Support for Enterprise JavaBeans in Component Broker

by C. F. Codella
D. N. Dillenberger
D. F. Ferguson
R. D. Jackson
T. A. Mikalsen
I. Silva-Lepe

Objects were introduced as programming constructs that encapsulate data and methods. The goal was to foster software reuse and simplify the developer's concept of how a task was implemented. The developer need only know the interfaces to an object to use its functionality. Distributed objects simplified conceptualization further by removing the need to know the locality of an object. Clients invoked methods on distributed objects as if the objects existed in the client's process. Beyond this location transparency, the need arose for distributed objects to survive beyond the life of one client, to be able to support thousands or millions of clients, and to participate in transactions. To support scalability, persistence, and transactional semantics with no dependencies on platform or data store, "component models" were developed. In this paper we look at various component models, focusing on two: IBM's Component Broker and Sun's Enterprise JavaBeans™. We show that they augment each other and propose how Enterprise JavaBeans can use the additional functions of Component Broker to provide a scalable, transactional, and persistent environment to clients of both worlds.

The emerging component model for Java**, JavaBeans**, defines "pluggable" program elements. These elements can be manipulated and run in a visual builder. They are customizable and portable and can be made persistent. They are capable of introspection, either by being self-describing or by adhering to interface-naming conventions that imply certain behavior patterns.

Enterprise JavaBeans** (EJB) extends the JavaBeans component model to distributed server components that support transactions. This capability is an important part of the Java programming model for multitier distributed applications.

Component Broker[1] (CB) is IBM's comprehensive implementation of the Object Management Group's[2] Common Object Request Broker Architecture** (CORBA**), which sets the standard for component-based interoperability of software from different vendors. It is imperative that Component Broker support the building, installation, and running of EJB components on CB servers and that it support Java programmers who build CB applications that use EJB components.

In this paper, we discuss component models and, in particular, Component Broker. We propose how EJB components can use Component Broker environments. We look at support for Java clients and CB clients, support for transactions, and run-time support. The paper concludes with a summary and some potential problems that are yet to be addressed.

0018-8670/98/$5.00 © 1998 IBM

Component software and distributed objects

As the industry grows, the need for software to be less complex, reusable, and platform-independent has resulted in the creation of component programming models. *Objects* encapsulate code and data. They provide software reuse through inheritance and defined method interfaces. However, only the language compiler that creates an object knows of its existence. An object "lives" within a single program. External workstations or systems do not know about the existence of these objects or how to reach them.[3]

Distributed objects can live anywhere on a network. They can be accessed via method invocations by clients, who need not be aware of the language and compiler used to create them, or on which machine or operating system the objects are executing. To achieve this transparency, distributed objects use services defined by a *component model*. Component models such as CORBA, CB, EJB, and Microsoft's Distributed Component Object Model, differ in the level of functionality and quality they provide. Some services that are needed to achieve object transparency are: security, licensing, versioning, life-cycle management, support for open tool palettes, event notification, configuration and property management, scripting, meta-data and introspection, transaction control and locking, persistence, ease of use, and self-installation.[1]

In this section we describe the JavaBeans, CORBA, Enterprise JavaBeans, and Component Broker component models.

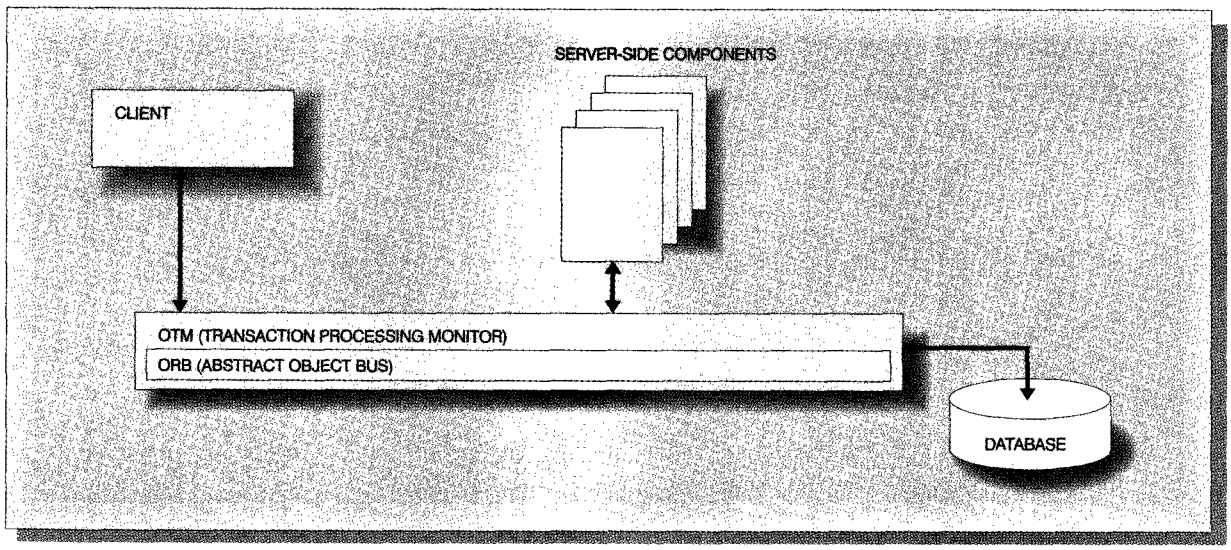
JavaBeans and Enterprise JavaBeans. The JavaBeans component model specifies how components expose their properties, methods, and events. Although beans can use the Java Development Kit (JDK**) as a component framework, a bean does not derive from some universal base class that gives it bean-like properties. Almost anything written in Java can be made into a bean. In fact, according to the JavaBeans specification, any Java class is a bean. This means that the JavaBeans specification defines no constraints. Introspection is what differentiates beans from ordinary Java classes. As long as a bean follows the defined conventions, a tool can look inside and discover its properties and behavior. The JavaBeans naming conventions, also known as JavaBeans design patterns, include conventions for naming simple properties, Boolean properties, indexed properties, multicast events, unicast events, and public methods. In addition, a developer can define a BeanInfo class to provide (via descriptors) all or parts of the bean's introspection information. This is optional, as the BeanInfo class can be generated for any bean that follows the naming conventions by using the Introspector class. The JavaBeans component model packaged with the JDK supports security (using the Java Security Manager), versioning, life-cycle management, support for open tool palettes, event notification, configuration and property management, scripting, meta-data and introspection, persistence (using serialization), ease of use (via the BeanBox, a simple test container), and is self-installing (via Java archive files).

In contrast, Enterprise JavaBeans defines a contract for how server-side components interact with their container, in which the container acts as a framework that expects application-specific beans to behave according to a given set of rules. To satisfy some of these rules, an EJB component must derive from an appropriate base class. Naming conventions are not defined as explicitly in the EJB specification and, although introspection can be used at the time an EJB component is deployed, the preferred way of communicating deployment information is via a deployment descriptor. A deployment descriptor is analogous to a BeanInfo class and is used by a deployment agent. EJB defines interfaces and functional behavior for security, life-cycle management, configuration and property management, meta-data and introspection, transaction control and locking, and persistence. We describe these services in more detail in subsequent sections. For more background on EJB, see Matena and Hapner[4] and Brackenbury et al.[5]

CORBA. The Common Object Request Broker Architecture (CORBA) defines a framework for multiplatform, multilanguage distributed object interaction.[6] CORBA defines: (1) an abstract object bus[7] for objects to send requests and receive replies, (2) an interface definition language (IDL) for defining abstract interfaces that objects can implement and invoke, (3) a number of object services that augment the functionality of the object bus (e.g., the naming service or the object transaction service), (4) a number of facilities that define "horizontal" and "vertical" application frameworks that are used by business objects, and (5) application objects that are the consumers of the CORBA framework services.

Enterprise JavaBeans uses remote method invocation (RMI) IDL for its distributed object model and the Java Transaction Service (JTS) for its distributed

Figure 1 OTM architecture



transaction model. A subset of RMI, Java's remote method invocation protocol and application programming interface (API), is mapped onto CORBA by RMI/IDL and RMI/IIOP (Internet Inter-Orb Protocol). JTS is a Java implementation of the CORBA Object Transaction Service (OTS). EJB also requires CORBA IIOP to interoperate across multivendor servers, propagate transaction and security contexts, service multilingual clients, and support ActiveX** clients via DCOM** (Distributed Component Object Model)-to-CORBA bridges.

Enterprise JavaBeans augments CORBA by defining the interfaces between a server-side component (bean) and its container. The container can be an Object Transaction Monitor (OTM). An OTM is a combination of a transaction processing (TP) monitor with an object request broker (ORB), more specifically, a TP monitor built on top of an ORB. A TP monitor behaves as an intermediary between client and server processes to manage transactions, route them across systems, balance their execution loads, and restart them after failures. This gives a server system with limited resources the scalability it needs to serve large numbers of client processes.

An ORB is simply an object bus. In contrast, an OTM provides a framework for running server-side components (see Figure 1). The OTM framework is the primary orchestrator of server-side components; it calls components at the right time and in the right sequence. An OTM maximizes the reuse of scarce system resources by components. It prestarts pools of objects, distributes their loads, provides fault tolerance, and coordinates multicomponent transactions. Some of the activities performed by an OTM are: activation and deactivation of components in memory, coordination of distributed transactions, notification of life-cycle events (such as creation, activation, deactivation, and destruction) to components, and management of the persistent state of a component.

IBM's Component Broker is an example of a CORBA OTM that can be used to deploy EJB components by implementing the contract between a component and its container.

Component Broker: Distributed object middleware

As mentioned earlier, an OTM coordinates large numbers of server-side components using a framework-based approach. In this approach, the framework embodies the main pieces of server-side infrastructure, such as activation and deactivation, distributed transaction coordination, and persistent state management. Components in this approach become subservient to the framework, providing the application-specific details (such as what application-specific activities to perform prior to deactivation)

by plugging into, and thus completing, the framework. Component Broker builds on a CORBA ORB, borrowing elements from TP monitors such as Encina**[8] and CICS* (Customer Information Control System), resulting in a framework for managed components that allows mixing in most of the object services defined by CORBA. In addition, Component Broker provides instance managers that manage component state, mapping it to data in a database management system (DBMS).

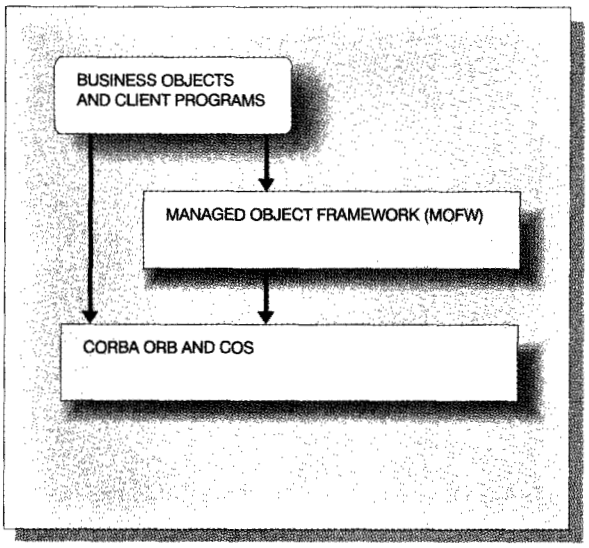
This section provides an overview of the main concepts in Component Broker. The Component Broker run-time infrastructure includes the Managed Object Framework, the Instance Manager Framework, the Object Transaction Monitor, and a collection of CORBA-compliant object services. Component Broker also provides tools for the creation of business objects and for installing, running, and monitoring servers. In addition, Component Broker defines a Client Programming Model: the activities that a client can perform using the Component Broker infrastructure.

The Managed Object Framework. The Managed Object Framework (MOFW) represents the set of interfaces and implementation conventions that must be followed in order to create and use business objects in Component Broker. The MOFW provides capabilities beyond those present in the basic CORBA ORB and object services defined by the Object Management Group (OMG). The MOFW also provides simplified interfaces to some of the basic CORBA interfaces. The MOFW is not the only set of interfaces supported by Component Broker. Component Broker architecture allows additional frameworks that can be used by business objects and client programs. The relationship between the MOFW and the CORBA ORB and Common Object Services (COS) is shown in Figure 2.

As the figure shows, business objects and client programs that use business objects can be written both to the MOFW interfaces and directly to the CORBA ORB and COS. The MOFW is not a complete layer over the CORBA services. It adds usability and function only in those places key to providing an integrated object server.

Managed objects and nonmanaged objects. Basically, two kinds of objects are dealt with in Component Broker: those that are managed by a Component Broker server and those that are not. All of the objects that client application programmers and

Figure 2 Overview of MOFW

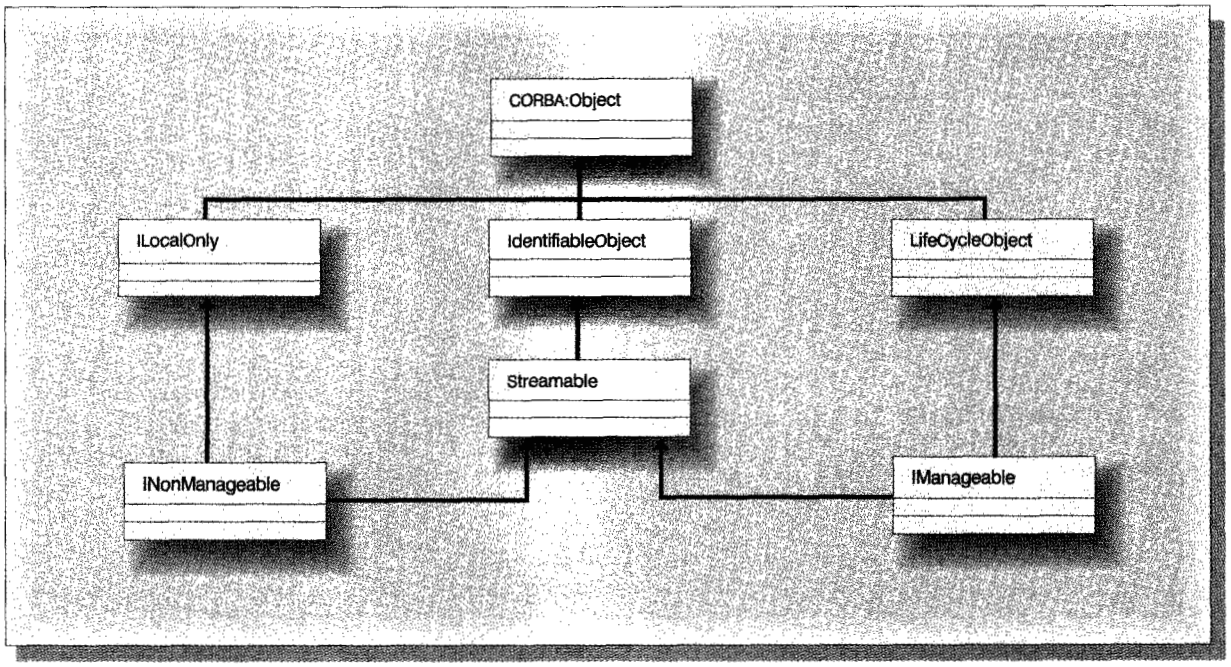


business object builders use will be instantiated from classes that descend, directly or most often indirectly, from either IManagedLocal::ILocalOnly or IManagedClient::IManageable.[9] This ensures a minimum "footprint" for the client, separation of server-only objects from those that may exist on either client or server, and, most importantly, simplicity for the programmer. No extra methods need to be used or implemented because of this distinction.

Those objects that are to be local only will be instantiated from classes that are descendants of ILocalOnly (perhaps through INonManageable). Those objects that are to be accessed remotely and managed by a Component Broker server will be instantiated from subclasses of the IManageable interface. Figure 3 shows the basic relationship between these important MOFW interfaces and the CORBA object services interfaces. In this diagram, the boxes represent classes and the arrow-terminated lines represent inheritance relationships.

A managed object has a rich structure that includes its client interface, business logic, business object state, and MOFW logic. Figure 4 illustrates a simplified view of this structure using an account business object as an example. In this diagram, the dotted arrows represent a dependency relationship from one class to another, e.g., an AccountBO object depends on an AccountDO object for the management of its state data.

Figure 3 MOFW basic abstractions



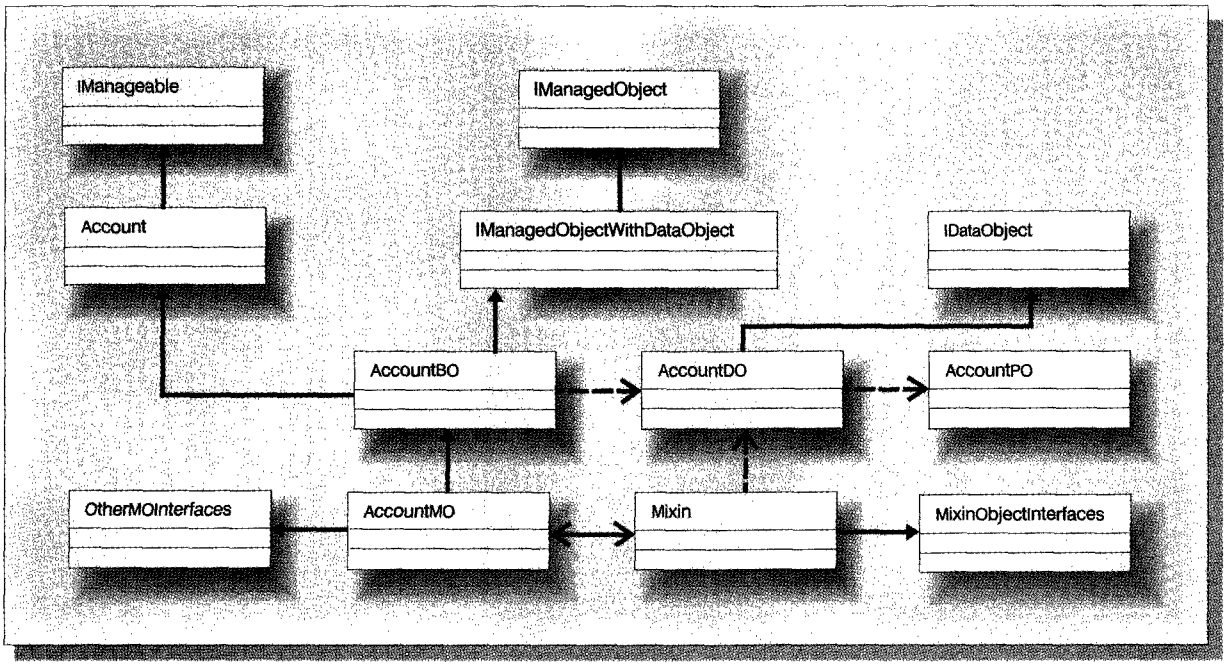
The box labeled Account defines the client interface of this business object, that is, the attributes and operations available to a client of the account business object. The fact that Account inherits from IManageable indicates that an account object is accessible remotely and that it can be persistent, accessed securely, participate in transactions, and take advantage of all the additional Component Broker features. AccountBO implements the business logic defined in the client interface, from which it inherits. In addition, AccountBO also inherits from IManagedObjectWithDataObject. The purpose of this class is to provide an approach for handling essential business-related state information. This approach is to delegate setting and getting attribute values to a data object, which handles all interactions with the storage mechanism. Other approaches include caching the attribute values in the business object and synchronizing with the data object at appropriate times, as well as using no data object at all.

