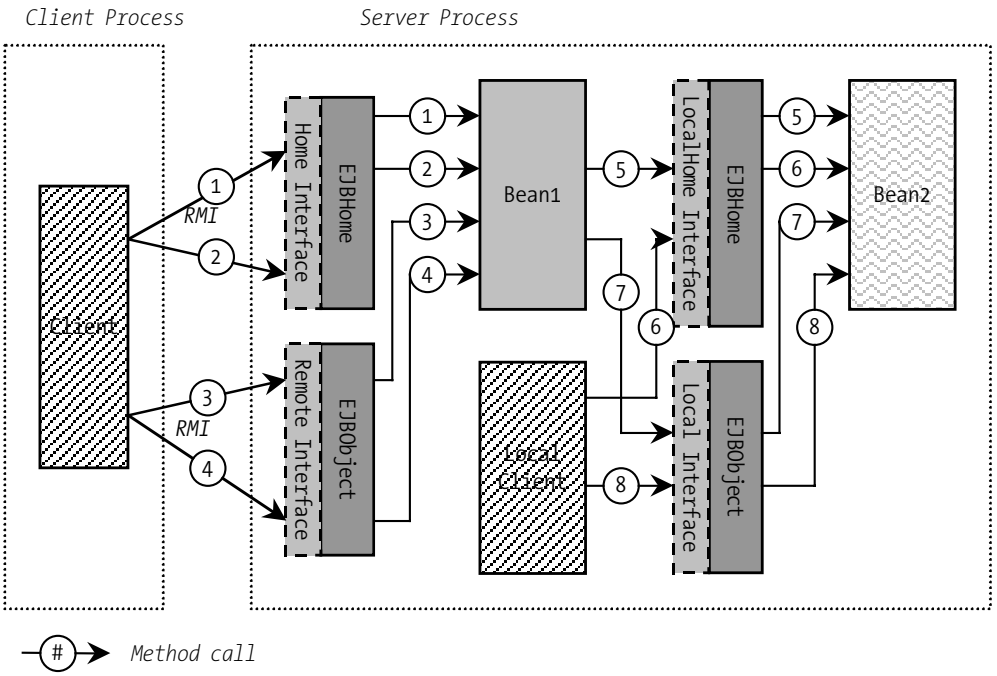

Figure 3-4. `EJBHome` *and* `EJBObject` *at run time.*

If the Enterprise Bean uses remote interfaces, then they are both remote objects in the sense of Java RMI. If the Enterprise Bean uses local interfaces, then they are traditional Java objects. In both cases the client calls upon the methods of the `EJBHome` and `EJBObject`, never directly those of the bean instance. `EJBHome` and `EJBObject` delegate the respective method calls (after or before particular container-related routines) to the bean instance.

With this indirection the EJB container is able to bring its implementation of the declarative instructions into the deployment descriptor. The content of the deployment descriptor influences considerably the generated code of the `EJBHome` and `EJBObject` classes.

The `EJBHome` class contains the code for the generation, location, and deletion of bean instances. Often, code is generated for resource management into the `EJBHome` class. The `EJBObject` class implements the transaction behavior, security checking, and, as needed, the logic for container-managed persistence. The implementation of these classes depends to a great extent on the creator. The specification makes no compulsory prescriptions for the creator with respect to the implementation of `EJBHome` and `EJBObject`.

No instance other than the `EJBHome` and `EJBObject` objects can cooperate with the bean instance. It is completely protected by the container classes. Communication between beans also always takes place by way of the container classes `EJBHome` and `EJBObject`.

From this state of affairs we obtain answers to questions that may have been left unanswered in the previous section. The home and remote interfaces do not need to be implemented, since the implementation classes are generated by the container tools. (The same holds in the case where a local home interface and local interface are used.) The Enterprise Bean is not a remote object, since it is never to be addressed externally. It should be able to be addressed via `EJBHome` and `EJBObject`. The EJB container would otherwise have no possibility of intervention to keep up with its tasks. Therefore, the interfaces `javax.ejb.EJBHome` and `javax.ejb.EJBObject` (the basic interfaces of the home and remote interfaces) are also derived from `java.rmi.Remote`.

It is even clear why the signatures of the bean methods and the methods in the home and remote interfaces must agree. The interfaces are implemented by `EJBHome` and `EJBObject` classes. They delegate the calls to the interface methods to the corresponding methods of the bean class. The code necessary for this is generated. If the signatures in the methods declared in the interfaces do not agree with the corresponding bean methods or if these methods don't even exist in the bean class, then a complaint will be registered in the generation of `EJBHome` and `EJBObject` by the container tools, or else the result will be a run-time error.

Finally, `EJBHome` and `EJBObject` ensure that in the case of session beans each client is able to work with an instance exclusive to it, while in the case of entity beans several clients can share the same instance.

