Sun Microsystems

Enterprise JavaBeans™

This is a draft specification of the Enterprise JavaBeans architecture for public review. We expect to finalize this specification after 60 - 90 days of public review.

Enterprise JavaBeans is a component architecture for development and deployment of object-oriented distributed enterprise-level Java applications. Applications written using Enterprise JavaBeans are scalable, transactional, and multi-user secure. These applications can be written once, and then deployed on any Java enabled server platform.

Sun Microsystems Inc. Proprietary and Confidential document

Please send technical comments on this specification to:

ejb-spec-comments@sun.com

Please send product and business questions to:

ejb-marketing@sun.com

Copyright © 1997 by Sun Microsystems Inc.
901 San Antonio Road, Palo Alto, CA 94303.
All rights reserved.

Version 0.8

Vlada Matena & Mark Hapner

December 9, 1997 2:09 pm

RESTRICTED RIGHTS: Use, duplication or disclosure by the government is subject to the restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software Clause as DFARS 252.227-7013 and FAR 52.227-19.

Sun, Sun Microsystems, the Sun logo, Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the Unites States and other countries.

THIS PUBLICATION IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MECHANTABILITY, FITNESS FOR A PARTICULAR USE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. SUN MICROSYSTEMS, INC., MAY MAKE NEW IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Contents

1. Introduction 6

1.1 Target audience 6
1.2 Acknowledgments 6
1.3 Organization 6

2. Goals 7

2.1 Overall goals 7
2.2 Goals for Release 1.0 7

3. Roles and scenarios 8

3.1 Roles 8
3.2 Scenario 1: Development, deployment, assembly 9

4. Fundamentals 11

4.1 Enterprise beans as components 11
4.2 Enterprise JavaBeans contracts 12
4.3 Session and entity objects 14
4.4 Standard CORBA mapping 15

5. Client view of a session bean 17

5.1 Overview 17
5.2 EJB container 17
5.3 EJB Object 19
5.4 Session object identity 19
5.5 Client's view of session bean's lifecycle 20
5.6 Creating and using a session bean 21

6. Session bean state management 22

6.1 Overview 22
6.2 Goals 22
6.3 A container's management of its working set 22
6.4 Conversational state 23
6.5 The protocol between a session bean and its container 23
6.6 Non-transactional session bean state diagram 26
6.7 Transactional session bean state diagram 28
6.8 Sequence diagrams 30

7. Example session scenario 35

7.1 Overview 35
7.2 Inheritance relationship 36

8. Client view of an entity 39

8.1 Overview 39
8.2 EJB container 39
8.3 Entity EJB object lifecycle 41

8.4 Primary key and object identity . . . 43
8.5 Enterprise bean's remote interface . . . 44
8.6 Enterprise bean's handle . . . 45

9. Entity container protocol . . . 46

9.1 The runtime execution model . . . 46
9.2 Entity persistence . . . 47
9.3 Instance lifecycle . . . 48
9.4 The entity container contract . . . 49
9.5 The design pattern for business method delegation . . . 54
9.6 The design pattern for the factory interface . . . 54
9.7 The design pattern for the finder interface . . . 57
9.8 Sequence diagrams . . . 57

10. Example entity scenario . . . 67

10.1 Overview . . . 67
10.2 Inheritance relationship . . . 68

11. Support for transactions . . . 71

11.1 Transaction model . . . 71
11.2 Relationship to JTS . . . 71
11.3 Scenarios . . . 71
11.4 Declarative transaction management . . . 76
11.5 Bean-managed demarcation . . . 79
11.6 Transaction management exceptions . . . 79

12. Support for distribution . . . 80

12.1 Overview . . . 80
12.2 Client-side objects . . . 80
12.3 Interoperability via network protocol . . . 80

13. Support for security . . . 82

13.1 Package java.security . . . 82
13.2 Security-related methods in InstanceContext . . . 82
13.3 Security-related deployment descriptor properties . . . 82
13.4 Examples . . . 82

14. Ejb-jar file . . . 84

14.1 ejb-jar file . . . 84
14.2 Deployment descriptor . . . 84
14.3 ejb-jar Manifest . . . 84

15. Enterprise bean provider responsibilities . . . 85

15.1 Classes and interfaces . . . 85
15.2 Environment properties . . . 86
15.3 Deployment descriptor . . . 87
15.4 Programming restrictions . . . 87
15.5 Component packaging responsibilities . . . 87

16. Container provider responsibilities 88
17. Enterprise JavaBeans API Reference. 89
18. Related documents. 134
Appendix A. Glossary of terms 135
Appendix B. Example application 136
Appendix C. Features deferred to future releases 137
C.1 Programmatic access to security 137
C.2 Enterprise beans with extended transactional semantics . . . 137
Appendix D. Issues and dependencies. 138
D.1 Pending issues 138
Appendix E. package java.jts. 140

1 Introduction

1.1 Target audience

The target audience for this specification are the vendors of transaction processing platforms, vendors of enterprise application tools, and other vendors who want to provide support for Enterprise JavaBeans in their products.

Many concepts described in this document are system-level issues that the application programmer using Enterprise JavaBeans will not have to deal with. Since the main goal of Enterprise JavaBeans is to hide these complex system level issues from the application programmer, we plan to provide a separate Enterprise JavaBean programmer's primer.

We thank the reviewers for their time and effort in helping us to improve the specification.

1.2 Acknowledgments

Rick Cattell, Shel Finkelstein, Graham Hamilton, Li Gong, Rohit Garg, Susan Cheung, and Anil Vijendran have provided invaluable input to the design of Enterprise JavaBeans.

Enterprise JavaBeans is a broad effort that includes contribution from numerous groups at Sun and at partner companies. The ongoing specification review process has been extremely valuable, and the many comments that we have received helped us to define the specification.

1.3 Organization

TODO - describe document organization

2 Goals

2.1 Overall goals

We have set the following goals for the Enterprise JavaBeans (EJB) architecture:

- Enterprise JavaBeans will be the standard component architecture for building distributed object-oriented business applications in Java. Enterprise JavaBeans will make it possible to build distributed applications by combining components developed using tools from different vendors.
- Enterprise JavaBeans will make it easy to write applications: Application developers will not have to deal with low-level details of transaction and state management; multi-threading; resource pooling; and other complex low-level APIs. However, an expert-level programmer will be allowed to gain direct access to the low-level APIs.
- Enterprise JavaBeans applications will follow the "write-once, run anywhere" philosophy of Java. An enterprise bean can be developed once, and then deployed on multiple platforms without recompilation or source code modification.
- The Enterprise JavaBeans architecture will address the development, deployment, and runtime aspects of an enterprise application's lifecycle.
- The Enterprise JavaBeans architecture will define the contracts that enable tools from multiple vendors to develop and deploy components that can interoperate at runtime.
- The Enterprise JavaBeans architecture will be compatible with existing server platforms. Vendors will be able to extend their existing products to support Enterprise JavaBeans.
- The Enterprise JavaBeans architecture will be compatible with other Java APIs.
- The Enterprise JavaBeans architecture will provide interoperability between enterprise beans and non-Java applications.
- The Enterprise JavaBeans architecture will be compatible with CORBA.

2.2 Goals for Release 1.0

In Release 1.0, we want to focus on the following aspects:

- Define the distinct "roles" that are assumed by the component architecture.
- Define the client's view of enterprise beans.
- Define the enterprise bean developer's view.
- Define the responsibilities of an *EJB container provider* and *server provider*; together these make up a system that supports the deployment and execution of enterprise beans.
- Define the format of the *ejb-jar file*, EJB's unit of deployment.

3 Roles and scenarios

3.1 Roles

The Enterprise JavaBeans architecture defines five distinct roles in the application development and deployment workflow. Each role may be performed by a different party. Enterprise JavaBeans specifies the contracts that ensure that the product of each role is compatible with the product of the other roles.

In some scenarios, a single party may perform several roles. For example, the container provider and the EJB server provider may be the same vendor. Or a single programmer may perform the role of the enterprise bean provider and the role of the application assembler.

The following sections define the five EJB roles.

3.1.1 Enterprise bean provider

An enterprise bean provider is typically an application domain expert. An enterprise bean provider develops reusable components called enterprise beans. An enterprise bean implements a business task.

An enterprise bean provider is not an expert at system-level programming. Therefore, an enterprise bean provider usually does not program transactions, concurrency, security, distribution and other services into the enterprise beans. An enterprise bean provider relies on an EJB container provider for these services.

The output of an enterprise bean provider is an ejb-jar file that contains enterprise beans. Each bean includes its Java classes, its deployment descriptor, and its environment properties.

3.1.2 Application assembler

An application assembler is a domain expert who composes applications out of enterprise beans. An application assembler produces an ejb-jar file that contains enterprise beans with their deployment descriptors and environment properties. The ejb-jar file might include additional files that are part of the assembled application but whose definition is outside of Enterprise JavaBeans.

The assembler may also write a GUI client-side for the applications.

3.1.3 Deployer

A deployer is an expert at a specific operational environment, and is responsible for the deployment of enterprise beans and their containers. A deployer typically uses tools provided by the container provider to adapt enterprise beans to an operational environment.

For example, a deployer is responsible for mapping the security roles assumed by the enterprise beans to those required by the organization that will be using the enterprise beans. A deployer typically reads the attribute settings in the enterprise beans' deploy-

ment descriptors and modifies the values of the enterprise beans' environment properties.

3.1.4 EJB container provider

The expertise of a container provider is system-level programming, possibly combined with some application-domain expertise. The focus of a container provider is on the development of a scalable, secure, transaction-enabled container system. The container provider insulates the enterprise bean from the specifics of an underlying EJB server by providing a simple, standard API between the enterprise bean and the container.

An entity[4.3.2] container is typically responsible for persistence of its entity enterprise beans. The container provider's tools are used to generate code that moves data between the enterprise bean's instance variables and a database. The container provider may be an expert in the area of providing access to legacy data sources.

A container provider is responsible for the evolution of enterprise beans. For example, the container provider should allow enterprise bean classes to be upgraded without invalidating existing clients or losing existing enterprise bean objects.

Enterprise JavaBeans defines the container contract that must be supported by every compliant EJB container. Enterprise JavaBeans allows container vendors to develop specialized containers that extend this contract. Examples of specialized containers include a container that supports an application-domain specific framework, a container that implements an Object/Relational mapping, or a container that is built on top of an object-oriented database system.

3.1.5 EJB server provider

An EJB server provider is a specialist in the area of distributed transaction management, distributed objects, and other lower-level system-level services. A typical EJB server provider is an OS vendor, middleware vendor, or database vendor.

Typically, the EJB server provider will provide a container that implements the EJB *session container*[4.2.2] contract, and may also provide an *entity container*[4.2.2] for one or more data sources supported on the EJB server.

An EJB server provider will typically publish its lower-level interfaces to allow third parties to develop containers.

> *A later release of Enterprise JavaBeans may standardize the interfaces between a container and an EJB server.*

3.2 Scenario 1: Development, deployment, assembly

Wombat Inc. is an enterprise bean provider that specializes in the development of software components for the banking sector. Wombat Inc. has developed the *AccountBean* and *TellerBean* enterprise beans, and packages them in an *ejb-jar* file.

Wombat sells the enterprise beans to banks that may use containers and EJB servers from multiple vendors. One of the banks uses a container from the Acme Corporation. Acme's tools that are part of Acme's container product facilitate the deployment of en-

terprise beans from any provider, including Wombat Inc. The deployment process results in generating additional classes used internally by Acme containers. The additional classes allow Acme containers to manage enterprise bean objects at runtime, as defined by the EJB container contract.

Since the *AccountBean* and *TellerBean* enterprise beans by themselves are not a complete application, the bank MIS department may use Acme's tools to assemble the *AccountBean* and *TellerBean* enterprise beans with other enterprise beans (possibly from another vendor) and possibly with some non-EJB existing software, into a complete application. The MIS department takes on both the EJB deployer and application assembler roles.

4 Fundamentals

This chapter defines the scope of the Enterprise JavaBeans specification.

4.1 Enterprise beans as components

Enterprise JavaBeans is an architecture for component based distributed computing. Enterprise beans are components of distributed transaction-oriented enterprise applications.

4.1.1 Component characteristics

The essential characteristics of an enterprise bean are:

- An enterprise bean is contained and managed at runtime by a container.
- An enterprise bean can be customized at deployment time by editing its environment properties.
- Various metadata, such as a transaction mode and security attributes, are separated out from the enterprise bean class. This allows the metadata to be manipulated using container's tools at design and deployment time.
- Client access is mediated by the container and the EJB server on which the enterprise bean is deployed.
- If an enterprise bean uses only the standard container services defined by the EJB specification, the enterprise bean can be deployed in any compliant EJB container.
- Specialized containers can provide additional services beyond those defined by the EJB specification. An enterprise bean that depends on such a service must be deployed only in a container that supports the service.
- An enterprise bean can be included in a composite application without requiring source code changes or recompilation of the enterprise bean.
- A client's view of an enterprise bean is defined by the bean developer. Its view is unaffected by the container and server the bean is deployed in. This ensures that both beans and their 100% Pure Java™ clients are write-once-run-anywhere.

4.1.2 Flexible component model

The enterprise bean architecture is flexible enough to implement components such as the following:

- An object that represents a stateless service.
- An object that represents a conversational session with a particular client. Such session objects automatically maintain their conversational state across multiple client-invoked methods.
- A persistent entity object that is shared among multiple clients.