The data object that interacts with the storage mechanism is defined by AccountDO. This object provides an abstraction of a persistent store, by presenting essential state data attributes in a storage-mechanism-independent fashion to the business object. The data object can use a "persistent object" to achieve improved performance. Interacting with a persistent object is one of the main responsibilities of the data object.

A persistent object provides transformations between storage mechanism data types and an object's attribute types. This object also interacts with the storage system's cache. There is a separate persistent object for each business object/data object pair, and it represents all of the data associated with the object.

Many of the interfaces that a business object class inherits from the MOFW, and which make it possible for the object to be actually managed, are not necessarily related to the essential logic of the business object itself and thus should not be the concern of the implementor. The managed object subclass AccountMO completes the implementation of inherited interfaces from the business class. It also inherits a few additional interfaces that allow a managed object to be fully managed within the Component Broker environment. AccountMO does not implement all of these interfaces by itself. It relies on a delegation scheme involving a "mixin" object.

Figure 4 Simplified managed object structure



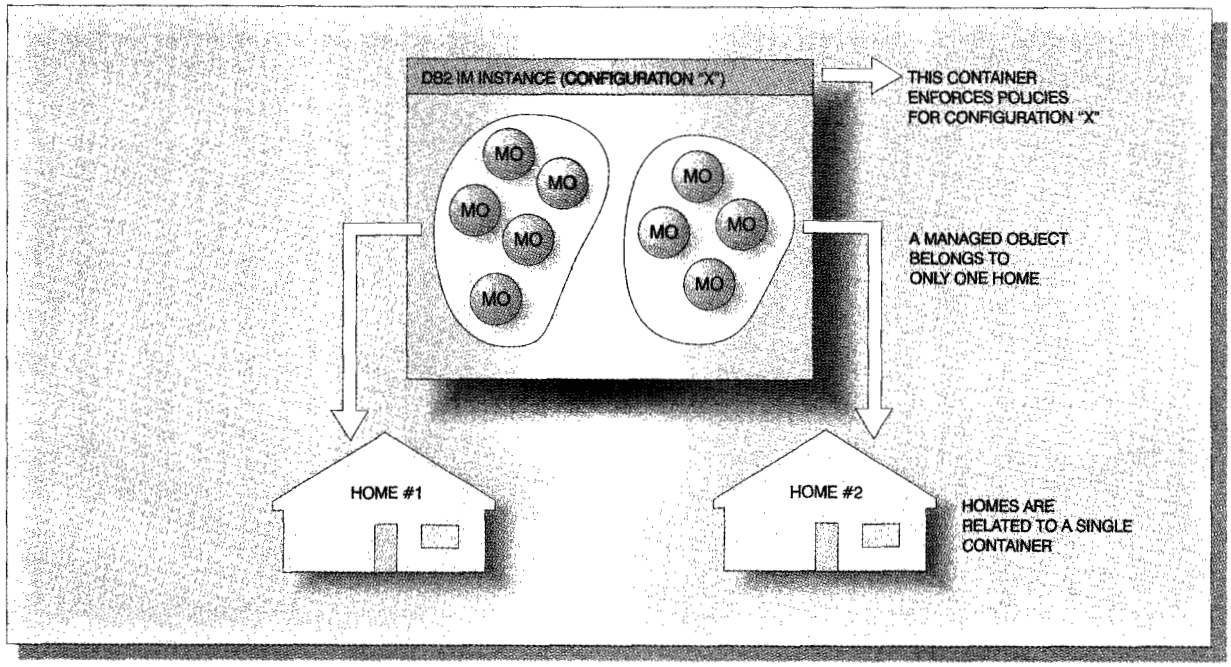
The mixin object is responsible for registering a business object with the transaction service, triggering the movement of data into and out of persistent storage, and so forth. A mixin object makes use of object services to provide appropriate behavior where necessary. Some mixin behavior can be controlled at run time based on interactions between the mixin object and systems management, as we discuss later.

The Instance Manager Framework. A managed object's relationship to an instance manager is similar to an object's relationship to an object-oriented database. That is, an instance manager provides capabilities such as identity, caching, persistence, recoverability, concurrency, and security for its managed objects. The Instance Manager (IM) Framework consists of interfaces that apply to all kinds of instance managers; for example, "container methods" that report the total number of active objects, passivate some or all objects (that is, force objects to remove themselves from memory and save their state as appropriate), and so on. This is in contrast to interfaces that apply to traditional legacy storage mechanisms such as DB2* (DATABASE 2*) and CICS, and that are provided by specializations of the IM Framework, such as the Business Object Instance Manager (BOIM) Framework.

The BOIM Framework actually constitutes a complete, runnable instance manager, as well as being a framework that can be further extended and implemented. As an example, Component Broker provides a DB2 instance manager that is an actual instantiation of the BOIM Framework and parameterizes BOIM with appropriate configurations. A particular kind of IM can only handle one type of legacy store. Thus the DB2 IM provides support for managed objects that are persistent in DB2 tables. A different kind of IM would be required for managed objects that are stored through CICS. An IM, or more specifically a BOIM, includes containers, "homes," mixin objects, and configuration objects, among others.

An IM container can be thought of as a further configuration of a particular IM. Thus, we could configure a DB2 IM in two different ways (to implement two different transaction policies, or to connect two different databases, for example). A container provides a configurable boundary for system administration and management as well as a facility for storing objects. In addition, a container has the responsibilities of interfacing with the ORB to resolve object references, interacting with homes to reactivate managed objects after they have been passivated, and

Figure 5 Instance Manager containers and homes



defining policy for managed objects (this includes decisions on passivation, caching, locking strategy, transaction modes, and concurrency control).

A home provides a way to create and locate managed objects. Thus, a given home is associated with a single type of managed object. For example, there is only one home to create and locate account objects and that home cannot create or locate any other kind of managed object. In addition to creating managed objects, a home brings them into existence by reactivating them if they have been passivated. In both of these cases, the home registers a managed object with its container after bringing it into existence. In this way, the container will be able to resolve references to a managed object until it is passivated or removed.

There can be many homes in a single container. Homes can be added to a container dynamically if it becomes necessary to support a new type of object. Homes that are associated with the same IM will necessarily be related to the same kind of legacy storage environment (for example, DB2), and their managed objects will be subject to the same kinds of policies.

Thus we see that a container represents a configuration of an IM instance, a home is related to a given container, and a managed object is owned by exactly one home. This is illustrated in Figure 5.

A mixin object is a special object provided to a business object by an instance manager. The mixin object integrates support for transactions and concurrency into the business object; it manages persistence on behalf of the business object; it externalizes the managed object key; and it participates in memory management activities. The mixin object accomplishes these tasks by relying on the special "before" and "after" methods provided by the mixin interface. These methods are invoked on the mixin object by other objects at appropriate times, and they allow transactions to be properly committed, states to be restored upon first reference, and so forth.

When a home creates a managed object, a special configuration object gets involved in the process. The configuration object is provided to the home by the container. It knows the policy information related to the container and thus the type of mixin that needs to be created. The configuration object is the instance

manager component that actually creates and initializes the mixin object.

The Object Transaction Monitor. Component Broker's Object Transaction Monitor (OTM) provides high availability and scalability of business objects by performing load balancing of Component Broker processes or servers in which business objects execute. The typical approach for load balancing across a cluster of servers is to assemble them into a server group. A server group consists of one or more Component Broker servers that can run on one or more hosts. Scalability is achieved by balancing the workload across the servers in a server group and by adding more servers if needed. High availability is achieved—if one server becomes unavailable, the other servers in the group can take over its responsibilities.

A server group-aware client, that is, the client side of the OTM, gathers information about servers in a server group and their policies by communicating with the group's control-point server. The client can then send requests to each server in the group according to the bind policies that are in effect. An alternative to an aware client would be to use a router. However, a single router implies a single point of failure and may become a performance bottleneck, and using multiple routers just shifts the client awareness problem from the servers to the routers.

A server group-aware object is any object that can be used to balance work in a server group. When a method is invoked on a server group-aware object, the OTM determines to which of the servers in the group to send the request. The object always appears to the client as a single object with a single object reference, even if it is activated concurrently in more than one server by different clients. From the client's point of view, there are two different approaches to distribute the requests to different servers: always use the same server if it is available, or alternate among servers.

Bind policies determine which server of a server group should be selected to receive the next request for a specific object. The bind policies that apply to a given object, according to the system configuration, are used to rank the available servers in a group. Each policy can adjust the weighting given to each server or mark the server as "impossible" for selection. The server with the lowest accumulated weight will be selected. Bind policies cover different aspects of server selection. Some may be used for load balancing, using a "round-robin" algorithm to distribute the load among servers. Other bind policies can determine the selection of objects involved in a transaction by marking all of the servers, except the one that runs the transaction, as "impossible" for selection. By combining these policies it is possible to determine which server best satisfies the availability and scalability requirements of a given request.

Object services. Component Broker supports CORBA-compliant implementations of the object services listed below. Rather than describing in detail what these services are, we indicate how the Component Broker implementations enhance their standard CORBA specification.

Naming service. In addition to implementing the standard naming service CosNaming module, Component Broker's naming service allows the manipulation of compound names as character strings by extending the CosNaming::NamingContext interface with the subinterface IExtendedNaming::Naming Context, which introduces parallel "_with_string" suffixed versions of the operations in CosNaming::Naming context. For example, it is possible to say something like the following to resolve the compound name "/host/applications" to an object:

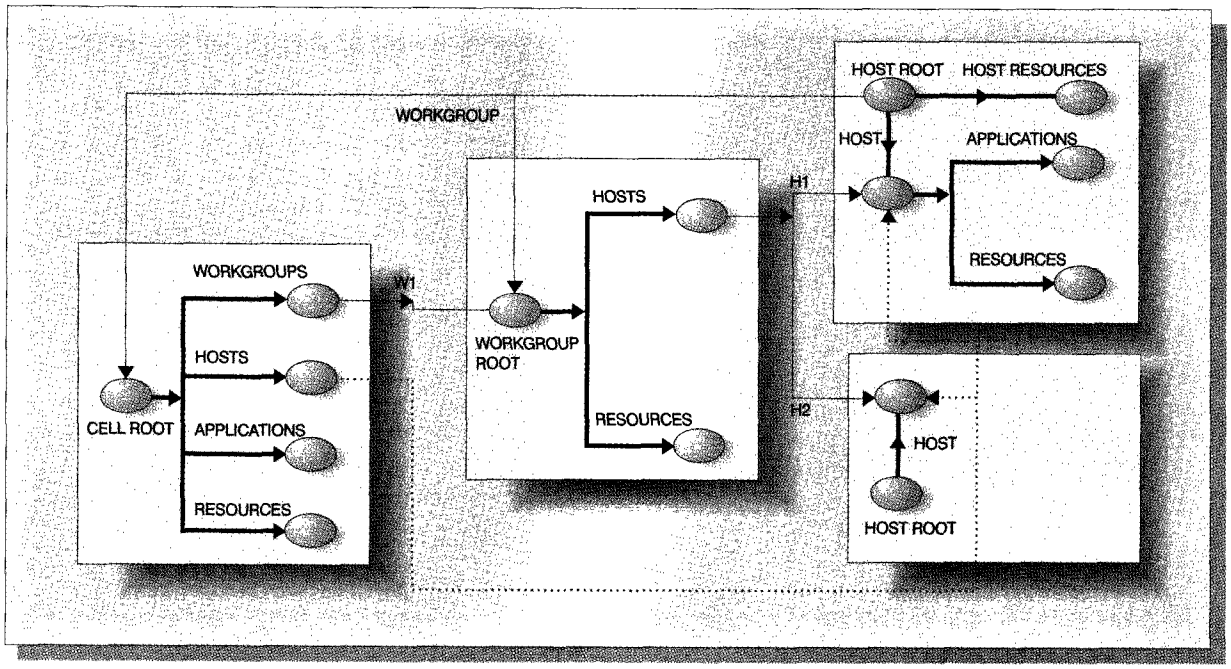
```
myRoot→resolve_with_string("/host/applications")
```

The CORBA naming service only defines an interface for handling name spaces; it does not define or mandate any structure the name spaces must adhere to. In contrast, Component Broker's naming service defines a structure, called the "System Name Tree," for its name space. An illustration of the System Name Tree is shown in Figure 6.

The System Name Tree is useful for ensuring that objects can be bound and located by "well-known" name paths. An instance of the host name tree exists on every (server) host. A workgroup is a logical collection of hosts whose aggregation creates some administrative or operational synergy for the business. A cell represents an administrative boundary for the name space. Notice that the local host root context is not bound to any other naming context. The ORB::resolve_initial_references("NamingService") operation returns the local host root for the host machine.

Security service. Component Broker's security service offers two major types of protection: authentication

Figure 6 Component Broker's System Name Tree



and message protection. In a Web-based environment, authentication must guarantee that clients can trust the servers they access, as well as guarantee that servers can trust their clients. Component Broker uses a DCE[10] security server as a mutually trusted third party that both clients and servers log on to before communicating and from which they obtain security tokens and have a security context established on their behalf.

A security context stores credentials and quality-of-protection information for clients and servers. A credential represents a user's secure identity and role in an interaction and allows the security server to restrict access to the system accordingly. Levels of quality of protection include no (or basic) protection, integrity, confidentiality, and both integrity and confidentiality. When more than one server (e.g., Server 1 and Server 2) is involved in carrying out a client's request, either both servers can use the same security server, or Server 1 can use its own security credentials to interact with Server 2 and its security server.

Life-cycle service. Component Broker's life-cycle service maintains a "factory" repository containing a collection of naming contexts in the System Name Tree. The life-cycle service also provides an interface for adding factories, at the time they are created, to the repository. To find a factory, the following steps must be followed: (1) decide on a location scope, for instance, which part of the repository will be searched for the factory; (2) determine which interface the desired object supports, as well as the key structure or string that identifies the interface; and (3) set the location scope and pass the interface identifier to the factory finder. After obtaining a factory, one of its methods can be called to create the desired object. In practice, these steps amount to a few lines of code, for example:

```
CORBA::Object_var obj;
obj = CBSeriesGlobal::nameService()→
                                    resolve_with_string
        ("host/resources/factory-finders/host-scope");
IExtendedLifeCycle::FactoryFinder_var finder =
      IExtendedLifeCycle::FactoryFinder::_narrow(obj);
obj = finder→find_factory_from_string
                              ("Person.object interface");
```

Event service. Component Broker offers a standard implementation of the CORBA event service, with consumers, suppliers, event channels, and "push" and "pull" communication models. In addition, Com-

ponent Broker offers two types of event services: transient and persistent. In the case of event service failure and subsequent recovery, consumers and suppliers of a transient event service are able to communicate through the same channel reference they used before the failure, but they must register again with the channel before using it, and events not communicated to consumers before the failure are lost. With the persistent event service, all objects, events, and event data are kept in a persistent store. In the case of failure all objects and data are recovered and, after recovery, consumers and suppliers do not need to register again with the channel.

Externalization service. Component Broker's externalization service extends the CORBA standard by providing the ability to control what is stored in the stream, according to the object's usage, using one of the following streaming policies during externalization and internalization: (1) reference, where a reference to a subobject is written into the stream, as opposed to the subobject itself; (2) raw, where the state of a subobject is externalized without its class information; and (3) value, where the state of a subobject is externalized with its class information. Mismatch of externalization and internalization policies are detected. Also, the externalization service provides an additional IBM stream format—a compact format that reduces the information in a stream whenever possible. This is in contrast to the OMG standard format, which is a pessimistic format that contains all information needed to move the state of an object from one machine to another, no matter how different the machine architectures and operating systems.

Identity service. Component Broker's identity service introduces a reference mechanism that associates two identities, absolute and constant random, with each object in a distributed system. Classes for objects that need these identities inherit the IdentifiableObject OMG standard interface. Absolute identities contain enough information to distinguish any two objects. The system includes the appropriate instance manager container information in the absolute identity of an object. The constant random identity is a compact representation of the absolute identity of an object, a four-byte key to a hash value. As such it is not unique for every object, but it can be used as a first-order comparison of object identities.

Query service. In Component Broker's query service, a collection is an aggregation of objects that can be one of the following: (1) a home, i.e., the "birthplace" and logical owner of objects of a given type; (2) a view, i.e., a subset of some other collection based on a predicate; (3) a reference collection, i.e., a collection that holds references to potentially heterogeneous objects; and (4) a single logical image, i.e., a collection of other collections that gives the client an image of one single collection. All of these collections can be queried using IBM's OO-SQL (object-oriented structured query language) and using the standard query evaluators. In addition, Component Broker's query service supports query pushdown, an optimization that delegates a query or parts of it to a database when the query evaluator receives a query on an object that has its state stored on the database.