The Client's Viewpoint

If a client now wishes to use the bank account bean installed in the EJB container, it must first locate the bean. For this it uses the naming and directory service. This is addressed via the JNDI interface. (See Figure 3-5.)

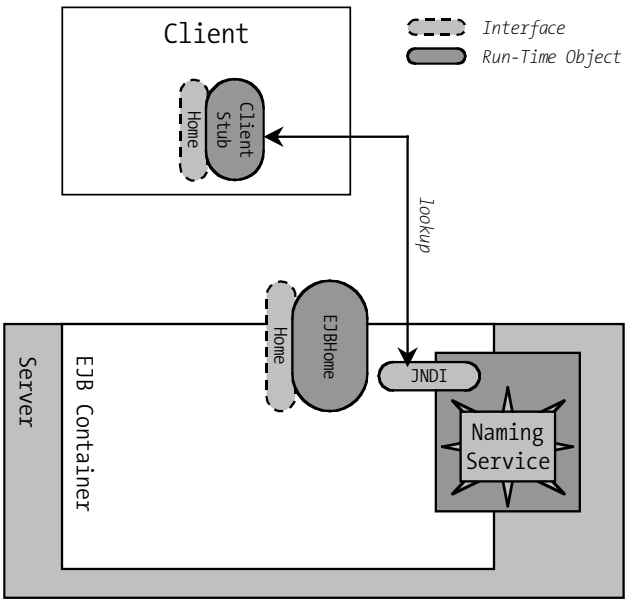

Figure 3-5. Finding an Enterprise Bean via JNDI.

Like the Enterprise Bean, the client uses the naming and directory service of the EJB container or of the server in which the bean is installed. The EJB container is responsible for making the Enterprise Bean accessible under a particular name via the naming and directory service. In many cases the name of the bean in the deployment descriptor is used for this purpose. The field of the deployment descriptor in which the name of the Enterprise Bean is entered is called `ejb-name` (and occurs in our `BankAccount` example). We now assume that the EJB container uses this name for the publication of the bean's name via the naming service (JNDI). Listing 3-7 shows how a client finds the bank account bean.

Listing 3-7. Finding the home interface using JNDI.

```
//depending on the creator of the container or server the appropriate settings
//in the environment should be made in order to generate the correct context.
final String BANK_ACCOUNT = "java:comp/env/ejb/BankAccount";
//generation of the context for access to the naming service
InitialContext ctx = new InitialContext();
//location of the bean BankAccount
Object o = ctx.lookup(BANK_ACCOUNT);
//type transformation; details on this in Section 4
BankAccountHome bh = (BankAccountHome)
 PortableRemoteObject.narrow(o,BankAccountHome.class);
```

Using the naming and directory service the EJB container provides the Enterprise Bean with an instance of the client stub of the implementation class of the home interface (`EJBHome`).

Using the methods of the home interface (and those of the interface `javax.ejb.EJBHome`) the client can govern the life cycle of the bean. For example, it can generate a new account with the following code fragment:

```
BankAccount ba = bh.create("0815", "sample account", 0.0);
```

The `EJBHome` object generates a new bean instance (or takes one from the pool), generates a new data set in the database, and generates an `EJBObject` instance (see Figure 3-6).

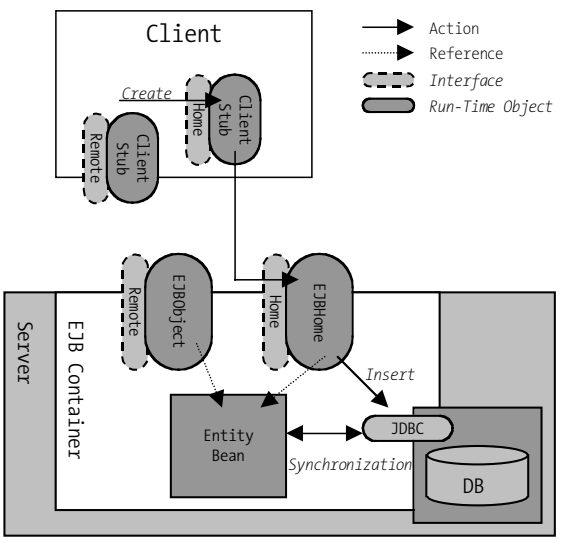

Figure 3-6. Generating a new bean via the home interface.

From the parameters of the `create` method a primary key object is generated, which is associated with the bean instance (via the context of the bean). In this way the bean instance acquires its identity. As a result of this operation the client is provided with the stub of the `EJBObject` instance. The stub represents the client with respect to the remote interface of the bean.

From this point on the client can use the functionality of the bean by calling the methods of the remote interface. For example, it can increase the bank balance by one hundred units:

```
ba.increaseBalance(100.0);
```

The call to the method `increaseBalance(float)` on the client stub passes to the `EJBObject` instance on the server. From there it is again delegated to the bean instance, which finally has the consequence of changing the data with the assistance of the transaction monitor (for this method it was specified in the deployment descriptor that it must take place in a transaction). (See Figure 3-7.)