Although the state management protocol defined by the Enterprise JavaBeans architecture is simple, it provides an enterprise bean developer great flexibility in managing a bean's state.

A client always uses the same API for object creation, lookup, method invocation, and destruction, regardless of how an enterprise bean is implemented, and what function it provides to the client.

4.2 Enterprise JavaBeans contracts

This section describes the Enterprise JavaBeans Release 1.0 contracts.

4.2.1 Client's view contract

This is a contract between a client and a container. The client's view contract provides a uniform development model for applications using enterprise beans as components. This uniform model enables using higher level development tools, and will allow greater reuse of components.

Both the enterprise bean provider and the container provider have obligations to fulfill the contract. This contract includes:

- Object identity.
- Lifecycle operations.
- Method invocation.
- Factory interface.
- Finder interface.

A client expects that an enterprise bean object has a unique identifier. The container provider is responsible for generating a unique identifier for each EJB object. For entity[4.3.2] enterprise beans, the EJB provider is responsible for supplying a unique primary key that the container embeds into the EJB object's identifier.

A client locates an enterprise bean container through the standard Java Naming and Directory Interface™ (JNDI). Within a container, the primary key is used to identify each EJB object.

An enterprise bean and its container cooperate to implement the create, find, and destroy operations callable by clients.

An enterprise bean provider defines a remote interface that defines the business methods callable by a client. The enterprise bean provider is also responsible for writing the implementation of the business methods in the enterprise bean class. The container is responsible for allowing the clients to invoke an enterprise bean through its associated remote interface. The container delegates the invocation of a business method to its implementation in the enterprise bean class.

An enterprise bean provider is responsible for supplying an enterprise bean's factory interface. The factory interface defines one or more *create(...)* methods, one for each way to create the EJB object. The enterprise bean provider is responsible for the imple-

mentation of the *ejbCreate(...)* methods in the enterprise bean class, whose signature must match those of the factory *create(...)* methods. The container provider tools are responsible for generating a class that implements the factory interface and for making the interface available to the enterprise bean's clients through the *getFactory()* method on the *Container* interface. The implementation of the create(...) methods generated by the container provider tools must invoke the matching *ejbCreate(...)* method.

TODO: describe finder

4.2.2 Container contract

This is a contract between an enterprise bean and its container. This contract includes:

- An enterprise bean class instance's view of its lifecycle. For a session enterprise bean, this includes the state management callbacks defined by the *java.ejb.SessionBean* and *java.ejb.SessionSynchronization* interfaces. For an entity enterprise bean, this includes the state management callbacks defined by the *java.ejb.EntityBean* interface. The container invokes the callback methods defined by these interfaces at the appropriate times to notify the instance of the important events in its lifecycle.
- The *java.ejb.SessionContext* interface that a container passes to a session enterprise bean instance at instance creation. The instance uses the *SessionContext* interface to obtain various information and services from its container. Similarly, an entity instance uses the *java.ejb.EntityContext* interface to communicate with its container.
- The environment *java.util.Properties* that a container makes available to an enterprise bean.
  See http://java.sun.com/products/jdk/1.1/docs/api/java.util.Properties.html.
- A list of services that every container must provide for its enterprise beans.

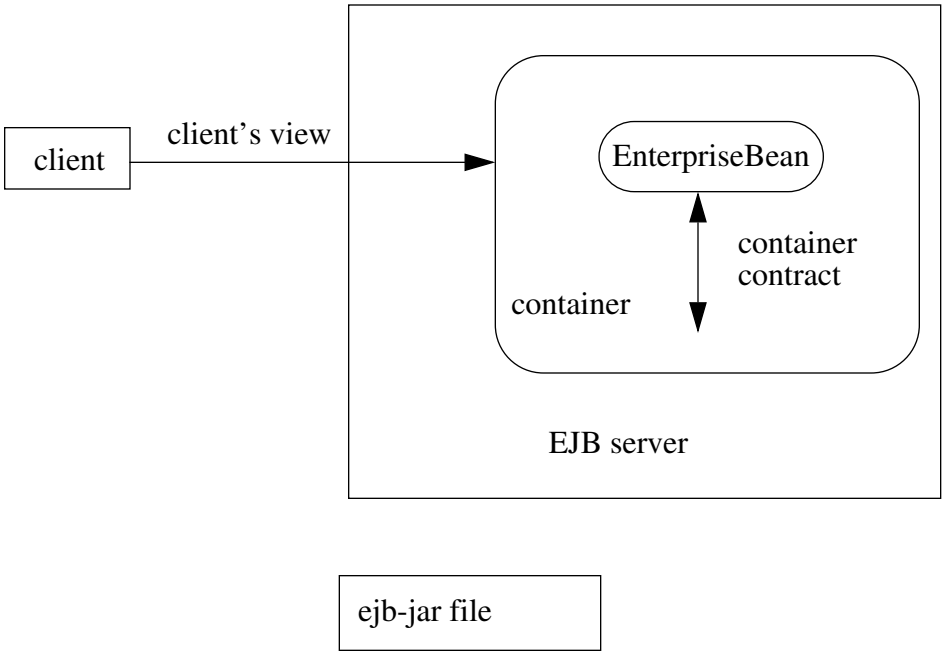
4.2.3 Ejb-jar file

An ejb-jar file is a standard format used by EJB tools for packaging enterprise beans with their declarative deployment information. All EJB tools must support ejb-jar files.

The ejb-jar contract includes:

- JAR file manifest entries that describe the content of the ejb-jar file.
- Java class files for the enterprise beans.
- Enterprise beans' deployment descriptors. A deployment descriptor includes the declarative attributes associated with an enterprise bean. The attributes instruct the container how to manage the enterprise bean objects.
- Enterprise beans' environment properties that the enterprise bean requires at runtime.

The following figure illustrates the Enterprise JavaBeans contracts that are defined in Release 1.0.



Note that while the figure illustrates only a remote client running outside of the container, the client-side API is also applicable to clients who themselves are enterprise beans installed in the same container.

4.3 Session and entity objects

Enterprise JavaBeans 1.0 defines two types of enterprise beans:

- A *session* object type.
- An *entity* object type.

The support for session objects is mandatory for an EJB 1.0 compliant container. The support for entity objects is optional for an EJB 1.0 compliant container, but it will become mandatory for EJB 2.0 compliant containers.

4.3.1 Session objects

A typical session object has the following characteristics:

- Executes on behalf of a single client.
- Can be transaction-aware.
- Updates data in an underlying database.
- Relatively short-lived.

- Is destroyed when the EJB server crashes. The client has to re-establish a new session object to continue computation.
- Does not represent data that should be stored in a database.

A typical EJB server and container provide a scalable runtime environment to execute a large number of session objects concurrently.

4.3.2 Entity objects

A typical entity object has the following characteristics:

- Represents data in the database.
- Transactional.
- Shared access from multiple users.
- Can be long-lived (lives as long as the data in the database).
- Survives crashes of the EJB server. A crash is transparent to the client.

A typical EJB server and container provide a scalable runtime environment for a large number of concurrently active entity objects.

4.4 Standard CORBA mapping

To ensure interoperability for multi-vendor EJB environments, we define a standard mapping of the Enterprise JavaBeans client's view contract to CORBA.

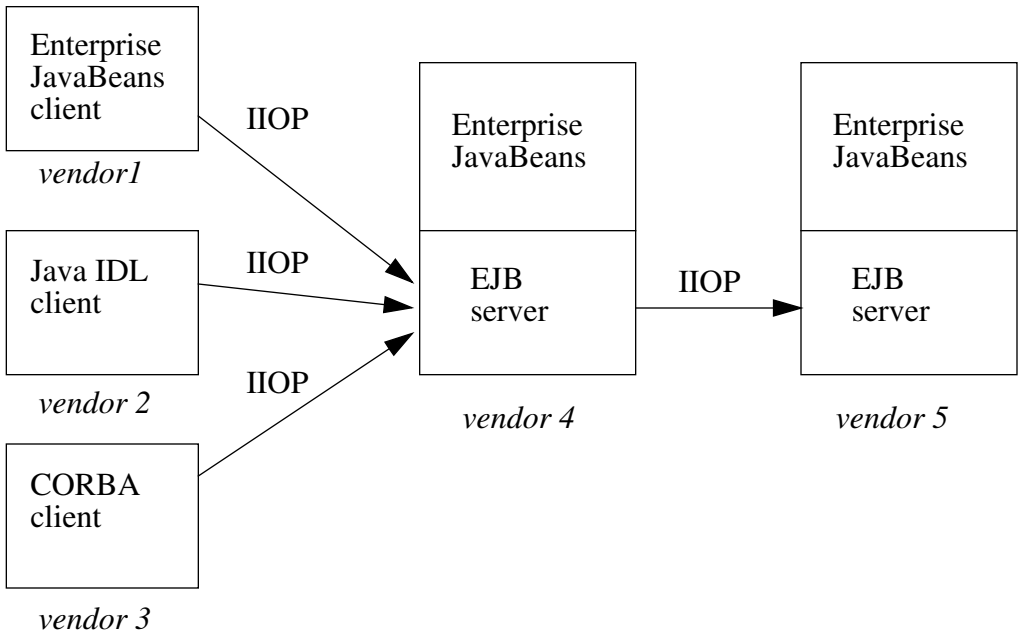
The mapping to CORBA covers:

1. Mapping of the EJB client interfaces to CORBA IDL.
2. Propagation of transaction context.
3. Propagation of security context.

The Enterprise JavaBeans to CORBA mapping not only enables on-the-wire interoperability among multiple vendors' implementations of an EJB server, but it also enables non-Java clients to access server-side applications written as enterprise beans through standard CORBA APIs.

The CORBA mapping is defined in an accompanying document [6].

The following figure illustrates a heterogeneous environment that includes systems from five different vendors.



5 Client view of a session bean

This chapter describes the client's view of a session enterprise bean. The session bean itself implements the bean's business logic. All the functionality for remote access, security, concurrency, transactions, etc. is provided by the bean's container.

5.1 Overview

A client-side programmer accesses an enterprise bean through an EJB object. An EJB object is a remote Java object accessible from a client through the standard Java APIs for remote object invocation [3, 5].

From its creation until destruction, an EJB object lives in a container. Transparently to the client, the container provides security, concurrency, transactions, persistence, and other services for the EJB object.

Each session EJB object has an identity which, in general, *does not* survive a crash and restart of the container.

The client's view of an EJB object is location-independent. A client running in the same JVM as the EJB object uses the same API as a client running in a different JVM on the same or different machine.

The client's view of an EJB object is the same, irrespective of the implementation of the enterprise bean and its container.

5.2 EJB container

An EJB container (container for short) is an object that functions as the "container" for enterprise beans. A container is where an enterprise bean object lives, just as a record lives in a database, and a file or directory lives in a file system.

5.2.1 Locating a container

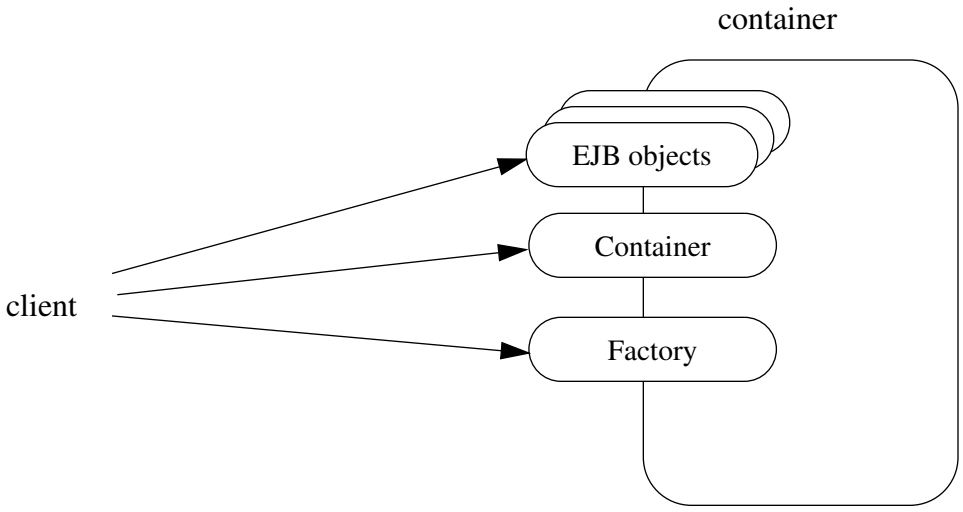
A client locates a container using JNDI. For example, a container for *Cart* EJB objects can be located using the following code segment:

```
Context initialContext = new InitialContext();
Container cartContainer = (Container)
        initialContext.lookup("applications/mall/freds-carts");
```

A client's JNDI name space may be configured to include EJB containers located on multiple machines on a network. The actual location of an EJB container is, in general, transparent to the client.

5.2.2 What a container provides

The following diagram illustrates the services that an session container provides to its clients.



5.2.3 Container interface

An EJB container implements the *java.ejb.Container* interface. The *java.ejb.Container* interface allows a client to do the following:

- Obtain a factory object that allows a client to create new EJB objects in the container.
- Destroy an EJB object.
- Get metadata describing the container's EJB object type.

5.2.4 Enterprise bean's factory

An enterprise bean's factory interface is an interface that allows a client to create new EJB objects in a container. A client obtains a factory interface using a container's *get-Factory* method.

A factory interface defines one or more *create(...)* methods, one for each way to create the EJB object. The arguments of the *create* methods are typically used to initialize the state of the created EJB object.