Transaction service. Component Broker's transaction service provides a standard implementation of the CORBA Object Transaction Service (OTS), with support for atomic, consistent, isolated, and durable (ACID) transactions, two-phase commit, and with planned support for nested transactions. Component Broker's transaction service implements the OTS standard interfaces Current and Coordinator and requires that classes of resources taking part in a transaction inherit from the Resource interface and implement its methods prepare, commit, rollback, commit_one_phase, and forget. Note that although Component Broker does not extend the CORBA OTS, it does offer a way to synchronize the management of a managed object's state with transactional boundaries. This is accomplished by the mixin object, which invokes synchronization methods on a managed object before actually executing a transaction commit.

Concurrency service. Component Broker's concurrency service implements the standard CORBA concurrency service, coordinating the granting of the five standard types of locks: intention read lock, read lock, upgrade lock, intention write lock, and write lock.

Component Broker Toolkit. Component Broker defines a suite of tools to support the development of server-based business object applications, including tools for building business and data objects, defining object behavior, generating code, and debugging and testing capabilities. The Component Broker Toolkit** provides bridge technology that allows the use of any tool that uses the MDL (model definition language) format of Rational ROSE**. Tools for the design of relational databases can also be bridged

by using the standard data definition language (DDL) format.

Component Broker Toolkit's central tool is the Object Builder, which integrates the activities performed during business object development. The Object Builder uses "smart guides" to assist in the development of business objects. It also generates a substantial amount of the code that implements a business object and facilitates the input of the business logic code that cannot be generated and the mapping between a data object and its underlying data store. The Object Builder uses a common data model to format the meta-data it creates and store it in version-controlled files. The Object Builder also packages and configures managed objects, containers, and homes into application families that can then be used by the Systems Management Tool to install the application. In addition to installing applications, the Systems Management Tool sets up Component Broker server networks with hosts and groups and performs other important systems management activities.

Systems management. Component Broker's systems management (SM) provides the ability to configure, deploy, monitor, and control a Component Broker network. From the point of view of systems management, there are three kinds of applications that are deployed on a network: systems management, agent, and client. The SM Application (SMAPPL) is the central point for definitional configuration data and contains a copy of the common data model (CDM). An agent application allows the SMAPPL to communicate with a Component Broker server in another host; an agent also contains a copy of the CDM and the portion of the configuration data that is associated with its own host. Client applications can run and are managed locally in their own hosts. These applications are installed using the Application Installation Tool.

The main user tool for systems management is the SM User Interface, which can be attached to a host that runs either the SMAPPL or an agent application. With this tool a Component Broker network can be set up, by creating and configuring management zones, configurations, applications, cells, workgroups, policy groups, and attributes of any of these elements. All these pieces contribute to making sure that error reporting, trace information, and performance data for "wellness" reporting can be collected and sent to responsible personnel, to be acted upon in a timely fashion.

Client Programming Model. The Client Programming Model defines how programmers use (develop objects that are clients of) business objects. Application developers building either Tier-1 (client) or Tier-2 (server) applications use the Client Programming Model whenever they implement a new object that makes use of a business object. Although in this paper we focus on client applications written in Java, Component Broker supports writing client applications in C++, Java, or Visual Basic.** Figure 7 presents a high-level overview of the Client and Programming Model, with VisualAge* for C++, Visual Basic, Visual C++**, and Java clients. The way clients deal with business objects in the programming model is consistent regardless of the underlying encapsulated "plumbing" that is required to support the various mechanisms that were used to build the business objects.

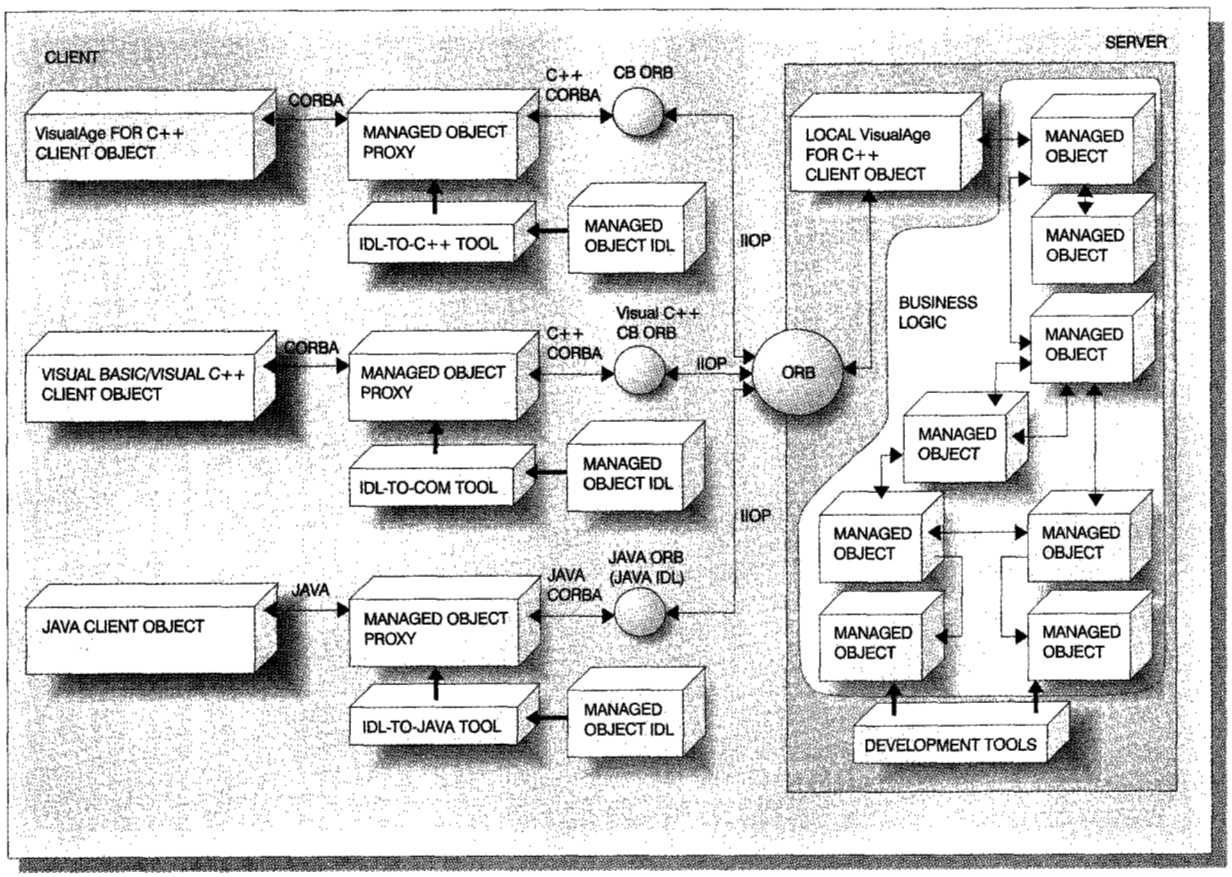
The client application accesses business objects, residing on the server, that implement the business logic of the application. As we have seen, a managed object is a business object that has been installed on a Component Broker server. In general, the client application is unaware of whether the managed object is remote or local, or implemented in the same or another language. This transparency is achieved by functions and architecture provided by Component Broker and CORBA.

The managed object implementor provides the client application with: (1) a set of interface files that define the interface to a managed object and any "helper" classes the client might use, and (2) DLL (dynamic link library) and Java class files that implement the classes in the interfaces and the helper classes.

A client application performs at least some of the following tasks:

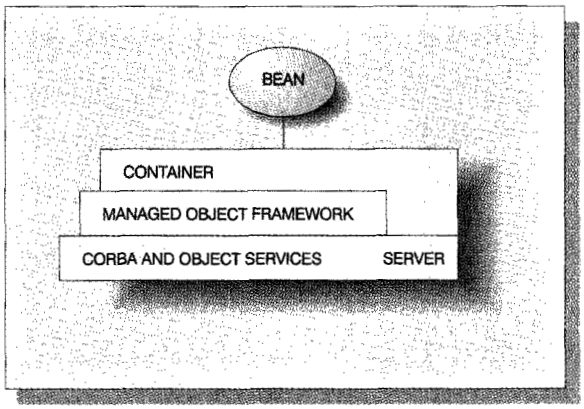
1. *Find objects.* To find a managed object, a client can use the naming service, navigating the System Name Tree with the well-known path and name for the desired managed object. In general, only a very small subset of the managed objects in a distributed system will be directly bound to a name in the naming service. Alternatively, a managed object can be found by first finding a known object, e.g., a factory or a collection, and then invoking a find method on this known object and passing it the managed object's primary key.

Figure 7 Client Programming Model overview



2. *Use objects.* This involves invoking methods on a reference to a managed object, once this reference has been found or created.
3. *Create objects.* A managed object can be created by invoking a "create" method on a factory and providing it with a key for the new managed object. This operation will return an uninitialized managed object and appropriate methods must then be invoked to initialize its attributes. This can result in significant overhead if every initialization call has to go "across the wire." Alternatively, a so-called copy-helper object can be created on the client side that contains all the data necessary to properly initialize a managed object upon creation. The client can then invoke a create-from-copy method on the factory and provide it with the key and copy-helper object.
4. *Use sets of objects.* A home in Component Broker represents a set of managed objects, all of the same type. A client can manipulate collections of managed objects by creating an iterator on the objects' home and invoking the iterator's navigation methods to obtain each subsequent object in the home.
5. *Remember interesting and important objects.* Component Broker supports CORBA's standard interoperable object references (IOR), which allow a client to refer to a managed object regardless of where it is on the network. A client can convert an IOR into a string and store this converted object reference for future use.
6. *Release or delete objects.* When a client releases a managed object, it loses its reference to it, but the managed object still exists on the server. Deleting a managed object not only removes the client's reference; it also removes the managed object from the back-end data store.

Figure 8 Enterprise bean on Component Broker implementation layers



We have provided a high-level overview of the main concepts in Component Broker. As we have seen, these concepts provide a rich framework for the development and operational reuse of distributed transaction-oriented enterprise applications.

Enterprise JavaBeans is an architecture for component-based distributed computing. Enterprise beans are components of distributed transaction-oriented enterprise applications. Thus, at least from this point of view, Component Broker is an ideal platform on which to implement the EJB architecture specification. In addition, the main concepts in the EJB specification, namely, Enterprise beans, containers, factories, finders, handles, and so on, correspond in a natural way to concepts in Component Broker such as business objects, homes, instance managers, and so on. Thus, as we shall see in the following section, providing support for EJB on Component Broker is guided by the natural mapping between the corresponding sets of concepts.

Architectural overview: EJB on CB

In providing support for EJB in Component Broker, there are several guiding principles. To ensure Java portability, the following apply:

- Any EJB component that adheres to the EJB specification should be usable in CB without modification.
- Any EJB component written for CB should be usable in other systems and tools supporting EJB. Thus any CB-related enhancements are encapsulated and ignored by other EJB containers or servers.
- EJB components can be used in a way that is natural to a Java programmer, i.e., without writing interface definition language (IDL) statements.
- The use of EJB constructs, classes, facilities, and services is through EJB and related Java APIs and no CB-specific APIs need be used from within a bean.
- The underlying EJB class and associated interfaces are completely provided.

Other principles provide separation of concerns between EJB and CB:

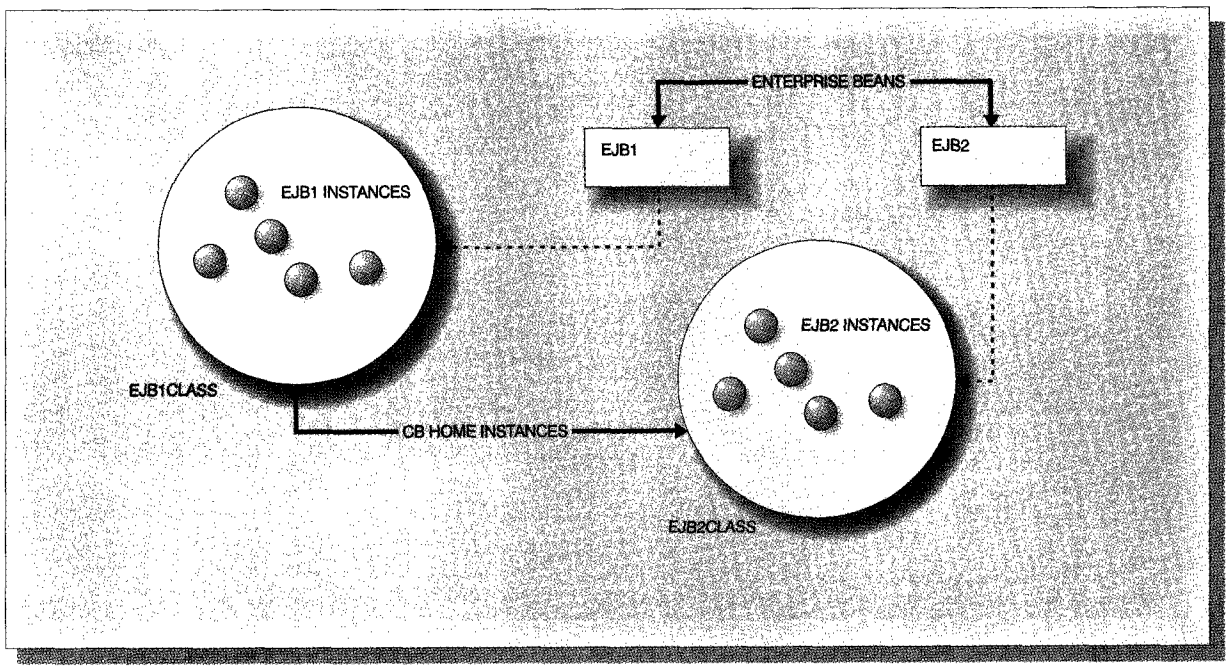
- Component Broker should play the roles of both container and server, as defined in the EJB specification.
- The specific mapping of EJB APIs and capabilities onto the CB system should be hidden from the Java programmer.
- Capabilities of CB that go beyond the EJB specification should be made available to the bean developer as installation time options (e.g., in Object Builder) and should not affect the specific bean class design itself.
- Any additions to an EJB package (Java archive [JAR] file) that are specific to CB should not interfere with the package's use in other systems.

Where there is ambiguity, incompleteness, or omission in the EJB specification, we will consider providing advanced capability for EJB components that are installed on CB systems. In all cases, however, we should properly separate such capability from the definition of the component itself and not "pollute" the actual bean class definition code with CB-specific extensions.

Of course it will always be possible for an EJB developer to create an EJB component on Component Broker that is not portable by including direct calls to MOFW classes. The component will then be usable only on a CB server. If a component uses other CB business objects (as CORBA objects) then it will be portable to other platforms, as long as either (1) the required business objects have also been ported, or (2) there is a CB server somewhere in the network that maintains the required objects.

Enterprise beans in Component Broker terms. In Component Broker, Enterprise beans are application object (a kind of business object) classes, and bean instances are created as managed objects on a CB server. CB fills the roles of both container and server, providing implementation of those interfaces

Figure 9 Mapping of EJB run-time objects to CB



via the MOFW and object services, tailored according to environment properties defined by the EJB provider and packaged with EJB (see Figure 8).

Since the implementation will only make use of Component Broker interfaces and not system-specific service APIs, the container can be easily ported to all of the platforms on which Component Broker is available.

Client view. An Enterprise bean on CB looks the same as any other Enterprise bean to a Java client. Component Broker Toolkit generates client bindings that present the bean as a local object on which the EJB interface methods can be called. Remote method calls are accomplished via the CB ORB on client and server systems, using IIOP.

Server view. On a CB server system, beans are contained in specialized homes that implement the *EJBHome* interface. Each EJB class is supported by an *EJBHome* instance that contains instances of the class. An *EJBHome* object is responsible for finding, creating, and destroying EJB object instances (see Figure 9).

A specialized CB home object is used to implement *EJBHome* objects. If we determine that there is common behavior it can be extracted into an EJBHome class.

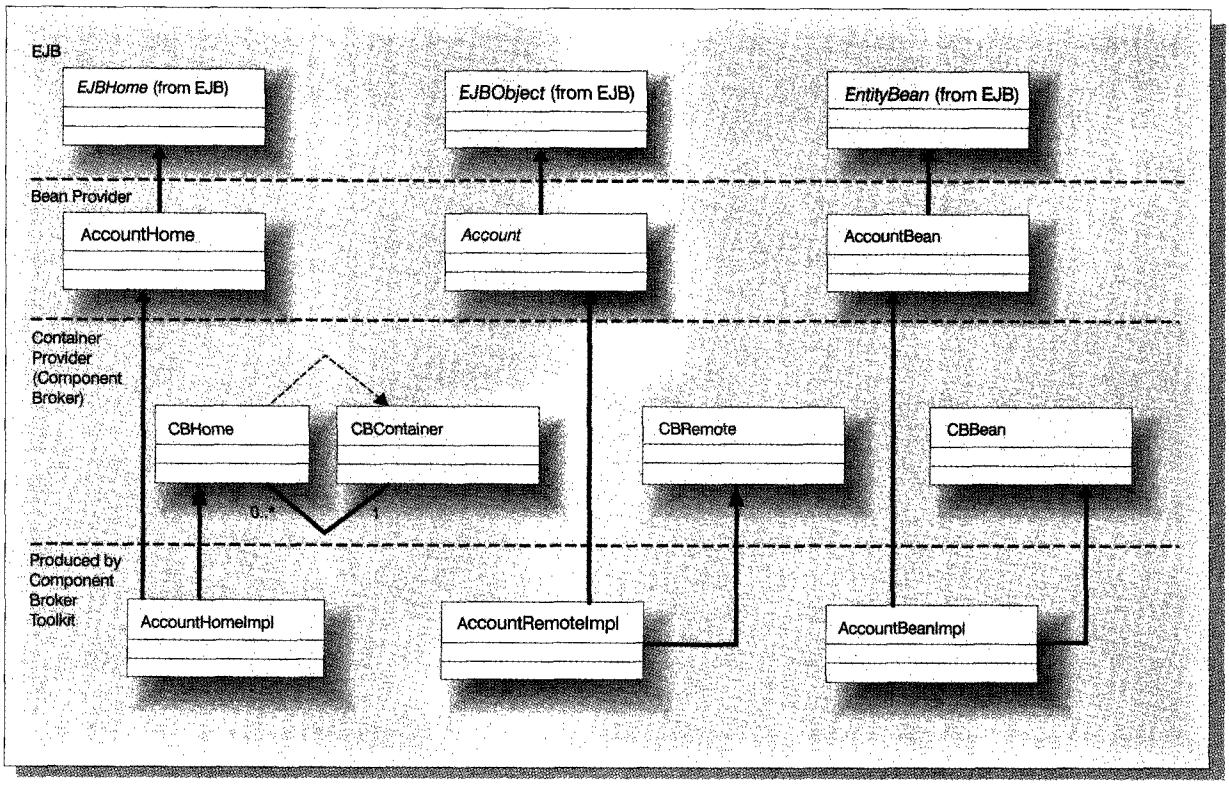
EJB classes and the CB MOFW. The diagram in Figure 10 illustrates the interfaces and classes involved in creating an example "Account" bean as a managed object on Component Broker. The classes in the diagram are grouped according to the roles defined in the EJB 1.0 specification.[4]

Support for clients of Enterprise beans

Clients of Enterprise beans perform the actions shown in Table 1. From this table, we see that the following objects need to be exposed to the client:

- EJB home. The exposed EJB home implements, in addition to the EJBHome interface, a number of create(. . .) methods that correspond to the ejbCreate(. . .) methods defined in the Enterprise bean class, as well as a number of find<method>(. . .) methods, with corresponding ejbFind<method> defined in the Enterprise bean class, defined by

Figure 10 Extension relationships for interfaces and classes of an Account component



the EJB provider at EJB development time. These methods can return single EJBObject instances, as well as collections of them, allowing the client to use sets of objects, as prescribed in the Component Broker Client Programming Model.