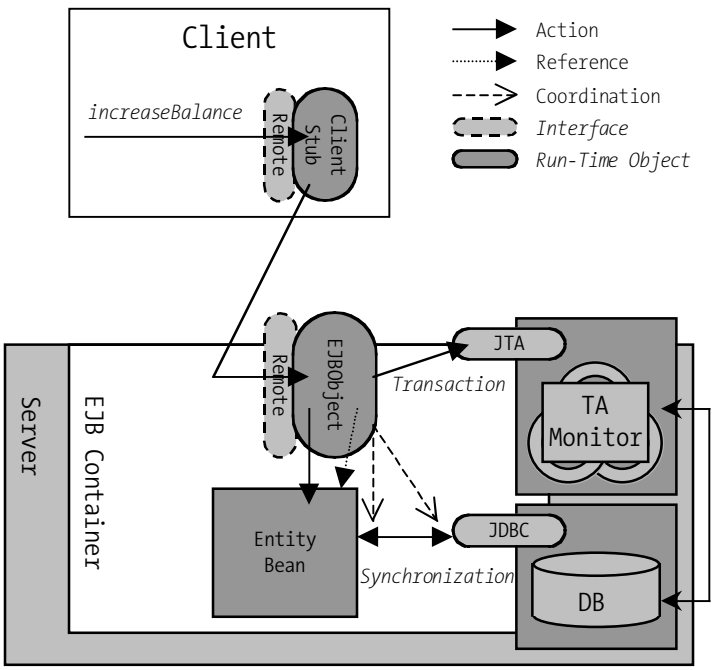

Figure 3-7. Invoking a method via the remote interface.

For the client there is no big difference between using an Enterprise Bean and a traditional Java object located in the same process, although it communicates with a transaction-protected component on a remote computer. What begins

with a one-line instruction in the code of the client can trigger very complex actions on the server.

From the client's point of view the way that an Enterprise Bean is used via the local interface is analogous to the use of an Enterprise Bean via the remote interface. Similar analogies hold in the processing of the `EJBHome` and `EJBObject` classes.

The information presented in this section applies primarily to session and entity beans. Message-driven beans differ from the other two types (as already mentioned) in that they cannot be addressed directly by a client. They can be addressed only indirectly via the asynchronous sending of a message. A message-driven bean is less complex than a session or entity bean and is also easier for the container to manipulate, since the message service takes over a large part of the work. Chapter 6 discusses this issue in detail and makes clear the differences between this type and the other two types of bean.

What an Enterprise Bean May Not Do

The specification of Enterprise JavaBeans is very restrictive for the developer of an Enterprise Bean as it relates to the use of certain interfaces. The most important restrictions will be presented in this section. What is most interesting is the question of the reason for such prohibitions. We shall investigate this question after we briefly introduce the most important restrictions (the complete list of restrictions can be found in [21]):

- An Enterprise Bean may not use static variables. Static constants, on the other hand, are permitted.
- An Enterprise Bean may not use a thread synchronization mechanism.
- An Enterprise Bean may not use the functionality of the AWT (Abstract Windowing Toolkit) to produce output via a graphic user interface or to read input from the keyboard.
- An Enterprise Bean may not use any classes from `java.io` to access files or directories in a file system.
- An Enterprise Bean may not listen in on a network socket; it may not accept a connection to a network socket; and it may not use a socket for multicast.
- An Enterprise Bean may not attempt to use introspection or reflection to access information from classes and instances that are supposed to remain secret according to the Java security policy.

- An Enterprise Bean may not generate a class loader; it may not use a class loader; it may not change the context of a class loader; it may not set a security manager; it may not generate a security manager; it may not stop the Java virtual machine; and it may not change the standard input the standard output, or the standard error.
- An Enterprise Bean may not use an object of the class `Policy`, `Security`, `Provider`, `Signer`, or `Identity` from the `java.security` package or attempt to change their values.
- An Enterprise Bean may not set a socket factory that is used by the classes `ServerSocket` and `Socket`. The same holds for the stream handler factory of the `URL` class.
- An Enterprise Bean may not use threads. It may neither start nor stop them.
- An Enterprise Bean may not directly read or write file descriptors.
- An Enterprise Bean may not load a native library.
- An Enterprise Bean may never pass `this` as an argument in a call to a method or `this` as the return value of a method call.