An enterprise bean's factory interface must extend the *java.ejb.Factory* interface. The following is an example of a factory interface:

```
public interface CartFactory extends java.ejb.Factory {
        public Cart create(String customerName, String account)
                    throws RemoteException;
}
```

The following example illustrates how a client obtains and uses a factory object:

```
CartFactory cartFactory = (CartFactory)
        cartContainer.getFactory();
cartFactory.create("John", "7506");
```

5.2.5 Destroying an EJB object

The container defines several methods that allow a client to destroy an EJB object.

5.2.6 A container's metadata

TODO

5.3 EJB Object

A client never accesses an EJB object directly.

An EJB object supports:

- The business logic methods of the object.
- The *java.ejb.EJBObject* interface. *EJBObject* includes methods that allow the client to
  - get the EJB object's container.
  - get the EJB object's handle.
  - test to see if another EJB object is identical.
  - destroy the EJB object.

The implementation of the methods defined in the *java.ejb.EJBObject* interface is provided by the container. The business methods are delegated to the particular EJB object implementation.

5.4 Session object identity

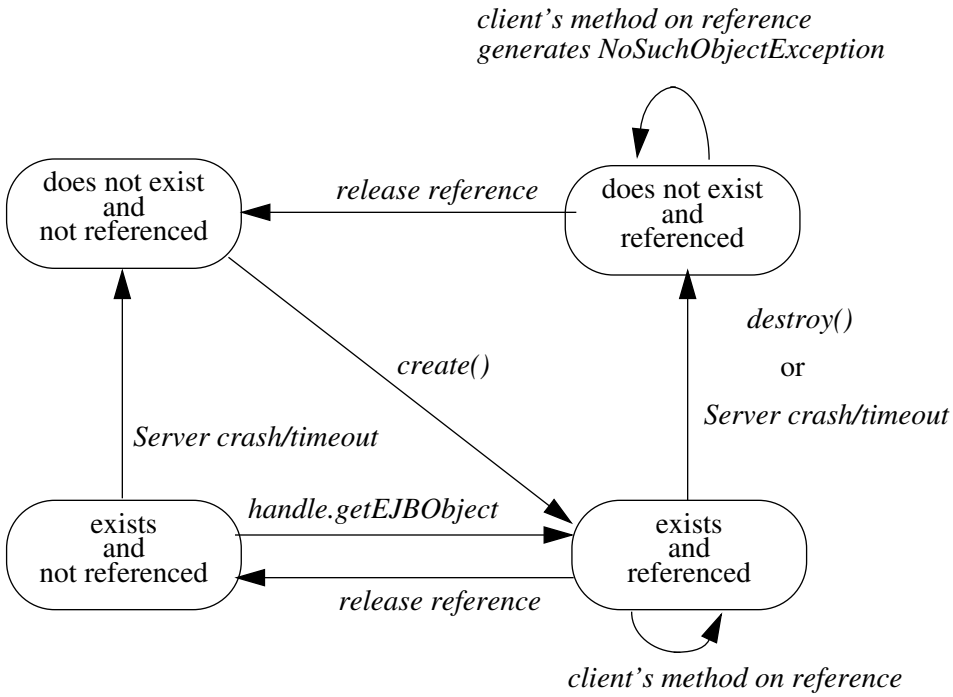
Session objects are intended to be private resources used only by the client that created them. For this reason, session EJB objects, from the clients perspective, appear anonymous. In contrast to entity EJB objects which expose their identity as a primary key, session objects hide their identity.

Since all session objects hide their identity, there is no need to provide a finder for them, so all their *Container.getFinder* methods return null.

A session EJB object handle can be held beyond the life of a client process by serializing the handle to persistent store. When the handle is later deserialized, the session EJB object it returns will work as long as the object still exists on the server (an earlier timeout or server crash may have destroyed it).

5.5 Client's view of session bean's lifecycle

From a client point of view, the lifecycle of a session bean object is illustrated below.



An object does not exist until it is created. When an object is created by a client, the client gets a reference to the newly created EJB object.

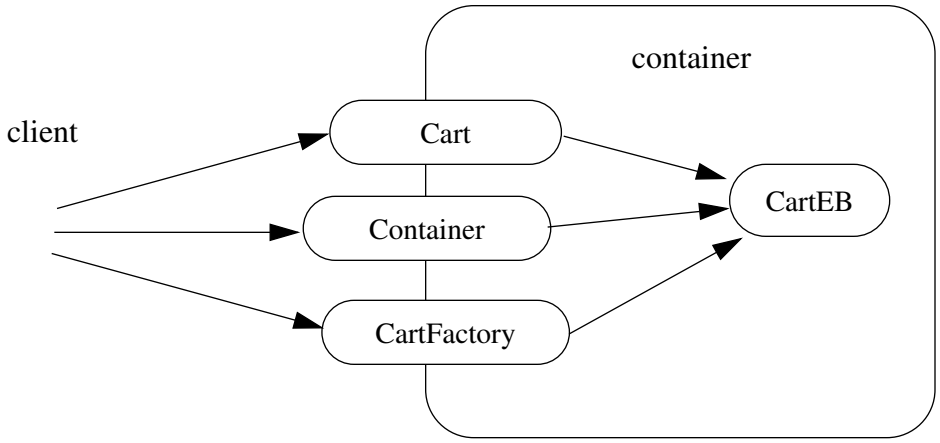
A client that has a reference to an object can then do any of the following:

- Invoke application methods on the object through the session bean's remote interface.
- Get a reference to the object's container.
- Get a handle for the object
- Pass the object as a parameter or return value within the scope of the client.
- Destroy the object. A container may also destroy the object when the object's lifetime expires.

References to an EJB object that does not exist are invalid. Attempted invocations on an object that does not exist will throw a *java.ejb.NoSuchObjectException*.

5.6 Creating and using a session bean

An example of the session bean runtime objects is illustrated by the following diagram:



A client creates a Cart session object (which provides a shopping service) using the cart Container's CartFactory. The client then uses this object to fill the cart with items and to purchase its contents.

If the client wishes to start his shopping session on his work machine and later complete this session from his home machine, this can be done by getting the session's handle, sending the serialized handle to his home, and using it to reestablish access to the original Cart.

For the following example, we start off by looking up a cart Container in JNDI. We then use this container to create a *Cart* EJBObject and add a few items to it:

```
Container store = (Container) lookup...
CartFactory cartFactory = store.getFactory();
Cart cart = cartFactory.create();
cart.addItem(66);
cart.addItem(22);
```

Next we decide to complete this shopping session at home so we serialize a handle to this cart session and mail it home:

```
Handle cartHandle = cart.gethandle;
serialize cartHandle, attach to message and mail it home...
```

Finally we deserialize the handle at home and purchase the content of the shopping cart:

```
Handle cartHandle = deserialize from mail attachment...
Cart cart = (Cart) cartHandle.getEJBObject();
cart.purchase();
cart.destroy();
```

6 Session bean state management

The state management of a session bean instance is part of the contract between a session bean and its container. It defines the lifecycle of a session bean instance.

This chapter defines the developer's view of session bean state management and the container's responsibility for managing it.

6.1 Overview

By definition, a session bean instance is an extension of the client that creates it:

- Its fields contain *conversational state* on behalf of the client. This state describes the conversation represented by a specific client/instance pair.
- It typically reads and updates data in a database on behalf of the client. Within a transaction, some of this data may be cached in the bean.
- Its lifetime is typically that of its client.

  A session bean instance's life may also be terminated by a container-specified timeout or the failure of the server it is running on. For this reason, a client must always be prepared to recreate a new instance if it loses the one it's using.

Typically, a session bean's conversational state is not written to the database. A bean developer simply stores it in the bean's fields and assumes its value is retained for the lifetime of the bean.

On the other hand, cached database data must be explicitly managed by the bean. A bean must write any database updates it is caching prior to the bean's transaction completion, and it must refresh any potentially stale database data it contains at the beginning of the next transaction.

6.2 Goals

The goal of the session bean model is to make developing a session bean as simple as developing the same functionality directly in a client.

The session bean container manages the lifecycle of the session bean, notifying it when bean action may be necessary, and providing a full range of services to ensure the bean implementation scales to support a large number of clients.

The remainder of this section describes the session bean lifecycle in detail and the protocol between the bean and its container.

6.3 A container's management of its working set

In order to efficiently manage the size of its working set, a session bean container may need to temporarily transfer the state of an idle session bean to some form of secondary storage. The transfer from the working set to secondary storage is called *passivation*. The transfer back is called *activation*.

A container may only passivate a session bean when that bean is *not* in a transaction.

In order to help its container manage its state, a session bean is specified at deployment as having one of the following state management modes:

- STATELESS - the bean contains no conversational state between methods; any bean instance can be used for any client.
- STATEFUL - the bean contains conversational state which must be retained across transactions.
- PINNED - the bean must not be passivated by its container; it contains conversational state that would be lost if it were passivated.

  The PINNED *style should be used with care. Since it prevents a container from passivating a session bean, it may not scale as well as the other styles.*

6.4 Conversational state

A STATEFUL session bean's conversational state is defined as its field values plus the transitive closure of the objects reachable from the session bean's fields. The transitive closure is defined in terms of the standard Java Serialization protocol—the fields that would be stored by serializing the enterprise bean instance are considered part of the enterprise bean state.

While a session container is not required to use the Java Serialization protocol to store the state of passivated beans, it must achieve the equivalent result. The one exception is that containers are not required to reset the value of *transient* fields.

In advanced cases, a session bean's conversational state may contain open resources. Examples of this are open files, open sockets and open database cursors. It is not possible for a container to retain open resources while a session bean is passivated. A developer of such a session bean must either designate the bean as PINNED, or close and open the resources using the *ejbPassivate* and *ejbActivate* notifications.

6.4.1 The effect of transaction rollback on conversational state

A session bean's conversational state is not transactional. It is not automatically rolled back to its initial state if the bean's transaction rolls back.

If a rollback could result in a session bean's conversational state being inconsistent, the bean developer must use the *afterCompletion* notification to manually reset its state.

6.5 The protocol between a session bean and its container

This section describes the interfaces that define the services a container provides to a session bean.

Containers themselves make no actual service demands on their session beans. The calls a container makes on a bean provide it with access to container services and deliver notifications issued by the container.

6.5.1 The required *SessionBean* interface

All session beans must implement the *SessionBean* interface.

The *setSessionContext* method is called by the bean's container to associate a session bean instance with its *Container* and *EJBObject*. Typically a session bean retains its session context as part of its conversational state.

The *ejbDestroy* notification signals that the instance is in the process of being destroyed by the container. Since most session beans don't have any database or resource state to clean up, this method is typically left empty.

The *ejbPassivate* notification signals the intent of the container to passivate the instance. The *ejbActivate* notification signals the instance it has just been reactivated. Since containers automatically maintain the conversational state of a session bean instance while it is passivated, most session beans can ignore these notifications. Their purpose is to allow advanced beans to maintain open resources that need to be closed prior to an instance's passivation and reopened when it is reactivated.

6.5.2 The container's *SessionContext* interface

All bean containers provide their bean instances with a *SessionContext*. This gives the bean instance access to its *Container* and its *EJBObject*.

The *getEJBObject* method returns the bean instance's EJB object.

The *getEnvironment* method returns the environment properties list the bean was deployed with.

The *getCallerIdentity* method returns the identity of the current invoker of the bean instance's EJB object.

The *isCallerInRole* predicate tests if the immediate caller has a particular role.

6.5.3 The optional *SessionSynchronization* interface

Most session beans will implement the *SessionSynchronization* interface. It provides the bean with notifications for transaction synchronization. Session beans use these notifications to manage database data they may cache within transactions.

The *beginTransaction* notification signals a session instance that a new transaction has begun. At this point, the instance is already in the transaction and may do any database work it requires.

The *beforeCompletion* notification is issued when a session instance's client has completed work on its current transaction but prior to committing the instance's resources. This is the time when the instance should write out any database updates it has cached.

The *afterCompletion* notification signals that the current transaction has completed. A completion status of *true* indicates the transaction committed; a status of *false* indicates a rollback occurred. Since a session instance's conversational state is not transactional, it may need to manually reset its state if a rollback occurred.

6.5.4 Business method delegation

TODO: describe how the container delegates a business method

6.5.5 Session bean factory

Session beans go through a two step creation process. First, a container calls the bean class' *newInstance* method to create a bean instance. Second, it calls one of the instance's *ejbCreate* methods.

Each session bean must have at least one *ejbCreate* method. The number and signatures of a session bean's *create* methods are open-ended.

A client creates a session bean instance using a factory object obtained from the bean's container. The bean's factory interface is provided by the bean developer; its implementation is generated by the bean's container. It has a set of *create* methods that duplicate the signatures of the bean's *ejbCreate* methods.

When a client calls a factory *create* method, the container creates a bean instance and then calls the bean's corresponding *ejbCreate* method, passing the original parameter values.

Since a session bean represents a specific, private conversation between the bean and its client, its create parameters typically contain the information the client uses to personalize the bean instance for its use.

6.5.6 Serializing session bean methods

A container serializes calls to each of its bean instances. Most containers will support many instances of a bean executing concurrently; however, each instance sees only a serialized sequence of method calls.

The method calls a container serializes includes those delivered via an instance's EJB object as well as the service calls made by the container itself.