- Bean remote interface. This interface defines, in addition to the methods defined by the EJBObject interface that it extends, the business methods implemented by the bean. Thus, the client only needs an object that implements this interface to make effective use of the bean.
- Handle. This is a local object for which remote stubs need not be generated.
- Context. To be able to find containers, the client must be able to make instances of InitialContext, which will be exposed by JNDI (Java Naming and Directory Interface).
- User transaction. This object exposes a minimal number of transaction control methods to the client, as opposed to the full power of the Java Transaction Service.

Consequently, code for Java clients will include the following kinds of items:

- Remote stubs. These include stubs for EJBHome objects and bean remote interfaces. These stubs are generated at deployment time.
- Local-only objects. In order to use handles, clients need to be able to have access to the handle class. This class is provided by the EJB provider at EJB development time.
- A Java ORB. This should be able to "marshal" actual arguments of a method invocation into an IIOP request, send an IIOP request to the server, receive an IIOP reply from the server, and "unmarshal" return values from a method invocation. In addition, the Java ORB must provide the necessary JNDI functionality for a client to look up bean stores and the current transaction object. Furthermore, the Java ORB must include the implementation of the CurrentTransaction interface. This ORB can either be

Table 1 Actions performed by a client of an Enterprise bean. "<BRIN>" represents the bean remote interface name.

Action	Supported By	Comments
lookup	Context	Obtain EJB class by JNDI name
create(. . .)	<BRIN>EJBHome	Supply initialization fields as arguments to creation
findByPrimaryKey	<BRIN>EJBHome	Supply key values for location
find<method>(. . .)	<BRIN>EJBHome	Perform search implemented by bean provider
<method>	<BRIN>	BRIN extends EJBObject
remove	<BRIN>	BRIN extends EJBObject
	EJBHome	Remove a bean either by handle or by primary key
begin	UserTransaction	
commit		
rollback		
isIdentical	<BRIN>	BRIN extends EJBObject, compare with another EJBObject
getHandle	<BRIN>	BRIN extends EJBObject
getEJBObject	Handle	Handle is always a local object
getPrimaryKey	<BRIN>	BRIN extends EJBObject
getEJBMetaData	EJBHome	Allow client to get various meta-data associated with EJBHome
getEJBHome	<BRIN>	Various ways of getting at the EJBHome
	EJBMetaData	
getClassName	<BRIN>	Obtain the class name of the EJB that provides the implementation of an EJB object, invoked from the EJB object or the meta-data of its EJBHome
	EJBMetaData	
getHomeInterfaceClass	EJBMetaData	
getPrimaryKeyClass		Class object for primary key is meta-data useful to the client
getRemoteInterfaceClass		
isSession		

downloaded with the bean stubs or already installed in the client's workstation.

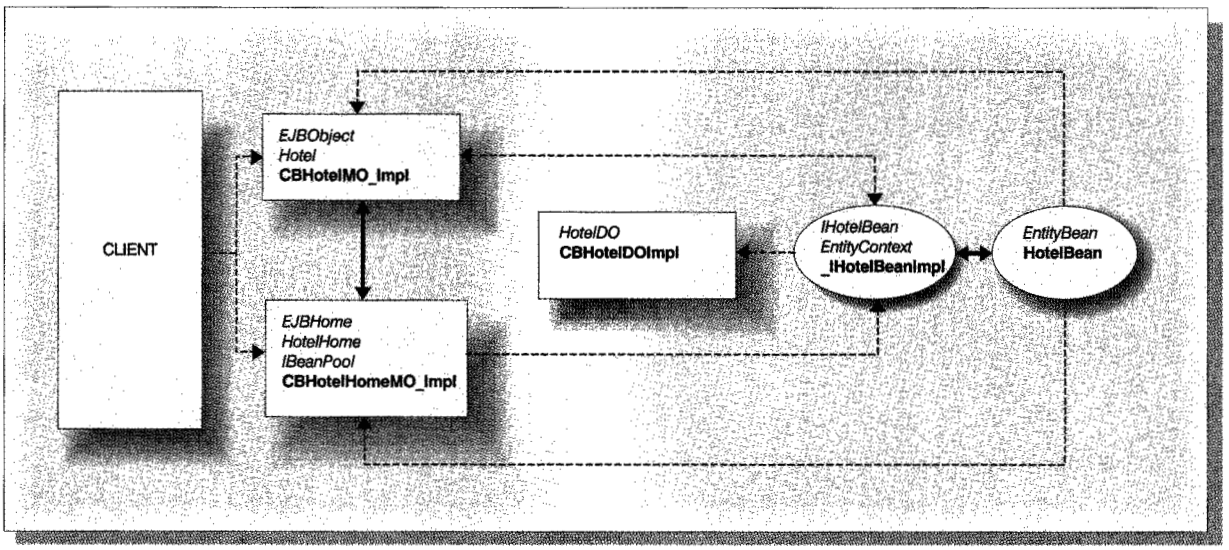
For non-Java clients, handle classes and stubs for the EJBHome and bean remote interface should be available. These could be generated in each language that a client could use to interact with an Enterprise bean, or they could be generated in IDL form and the language-specific stubs could be generated at the client site on demand. In addition, an ORB for the corresponding language should also be available on the client's workstation.

Support for Enterprise beans

Enterprise beans can be of at least two kinds: session beans and entity beans. In turn, entity beans can have their persistence managed by the container or by the bean itself. In Component Broker, session beans correspond to application objects and entity beans correspond to business objects. One of Component Broker's strengths is that it can manage the persistence of business object data items automatically and that business objects can use their data items independently of the actual back-end data store. Thus entity beans with container-managed persistence are most naturally mapped to business objects, as Component Broker defines them. However, it is possible to define business objects to implement entity beans with bean-managed persistence.

The main elements in the implementation of an Enterprise bean include a remote interface that extends *EJBObject*, a home interface that extends *EJBHome*, and an Enterprise bean class that implements either *SessionBean* or *EntityBean*. Figure 11 illustrates the mapping of these elements for a hotel entity bean—a *Hotel* remote interface, a *HotelHome* interface, and a HotelBean class—at a high level. Here boxes represent C++ objects, ovals represent Java objects, a boldface label stands for the name of the class instantiated by an object and an italicized label stands for the name of an interface implemented by a class. Solid lines represent interactions within the same language and dotted lines represent cross-language interactions; the rounded rectangle representing a client indicates that this could be either a C++ or a Java object.

Figure 11 Overview of the mapping of an entity bean to Component Broker



Separation of an EJB object from its Enterprise bean. The EJB specification allows for, but discourages, the implementation of a bean's remote interface by the Enterprise bean class. This is in contrast to Component Broker, where the business object interface is implemented by the business object class and ultimately instantiated by a managed object. On the other hand, the current implementation of the Component Broker infrastructure is written in C++; thus a managed object that implements the bean's remote interface will need to use a C++-to-Java interaction mechanism to invoke the remote interface methods implemented in the Enterprise bean class.

Using a delegation scheme from the managed object that implements an Enterprise bean's remote interface to the bean's class both achieves the separation between remote interface and bean class and allows a clean C++-to-Java interaction. This delegation scheme involves the definition of a Java "bean tie" class that implements an IDL interface exposing the Enterprise bean class to the C++ managed object and home object. In Figure 11, the hotel bean tie is the _IHotelBeanImpl object that implements the *IHotelBean* interface. This way, all access to a bean passes through the bean tie, which then delegates to the bean instance. In addition, as seen in Figure 11, the bean tie implements the *EJBContext* interface (via either the *EntityContext* or *SessionContext* interfaces) that allows the bean to interact with its container.

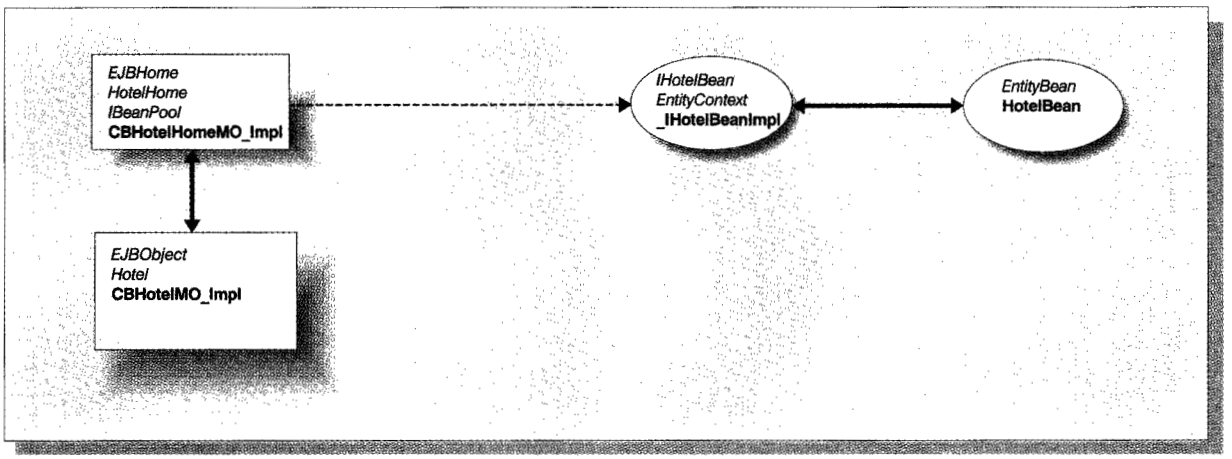
EJB objects and managed objects. An Enterprise bean's remote interface is written as an RMI remote interface. However, the managed object that implements it is CORBA IDL-based, so an RMI-to-IDL translation is required, which implies support for RMI/IIOP. In a later section we describe the requirements on RMI/IIOP posed by an effective support of Enterprise beans on Component Broker. In Figure 11, CBHotelMO_Impl is the managed object that implements *Hotel* and *EJBObject* as IDL interfaces that have been derived from their RMI counterparts provided by the bean developer.

Cross-language interactions can be implemented either by local proxies or by remote proxies. For example, CBHotelMO_Impl uses a local proxy to send messages to the bean tie. On the other hand, HotelBean could send messages to a CBHotelMO_Impl that implements *Hotel* using a local proxy, if the managed object is located in the same process, or using a remote proxy if it is not. Either way, HotelBean sees only an RMI *Hotel* interface.

The methods defined by the *EJBObject* interface and their implementation by a managed object are as follows:

- The *remove* method delegates to the LifeCycleObject::remove method implemented in the managed object. This method must reactivate the corresponding bean instance if it was passivated. No-

Figure 12 Implementation of a bean pool



tice that since the managed object implements two interfaces that define a *remove* method (namely, *EJBObject* and *LifeCycleObject*), one of these two method definitions will need to be renamed to avoid the name collision. One option is to rename the method—to something like *ejbRemove*—in the IDL interface that gets generated from the *EJBObject* interface.

- The *getEJBHome* method delegates to the IManageable::getHome method implemented in the managed object.
- The *getHandle* method should return the CB handle of the managed object associated with the Enterprise bean. See also the section on handles that follows.
- The *getPrimaryKey* method is implemented by the managed object, for entity beans only, by getting the primary key fields from the data object and building and returning a primary key object.
- The *isIdentical* method delegates to the IdentifiableObject::is_identical method implemented in the managed object.

In addition, application-specific extensions of *EJBObject*, such as *Hotel*, define application-specific methods. Each one of these methods is implemented by the managed object, such as CBHotelMO_Impl, by delegating to the bean via the bean tie, as we have seen.

EJB homes and beans' life cycle. An Enterprise bean home can be supported in Component Broker by a specialized home that implements its create and find methods. In Figure 11, the hotel home is given by the CBHotelHomeMO_Impl object that implements the *HotelHome* and *EJBHome* interfaces. In addition, this specialized home implements an *IBeanPool* interface that manages the life cycle of Enterprise beans via their bean ties. Upon creation, a bean tie instantiates a bean and passes itself to the instance as an *EJBContext* object. The bean instance remains attached to the bean tie until the home releases the bean tie. This occurs when the bean pool invokes *release* on the bean tie, which in turn tells the bean instance to release its *EJBContext* object. After releasing a bean tie, the bean pool makes null its reference to it, at which point the bean tie and the bean instance are candidates for garbage collection. Figure 12 illustrates a high-level structure of the objects involved in implementing a bean pool.

Bean ties are owned by the bean pool implemented by the bean's home. They are lent to managed objects upon activation and creation of a bean, and they are returned to the bean pool when the managed object is destroyed. In the special case of home finder methods, the home temporarily borrows a bean tie from the bean pool to perform the find. Each home implements the *IBeanPool* interface. This interface provides methods for managing the acquisition and release of bean ties. The *IBeanPool* interface is used by the home and the managed object. Notice that the EJB specification discusses bean pools only in relation to stateless session beans and entity beans.

The methods defined by the *EJBHome* interface and their implementation by a specialized home are as follows:

- The *remove* method delegates to the remove method implemented by the mixin associated with the home, which must reactivate the corresponding bean instance if it was passivated. Notice that *EJBHome.remove* can take either a handle or a primary key as an argument, the latter when the home hosts entity beans. In order to support passing this argument from the client to the home object, the class that implements the handle or the primary key must be serializable[11] and it must be a valid RMI/IDL type, which implies that an implementation of RMI/IIOP must be available.
- The *getEJBMetaData* method returns the home object's meta-data. See the following section for a description of the mapping of the *EJBMetaData* interface and a later section for a description of how to instantiate these meta-data at run time.

In addition, application-specific extensions of *EJBHome*, such as *HotelHome*, define application-specific methods, each of which must be either a create method or a finder method. Finder methods can be defined only on homes hosting entity beans. See the sections on entity beans and session beans for issues related to implementing these methods.

EJB meta-data. The *EJBMetaData* interface allows a client to obtain various meta-data associated with a home object. Since *EJBMetaData* is not a remote interface, the class that implements it must be serializable and it must be a valid RMI/IDL type. This implies that in order to fully implement the *EJBMetaData* interface, an implementation of RMI/IIOP must be available. The following methods are defined in the *EJBMetaData* interface:

- The *getEJBHome* method returns a reference to the CB home class that implements the application-specific *EJBHome* interface, e.g., CBHotelHomeMO_Impl, which implements *HotelHome*. The returned reference is cast to the application-specific *EJBHome* interface, e.g., *HotelHome*. Notice that since this method is invoked on a local *EJBMetaData* instance there is an opportunity for optimizing access to the returned reference to the home. Otherwise the issue arises as to whether this reference can be effectively serialized by RMI/IIOP.
- The *getHomeInterfaceClass* method returns the class object of the application-specific *EJBHome* interface, e.g., *HotelHome*.
- The *getPrimaryKeyClass* method obtains the class object for the primary key class. This will return the primary key class associated with the CB home that implements this *EJBHome* object.
- The *getRemoteInterfaceClass* method returns the class object of the application-specific EJB remote interface, e.g., *Hotel*.
- The *isSession* method returns true if the current bean is a session bean and false otherwise. At creation time of an *EJBMetaData* object, the container can set the value of this method on the meta-data object. The container knows this value, since it either is or is not a home for session beans.

Handles. Handles can be implemented using Component Broker handles. Since *Handle* is not a remote interface, the class that implements it must be serializable and a valid RMI/IDL type. In order to fully implement the *Handle* interface, an implementation of RMI/IIOP must be available.

The *getEJBObject* method is defined by the *Handle* interface. This method returns a valid reference to the EJB object represented by this handle. In CB, this will return a reference to the associated managed object for the EJB instance. This should be a local operation. As an implementation consideration, the handle can incorporate additional information along with a managed object's IOR (interoperable object reference), so that if the ObjectNotFound exception is returned by the ORB, the handle can use an alternative mechanism, such as the directory service, to locate the managed object.

EJB context. A bean instance uses an EJB context object to obtain contextual information, such as the bean's environment properties or its primary key. An *EJBContext* object is provided to a bean instance by its container at bean instance creation time. As we have seen, the bean tie implements the context, creates the bean instance, and passes itself to the bean instance as its context. The bean tie uses a reference to the managed object to which it delegates the implementation of some of the *EJBContext* methods. These methods are as follows:

- The *getCallerIdentity* method obtains the security ID (identity) of the immediate caller. This method can be delegated to the CB *Current* object, which can be obtained from the ORB. Given that this method returns a *java.security.Identity* object, the *Credentials* type returned by the *Current* object will need to be mapped to the *java.security.Identity* type.
- The *getEJBHome* method delegates to the

IManageable::getHome method implemented in the managed object.