To "program" an Enterprise Bean means to develop server-side logic at a relatively high level of abstraction. The specification of Enterprise JavaBeans describes a component architecture for distributed applications. Through the component model the specification seeks to establish a clear distribution of labor among various tasks. The server and the EJB container are responsible for system-specific functionality. The bean uses this infrastructure as its run-time environment and thus need not concern itself with system-specific functionality. The task of the bean is to concentrate on the logic for enterprise-related processes. In order to be able to carry out this task, it should use the services made available to it by the EJB container, and no others. These restrictions imposed by the specification of Enterprise Beans should serve to ward off conflicts between the EJB container and the Enterprise Beans.

EJB Assignment of Roles

The specification of Enterprise JavaBeans splits the responsibilities for development into various roles. The idea is to achieve a certain level of abstraction in the EJB model in order to encourage diversification through the allocation of concrete tasks to various expert groups and thereby achieve a synergistic effect. To put it another way, everyone should develop what he or she can best develop. Figure 3-8 provides an overview of the various roles and their interaction.

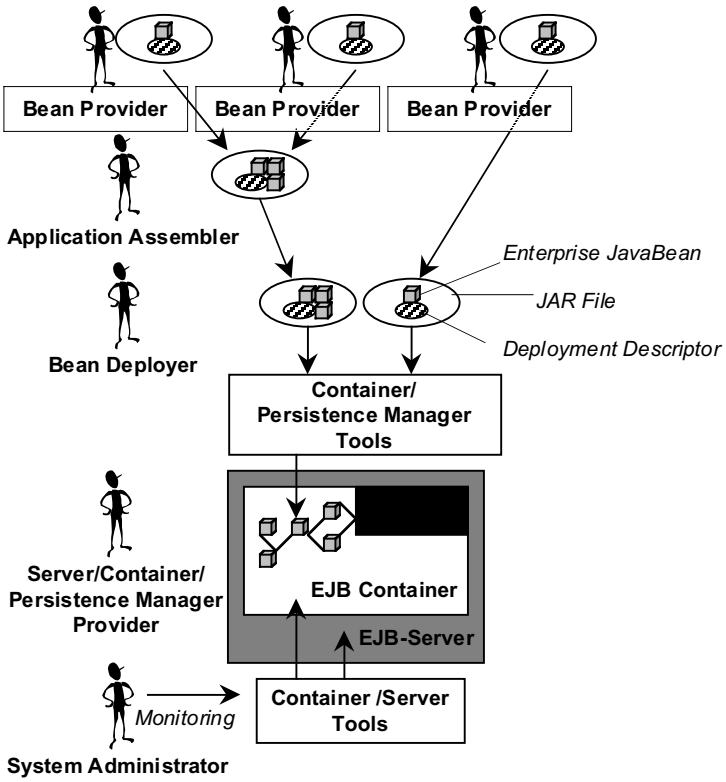

Figure 3-8. EJB assignment of roles.

Server Provider

The server provider is responsible for the provision of basic server functions. It must ensure that a stable run-time environment is available to various containers (e.g., above all, EJB containers). This includes—as already mentioned in the previous chapter—the network connection, thread and process management, sizing (clustering), and resource management.

Container Provider

The container provider sits on top of the interfaces of the server provider and offers the components (Enterprise Beans) a convenient run-time environment. The implementation of a container provider must be in accordance with the conditions set forth in this chapter. It must ensure that accesses to an Enterprise Bean take place solely via the container, and the same holds for the Enterprise Bean's communication with its environment. Primarily, the container

provider provides for persistence of components, mechanisms for dealing with transaction-oriented processes, security features, resource pooling, and support for making available versions of components (this last feature is not more precisely defined in the specification). For beans to be installed in a container, the container provider must make available tools that enable the bean deployer (see below) to generate the necessary interface code (`EJBHome` and `EJBObject`). The container provider must make tools available to the system administrator for monitoring the container and the beans.

Persistence Manager Provider

The persistence manager provider must provide tools with which the code needed for the persistence of a container-managed entity bean can be generated. The tools are used when an Enterprise Bean is installed in a particular container. The persistence manager provider is typically a specialist in the area of databases. It is therefore necessary that the following conditions hold:

- The state of an entity bean is stored with container-managed persistence in the database.
- The referential integrity of the entity bean is ensured with respect to other entity beans to which it is related.

Since the EJB container governs the persistence of an entity bean (that is, it determines, for example, when the state should be saved), an interface between container and persistence manager is necessary. The specification leaves it to the container and persistence manager providers to define such an interface.

Bean Provider

The role of the bean provider belongs to those developers who implement the actual business logic. They package their application knowledge into components that make this knowledge reusable. Building on the products of the server and container providers, the bean developer is freed from the development of basic server tasks (such as multithreading, network connection, transactions). Therefore, such a developer can devote his or her complete energy to the task at hand. In addition to the component, the bean provider provides the deployment descriptor, which contains information about the component itself, as well as about its external dependencies (e.g., on what services from the server the component depends). The deployment descriptor (an extensive example was presented in Listing 3-5) provides all information that the bean deployer (to be discussed shortly) requires for installing the component in a server.