6.5.7 Transaction context of session bean methods

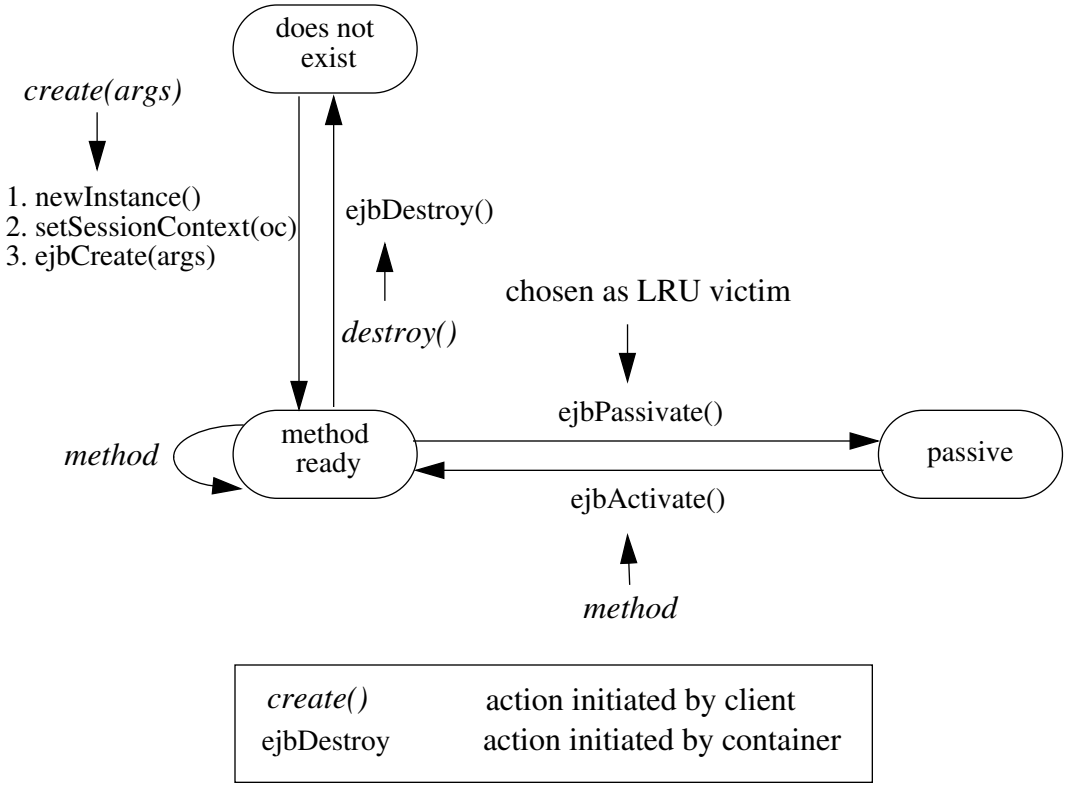
A session bean's *beginTransaction* and *beforeCompletion* methods are always called within a transaction.

A session bean's *setSessionContext*, *ejbCreate*, *ejbDestroy*, *ejbPassivate*, *ejbActivate* and *afterCompletion* methods are always called without a transaction. So, for instance, it would usually be wrong to make database updates within a session bean's *ejbCreate* or *ejbDestroy* methods.

A session bean's deployment descriptor determines whether or not its business methods are called with a transaction.

6.6 Non-transactional session bean state diagram

The following figure illustrates the lifecycle of a non-transactional session bean instance.



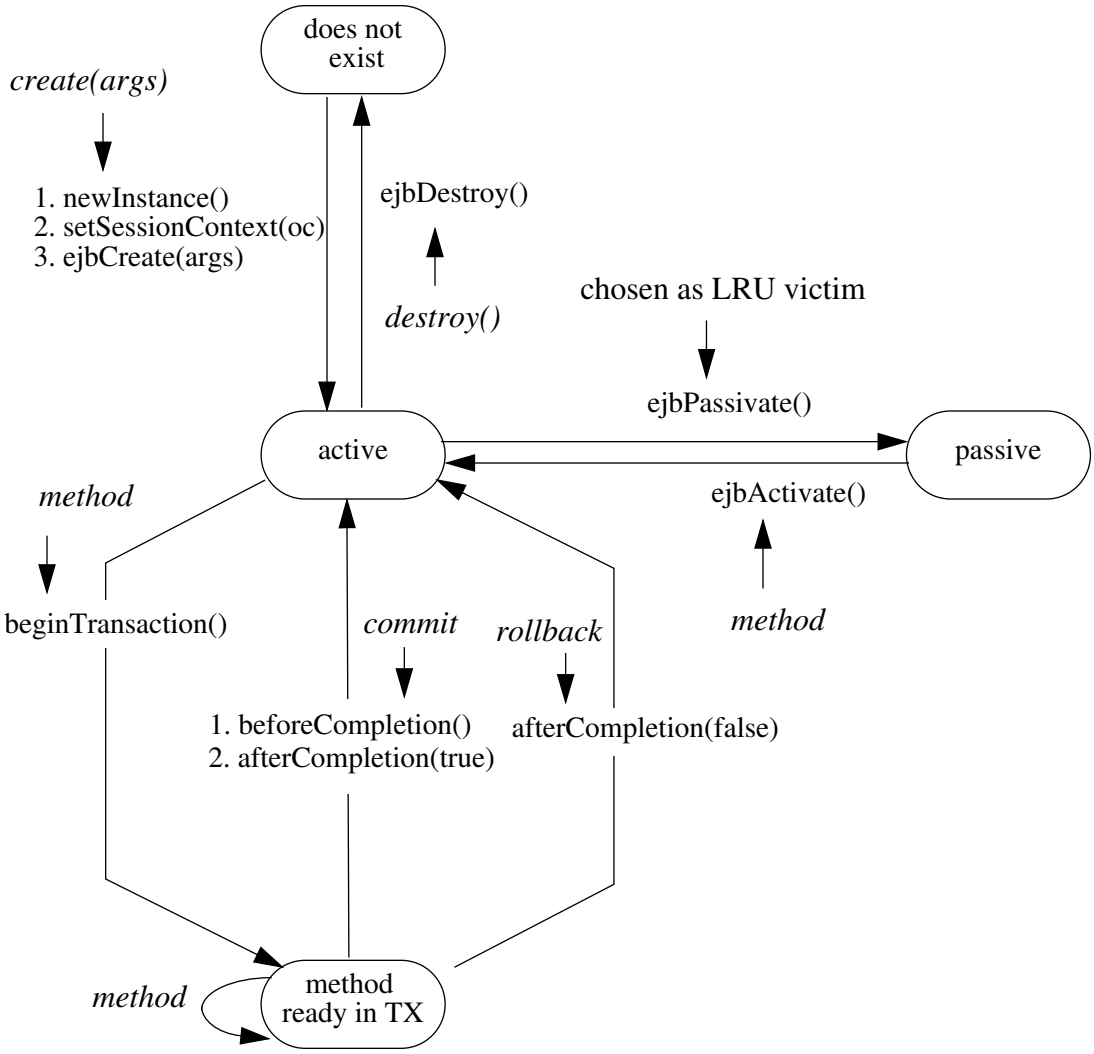
The following is a walk-through the lifecyle of a session bean instance:

- A non-transactional session bean's life starts when a client invokes a c*reate(...)* method on its container's factory. This causes the container to invoke *newInstance()* on the bean class to create a new memory object for the enterprise bean. Next, the container calls *setSessionContext()* followed by *ejbCreate(...)* on the instance, and returns an EJB object to the client.
- The bean instance is now ready for client methods.
- The container's caching algorithm may decide that the bean instance should be evicted from memory (this could be done at the end of each method, or by using an LRU policy). If the bean instance is STATELESS, the container can simply let the instance be garbage collected. If it is PINNED, the container cannot evict the instance. If it is STATEFUL, the container will issue *ejbPassivate()* on the instance. After this completes, the container must save the instance's state to secondary storage.

- If a client invokes a method on this passivated instance (only STATEFUL beans are passivated), the container activates the session instance prior to delegating the method invocation to it. To activate the session instance, the container restores the instance's state from secondary storage and issues *ejbActivate()* on it.
- The enterprise bean is again ready for client methods.
- When the client calls *destroy()* on the EJB object, this causes the container to issue *ejbDestroy()* on the bean instance. This ends the life of the session bean instance. Any subsequent attempt by its client to invoke the instance will result in throwing the *java.rmi.NoSuchObjectException*. Note that a container can implicitly call the *destroy()* method on the instance after the lifetime of the EJB object has expired.

6.7 Transactional session bean state diagram

The following figure illustrates the lifecycle of a transactional session bean instance.



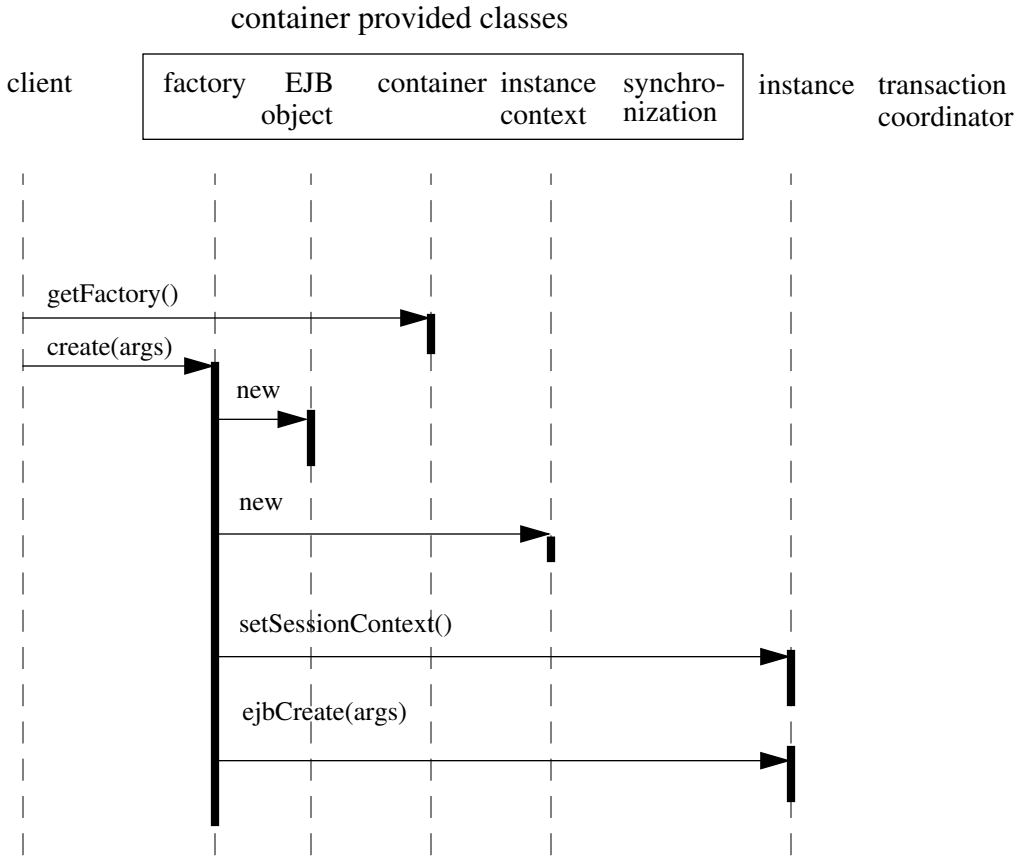
The following is a walk-through of the lifecycle of a transactional session bean instance:

- A transactional session bean's life starts when a client invokes a c*reate(...)* method on its container's factory. This causes the container to invoke *newInstance()* on the bean class to create a new memory object for the enterprise bean. Next, the container calls *setSessionContext()* followed by *ejbCreate(...)* on the instance, and returns an EJB object to the client.
- The bean instance is now ready to be included in a transaction.
- After the bean instance is included in a transaction and before any of its other methods are executed within the transaction, the container issues *beginTransaction* on it.
- Bean methods invoked by the client in this transaction can now be delegated to the bean instance.
- If a transaction commit has been requested, prior to actually committing the transaction, the container issues *beforeCompletion* on the instance. This is when the instance should write any cached updates to the database.
- The container then attempts to commit the transaction, resulting in either a commit or rollback. If, in the previous step, transaction rollback had been requested, rollback status is reached without issuing *beforeCompletion*.
- When the transaction completes, the container issues *afterCompletion* on the instance, specifying the status of the completion (commit or rollback). If a rollback occurred, the bean instance may need to reset its conversational state back to the value it had at the beginning of the transaction.
- The container's caching algorithm may decide that the bean instance should be evicted from memory (this could be done at the end of each method, or by using an LRU policy). If the bean instance is STATELESS, the container can simply let the instance be garbage collected. If it is PINNED, the container cannot evict the instance. If it is STATEFUL, the container issues *ejbPassivate()* on the instance. After this completes, the container must save the instance's state to secondary storage.
- If this passivated instance's EJB object is included in a transaction (only STATEFUL beans are passivated), the container will activate the session instance. To activate the session instance, the container restores the instance's state from secondary storage and issues *ejbActivate()* on it.
- The enterprise bean is again ready to be included in a transaction.
- When the client calls *destroy()* on the EJB object, this causes the container to issue *ejbDestroy()* on the bean instance. This ends the life of the session bean instance. Any subsequent attempt by its client to invoke the instance will result in throwing the *java.rmi.NoSuchObjectException*. Note that a container can implicitly invoke the *destroy()* method on the instance after the lifetime of the EJB object has expired.