- The *getUserTransaction* can return the CB *Current* object, which is then mapped to a *UserTransaction* object.
- The *getEnvironment* method returns a *Property* object containing the bean's environment properties. This method can use the SM API to read the environment properties from internalized DDL (as discussed in a later section), create a property object, populate the return values, and send the property object back to the caller.
- The *isCallerInRole* method accepts a security ID and checks to see if the caller of the bean has the input security ID. This method can invoke *this.getCallerIdentity* and compare the result with the input *Identity*.
- The *setRollbackOnly* method can be delegated to the *Current* object's *rollback_only* method.
- The *getRollbackOnly* method can be delegated to the CB *Coordinator* object.

The *EJBContext* interface is extended by the *SessionContext* and *EntityContext* interfaces to provide contextual information specific to session beans and entity beans, respectively. These two interfaces and the methods they define are described in subsequent sections.

Support for entity beans

An entity bean represents data in a database and it provides shared access to multiple users. Entity beans are transactional and long-lived, and they survive crashes of the EJB server. Entity beans correspond naturally to CB's business objects.

An entity bean can implement its persistence directly, using bean-managed persistence, or by relying on its container, using container-managed persistence. An entity bean must implement the *EntityBean* interface. Notice that this means that an entity bean must provide implementation for the state control methods *ejbLoad* and *ejbStore*, regardless of whether the entity bean uses bean-managed or container-managed persistence. Thus, an entity bean with container-managed persistence can be mapped to a business object that has a cached data object. For an entity bean with bean-managed persistence, one option is to map to a business object with a cached data object in order to take advantage of the *syncFromDataObject* and *syncToDataObject* callbacks, ignoring the data object itself. Another option is to use a business object without a data object and to subclass the business object's transactional mixin to call the state control methods. Figure 13 illustrates the mapping of a hotel entity bean, showing a structure of the classes to be instantiated. Notice that *IHotelBean*, implemented by _IHotelBeanImpl, extends *IBean* and *IBeanWithCDO*. The bean tie is responsible for providing the callbacks defined in the *EntityBean* interface. These callbacks are defined and exposed to the C++ side managed object by the *IBean* and *IBeanWithCDO* interfaces and they are implemented by the bean tie via delegation to the bean. These two interfaces separate the state control methods from the more generic callbacks that can also be used by session bean ties. In addition to bean delegation, a bean tie provides MOFW-like services for the bean. For example, during the activation and passivation of container-managed entity beans, the bean tie is responsible for synchronizing the bean's state with the container's data object, in addition to performing the state control callbacks. Notice that in this situation it is the bean tie that actually interacts with the data object, instead of the managed object. This is why _IHotelBeanImpl has a reference to a *HotelDO* object, implemented by CBHotelDOImpl.

The methods defined by the *EntityBean* interface are mapped to Component Broker through the bean tie. The *setEntityContext* method can be called from the bean tie's constructor when it instantiates the bean and is given a reference to the bean tie object itself, which implements the EntityContext interface. Other EntityBean interface methods can be called from methods defined in and exposed by the bean tie for this purpose, as shown in Table 2.

Create and finder methods. In addition to the methods defined in the *EntityBean* interface, an entity bean must implement a number of application-specific create and finder methods. Create methods for bean-managed entity beans as well as finder methods for both kinds of entity beans are required to return the primary key (or keys) of the bean (or beans) just created or retrieved. Keys are discussed in a subsequent section. If an entity bean's home finder method returns a collection of keys, then the collection must be of a valid RMI/IIOP type. For the time being all that is needed is that *java.util.Enumeration* be a supported RMI/IIOP type.

Finder methods for entity beans are to be generated by the container tools, presumably from some specification given by the bean developer. Unless the implementation of this specification can be readily generated automatically, it is assumed that it will require

Figure 13 Mapping of a Hotel entity bean to Component Broker

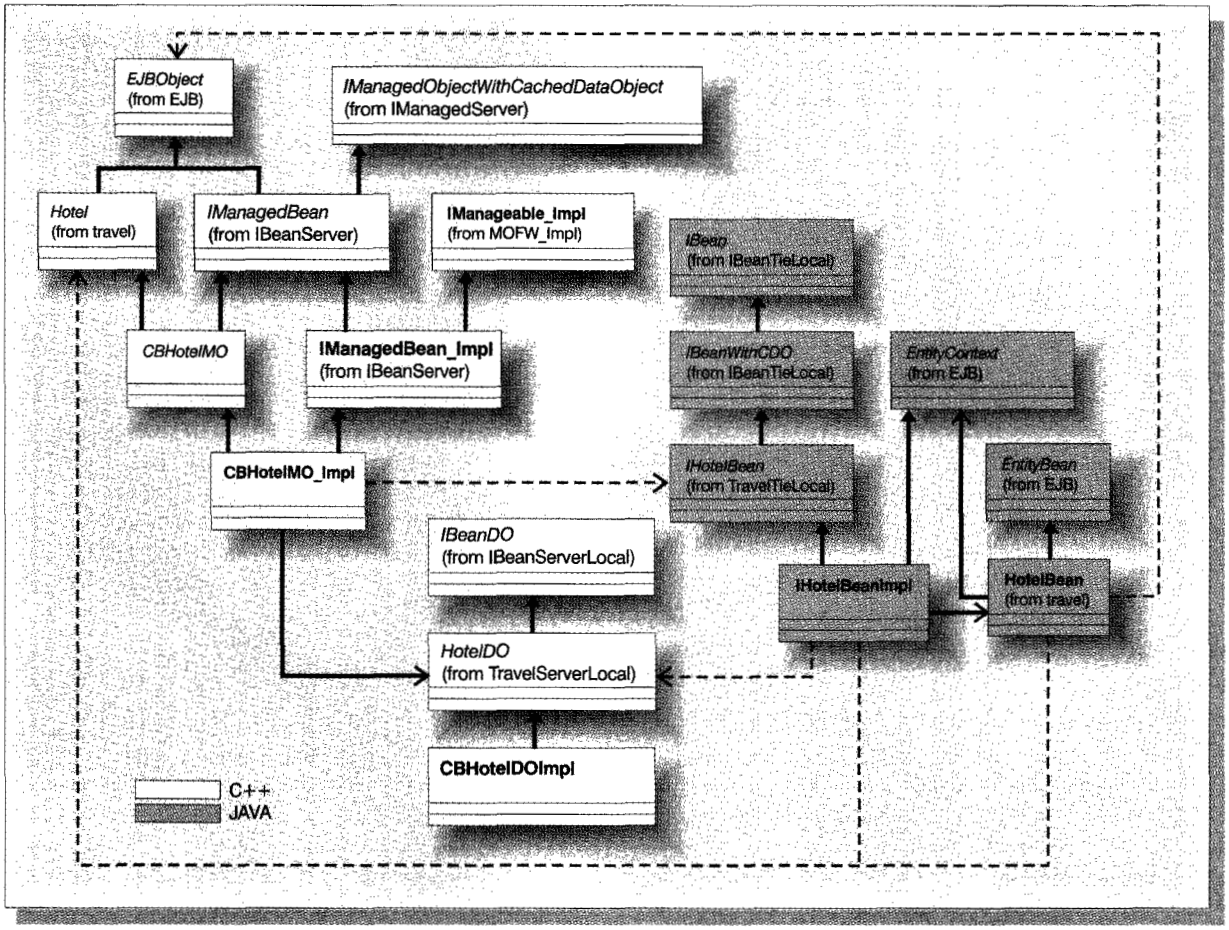


Table 2 EntityBean methods and corresponding bean-tie methods

EntityBean Method	Bean-Tie Method
setEntityContext	(Called from bean tie's constructor)
unsetEntityContext	*release*
ejbRemove	*uninitForDestruction*
ejbActivate	*initForReactivation*
ejbPassivate	*uninitForPassivation*
ejbLoad	*ejbLoad*
ejbStore	*syncToDataObject*

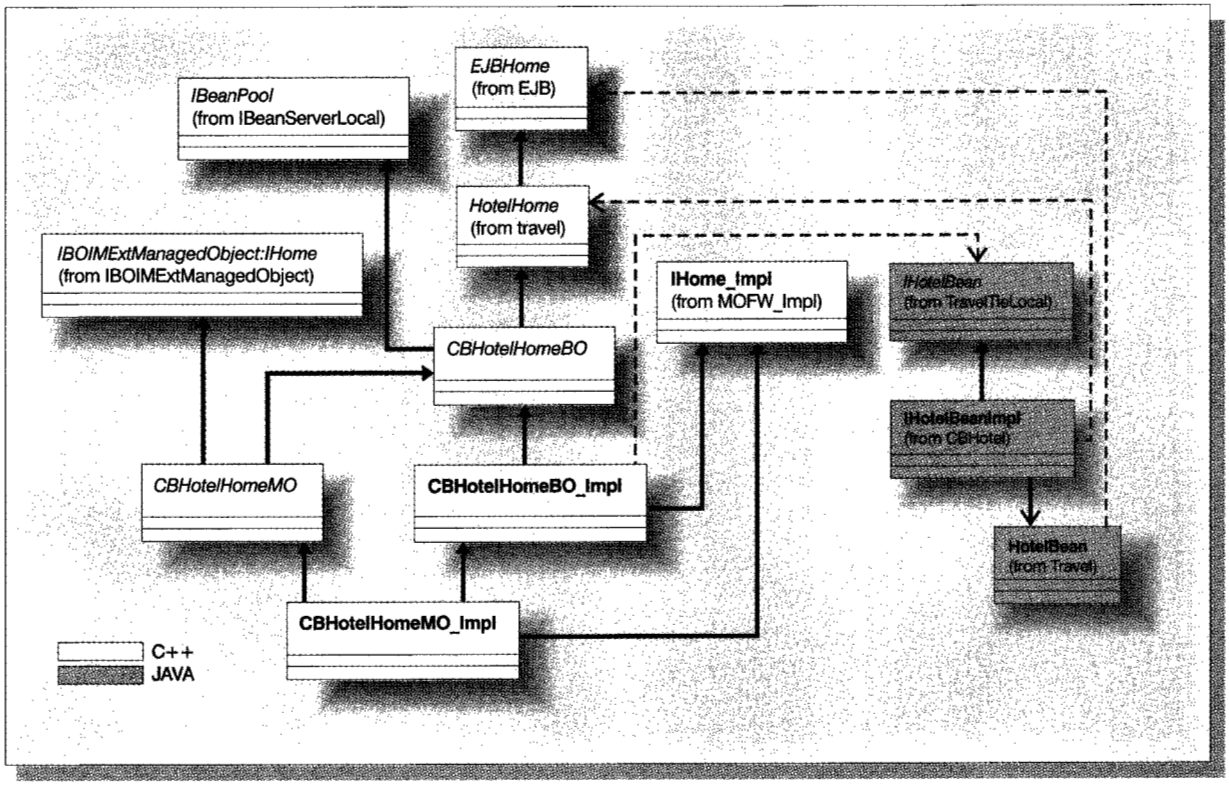
human intervention, presumably by the bean deployer.

Entity bean homes. The diagram in Figure 14 shows the mapping of a Hotel entity bean's home to Component Broker. The CBHotelHomeBO_Impl class implements the functionality in the *EJBHome* interface in terms of IHome_Impl. In Figure 14 we see references from HotelBean to a proxy of the CBHotelHomeMO_Impl class. This reference can be obtained, for example, by a call to *getEJBHome* on the *EJBObject* implemented by CBHotelMO_Impl.

An entity bean's home class, CBHotelHomeMO_Impl in Figure 14, also implements the *IBeanPool* interface. This interface is used by the home itself:

- When servicing finder methods, the home acquires a "finder" bean tie, uses it to invoke the appropriate ejbFind method, and returns it to the bean pool.
- When servicing create methods, the home acquires a bean tie and lends it to the managed object.

Figure 14 Mapping of a Hotel entity bean's home to Component Broker



The *IBeanPool* interface is also used by the managed object:

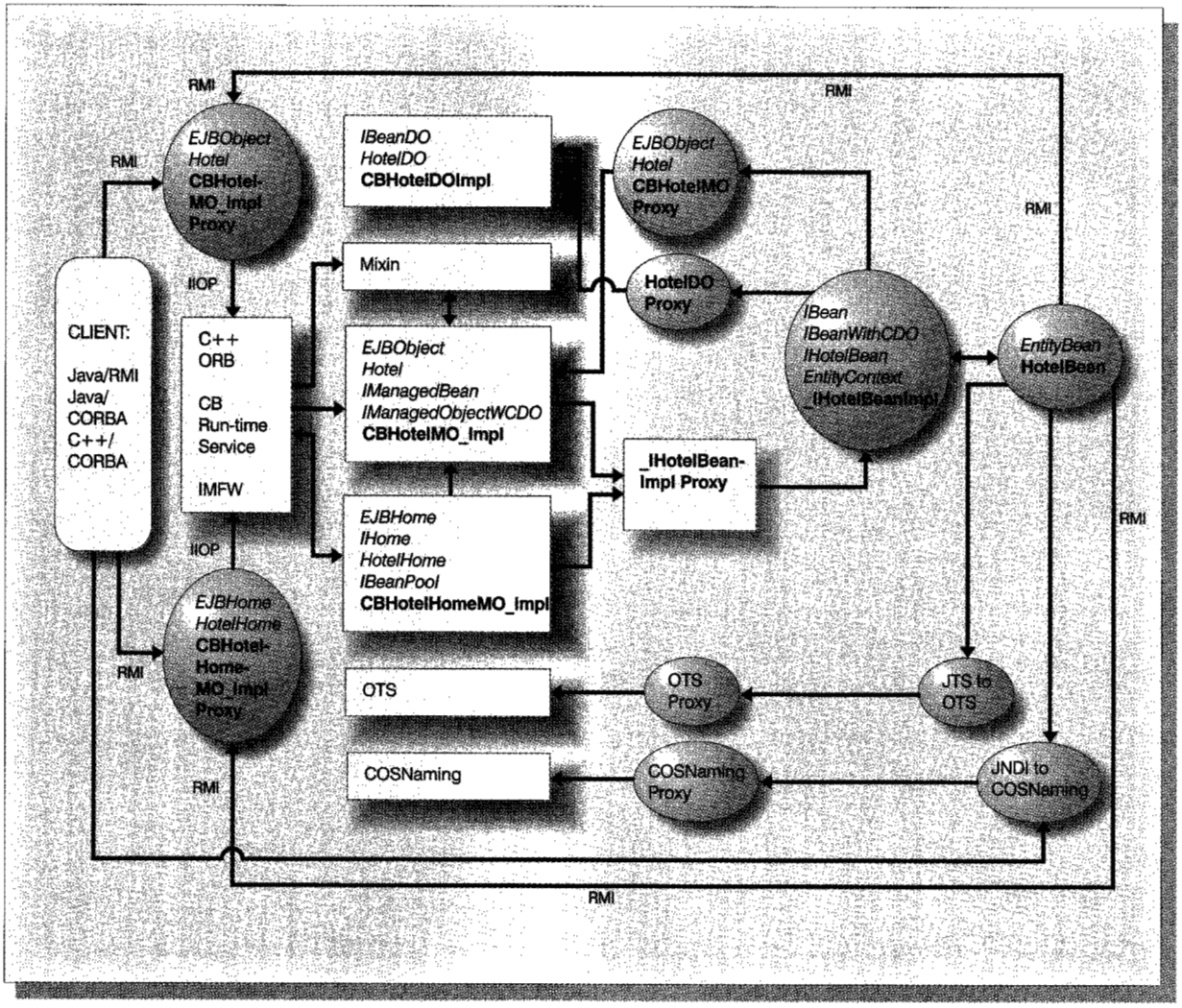
- During execution of *initForReactivation*, the managed object borrows a bean tie from the bean pool.
- During execution of *uninitForPassivation* and *uninitForDestruction*, the managed object returns the borrowed bean tie to the bean pool.

Keys. For a bean with identity (that is, a bean in a nonpooled state), the primary key is stored as attributes in the bean's associated data object. This is true for both bean-managed and container-managed entity beans. A bean's primary key class is packaged within the bean's JAR file and must be a valid RMI/IIOP value type, allowing an IDL value type to be generated from it. The bean's Component Broker primary key (IDL and C++ implementation) is generated from this generated IDL.