The application assembler (see the next heading) also needs this information in order to assemble applications or modules of an application from various components. The result of the work of a bean provider is a file in JAR (Java Archive) format that contains the bean class(es) and the interfaces of the bean(s) as well as the deployment descriptor.

Application Assembler

Finally, the application assembler is responsible for linking the functionalities of the beans installed in the container to applications. To this can belong the development of client applications and the associated control of information exchange with the Enterprise Beans. The application assembler can, however, also assemble several (smaller) components provided by bean providers into a new (larger) component. The application assembler can enlarge this conglomerate with its own Enterprise Beans, whose task consists, for example, in the coupling of other Enterprise Beans. The resulting "supercomponents" already represent an aspect of a concrete application.

The application assembler documents its work, just like the bean provider in the deployment descriptor. Thereby the bean deployer (see the next section) understands how to install the components so that from the individual building blocks a complete application results. Additionally, the application assembler can provide instructions and information for the bean deployer in regard to the user interface or dependencies on non-EJB components in the deployment descriptor.

Bean Deployer

To install the components selected for a particular business problem in a container is the task of the bean deployer. To this belongs, above all, the provision of the tools of the container and persistence manager providers as well as the correct parameterization of the components to be installed. This operation assumes an extensive knowledge of the application and system contexts as well as of the internal system linkages. In particular, the deployer must resolve the external dependencies defined in the deployment descriptor (for example, by ensuring the existence of required services or related Enterprise Beans) and take into account the instructions of the application assembler contained in the deployment descriptor.

System Administrator

The system administrator monitors the EJB server with the assistance of the tools provided by the producer and takes care of the operation of the required infrastructure (e.g., for an operational network) of an EJB server.

The EJB assignment of roles represents an ideal scenario, and the translation of the EJB specification into practice will show whether it can be maintained in this form. Thus today servers and containers are offered in a single product. The Java-2 platform, Enterprise Edition (J2EE), already provides a variety of containers (at least an EJB container and a web container) for an application server. In the meantime, the specification of Enterprise JavaBeans assumes that server and container are always offered by the same producer and cannot be exchanged for others. The same holds for the persistence manager. As long as the specification defines no protocol between EJB containers and persistence managers, both will be offered in a single product.

This role-playing is supremely adapted (in theory) to component-oriented software, at least from the point of view that one is dealing with a number of individuals. Thus a container provider (a container is, after all, a type of component) does not know during development what beans will later be put into this container. Nor does a bean provider know at the time of development to what purpose those components will later be used. The provider merely packs a well-rounded functionality into a component with the goal of optimal reusability. If the correct components have been installed and parameterized by the bean deployer, then it is simple for the application assembler to link the functionality that is offered to applications or to create modules out of available components. The work of the application assembler is, moreover, not bound temporally to the work of the bean deployer. The application assembler can use, via the deployer, already installed components, or can transmit "supercomponents" that have assembled to the deployer for installation.

The roles of the server and container providers are intended for system specialists, who provide a stable basis. For an application developer (such as described in the introductory chapter, "Motivation") there are a variety of approaches. One could slip into the role of the bean provider by encapsulating one's knowledge into beans and then installing them in a purchased application server that offers an EJB container. Or instead, one could leaf through the catalog of a software manufacturer and look for components that offer a solution to one's problem(s). One's task would then be limited to installing the purchased beans on the server and correctly parameterizing them (corresponding to the role of the bean deployer).

In the next step our developer (or a colleague) could write an application using the installed beans that takes over the interaction between the user and the communication with the components (application assembler, limited to the development of the client scenario).

In any case, one thing is clear to the application developer (above all in the role of the bean provider): He or she can concentrate fully on the solution to his or her problems. Technical problems such as the implementation of a stable, efficient, and scalable server are taken over by the server and container providers.

In the rest of this book we shall for the most part restrict our attention to the roles of the bean provider, the bean deployer, and the application assembler.

Points of View

Once the architecture and the significant features and concepts of the specification of the Enterprise JavaBeans are known, we would like, in conclusion, to clarify various perspectives on the architecture.

EJB from the Point of View of Application Development

It is interesting to consider Enterprise JavaBeans from the point of view of application development. For this we should return to the scenario of Chapter 1. The application developer described there could use Enterprise JavaBeans as a basis for the prototype to be developed. He or she could use a J2EE-compatible server including an EJB container from any supplier and develop the prototype in the form of Enterprise Beans. We would like to consider whether the secondary problems that arise from the business problem can be neutralized using Enterprise JavaBeans as the basis technology. We wish to investigate whether the application developer can concentrate completely on the problems of the application domain. To this end we recall a figure from Chapter 1 (see Figure 3-9).