6.8 Sequence diagrams

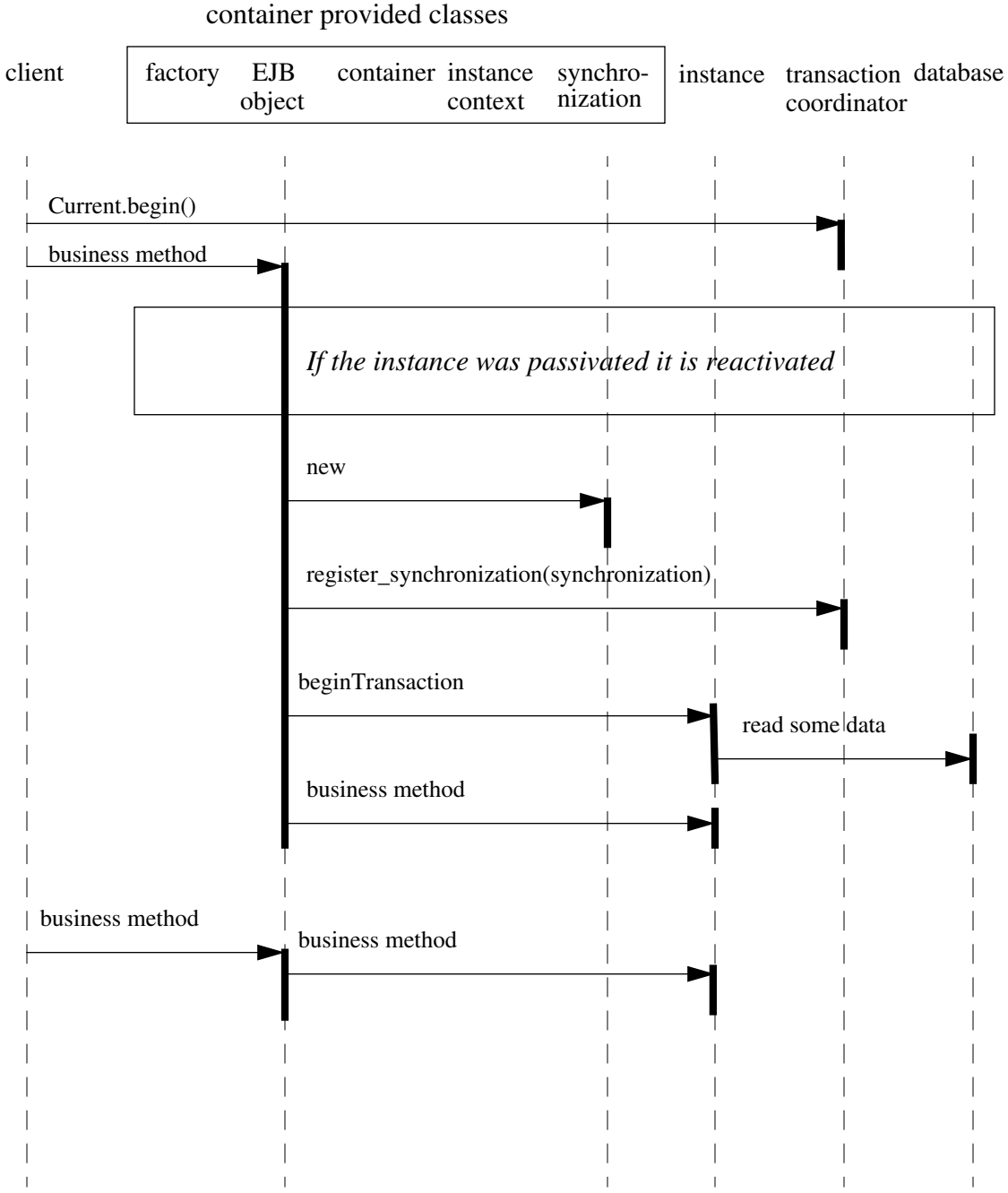
6.8.1 Creating a session object

The following diagram illustrates the creation of a transactional session enterprise bean.



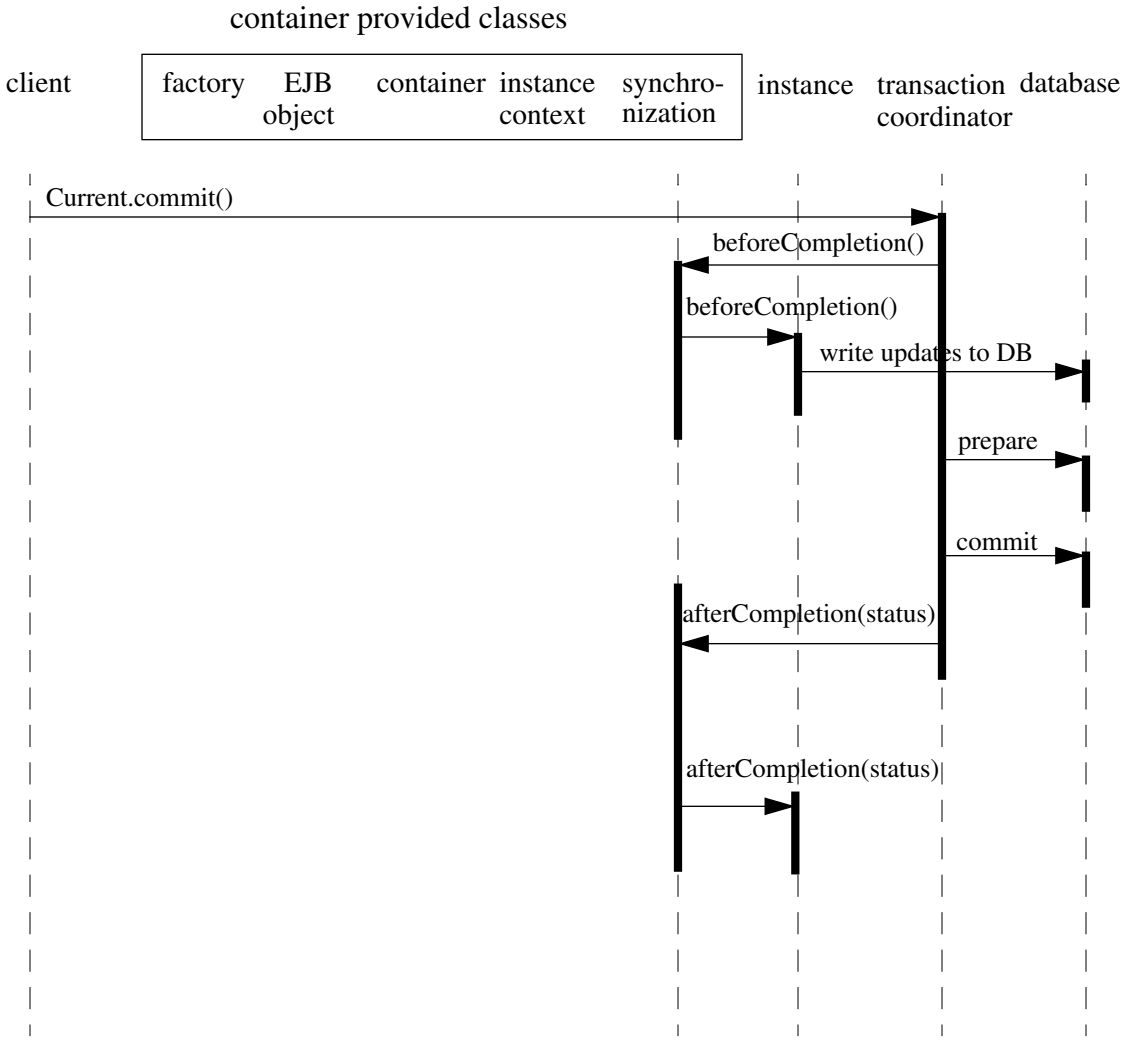
6.8.2 Starting a transaction

The following diagram illustrates the protocol performed at the beginning of a transaction.



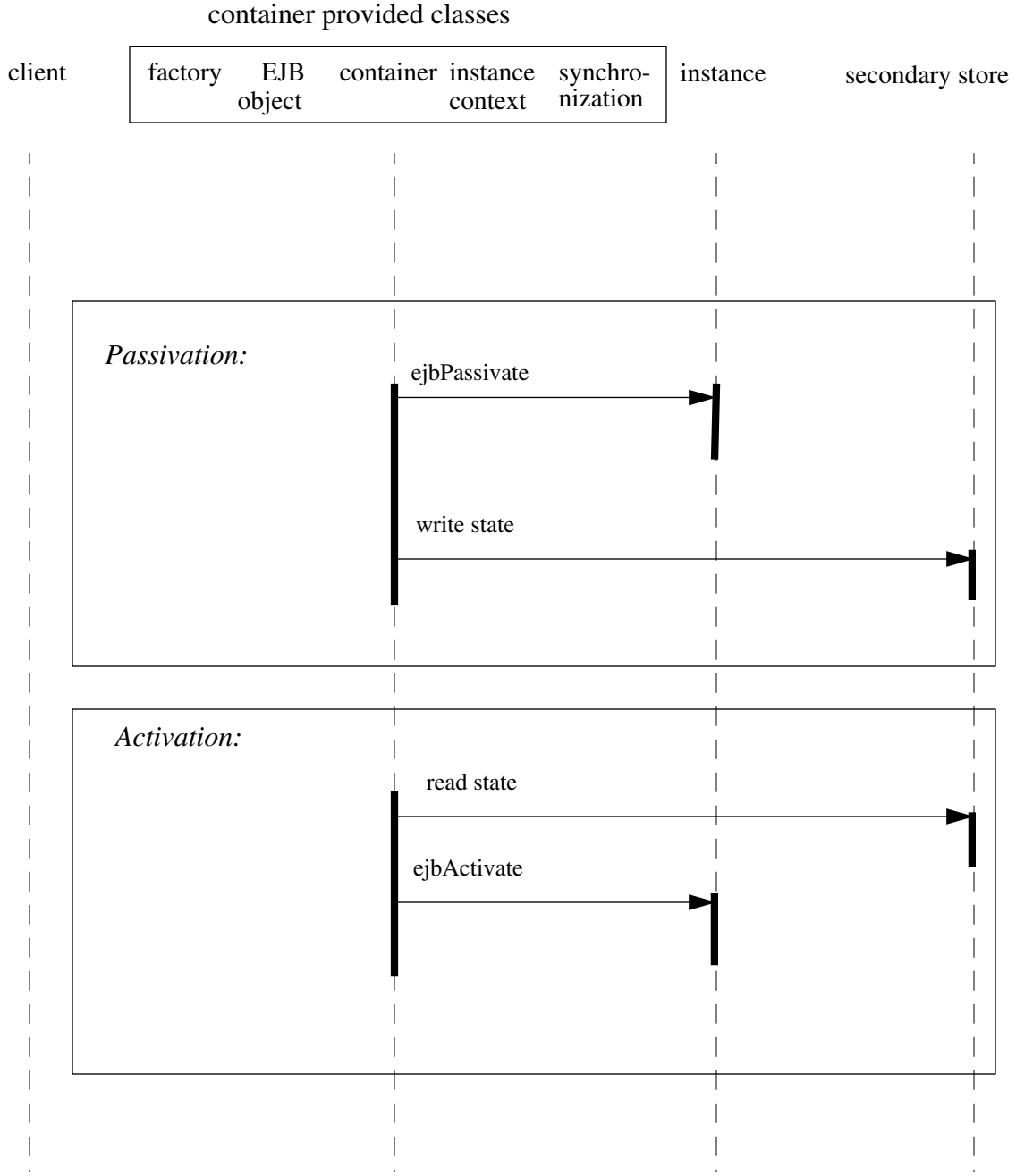
6.8.3 Committing a transaction

The following diagram illustrates the transaction synchronization protocol for a session enterprise bean instance.



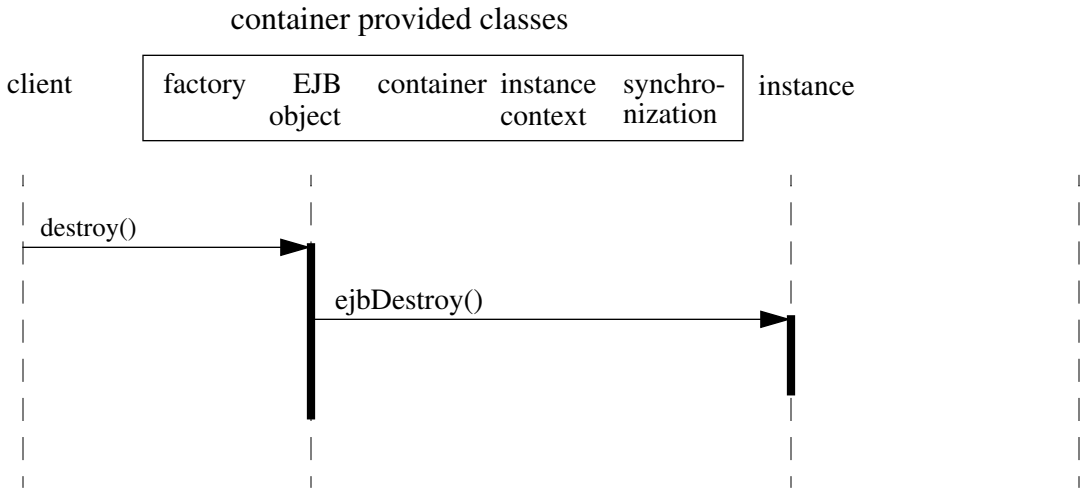
6.8.4 Passivating and activating an instance between transactions

The following diagram illustrates the passivation and reactivation of a session enterprise bean instance. Passivation typically happens spontaneously based on the needs of the container. Activation typically occurs when a client calls a method.