For container-managed entity beans, the data object is generated with an attribute for each of the bean's persistent fields. A subset of these attributes represents the bean's primary key. For bean-managed entity beans, the data object is generated from the bean's primary key IDL. It contains attributes representing the bean's primary key. The following transformations are performed on the various representations of the primary key:

1. The bean's Component Broker primary key is transformed to data object attributes. This form is used by the data object in the implementation of *internalizeFromPrimaryKey*.
2. Data object attributes are transformed to the bean's Component Broker primary key. This is used by the managed object in the implementation of *getPrimaryKeyString*.
3. The bean's primary key value type is transformed to the bean's Component Broker primary key. This is used by the home, in the implementation of the create and finder methods.
4. Data object attributes are transformed to the bean's primary key value type. This is used by the

Figure 15 An object interaction diagram for the Itinerary bean mapping



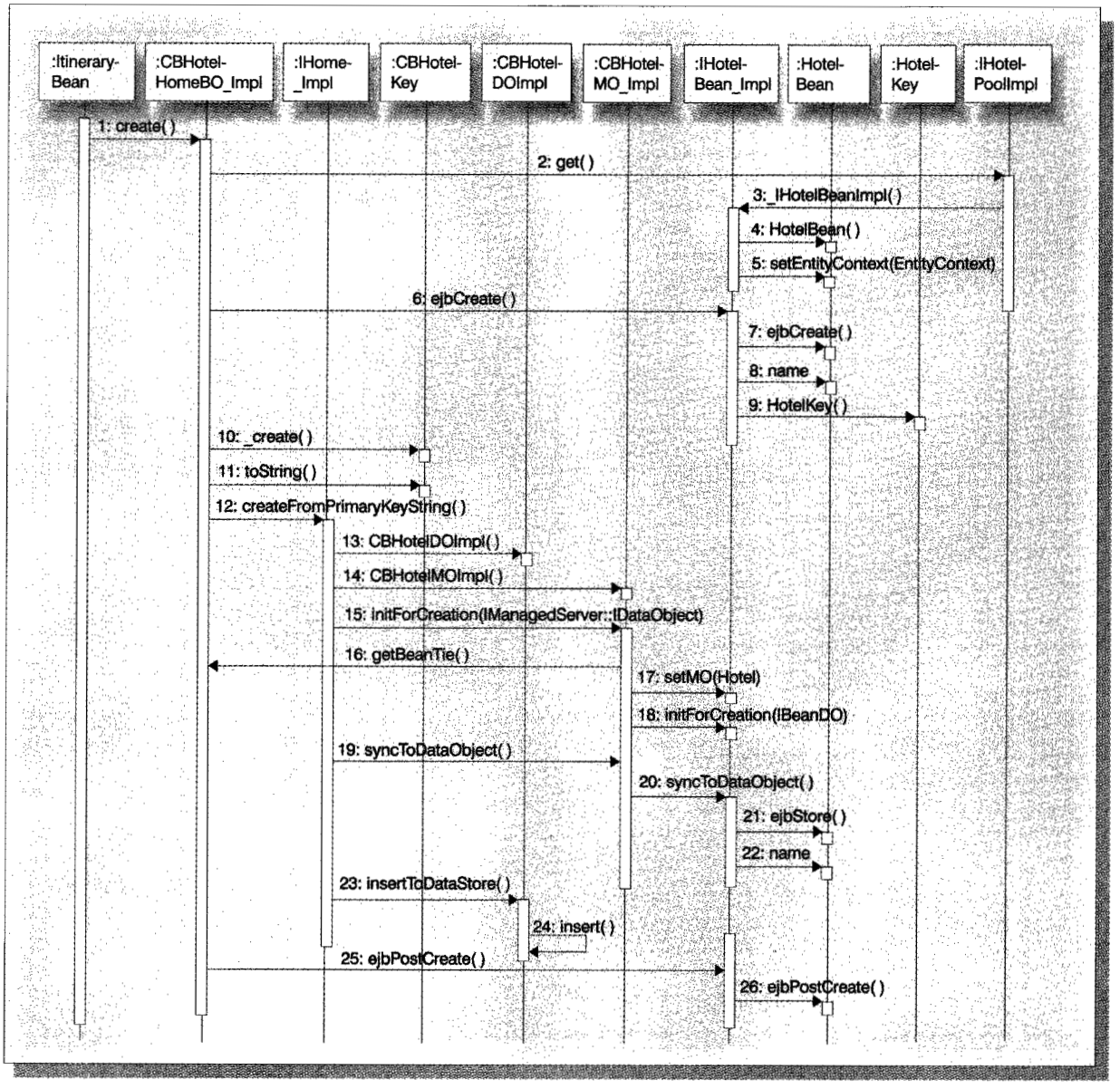
bean tie in the implementation of *EntityContext.getPrimaryKey* and by the managed object in the implementation of *EJBObject.getPrimaryKey*.

Entity context. The *EntityContext* interface extends *EJBContext*, and like *EJBContext*, it is implemented by the bean tie. The methods defined by the *EntityContext* interface and their mappings are as follows:

- The *getEJBObject* method can return a reference to the managed object's proxy, cast as an EJBObject.
- The *getPrimaryKey* method is implemented by the bean tie by getting the key's fields from the data object and building and returning a primary key object.

Entity bean container implementation issues. An entity bean's deployment descriptor can specify that a bean is re-entrant, although it is not clear whether it is possible for a client to perform concurrent calls to the same bean in the same transaction, or whether a bean is allowed to perform calls on itself. Given that a mixin already performs thread serialization

Figure 16 Sequence diagram: Creation of a HotelBean object

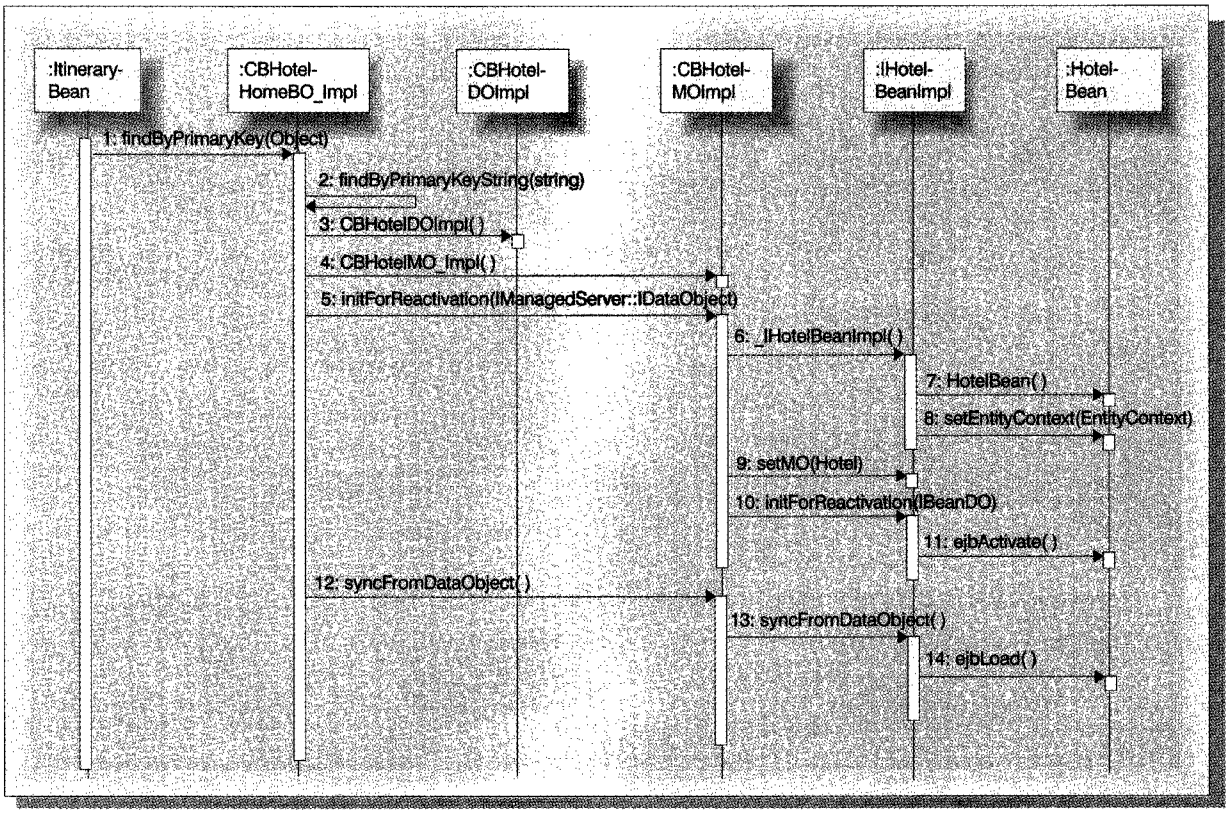


at the transaction level, an option to support thread serialization is for the mixin to make sure that only one method is being executed by the managed object implementing the bean.

Scenarios. Figure 15 shows an object interaction diagram that elaborates the diagram in Figure 11, with the major objects defined in the mapping of the *Hotel* and *HotelHome* interfaces and the HotelBean class.

The sequence of operations in Figure 16 illustrates the activation of a HotelBean instance when a client calls *findByPrimaryKey* on a *HotelHome* object. It is assumed that the CBHotelHomeMO_Impl object ex-

Figure 17 Sequence diagram for activation of a HotelBean object



ists at this point. This can be created at the time the CB server is launched.

The sequence of operations in the diagram of Figure 17 illustrates the creation of a HotelBean instance when a client calls *create* on a *HotelHome* object.

Support for session beans

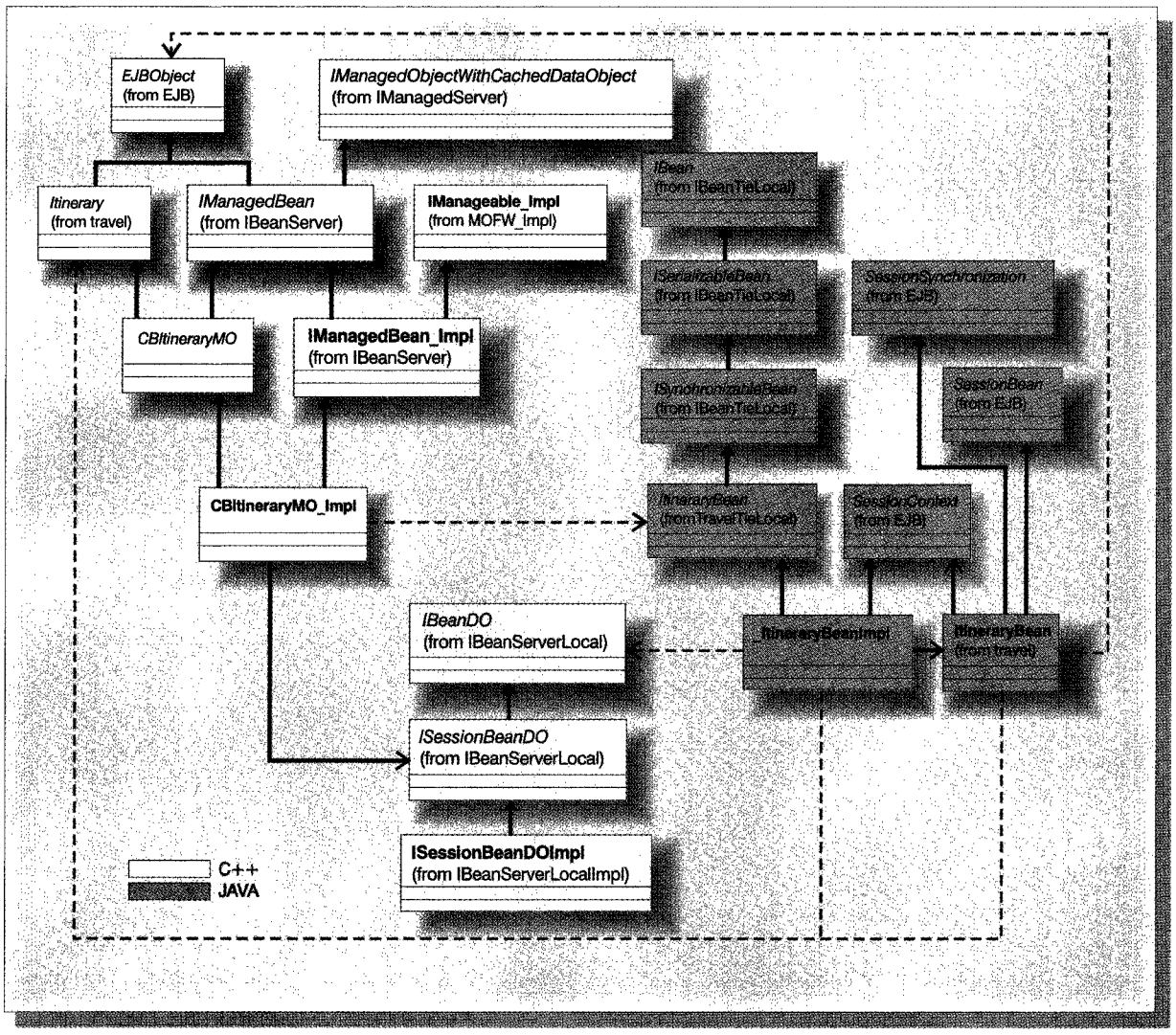
A session bean executes on behalf of a single client, can be transaction-aware, and can update data in an underlying database on behalf of the client, although it does not represent such data. A session bean is relatively short-lived and is destroyed when the EJB server crashes.

Session beans have a natural correspondence to Component Broker's application objects, which can be used to manage processes, tasks, or control flows. However, given that application objects are not yet fully available, and that a business object's data object can be useful, a session bean can be mapped to Component Broker using a managed object with a data object. The diagram in Figure 18 illustrates the mapping of an itinerary session bean.

There are three differences between this mapping and that of an entity bean:

1. ItineraryBean implements the *SessionSynchronization* interface.
2. The *IItineraryBean* interface, which exposes the bean tie to the C++ side, extends the *ISynchronizableBean*, *ISerializableBean*, and *IBean* interfaces. *ISynchronizableBean* exposes the *SessionSynchronization* interface to the C++ side so that the mixin can invoke its methods via CBItineraryMO_Impl. *ISerializableBean* exposes serialization and deserialization methods to the C++ side so

Figure 18 Mapping of an Itinerary session bean to Component Broker



that CBItineraryMO_Impl can perform such operations on the bean at passivation and reactivation. *IBean* is a generalization of *IBeanWithCDO*, which does not define methods to synchronize with a data object.

3. CBItineraryMO_Impl uses a generic data object, implemented by ISessionBeanDOImpl. This data object is used to store a primary key that the home can use to locate the bean. This is where the serialized conversational state can be stored for a "stateful" session bean when it is passivated.

A session bean implements the *SessionBean* interface. As with the EntityBean, the methods defined by this interface are mapped to Component Broker through the bean tie. The *setSessionContext* method, as we have seen, can be called from the bean tie's constructor when it instantiates the bean and is given a reference to the bean tie object itself, which implements the *SessionContext* interface. Other SessionBean interface methods can be called from methods defined in and exposed by the bean tie for this purpose, as shown in Table 3.

Table 3 SessionBean methods and corresponding bean-tie methods

SessionBean Method	Bean-Tie Method
setEntityContext	(Called from bean tie's constructor)
ejbRemove	*uninitForDestruction*
ejbActivate	*initForReactivation*
ejbPassivate	*uninitForPassivation*

Notice that *SessionBean* does not define *ejbLoad* and *ejbStore*. This is because a session bean has no persistent data to be transferred to and from a backend data store.

Also notice that *SessionBean* does not define *unsetSessionContext*, without which the container cannot ensure that a bean's session context is properly released when the bean is removed (in a similar fashion as an entity bean can be requested to unset its entity context). This method seems necessary and it could be invoked from the *release* method, defined for this purpose, on the bean tie.

Stateful session beans' conversational state and passivation. A stateful session bean can maintain conversational state, on behalf of its client, that must be retained across transactions. This state is not data that must be stored persistently in a database. However, passivation of a session bean requires holding on to its conversational data until it is reactivated. Thus, the data object, associated with the managed object that implements the session bean's remote interface, can be used to store the session bean's serialized conversational state as a BLOB (binary large object). This actually makes the session bean recoverable in the event of a crash, at least to the point of the latest passivation, which, combined with a passivate-at-end-of-transaction policy, would be to the point of the latest commit or rollback. Another issue has to do with what to serialize, in particular, what to do if a piece of the session bean's conversational state is not serializable. Options for dealing with this include requiring the bean developer to explicitly mark pieces of its conversational state that are to be passivated, using the transient keyword.

Session synchronization. If a session bean is transaction-aware, that is, if its methods are called within a transaction, as indicated in the session bean's deployment descriptor, then the session bean may implement the *SessionSynchronization* interface. Notice that this interface can only be implemented by stateful session beans. The methods defined by this interface and their mapping to Component Broker are as follows:

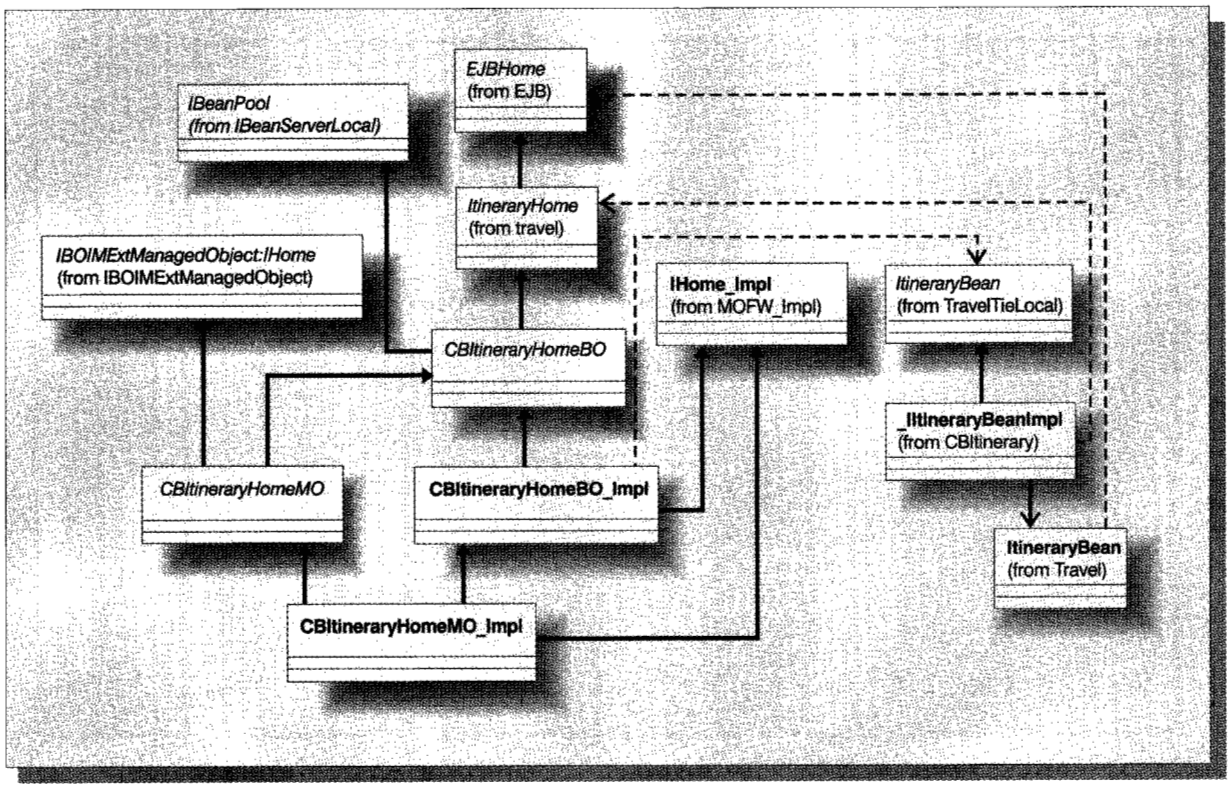
- The *afterBegin* method can be called from the mixin's *before* method, which can check whether a transaction has just begun. Alternatively, wherever the mixin starts a transaction it can perform this call.
- The *beforeCompletion* method can be called from the mixin's *beforeCompletion* method.
- The *afterCompletion* method can be called from the mixin's *afterCompletion* method and is given the mixin's status of the completion.