There is certainly the requirement of *reusability* due to the client–server-oriented architecture. There is also the necessity for security characteristics of the transaction service of Enterprise JavaBeans in order to ensure a smooth multiuser operation.

The characteristic of scalability means that applications that are realized as Enterprise Beans or as aggregates of beans can be partitioned among several servers. For example, the servers can use the same database as back-end system. Thus the burden of client queries, depending on the application, is divided among several servers, while the database is centrally located. Through locational transparency of Enterprise Beans with a remote interface (which is ensured by a central naming and directory service) the following are possible:

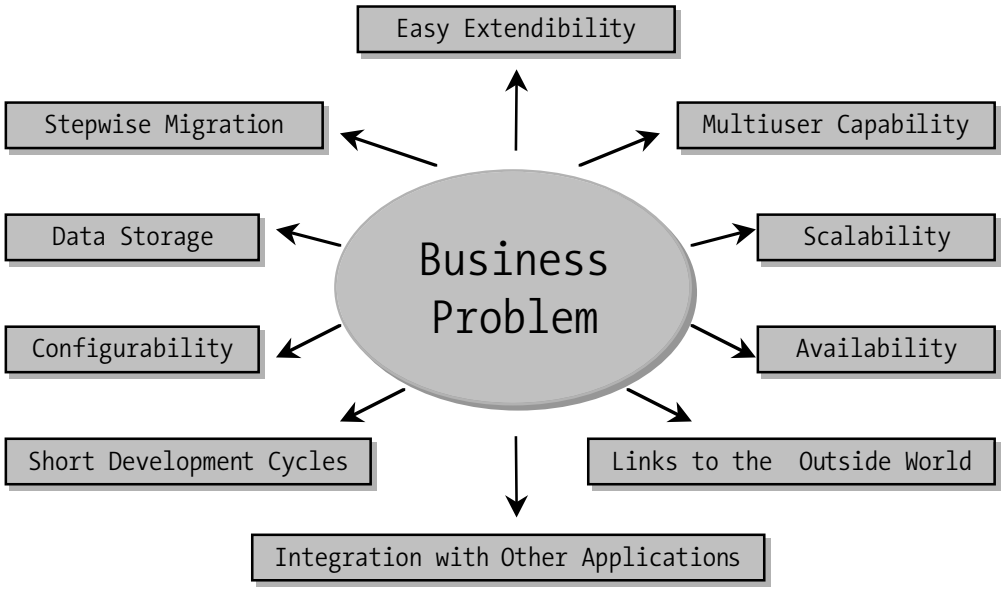

Figure 3-9. Secondary issues in a business problem.

- Distribute individual applications of an application system among several servers.
- Store individual beans from one server on another server.
- Set up additional servers and store beans or aggregates of beans there.

Moreover, many application servers offer mechanisms for the allocation of tasks. In this regard a cluster of application servers (independent of the application that the client uses) concerns itself with the optimal servicing of the client's queries. However, this feature is provider-dependent and not required by the architecture of Enterprise JavaBeans.

The subject of availability becomes significant in regard to the locational transparency of Enterprise Bean components, which is ensured by the central naming and directory service. If an application server goes out of service, the beans installed there are installed on another server. The information necessary for the installation is already contained in the deployment descriptor of the Enterprise Beans, and they can generally be transported unaltered. A change in the configuration in the naming and directory service allows them to be placed at the service of clients on another server without the configuration of the client having to be changed.

Since Enterprise JavaBeans are ever more tightly bound in the context of the Java 2 platform, Enterprise Edition, the *connection with the outside world* is not a critical topic. With the support of a web container (see Figure 3-2), at

least the basic technological and architectural prerequisites are given to create a connection with the outside world. In Chapter 9 we shall devote ourselves to this topic in somewhat greater detail.

Once all the uses of an application system have been realized based on Enterprise JavaBeans, then the potential for integration of the applications among one another is very high. The exchange of data within the application is simple in this case (even more so if a central database is used). Enterprise JavaBeans can also be used to integrate applications belonging to various systems. Using a message service, which the EJB container has had to integrate since version 2.0, it is even possible to achieve dynamic coupling of applications and systems. A message service makes a number of communication channels available over which n-to-n communication with dynamically changing partners is possible.

Through the use of specialized EJB containers, which use an application system other than a database system as persistence system, these systems can be integrated into EJB-based systems. Under certain conditions one can also envision employing Enterprise Beans as wrappers and interfaces to simple systems (to the extent to which that is possible within the limits of programming restrictions).

Applications are constructed from a number of Enterprise Beans, each of which encapsulates a particular functionality. This granularity, as established by the component paradigm, makes it possible for the development of an application system to be structured from a variety of viewpoints. Each component can be a partial project. It is easier to manage each subproject than to keep track of the entire project all at one time.