6.8.5 Destroying a session object

The following diagram illustrates the destruction of a session bean.



7 Example session scenario

This chapter describes an example development and deployment scenario of a session enterprise bean. We use the scenario to explain the responsibilities of the enterprise bean provider and those of the container provider.

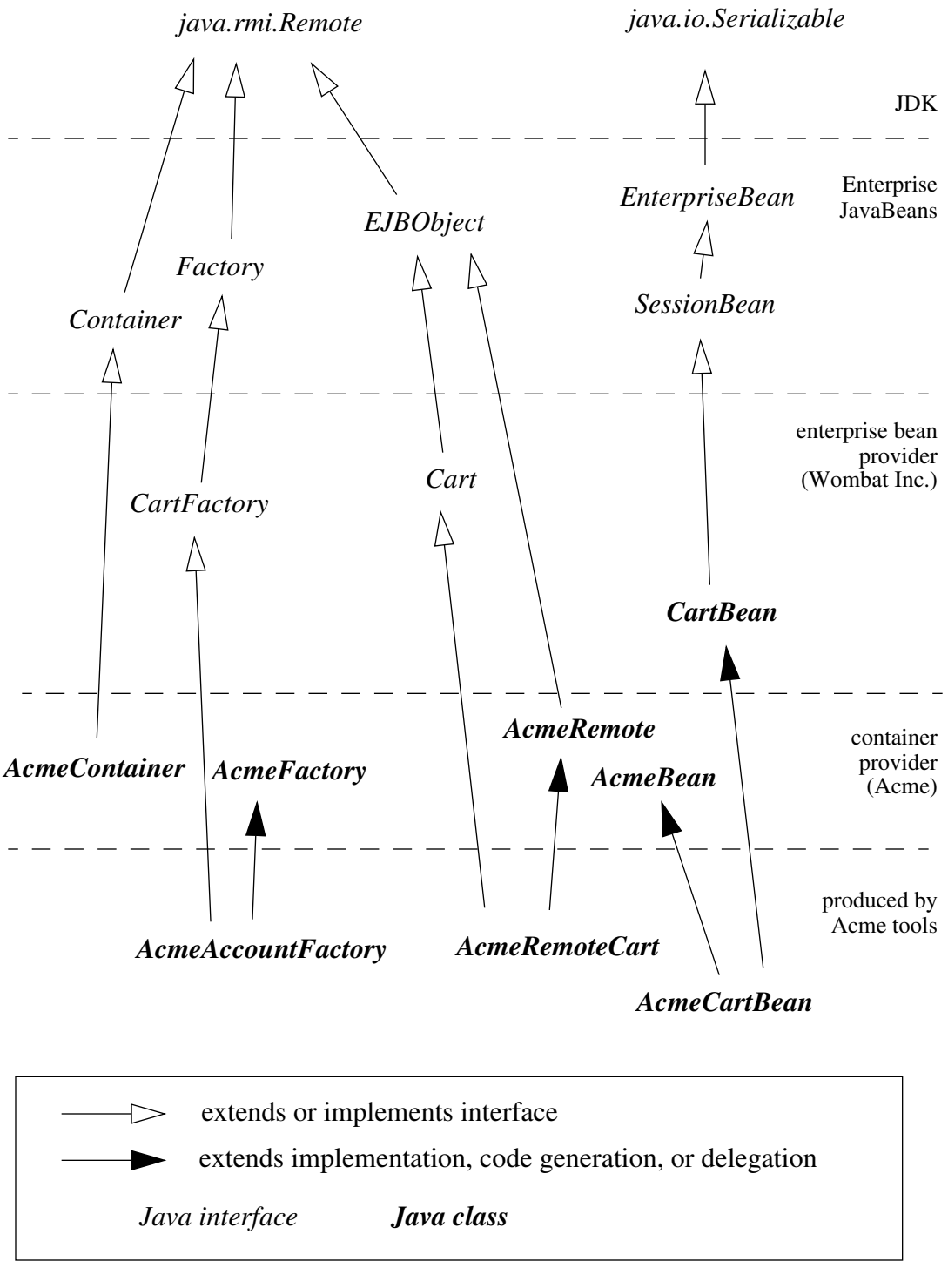
The classes generated by the container provider's tools in this scenario should be considered illustrative rather than prescriptive. Container providers are free to implement the contract between a session enterprise bean and its container in a different way that achieves an equivalent effect (from the perspectives of the enterprise bean provider and the client-side programmer).

7.1 Overview

Wombat Inc. has developed the *CartBean* session bean. The CartBean is deployed in a container provided by the Acme Corporation.

7.2 Inheritance relationship

An example of the inheritance relationship between the interfaces and classes is illustrated in the following diagram:



7.2.1 What the session bean provider is responsible for

Wombat Inc. is responsible for providing the following:

- Define the session bean's remote interface (Cart). The remote interface defines the business methods callable by a client. The remote interface must extend the *java.ejb.EJBObject* interface, and follow the standard rules for a Java RMI remote interface. The remote interface must be defined as *public*.
- Write the business logic in the session bean class (CartBean). The enterprise bean class must not implement the enterprise bean's remote interface (Cart). The enterprise bean must implement the *java.ejb.SessionBean* interface, and define the *ejbCreate(...)* methods invoked at an EJB object creation. The *ejbCreate(...)* methods must follow the factory design pattern[9.6].
- Define a factory interface (CartFactory) for the enterprise bean. The signatures of the methods of the factory interface must follow the factory design pattern[9.6]. The factory interface must be defined as *public*, extend the *java.ejb.Factory* interface, and follow the standard rules for Java RMI remote interfaces.
- Specify the environment properties that the session bean needs at runtime. The environment properties is a standard *java.util.Properties* file.
- Define a deployment descriptor that specifies any declarative metadata that the session bean provider wishes to pass with the bean to the next stage of the development/deployment workflow.

7.2.2 Classes supplied by container provider

The following classes are supplied by the container provider Acme Corp:

- The AcmeContainer class provides the Acme implementation of the *java.ejb.Container* interface.
- The AcmeFactory class provides the Acme implementation of a factory base class.
- The AcmeRemote class provides the Acme implementation of the *java.ejb.EJBObject* methods.
- The AcmeBean class provides additional state and methods to allow Acme's container to manage its session bean instances. For example, if Acme's container uses an LRU algorithm, then AcmeBean may include the clock count and methods to use it.

7.2.3 What the container provider is responsible for

The tools provided by Acme Corporation are responsible for the following:

- Generate the remote bean class (AcmeRemoteCart) for the session bean. The remote bean class is a "wrapper" class for the enterprise bean and provides the client's view of the enterprise bean. The tools also generate the classes that implement the communication stub and skeleton for the remote bean class.

- Generate the implementation of the session bean class suitable for the Acme container (AcmeCartBean). AcmeCartBean includes the business logic from the CartBean class mixed with the services defined in the AcmeBean class. Acme tools can use inheritance, delegation, and code generation to achieve a mix-in of the two classes.
- Generate the class for the session bean's factory interface (AcmeCartFactory). The tools also generate the classes that implement the communication stub and skeleton for the factory class.

Many of the above classes and tools are container-specific (i.e., they reflect the way Acme Corp implemented them). Other container providers may use different mechanisms to produce their runtime classes, and the generated classes most likely will be different from those generated by Acme's tools.

8 Client view of an entity

Note: Container support for entity enterprise beans is an optional feature for EJB 1.0 compliance. Container support for entity enterprise beans will become mandatory in EJB 2.0.

This chapter describes the client's view of an entity EJB object. It is actually a contract fulfilled by an enterprise bean's container, with only the business part supplied by the enterprise bean itself.

We are considering whether to extend the Container interface to support multiple enterprise bean classes. This feature would be desirable for supporting evolution of entity objects.

8.1 Overview

A client-side programmer accesses an enterprise bean through an EJB object. An EJB object is a remote Java object accessible from a client through the standard Java APIs for remote object invocation [3, 5].

From its creation until destruction, an EJB object lives in a container. Transparently to the client, the container provides security, concurrency, transactions, persistence, and other services for the EJB object.

Each entity object has an identity which, in general, survives a crash and restart of the container in which the entity object has been created. The object identity is implemented by the container.

The client's view of an EJB object is location independent. A client running in the same JVM as the EJB object uses the same API as a client running in a different JVM on the same or different machine.

The client's view of an EJB object is the same, irrespective of the implementation of the enterprise bean and its container.

8.2 EJB container

An EJB container (container for short) is an object that functions as the "container" for enterprise beans. A container is where an enterprise bean object lives, just as a record lives in a database, and a file or directory lives in a file system.

8.2.1 Locating a container

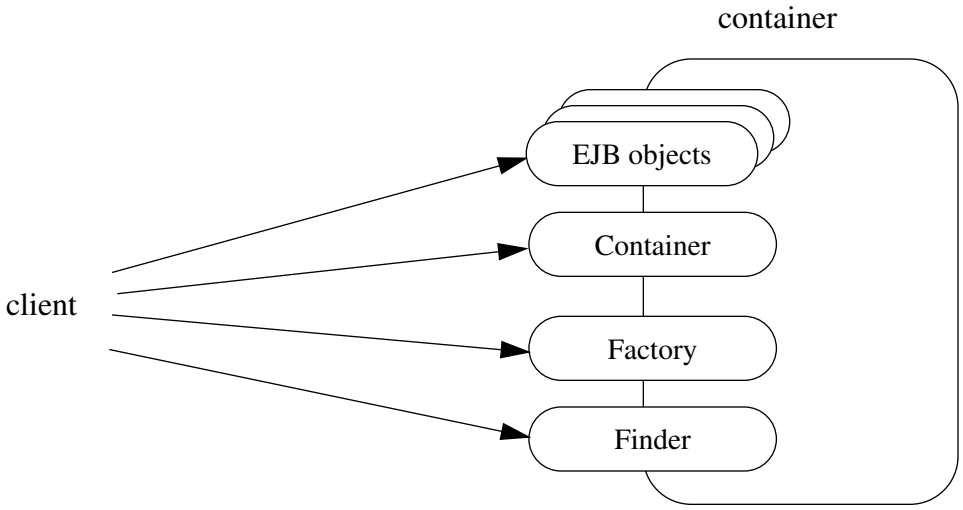
A client locates a container using JNDI. For example, a container for account EJB objects can be located using the following code segment:

```
Context initialContext = new InitialContext();
Container accountContainer = (Container)
        initialContext.lookup("applications/bank/accounts");
```

A client's JNDI name space may be configured to include EJB containers located on multiple machines on a network. The actual location of an EJB container is, in general, transparent to the client.

8.2.2 What a container provides

The following diagram illustrates the services that an entity container provides to its clients.



8.2.3 Container interface

An EJB container implements the *java.ejb.Container* interface. The *java.ejb.Container* interface allows a client to do the following:

- Obtain a factory object that allows a client to create new EJB objects in the container.
- Obtain a finder object that allows a client to look up existing EJB objects in the container.
- Destroy an EJB object.

8.2.4 Enterprise bean's factory

An enterprise bean's factory is an object that allows a client to create new EJB objects in a container. A client obtains a factory object using a container's *getFactory* method.

A factory interface defines one or more *create(...)* methods, one for each way to create the EJB object. The arguments of the *create* methods are typically used to initialize the state of the created EJB object.

An enterprise bean's factory interface must extend the *java.ejb.Factory* interface. The following is an example of a factory interface:

```
public interface AccountFactory extends java.ejb.Factory {
        public Account create(String firstName, String lastName,
                  double initialBalance) throws RemoteException;
        public Account create(String accountNumber,
                  initialBalance) throws RemoteException;
}
```

The following example illustrates how a client obtains and uses a factory interface:

```
AccountFactory accountFactory = (AccountFactory)
        accountContainer.getFactory();
accountFactory.create("John", "Smith, 500.00);
```

8.2.5 Enterprise bean's finder

TODO: describe how a client uses a finder

An enterprise bean's finder interface must extend the *java.ejb.Finder* interface.