Notice that the managed object's transactional mixin needs to be subclassed in order to call these methods at appropriate times. This subclassing needs to occur from whatever mixin class is used to implement EJB transaction attributes, as discussed later.

Stateless session beans. Since stateless session bean instances are identical when they are not serving a client-invoked method, they do not need to be passivated; furthermore, there is no fixed mapping between clients and stateless instances. This means that from one method invocation to the next, the client may be interacting with different stateless instances. However, although a stateless bean cannot implement the *SessionSynchronization* interface, it can still have transaction attribute values defined for it or its methods. This can potentially create a situation where a stateless instance may be shared by two transactions. For example, consider a stateless session bean with the TX_REQUIRED attribute value and two methods *m1* and *m2*. A client may begin a transaction, invoke *m1* on an instance of this bean and then invoke *m2* after a considerable amount of "think" time. Meanwhile, another client invokes *m1* on an instance of this bean. Given that the first client has been holding its instance of the bean idle for some time, the container may decide to assign this instance to the second client, to improve performance. To avoid this situation, it would be useful to require a stateless instance to remain associated with a client for the duration of a transaction. This would make the sharing model for stateless session beans analogous to that of entity beans, which cannot be shared by two transactions or within the same transaction.

Session bean homes. Figure 19 shows the mapping of an itinerary entity bean's home to Component

Figure 19 Mapping of an Itinerary session bean's home to Component Broker



Broker. The CBItineraryHomeBO_Impl class implements the functionality in the *EJBHome* interface in terms of IHome_Impl. Here we see references from ItineraryBean to a proxy of the CBItineraryHomeMO_Impl class. This reference can be obtained, for example, by a call to *getEJBHome* on the *EJBObject* implemented by CBItineraryMO_Impl.

As we saw earlier for an entity bean, a session bean's home class also implements the *IBeanPool* interface.

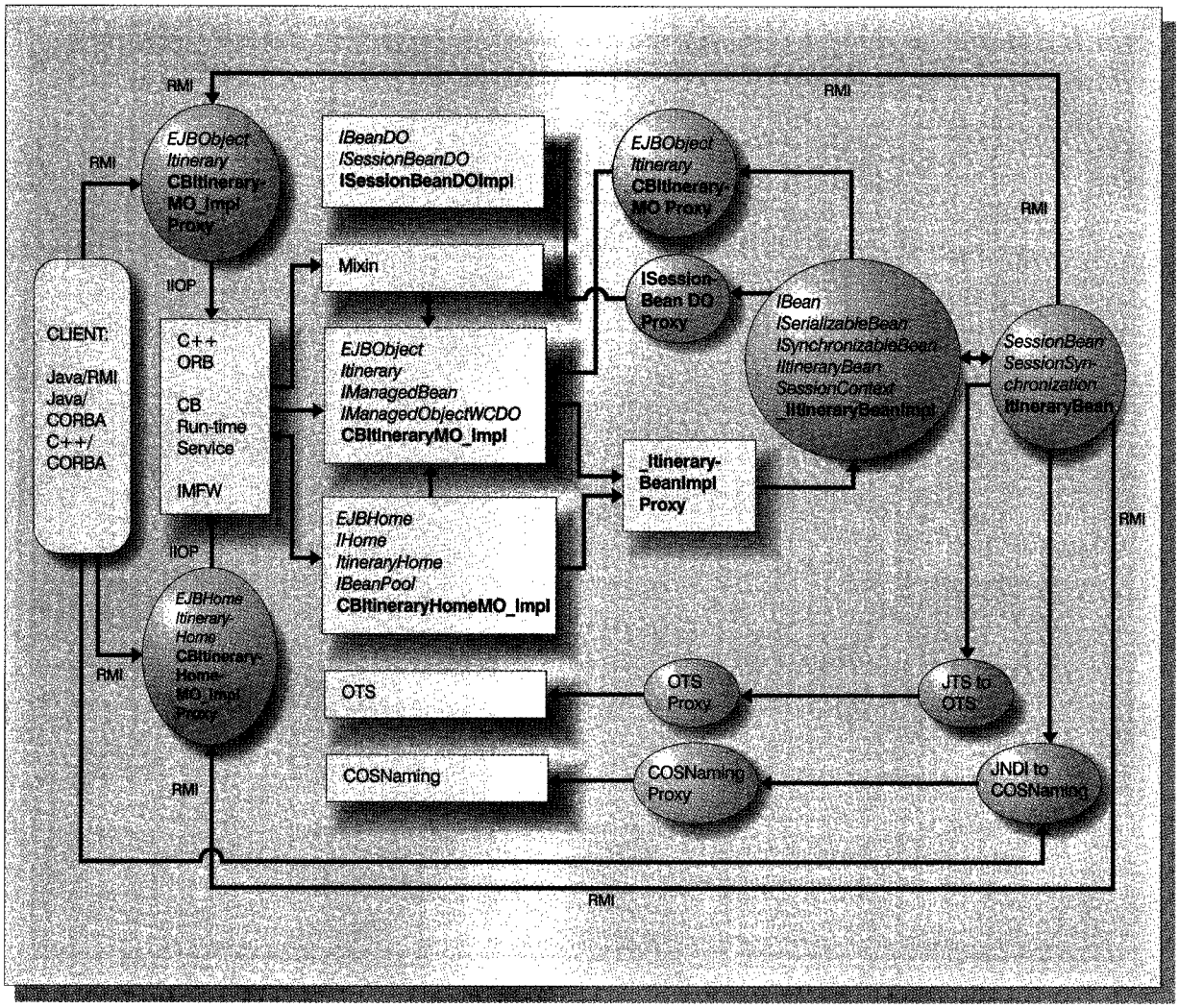
Identity. Session beans hide their identity from the client and from the bean itself. That is, at run time, neither the client nor the bean instance can obtain the bean's identity. A session bean is an anonymous resource, which exists on the client's behalf.

The managed object that represents the session bean's remote interface, however, requires identity, even though this identity is not visible to the client or the bean. The identity is needed to establish an object reference, for the managed object, that uniquely identifies the object. For stateful session beans, this identity serves as a key during passivation and activation.

Like entity beans, the session bean's identity is represented by a primary key. Unlike entity beans, however, session beans do not have state data that can be used as a primary key. For this reason, session beans use a special primary key, UUID (universal unique identifier), that provides a globally unique key for the bean. A common data object class ISessionBeanDOImpl, which extends *IUUIDDataObject*, is used for all session beans. *UUIDDataObject* instantiations are transient and support UUID primary keys.

Session context. The *SessionContext* interface extends *EJBContext*, and like *EJBContext*, it is implemented by the bean tie. One method is defined by the *SessionContext* interface: *getEJBObject*. This method can return a reference to the managed object's proxy, cast as an EJBObject.

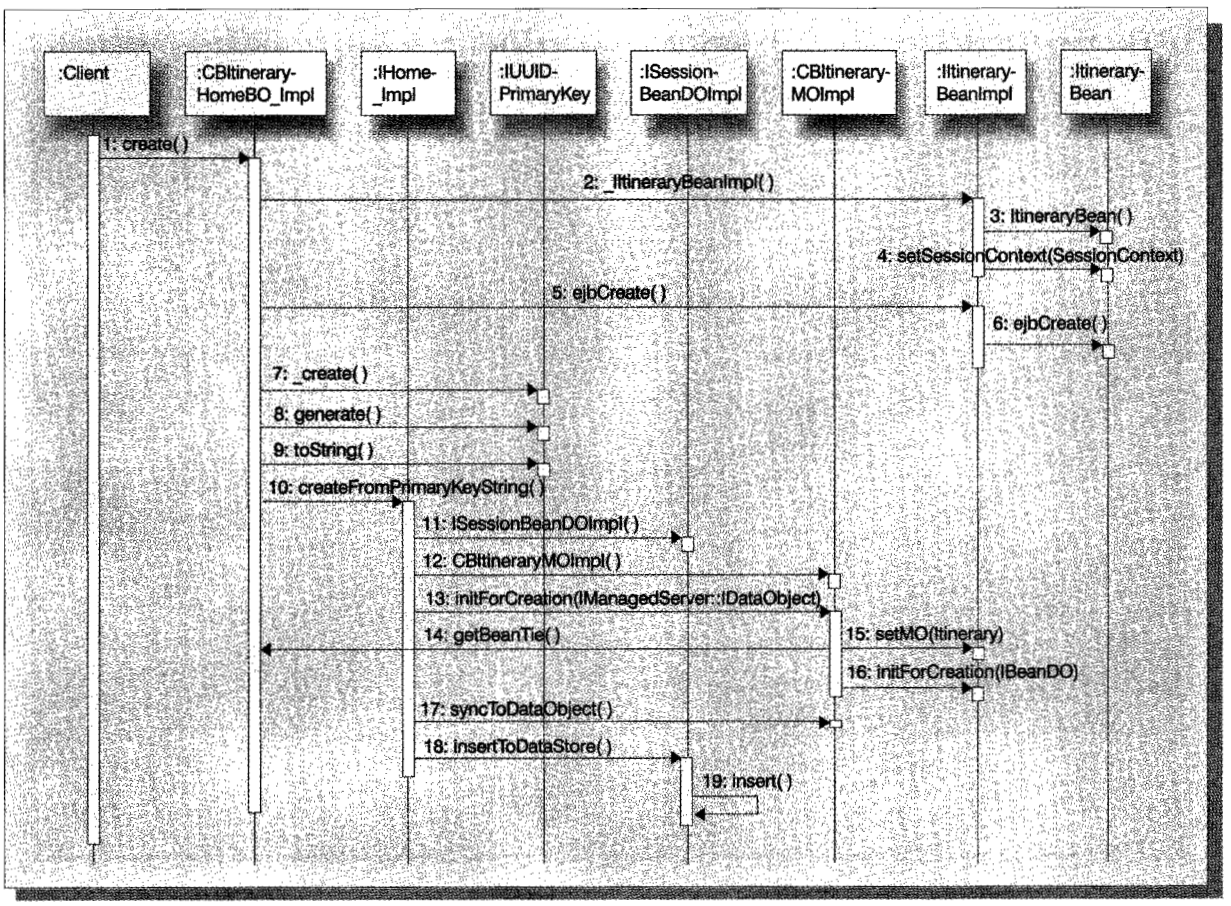
Figure 20 An object interaction diagram for the Itinerary bean mapping



Session bean container implementation issues. The discussion on serialized access to managed objects, in the section on entity bean container implementation issues, applies here as well. Additional issues for session beans are:

- *Thread serialization.* It is illegal to make a "loopback" call to a session bean. Also, session beans cannot be shared among clients, so it is illegal for two clients to make concurrent calls on a session bean instance. Thus, it is not necessary to allow for the case of a loopback call when enforcing thread serialization, or to differentiate between loopback calls from client concurrent calls, as is the case for entity beans.
- *Timing out.* A session bean's deployment descriptor can specify a session time-out value in seconds. One implementation option is to have the mixin start a timer in either its *after* method or its *afterCompletion* method. This timer can signal an event that the container can listen for to time out the session bean.
- *Transaction context of session bean methods.* If the methods *newInstance*, *setSessionContext*, *ejbCreate*, *ejbRemove*, *ejbPassivate*, *finalize*, *ejbActivate*, and *afterCompletion* of a session bean are not already

Figure 21 Sequence diagram: Creation of an ItineraryBean object



called without a transaction context, given the mapping defined in this and the previous sections, then it will be necessary for the container to suspend a potentially active transaction before calling these methods, and to resume it afterwards.

- *Reactivation.* Although the state diagram for stateful session beans (in the EJB specification) does not indicate it, it seems that it is necessary to invoke *newInstance* as well as *setSessionContext* prior to invoking *ejbActivate*, unless the diagram implies that the method "ready" state is also a pool state.

Scenarios. Figure 20 shows an object interaction diagram, analogous to the diagram in Figure 15, with the major objects defined in the mapping of the *Itinerary* and *ItineraryHome* interfaces, and the Itinerary-Bean class.

The sequence of operations in Figure 21 illustrates the creation of an ItineraryBean instance when a client calls *create* on an *ItineraryHome* object. It is assumed that the CBItineraryHomeMO_Impl object exists at this point. This can be created at launch time by the server.

Distribution

Given that Component Broker is CORBA/IIOP-based, to support the RMI-based programming model assumed by Enterprise beans, an implementation of RMI/IIOP is necessary. In particular, the following items are required in order to implement the support for EJB by Component Broker outlined in this document:

- Definition of remote interfaces, such as *EJBObject* and *EJBHome*, as RMI interfaces and compiled

Table 4 Context source and commit action for valid transaction attributes

	Values/Options	Transaction Started	No Transaction Started	Commit at Method Completion
EJB transaction attributes	TX_NOT_SUPPORTED	No context	No context	No
	TX_REQUIRED	Inherit context	New context	If no transaction started
	TX_SUPPORTS	Inherit context	No context	No
	TX_REQUIRES_NEW	New context	New context	Yes
	TX_MANDATORY	Inherit context	Throw exception	No
CB transaction models	Default	Inherit context	Throw exception	No
	Atomic	Inherit context	New context	If no transaction started
	Neutral	Inherit context	No context	No
	No	No context	No context	No

into IDL, with the subsequent generation of CORBA stubs and skeletons
- Support for "objects by value," to allow passing objects of types such as *EJBMetaData* and *java.util.Enumeration* (to return collections of primary keys), as well as handles and keys between clients and servers
- Translation of exceptions from IDL to Java. In addition to generating the appropriate value-based IDL exception for a given RMI exception, EJB-specified exceptions need to be mapped into their corresponding Component Broker exceptions. For example, an EJB client's reference on a referenced but nonexisting instance generates the NoSuchObjectException in RMI. This exception must be mapped into IDL and into the InvalidObjRef exception raised by Component Broker.

Transactions

Table 4 summarizes where the transaction context comes from when a client invokes an Enterprise bean object's method with valid transaction attribute values. The table also states whether or when a commit is issued by the transaction server when the invoked method completes.

The table also lists the types of transaction models supported by Component Broker. The options available (default, atomic, neutral, and no transaction) affect the way a managed object behaves when no transaction is active. No transaction is active if there is no transactional current object on the thread used to invoke a method on the object. Thus it is assumed that if a transaction is active then the mixin object and the managed object that it supports will execute in the context of this transaction. This is what is meant by the entries in the table marked "inherit context." This assumption is supported by the functional specification of the different types of mixin classes.

Mapping EJB transaction attributes. Based on Table 4, it seems that the following mapping from transaction attribute values to Component Broker transaction model options applies:

TX_NOT_SUPPORTED ⇨ No
TX_REQUIRED ⇨ Atomic
TX_SUPPORTS ⇨ Neutral
TX_MANDATORY ⇨ Default

As a result, it would be necessary to introduce a new mixin class to support the behavior prescribed by the "TX_REQUIRES_NEW" transaction attribute value.

In addition, the transaction attribute for an Enterprise bean can have the value TX_BEAN_MANAGED, which means that the bean is allowed to perform its own transaction demarcation. A method of a stateful session bean that begins a transaction could complete without committing or rolling back the transaction. Thus, the transaction context in which any given method of such a bean executes depends on whether or not the instance of the bean has itself begun a transaction but not committed it or rolled it back. This is in contrast to the transaction context depending on whether or not a client that invokes the given method was associated with a transaction context. This also has implications not only for a transaction context created by the Enterprise bean instance, but also for the transaction context associated with a method invocation coming from a client. Specifically, if a client's invocation has a transaction context associated with it, the transaction context must be suspended when the invoked method

Table 5 Atomic actions for mixin

	Client's Transaction Context	Container's Transaction Context	Bean's Transaction Context
At method start	Suspend Inherit Throw exception Do nothing	Begin Do nothing	Resume Do nothing
At method completion	Resume Do nothing	Commit Rollback Throw exception Suspend	Suspend Throw exception

starts and resumed when the invoked method completes. In addition, if a method of an Enterprise bean instance begins a transaction but does not commit it or roll it back, the corresponding transaction context must be suspended when the method completes and it must be resumed when any other method of the bean instance starts, until a method of the bean instance either commits the transaction or rolls it back. This nontrivial behavior for stateful session beans is currently not supported by the CB transaction model and thus a new mixin class or some other mechanism would have to be introduced to support it.

Transaction attribute values on a per-method basis. If these were all the changes to the Component Broker transaction model needed to support the EJB transaction attribute values, then the current architecture for transactional mixins could be preserved, with additions to support the TX_REQUIRES_NEW and TX_BEAN_MANAGED EJB transaction attribute values. But an EJB's transaction attribute can also be associated with an individual method, as opposed to being associated with an entire Enterprise bean. This means that from one method invocation to another, an Enterprise bean may be using different transaction attribute values. For example, let us assume that a client invokes *method1* on EJB1 with no transaction context and it subsequently invokes *method2* on EJB1, again with no transaction context. Let us further assume that *method1* has the TX_NOT_SUPPORTED transaction attribute value and that *method2* has the TX_REQUIRED transaction attribute value. Then the container would not need to do anything when *method1* is invoked or when it completes, but it would have to begin a transaction when *method2* is invoked and commit the transaction when *method2* completes. This behavior goes beyond the capabilities of the current transactional mixins provided by Component Broker.