The interfaces to the outside world are defined for each component (and thus for each subproject). This results in a trend toward shortening the development cycle. Not least, the architecture of Enterprise JavaBeans plays an active role, since the application developer (i.e., the bean developer) is freed from the development of system-technical functionality. The presence of a unified platform allows this system-technical functionality to be reused, and the bean developer can concentrate fully on the solution of the application-related problems.

The configurability of an EJB-based system is given primarily via the deployment descriptor. This makes it possible for beans, at the time of installation in a server, to adapt themselves to a variety of situations. The run-time behavior of an Enterprise Bean can also be influenced by the settings in the deployment descriptor. Using the bean environment, parameters can be set that the bean can evaluate at run time.

An EJB-based system can also be configured via the exchange of bean classes. In fact, the new bean classes offer the same interface; that is, they use the same home and remote interfaces as the old bean classes, but they offer a different implementation. To the client the beans with altered implementation appear under the same names in the JNDI as before. For the client the exchange

of implementations plays no role, in that the old agreement (the promised functionality of the interface) is preserved.

In the case of container-managed persistence, Enterprise JavaBeans offer the concept of transparent, automatic data storage. The Enterprise Bean leaves data storage to the EJB container and the persistence manager. It need not concern itself with the technical aspects of data storage. Such a bean can be used without difficulty in other EJB containers.

Through the deployment of specialized EJB containers (as can be seen in Figure 3-10) it is possible to achieve a stepwise migration from the old system. The state of the data and the systems can be reused. For the bean the EJB container is not only a run-time environment and service provider but also the interface to the old system (which to the bean is essentially transparent). After the complete takeover of the data (for example, into a relational database system) the beans already developed can be reused through installation in another EJB container.

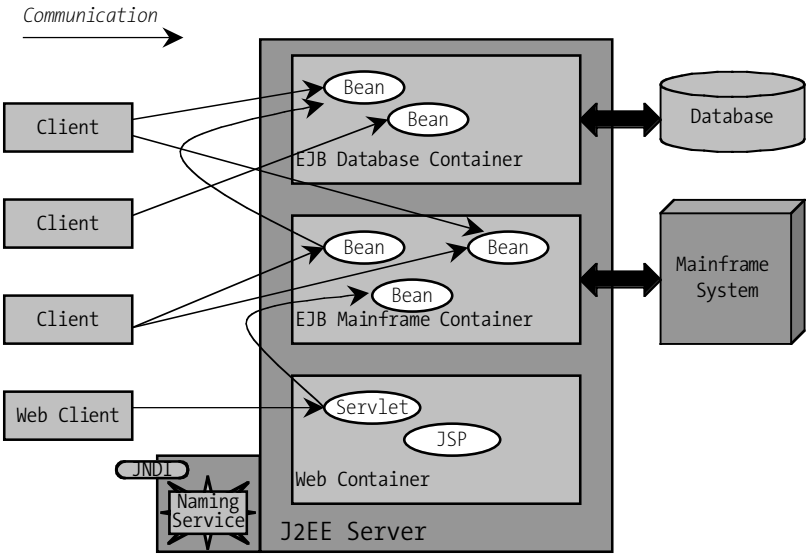

Figure 3-10. An Enterprise JavaBean scenario.

It is possible to extend the functionality of an EJB-based system without difficulty by installing new Enterprise Beans. Under certain conditions new beans can make use of the functionality of already existing beans.

From the viewpoint of the application developer as described in the scenario of Chapter 1 all the secondary problems that arise can be neutralized by the use of Enterprise JavaBeans. EJB was the correct approach to the solution of the problem.

EJB from the Point of View of the Component Paradigm

An additional point of view from which Enterprise JavaBeans can be considered is that of the component paradigm. In the preceding chapters some basic points about this were presented. Two questions then arise:

- Are Enterprise Beans genuine components?
- Do Enterprise JavaBeans fulfill the requirements of the component architecture?

To elucidate these questions we must measure the properties of Enterprise Beans and the EJB architecture against the requirements presented in Chapter 2.

In view of the definition presented in Chapter 2, Enterprise Beans are genuine components. The fundamental characteristic of the standardized interface is given in the form of the home and remote interfaces or the local home and local interfaces. The interface is standardized by the guidelines of the component model of the Enterprise JavaBeans specification. The interface represents a sort of contract to which an Enterprise Bean is compelled to adhere. This interface is independent of the implementation. A bean can be developed and maintained as a single unit. Whether this represents a self-contained functionality depends on the developer. This characteristic of a component cannot be compelled by the specification, but it should serve as a guideline to the developer of an Enterprise Bean.