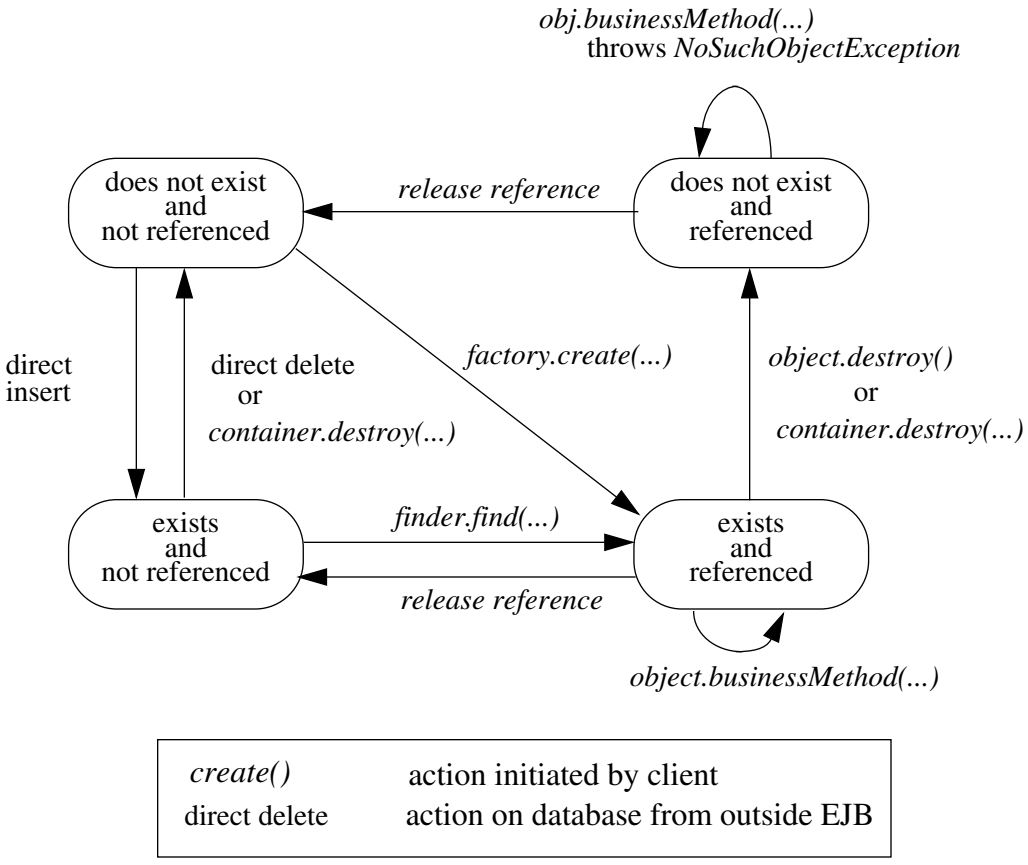
8.2.6 Destroying an EJB object

The container defines several methods that allow a client to destroy an EJB object.

8.3 Entity EJB object lifecycle

This section describes the lifecycle of an EJB object from the perspective of a client.

The following diagram illustrates a client's point of view of an entity EJB object lifecycle (the term *referenced* in the diagram means that the client program has a reference of the EJB object).



An EJB object does not exist until it is created. Until it is created, it has no identity. After it is created, it has identity. A client creates an EJB object using a factory object that is provided by the container. When an EJB object is created by a client, the client obtains a reference to the newly created EJB object.

In an environment with a legacy data, EJB objects may "exist" before the container is deployed. In addition, an entity EJB object may be "created" in the environment via a mechanism other than by invoking a factory *create(...)* method (e.g. by inserting a database record), but still may be accessible by a container's clients via the finder interface. Also, an EJB object may be deleted directly using other means than the *destroy()* operation (e.g. by deletion of a database record). The "direct insert" and "direct delete" transitions in the diagram represent such direct database manipulation.

For an existing EJB object, a client can get a reference to an EJB object in any of the following ways:

- Receive a reference as a parameter in a method call (input parameter or result).

- Look up the EJB object using a container's finder interface.
- Obtain the reference from a bean's handle (handles are described later in Section 8.6).

A client that has a reference to an object can then do any of the following:

- Invoke business methods on the object through the EJB object's remote interface.
- Obtain a reference to the object's container.
- Obtain the name of the enterprise bean class that provides the implementation of the business methods.
- Pass the reference as a parameter or return value.
- Obtain the EJB object's primary key.
- Obtain the EJB object's handle.
- Destroy the EJB object.

All references to an object that does not exist are invalid. All attempted invocations on an object that does not exist will result in an *java.rmi.NoSuchObjectException* being thrown.

All entity EJB objects are considered *persistent objects*. The lifetime of an entity EJB object is not limited by the lifetime of the Java Virtual Machine process in which it executes. A crash of the Java Virtual Machine may result in a rollback of current transactions, but does not destroy previously created EJB entity objects, or invalidate their references held by clients.

8.4 Primary key and object identity

Every entity EJB object has a unique identity within its container. The object's identity relative to its container is determined by the EJB object's primary key.

Enterprise JavaBeans allows a primary key object to be any *java.io.Serializable* class. The primary key class is specific to an enterprise bean class (i.e. each enterprise bean class may have a difference class for its primary key).

A client that holds a reference to an EJB object can determine the object's identity by invoking the *getPrimaryKey()* method on the reference.

A client can test whether two EJB object references refer to the same entity by any of the following methods:

- Invoke the *isIdentical(object)* method on one of the references and pass the other reference as the method's argument.
- Obtain the primary keys, and compare them using the Java equality operator.

TODO: Add code showing obtaining primary key and test of object identity.

A client that know the primary key of an entity EJB object can obtain a reference to the object by invoking the *getByPrimaryKey(key)* method on the finder interface provided by the container.

Note that Enterprise JavaBeans does not specify "object equality" for EJB object references. The result of comparing two object references using the Java *Object.equals(Object obj)* method is unspecified. Performing the *Object.hashCode()* method on two object references that represent the same object is not guaranteed to yield the same result.

ISSUE: Should we prescribe comparison of EJB object references as part of the EJB architecture? Making this a requirement might put some burden on ORB providers, but it would simplify programming of client-side applications.

8.5 Enterprise bean's remote interface

A client accesses an entity bean through the enterprise bean's remote interface. An enterprise bean's remote interface must extend the *java.ejb.EJBObject* interface. A remote interface defines the business methods that are callable by clients.

The following example illustrates the definition of an entity bean's remote interface:

```
public interface Account extends java.ejb.EJBObject {
        void debit(double amount)
                  throws java.rmi.RemoteException;
        void credit(double amount)
                  throws java.rmi.RemoteException;
        double getBalance()
                  throws java.rmi.RemoteException;
}
```

The *java.ejb.EJBObject* interface defines methods that allow the client to do the following operations on an EJB object's reference:

- Obtain a reference of the EJB object's container.
- Destroy the EJB object.
- Obtain the EJB object's handle.
- Obtain the EJB object's primary key.

The implementation of the methods defined in the *java.ejb.EJBObject* interface is provided by the container. The business methods are delegated to the enterprise bean class.

Note that the EJB object does not expose the enterprise bean's methods introduced by the *java.ejb.EnterpriseBean* and *java.ejb.EntityControl* interfaces. These interfaces are not intended for the client—they are for the container to manage the object.

8.6 Enterprise bean's handle

A handle is an object that identifies an EJB object. A client that has a reference to an EJB object can obtain the object's handle by invoking *getHandle()* method on the reference.

Since the handle's class implements the *java.io.Serializable* interface, a client may serialize it. The client may use the serialized handle later, possibly in a different process, to re-obtain a reference to the EJB object identified by the handle.

Containers that store long-lived entities will typically provide handle implementations that allow clients to store a handle for a long time (possibly many years). Such a handle will be usable even if parts of the technology used by the container (e.g. ORB, DBMS, server) have been upgraded or replaced while the client has stored the handle.

The use of a handle is illustrated by the following example:

```
// A client obtains a handle of an account EJB object and
// stores the handle in stable storage.
//
ObjectOutputStream stream = ...;
Account account = ...;
Handle handle = account.getHandle();
stream.writeObject(handle);

// A client can read the handle from stable storage, and resurrect
// an object reference to the account EJB object from the handle.
//
ObjectInputStream stream = ...;
Handle handle = (Handle) stream.readObject(handle);
Account account = (Account) handle.getEJBObject();
account.debit(100.00);
```

9 Entity container protocol

Note: Container support for entity enterprise beans is an optional feature for EJB 1.0 compliance. Container support for entity enterprise beans will become mandatory in EJB 2.0.

The goal is to define a “basic” state management protocol between an entity enterprise bean and its container that must be supported by all entity containers.

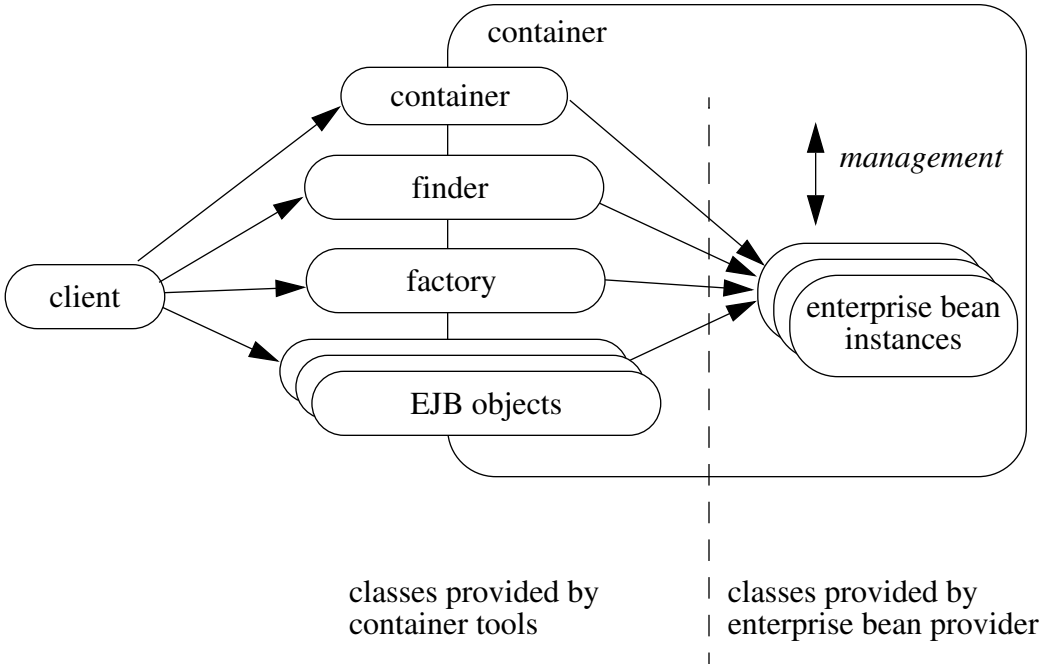
This chapter both describes the enterprise bean provider’s view of an entity object, and defines the responsibility of the entity container provider.

Note: This chapter is still work-in-progress.

9.1 The runtime execution model

TODO - explain the difference between an EJB object and enterprise bean instance

This section describes the runtime model and the classes used in the description of the contract between an enterprise bean and its container.



The *enterprise bean* is an object whose class was provided by the enterprise bean developer.

An *EJB object* is an object whose class was generated at deployment time by the container provider’s tools. The EJB object class implements the enterprise bean’s remote

interface. A client never references an enterprise bean instance directly—a client always references the corresponding EJB object.

The *container* object provides the lifecycle operations for its EJB objects. The class for the container object is provided by the container provider, or generated by the container provider's tools.

The *factory* is a remote object that implements an enterprise bean's factory interface. The class for the factory object was generated by the container provider's tools.

The *finder* is a remote object that implements an enterprise bean's finder interface. The class for the factory object was generated by the container provider's tools.

9.2 Entity persistence

The entity container protocol allows the enterprise bean provider either to implement the enterprise bean's persistence directly in the enterprise bean class (we call this bean-managed persistence), or delegate the enterprise bean's persistence to the container (we call this container-managed persistence).

9.2.1 Bean-managed persistence

In the bean-managed case, the enterprise bean provider writes database access calls (e.g. using JDBC™ or JSQL) directly in the methods of the enterprise bean class. The database access calls are performed in the *ejbCreate(...)*, *ejbDestroy()*, *ejbLoad()*, and *ejbStore()* enterprise bean callback methods.

The advantage of using bean-managed persistence is that the enterprise bean can be installed into a container without human assistance. The main disadvantage is that the persistence is hard-coded into the enterprise bean class, which makes it hard to adapt the enterprise bean to a different data source.

9.2.2 Container-managed persistence

In the container-managed case, the container provider's tools generate the database access calls at deployment time (i.e. when the enterprise bean class is installed into the container). The enterprise bean provider must specify the *containerManagedFields* deployment descriptor property to specify the list of instance fields for which the container provider tools shall generate database access calls.

The enterprise bean provider must declare the container-managed fields as *public* to allow the container tools to generate the additional classes that transfer data between the instance fields and the data source. The container-managed fields must be of Java primitive types.

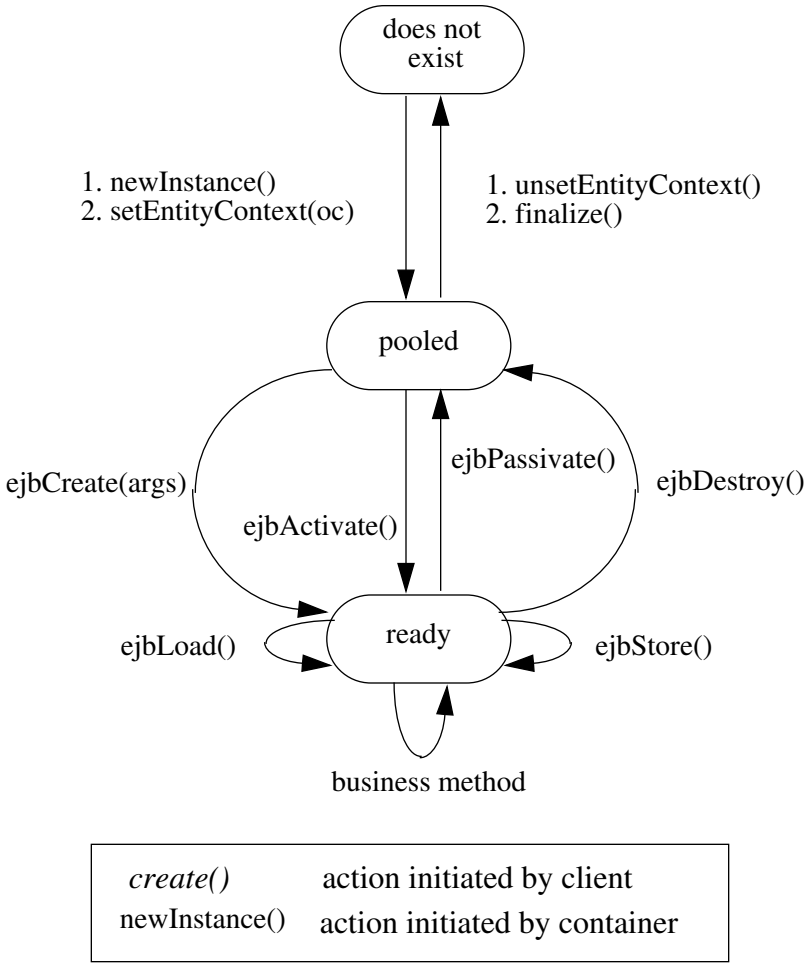
> *This restriction to only primitive types can, in the future, be relaxed to allow persisting bean handles, references to other enterprise beans, and any java.io.Serializable objects.*

The advantage of using container-managed persistence is that the enterprise bean class is independent from the data source in which the entity is stored. The container tools can generate classes that use JDBC or JSQL to access the entity state in a relational da-

tabase, or classes that implement access to a non-relational data source, such as CICS or IMS databases. The disadvantage is that tools must be used at deployment time to map the enterprise bean's fields to a specific data source.