To support EJB transaction attribute values on a per-method basis it is necessary to define a new kind of mixin. It must behave like any of the currently defined transactional mixins when any method of an Enterprise bean is invoked. Let us consider what atomic actions this mixin can perform, prior to starting an invoked Enterprise bean method and after such a method has completed, for any of the EJB transaction attribute values, including TX_REQUIRES_NEW and TX_BEAN_MANAGED. These atomic actions can be performed on a client's transaction context that comes in a method invocation, on a transaction context that can be or may have been begun by the container via the mixin itself, and on a transaction context that may have been begun by the bean (for the case of an Enterprise bean with the TX_BEAN_MANAGED value). Table 5 shows these atomic actions.

Before starting a given method, a client's transaction context can be suspended, or it can be propagated to the method's thread, i.e., it can be inherited, or the mixin can throw an exception, or it can do nothing if a client's transaction context is not present in the request to invoke the method. Similarly, at method completion the mixin can resume a client's transaction context if it is present, or do nothing otherwise. Also at method completion, the mixin can throw an exception if it needs to commit a transaction it started but the transaction context is not there anymore, perhaps because the bean committed it or rolled it back by mistake. The mixin could also suspend a transaction it started; although this action is not called for by any of the EJB transaction attribute values, it could be used to support behav-

Table 6 Actions performed by mixin for transaction attributes other than TX_BEAN_MANAGED

		Incoming Client's Transaction Context		No Incoming Client's Transaction Context	
	Transaction attribute value	Action on client's transaction context	Action on container's transaction context	Action on client's transaction context	Action on container's transaction context
At method start	TX_NOT_SUPPORTED	Throw exception or suspend	Do nothing	Do nothing	Do nothing
	TX_REQUIRED	Inherit	Do nothing	Do nothing	Begin
	TX_SUPPORTS	Inherit	Do nothing	Do nothing	Do nothing
	TX_REQUIRES_NEW	Throw exception or suspend	Begin	Do nothing	Begin
	TX_MANDATORY	Inherit	Do nothing	Throw exception	Do nothing
At method completion	TX_NOT_SUPPORTED	Resume	Do nothing	Do nothing	Do nothing
	TX_REQUIRED	Do nothing	Do nothing	Do nothing	Commit or rollback
	TX_SUPPORTS	Do nothing	Do nothing	Do nothing	Do nothing
	TX_REQUIRES_NEW	Resume	Commit or rollback	Do nothing	Commit or rollback
	TX_MANDATORY	Do nothing	Do nothing		

ior similar to disabling autocommit in JDBC (Java database connectivity). Finally, if a stateless session bean or an entity bean that uses TX_BEAN_MANAGED starts a transaction in a given method but it does not commit or roll back the transaction before the method completes, the mixin must throw an exception.

To perform these actions the mixin needs the following objects or pieces of information:

- A client's transaction context in a method request (or that there is no transaction context)
- The transaction policy for the method being invoked. To know this the mixin will need to have access to the run-time representation of the transaction attribute value for the method invoked or for the bean that contains it that was defined in the deployment descriptor.
- The kind of bean on which the request is being made. This allows the mixin to throw an exception at method completion on a bean's transaction that was not committed or rolled back, for stateless session or entity beans.
- A transaction context started by the mixin (if any)
- A bean's transaction context (if the bean has started one)

Given this information, a mixin can decide what to do for a given request on any method of its Enterprise bean. Whether or not an action is to be performed on a client's incoming transaction context will not depend on the value of the EJB method's transaction attribute; the mixin will always need to know whether there is a transaction context coming in from the client. This is in contrast with a transaction context the mixin may have created or with one the bean itself may have created. If the transaction attribute value is other than TX_BEAN_MANAGED, then the mixin need not be concerned about a transaction context having been created by the bean. The bean is prohibited from actually beginning a transaction if it does not have this transaction attribute value defined for one of its methods. On the other hand, if the transaction attribute value is TX_BEAN_MANAGED, then the mixin need not be concerned about a transaction context that it created. This is because the container, and thus the mixin, need not ensure that an EJB's method executes within a transaction; this is left to the bean itself.

Table 6 outlines the actions a mixin needs to perform on a client's incoming transaction context, as well as on a transaction context the mixin may need to create or may have created (depending on whether there actually is a client's incoming transaction context), and for each of the transaction attribute values other than TX_BEAN_MANAGED.

Notice that when there is a client's incoming transaction context the action on it at method start for TX_NOT_SUPPORTED and TX_REQUIRES_NEW could be either "throw exception" or "suspend." It could be "throw exception" because the container must guard against the bean being associated with a transaction and then having a method on the bean invoked outside the transaction, as is indicated in Section

Table 7 Actions performed by mixin for TX_BEAN_MANAGED

		Incoming Client's Transaction Context		No Incoming Client's Transaction Context	
		Action on client's transaction context	Action on bean's transaction context	Action on client's transaction context	Action on bean's transaction context
At method start	Existing bean's transaction context	Suspend	Resume	Do nothing	Resume
	No existing bean's transaction context	Suspend	Do nothing	Do nothing	Do nothing
At method completion	Existing bean's transaction context	Resume	Throw exception or suspend	Do nothing	Throw exception or suspend
	No existing bean's transaction context	Resume	Do nothing	Do nothing	Do nothing

11.7.2 of the EJB specification. If the client had begun a transaction and invoked a method with TX_SUPPORTS, for example, then the subsequent invocation of a method with either TX_NOT_SUPPORTED or TX_REQUIRES_NEW would require the client's incoming transaction context to be suspended, assuming the client did not commit or roll back the transaction between the two method invocations. However, suspending the client's incoming transaction context is not enough, because the bean—depending on the implementation of the container—may already have the client's transaction associated with its thread, thus violating the rule prescribed by Section 11.7.2 of the EJB specification.

Also notice that the two entries for the case where there is no client's incoming transaction context at method completion for TX_MANDATORY are left blank. This is because this situation cannot occur, given that at method start for this case an exception was thrown and the thread never reaches method completion.

Table 7 outlines the actions a mixin needs to perform on a client's incoming transaction context, as well as on a transaction context the bean may have created, for the transaction attribute value TX_BEAN_MANAGED.

Notice that at method completion, when there is an existing bean's transaction context, the action on it could be either "throw exception" or "suspend." This is because only stateful session beans with a TX_BEAN_MANAGED method can begin a transaction within that method and not commit it or roll it back, which is the only case where an existing bean's transaction can be suspended. In the remaining cases—either stateless session or entity bean—the container throws an exception.

Isolation levels. Container-managed entity beans, just like their counterpart Component Broker business objects, cache their persistent data items. On the other hand, the CB programming model does not provide any facility for supporting isolation levels at this level of caching. The client(s) are supposed to provide their own concurrency control.

One option to support the isolation levels specified by EJB is to use the concurrency service to implement the isolation levels declared in the deployment descriptor, at least for support on Component Broker for the workstation. Component Broker for OS/390* (Operating System/390) does not do any caching of data at any level. So in Component Broker for OS/390, support for isolation levels would require mapping the isolation level options to whatever isolation levels its back-end data store provides. For example, DB2 provides isolation levels that are similar to those defined by EJB: uncommitted read, cursor stability, read stability, and repeatable read.

Security

Supporting EJB security-related functions in Component Broker requires mapping a subset of the Java security model into the security model defined for Component Broker. There are two areas of concern: security between the client and the server, and security within Component Broker. Security between

the client and the server can be handled by Component Broker's Secure Socket Layer implementation.

The EJB specification defines methods such as *getCallerIdentity* and *isCallerInRole*, as well as deployment descriptor items that deal with authenticating the identity of an Enterprise bean to a Component Broker container. In addition, the *AccessControlEntry* interface deals with deciding what entities can run methods on an Enterprise bean.

Identity. The *java.security.Identity* interface can be mapped to CORBA's *SecurityLevel2::Credentials* by providing support for the privilege attributes security name, group, role, capabilities, clearance level, and host authentication. These attributes can be part of a CBIdentity class that implements *java.security.Identity*.

Component Broker defines three kinds of credentials: received, own, and invocation. It creates and places a received credential on the thread of execution for any method request that comes into a server. A call such as

```
CBIdentity id = EJBContext.getCallerIdentity()
```

can access and return the received credential, generated by Component Broker, as a CBIdentity object.

The "own" credential is the identity of the principal that owns the object. By default, the own credential points at the server process. This credential can be changed. Given that an EJB's ControlDescriptor defines runAsMode and runAsIdentity attributes, it should be possible to set the own credential of an EJB's implementation. An issue that arises here is how to authenticate an object or method whose mode or identity is being changed. One possibility is to use DCE to create a set of authorized log-in sets. When an Enterprise bean is deployed, the principal name of a control descriptor's mode or identity attribute must correspond to one of these authorized log-in entities. This authentication can be done in the constructor for CBIdentity, which takes a principal name as argument and performs the authentication when a CBIdentity is created, returning a null object if the authentication fails.

The invocation credential is the credential of the principal on whose behalf down-stream requests are performed. If delegation is disabled, the invocation credential is set equal to the own credential, that is, down-stream requests are invoked under the own credential. If delegation is enabled, the invocation credential is set equal to the received credential, that is, down-stream requests are invoked under the credential of the requester.

Invocation. Invocation deals with deciding which entities can run methods on an Enterprise bean. The EJB AccessControlList class allows the specification, on a per-bean or per-method granularity, of which principals are allowed to execute a method.

All inter-Enterprise bean method requests are intercepted by the managed object's *before* method. Thus, the basic approach for invocation authentication would be to catch each method request and check the invocation credential against the list of CBIdentities in the EJB AccessControlList for the given method.

Naming

A client of an Enterprise bean locates the bean's home interface using JNDI. An Enterprise bean's home interface name is defined in the bean's deployment descriptor. This name is used as the trailing part of the actual name that the container binds into its version of the JNDI name space. This means that the container can prefix the name of an Enterprise bean's home as defined in the bean's deployment descriptor with an arbitrary JNDI path. An Enterprise bean home is mapped by Component Broker into a CB home, which can be found using CB's implementation of the COS naming service. As we saw earlier, in addition to extending the COS naming interfaces, CB's naming service defines an architected structure for its name space, called the System Name Tree.

In Component Broker's programming model, homes are found using factory finders, which in turn are stored in the System Name Tree in a structured location. This location is typically denoted by the name "host/resources/factory-finders/host-scope." Notice how this name is traversed in the System Name Tree by starting at the local host root node and following the host name.

Thus, to map the name of an Enterprise bean's home to Component Broker, two issues must be dealt with:

- Factory finders are not used in the EJB programming model.
- The name of the Enterprise bean's home is an ar-

bitrary JNDI name that does not denote any structured location.

It is possible in principle to deal with the last issue by prefixing the location of the factory finder to the Enterprise bean's home name as defined in the deployment descriptor. In most cases, this will not be enough, as additional deployment-time configuration may be required. For example, the deployer may need to further prefix the home's path with a name appropriate to the enterprise. Additionally, it is not specified how clients (including bean instances) discover the full path to the home of a deployed bean. It seems reasonable to expect a bean instance, in the client role, to generate JNDI paths at run time, based on some exposed environment property (which would have to be configured at deployment time). This would essentially lead to nonstandard, bean-specific, deployment descriptors masquerading as environment properties. Furthermore, it does not address the client side.

Conclusion

Focusing on the architectural mapping of Enterprise beans to CB alone does not address the run-time and system management requirements that are needed. Future work is needed to define new functions in Component Broker administrative tools. These functions would include the ability to construct and execute the necessary Java Naming and Directory Interface services to seamlessly register Enterprise beans to clients from a CORBA naming tree. Interfaces to native security controls to govern access to beans need to be incorporated into these deployment tools. Configuring Component Broker servers that load JAR files from application-specific class paths are needed for production-level isolation. Packaging CB-generated classes with specific EJB JAR files also needs to be elucidated. Where Enterprise beans take advantage of Component Broker function for capabilities richer than the EJB specification, additional deployment and packaging capabilities are needed from these tools.

Enabling Enterprise JavaBeans to run on Component Broker servers merges the strengths of two worlds. To EJB users, a CB environment provides a secure, transactional, scalable environment. For Component Broker application developers, the plethora and richness of JavaBeans tools can be used to develop CB applications. The pervasiveness of Java and the larger functionality of Component Broker are made available to both the Java and CB communities. EJB developers do not require any knowledge of CB to run Enterprise beans on Component Broker. Indeed, CB can be viewed as an extension to EJB. Enterprise beans can be made to run in CB with little effort. Writing applications using EJB classes makes Java a fine language for creating enterprise applications.

At the time this article was written, Gemstone Systems, Inc., Oracle Corporation, WebLogic, Inc., Netscape Communications Corporation, IBM, and SUN Microsystems, Inc. had announced general availability dates for EJB implementations. For platforms supporting Component Broker, the design for scalable and transactional EJB servers is already in place.

Cited references and notes

1. O. Gample, A. Gregor, S. B. Hassen, D. Johnson, W. Jönsson, D. Racioppo, H. Stöllinger, K. Washida, and L. Widengren, *Component Broker Connector Overview*, IBM International Technical Support Organization, SG24-2022-00 (May 1997); available from IBM branch offices or at http://www.redbooks.ibm.com/.
2. See http://www.omg.org for more information about the Object Management Group.
3. R. Orfali, D. Harkey, and J. Edwards, *The Essential Distributed Objects Survival Guide*, John Wiley & Sons, Inc., New York (1995).
4. V. Matena and M. Hapner, *SUN Microsystems Enterprise JavaBeans*, Version 1.0 (March 1998); available at ftp://ftp.javasoft.com/docs/ejb/.
5. I. F. Brackenbury, D. F. Ferguson, K. D. Gottschalk, and R. A. Storey, "IBM's Enterprise Server for Java," *IBM Systems Journal* **37**, No. 3, 323–335 (1998).
6. R. Orfali and D. Harkey, *Client/Server Programming with Java and CORBA*, 2nd edition, John Wiley & Sons, Inc., New York (1998).
7. By abstract object bus, we mean a high-level transport mechanism that allows objects to interoperate, regardless of location, source language, or operating system.
8. For more information on Encina, see http://www.transarc.com/dfs/public/www/htdocs/hosts/external/Product/Txseries/Encina/Brochure2.0/encina.html.
9. This notation indicates *module name::interface*.
10. DCE is the Distributed Computing Environment, from the Open Software Foundation, now called The Open Group. DCE is "middleware" that consists of multiple components that have been integrated to work closely together. See http://www.camb.opengroup.org/tech/dce/info.
11. That is, it must implement the Java *Serializable* interface.

Accepted for publication July 1, 1998.

Christopher F. Codella *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: codella@us.ibm.com).* Dr. Codella received the B.S. degree from Rutgers University in 1977, the M.S.E. degree from the University of Michigan in 1978, and the Ph.D. degree from Cornell University in 1984, all in electrical engineering. He is currently a research staff member and manager of the Component Software group at the Thomas J. Watson Research Center. Dr. Codella joined IBM in 1979, doing semiconductor device design and simulation at the East Fishkill, New York, semiconductor lab and the Watson Research Center. In 1989 he joined the Computer Science department at Watson, working in simulation, 3D graphical virtual environments, and distributed object systems, and became manager of the Virtual Worlds group there in 1993. In 1996 he worked on assignment as manager of Enterprise Application Architecture in the IBM Consulting Group. Returning to the Research Division in 1997, he formed the Component Software group involved in research and development in distributed object technology, enterprise application middleware, and enterprise Java, as well as pilot customer engagements with Component Broker. Dr. Codella is a Senior Member of the Institute of Electrical and Electronics Engineers.

Donna N. Dillenberger *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: engd@us.ibm.com).* Ms. Dillenberger joined IBM in 1988 at the former Data Systems Division in Poughkeepsie. She joined the Research Division in 1994. She has worked on future hardware simulations, workload management, objects, and the Web.

Donald F. Ferguson *IBM Software Group, Thomas J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, New York 10532 (electronic mail: dff@us.ibm.com).* Dr. Ferguson joined IBM as a research staff member in 1987 and is currently a Distinguished Engineer in the IBM Software Group. He is also Chief Architect and Technical Leader for IBM's Component Broker product and Enterprise JavaBeans implementations. During his career in IBM, he has contributed to IBM cache management, operating system and transaction processing workload management, multimedia content server, system management, and distributed object-oriented products. He is an author of seven current or pending patents and over two dozen technical publications. Dr. Ferguson has received two IBM Outstanding Innovation Awards, four Research Division Technical Awards, two IBM Invention Plateau Awards, an IEEE best paper award, and was elected to the IBM Academy of Technology in 1997.

Rory D. Jackson *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (roryj@us.ibm.com).* Mr. Jackson is an advisory engineer in the Component Software group at the Watson Research Center. He received the B.E. degree in electrical engineering from New York University, the M.S. degree in computer engineering from Syracuse University, and the M.B.A. degree in management information systems from Iona College, New Rochelle, New York. Mr. Jackson has been with IBM since 1975 and has worked on the design and development of EBeam lithography systems, parallel processor machines, the 1996 Atlanta Olympics Web database, and enterprise software products.

Thomas A. Mikalsen *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: tommi@watson.ibm.com).* Mr. Mikalsen is an advisory software engineer and member of the Component Software group at the Watson Research Center. He received the B.S. degree in computer science in 1991 from the Rochester Institute of Technology and is presently working toward the M.S. degree in computer science from Polytechnic University, Brooklyn, New York. His interests include distributed systems, distributed object frameworks, and application development tools.

Ignacio Silva-Lepe *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598 (electronic mail: isilval@us.ibm.com).* Dr. Silva-Lepe joined IBM in 1997 and is an advisory software engineer with the Component Software group. He participated in the design and implementation of the mapping of the Enterprise JavaBeans specification onto Component Broker, the definition of the composite business object for Component Broker's Object Builder, and more recently on the definition of a new type of Enterprise bean that can communicate via asynchronous messaging. He received the B.S. degree in computer systems engineering from Universidad ITESO, Guadalajara, Mexico in 1985, and the M.S. and Ph.D. degrees in computer science from Northeastern University, Boston, Massachusetts in 1989 and 1994.

Reprint Order No. G321-5688.