In order to determine whether the architecture of the Enterprise JavaBeans does justice to the requirements of the component paradigm, we would like to evaluate the various characteristics that we presented in Chapter 2:

- *Independence of the environment*: Enterprise JavaBeans are based completely on the programming language Java. Enterprise Beans can be deployed only in a Java environment. However, via CORBA they can be used in conjunction with other environments. The specification of version 2.1 designates RMI-IIOP as communications protocol, which should ensure compatibility with CORBA.
- *Locational transparency*: The location of a component with a remote interface is fully transparent, since the Enterprise Beans can be located via the naming and directory service as well as RMI, and also can be addressed on remote computers. The specification also requires a producer to include a naming and directory service in the system environment and emphasizes the distributed nature of Enterprise Beans. Locational transparency is not a characteristic of Enterprise Beans with a local interface.
- *Separation of the interface and implementation:* This property is provided to Enterprise JavaBeans through the concept of the home and remote, respectively local home and local remote, interfaces.

- *Self-descriptive interfaces:* In the case of Enterprise JavaBeans it is possible to obtain information about the remote interface(s) of a component. The home object uses the method `getEJBMetaData( )` to return an object of type `javax.ejb.EJBMetaData` (for relevant details see [21]). This object provides, among other things, information about the home and remote interfaces of an Enterprise Bean. Using the Java reflection API (see [4]) it is possible to investigate the interfaces at run time and to program dynamic calls to Enterprise Bean methods. The combination of naming and directory services (JNDI), the metadata interface (`javax.ejb.EJBMetaData`), and the Java reflection API offers possibilities comparable to those of the interface repository of CORBA (cf. [17]). It is the task of the EJB container to generate for an Enterprise Bean a corresponding implementation class for the metadata interface from the specifications of the deployment descriptor.
- *Problem-free immediate usability (Plug & Play):* At the time of installation of an Enterprise Bean the implementation classes for the home and remote interfaces must be generated and compiled. By means of these an Enterprise Bean component becomes usable (without it having to be altered in the process) by the EJB container of a particular producer. All information that is necessary for the installation process is contained in the bean class itself and in the deployment descriptor (which is a part of the component). The binary independence of the component code of an Enterprise Bean exists to the extent that the components comply with the binary standard defined by the Java programming language. This results is problem-free usability of an Enterprise Bean component in every EJB container.
- *Ability of integration and composition:* This issue is accommodated by a certain role in the EJB process model, the application assembler. The composition of Enterprise Beans into an aggregate is specifically provided for.

Enterprise JavaBeans can be considered a component architecture. It will be interesting to see what extensions and improvements are offered in future versions.

EJB from the Enterprise Point of View

A final point of view is that of Enterprise JavaBeans from the point of view of the Enterprise. In Chapter 2 we defined criteria that a basic technology should satisfy from this viewpoint: economic viability, security, and meeting the enterprise's requirements. It is difficult to give an objective evaluation of a technology according to such global criteria. Nonetheless, we may certainly

conclude that Enterprise JavaBeans, on grounds of economic viability as defined in Chapter 2, are certainly of interest to an enterprise, particularly for those businesses that engage in application development (whether internally or for external customers).

In the sections above we have already seen the advantages of such a technology. In the development of enterprise-related logic in Enterprise Beans the developer is freed totally from having to deal with technical system issues. The technological basis has been developed by specialists in this field and conforms to a (quasi) standard. Thus development can be focused directly on the problems of the particular enterprise. The component-oriented point of view enables a greater granularity and thereby makes development more transparent and more easily kept under control. The EJB technology offers all the prerequisites for allowing the application systems to grow with the enterprise (with respect both to greater functionality and to growth in the number of employees).

With respect to security the enterprise is supported not only by the security mechanisms of the specification and the programming language Java. Since Enterprise JavaBeans defines a standard in a certain sense that is already supported by a number of producers, there are relatively few dependencies on a particular application server vendor. It would, however, be desirable to achieve an official standardization of Enterprise JavaBeans through the work of a standardization committee. Nevertheless, an investment in this technology could pay off over a long period of time.

The specification defines many conditions that ensure the fail-safe operation of EJB-based systems. An example is the stringent restrictions on the development of beans. Among other things, this should help to avoid unstable conditions in the system. Another example is the separation of system functionality and enterprise-specific functionality. It benefits reliability and security that the system functionality has been developed by a specialist. A wide deployment of the system among many customers by a responsible developer should lead more rapidly to a higher quality than what could be obtained by in-house development.

As for the needs of enterprises, the EJB specification is directed precisely at those enterprises that develop their own applications or act as a service provider for other enterprises. As a rule, the developers in such enterprises are specialists in particular application areas, but not in the domain of system development. The result is often applications that deal with the technical problems, but cause dissatisfaction through shortcomings in the technological foundations. With the model of Enterprise JavaBeans it is precisely this issue that is dealt with. The demand for a stable basic architecture that offers a convenient component model that can be embedded in the application logic should exist in many areas and in many enterprises. Precisely because applications deployed in businesses are becoming ever more complex, there is ever increasing demand for stable, secure, and flexible basic architectures.