9.3 Instance lifecycle

The following diagram illustrates the lifecycle of an enterprise bean's instance.



The following is a walk-through of the lifecycle of an entity enterprise bean instance:

- An enterprise bean's instance life starts when the container creates the instance using *newInstance()*. The container then invokes the *setEntityContext()* method to pass the instance a reference to an entity context object associated with the instance. The entity context object allows the instance to invoke services provided by the container and obtain the information about the caller of a business method.

- The instance enters the pool of available instances of the enterprise bean class. While the instance is in the available pool, the instance is not associated with an identity of a specific EJB object. All instances in the pool are equivalent, and therefore can be assigned by the container to any EJB object at the transition to the ready state.
- An instance transitions from the pooled state to the ready state when the container picks that instance to service a client call on an EJB object for which there is no instance in the ready state. There are two possible transitions from the pooled to the ready state: through the *ejbCreate(...)* method, and through the *ejbActivate()* method. The container invokes the *ejbCreate(...)* method when the instance is assigned to an EJB object during EJB object creation (i.e. when the client invokes a factory method to create the EJB object). The container invokes the *ejbActivate()* method on an instance when the instance needs to be activated to service an invocation on an existing EJB object.
- When an enterprise bean instance is in the ready state, the instance is associated with a specific EJB object. While the instance is in the ready state, the container can invoke the *ejbLoad()* and *ejbStore()* methods zero or more times, at anytime. A business method can be invoked on the instance zero or more times. Invocations of the *ejbLoad()* and *ejbStore()* methods can be arbitrarily mixed with invocations of business methods.
- Eventually, the container will transition the instance to the pooled state. There are two possible transitions from the ready to the pooled state: through the *ejbPassivate()* method, and through the *ejbDestroy()* method. The container invokes the *ejbPassivate()* method when the container wants to disassociate the instance from the EJB object without destroying the EJB object. The container invokes the *ejbDestroy()* method when the container is destroying the EJB object (i.e. when the client invoked the *destroy()* method on the EJB object, or one of the *destroy()* methods on the container).
- When the instance is put back into the pool, it is no longer associated with the identity of the EJB object. The container can assign the instance to any EJB object of the same enterprise bean class.
- An instance in the pool can be terminated by calling the *unsetEntityContext()* method on the instance. The Java runtime will eventually invoke the *finalize()* method on the instance.

9.4 The entity container contract

This section specifies the state management contract between an entity container and an enterprise bean.

9.4.1 Enterprise bean instance's view:

The following describes the enterprise bean instance's side of the contract:

An enterprise bean is responsible for doing the following in the callback methods:

- *public void setEntityContext(EntityContext ic);*

  A container uses this method to pass a reference to the entity context object to the enterprise bean instance. If the enterprise bean instance needs to use the entity context during its lifetime, it must remember the entity context in an instance variable.

  It is unspecified in which transaction context this method is called. An identity of an EJB object is not available during this method.

  The instance can take advantage of the *setEntityContext(ic)* method to allocate any resources that are to be held by the instance for its lifetime. Such resources cannot be specific to an EJB object identity since the instance might be reused during its lifetime to serve multiple EJB objects.

- *public void unsetEntityContext(EntityContext ic);*

  A container invokes this method before terminating the life of the instance.

  It is unspecified in which transaction context this method is called. An identity of an EJB object is not available during this method.

  The instance can take advantage of the *unsetEntityContext(ec)* method to free any resources that are held by the instance (these resource typically had been allocated by the *setEntityContext()* method).

- *public void ejbCreate(...);*

  There are zero[1] or more *ejbCreate(...)* methods, whose signatures match the signatures of the *create(...)* methods of the enterprise bean's factory interface (See Section 9.6 for the *ejbCreate(...)* design pattern). The container invokes an *ejbCreate(...)* method on an enterprise bean instance when a client invokes a matching factory *create(...)* function to create an EJB object.

  An *ejbCreate(...)* method gives the enterprise bean instance a chance to validate the client supplied arguments and initialize the instance variables from the input arguments before the instance enters the ready state.

  The instance must invoke the *setPrimaryKey(primaryKey)* method on its entity context during the *ejbCreate(...)* method so that the container knows the primary key that needs to be inserted into the EJB object reference that is returned to the client as the result of the factory *create(...)* method.

  An *ejbCreate(...)* method executes in the proper transaction context.

  An instance of an enterprise bean with bean-managed persistence should insert a record into the database with its initial values copied or computed from the input arguments of the *ejbCreate(...)* method. An instance of an enterprise bean with container-managed persistence must set the values of the container-

1. An entity enterprise bean has no *ejbCreate(...)* methods if it does not define a factory interface. Such an entity enterprise bean does not allow the clients to create new EJB objects. The enterprise bean restricts the clients to accessing entities that were created through direct database inserts.

managed fields such that when the *ejbCreate(...)* method completes, the container can extract the values and insert a record containing the values into the database.

- *public void ejbActivate();*

  The container invokes this method on the instance when the container picks the instance from the pool and assigns it to a specific EJB object identity. The *ejbActivate()* method gives the enterprise bean instance the chance to acquire additional resources that it needs while it is in the ready state.

  This method executes in an unspecified transaction context. The instance can obtain the identity of the EJB object via the *getPrimaryKey() or getEJBObject()* method on the entity context.

  public void ejbPassivate();
  The container invokes this method on an instance when the container decides to disassociate the instance from an EJB object identity, and put the instance back into the pool of available instances. The *ejbPassivate()* method gives the enterprise bean the chance to release any resources that should not be held while the instance is in the pool (these resource typically had been allocated during the *ejbActivate()* method).

  This method executes in an unspecified transaction context. The instance can still obtain the identity of the EJB object via the *getPrimaryKey() or getEJBObject()* method on the entity context.

- *public void ejbDestroy();*

  The container invokes this method on an instance as a result of a client's invoking a destroy method. The instance is in the ready state when *ejbDestroy()* is invoked and it will be entered into the pool when the method completes.

  This method executes in the effective transaction context of the client's *destroy* method. The instance can still obtain the identity of the EJB object via the *getPrimaryKey() or getEJBObject()* method on the entity context.

  An enterprise bean instance with bean-managed persistence should use this method to destroy its entity state in the database.

  Since the instance will be entered into the pool, the state of the instance at the end of this method must be equivalent to the state of a passivated instance. This means that the instance must release any resource that it would normally release in the *ejbPassivate()* method.

- *public void ejbLoad();*

  The container invokes this method on an instance in the ready state to advise the instance that it must synchronize its instance variables from the entity state in the database. The instance must be prepared for the container to invoke this method at any time that the instance is in the ready state.

An instance of an enterprise bean with bean-managed persistence should refresh its state in the *ejbLoad()* method by reading the entity state from the database.

An instance of an enterprise bean with container-managed persistence should refresh any fields whose content is computed from the fields managed by the container. The container invokes this method after it has read the container-managed fields from the database.

This method executes in the proper transaction context.

- *public void ejbStore();*

  The container invokes this method on an instance to advise the instance that the instance must synchronize the entity state in the database with its instance variables. The instance must be prepared for the container to invoke this method at any time that the instance is in the ready state.

  An instance of an enterprise bean with bean-managed persistence should write its state to the database in the *ejbStore()* method.

  An instance of an enterprise bean with container-managed persistence should prepare the fields that are managed by the container for being written to the database. The container invokes this method before it writes the container-managed fields to the database.

  This method executes in the proper transaction context.

9.4.2 Container's view:

The following describes the container's side of the state management contract. The container must call the following methods as indicated below:

- *public void setEntityContext(oc);*

  The container invokes this method to pass a reference to the enterprise bean's entity context to the enterprise bean. The container must invoke this method after it creates the instance, and before it puts the instance into the pool of available instances.

  It does not matter whether the container calls this method inside or outside of a transaction context.

- *public void unsetEntityContext(oc);*

  The container invokes this method when the container wants to reduce the number of instances in the pool. After this method completes, the container is not allowed to reuse this instance, and therefore it should drop any references to the instance to allow the Java garbage collector to eventually invoke the *finalize()* method on the instance.

  It does not matter whether the container calls this method inside or outside of a transaction context.

- *public void ejbCreate(...);*

The container invokes an *ejbCreate(...)* method on an instance during the creation of an EJB object as a result of a client invoking a create method on a container-generated class that implements the enterprise bean's factory interface. The container invokes the *ejbCreate(...)* method whose signature matches the *create(...)* method invoked by the client (See Section 9.6).

A container of an enterprise bean with bean-managed persistence invokes its *ejbCreate(...)* method to allow the instance to initialize its fields and create a representation of the entity in the database.

A container of an enterprise bean with container-managed persistence invokes the *ejbCreate(...)* method on the instance to allow the instance to initialize the container-managed fields from the *create(...)* arguments before the container extracts those fields from the instance. On return from the *ejbCreate(...)* method, the container creates a representation of the entity in the database using the values extracted from the instance.

The container invokes this method in the transaction context of the client's *create(...)* method.

- *public void ejbActivate();*

  The container invokes this method on an enterprise bean instance at activation time (i.e., when the instance is taken from the pool and assigned to an EJB object). The container must ensure that the primary key of the associated EJB object is available to the instance if the instance invokes the *getPrimaryKey()* or *getEJBObject()* method on its entity context.

  A container may call this method inside or outside of a transaction context.

- *public void ejbPassivate();*

  The container invokes this method on an enterprise bean instance at passivation time (i.e., when the instance is being disassociated from an EJB object and moved into the pool). The container must ensure that the primary key of the associated EJB object is still available to the instance if the instance invokes the *getPrimaryKey()* or *getEJBObject()* method on its entity context.

  A container may call this method inside or outside of a transaction context.

- *public void ejbDestroy()*;

  The container invokes this method before it ends the life of an EJB object as a result of a client's invoking a destroy operation.

  If the enterprise bean uses container-managed persistence, the container must invoke this method before deleting the database representation of the entity.

  The container invokes this method in the transaction context of the client's *destroy* method. The container must ensure that the primary key of the associated EJB object is still available to the instance if the instance invokes the *getPrimaryKey() or getEJBObject()* method on its entity context.

- *public void ejbLoad();*

The container must invoke this method on the instance whenever it becomes necessary for the instance to synchronize its instance state from its state in the database.

If the enterprise bean's persistence is container-managed, the container reloads the instance's container-managed state from the database before invoking this method on the instance.

The container invokes this method in the proper transaction context.

- *public void ejbStore();*

  The container must invoke this method on the instance whenever it becomes necessary for the instance to synchronize its state in the database with the state of the instance's fields.

  If the enterprise bean's persistence is container-managed, the container must invoke this method on the instance before it stores the instance's container-managed state to the database.

  The container invokes this method in the proper transaction context.

9.4.3 Single-threaded rule

The container must ensure that only one thread can be executing the instance at any time. Consequently, the container must not invoke any of the state management call-back methods while a business method invocation is in-progress.

9.5 The design pattern for business method delegation

TODO:

9.6 The design pattern for the factory interface

An enterprise bean's factory interface is defined by the enterprise bean provider. The factory interface defines one or more *create(...)* functions that a client can invoke to create a new EJB object.

9.6.1 Enterprise bean provider responsibility

The factory *create(...)* methods must follow these rules:

- The return value of a *create(...)* method must be the enterprise bean's remote interface.

For each factory interface *create(...)* method, the enterprise bean provider must also define a matching *ejbCreate(...)* method in the enterprise bean class. The enterprise bean provider must match the enterprise bean's *ejbCreate(...)* methods with the factory *create(...)* methods through the following design pattern:

- Each *create(...)* method of the factory interface must have a matching *ejbCreate(...)* method in the enterprise bean class.
- An *ejbCreate*(...) method must be declared as *public*.

- The input arguments of an *ejbCreate(...)* method need to be the same as the arguments of its corresponding *create(...)* method.
- The return value of an *ejbCreate(...)* method must be *void*.
- The *throws* clause of an *ejbCreate(...)* method can include an arbitrary set of exceptions.

For example, an enterprise bean class with the following factory interface:

```
public interface AccountFactory extends java.ejb.Factory {
        public Account create(Foo foo)
                  throws RemoteException;
        public Account create(Foo foo, Bar bar);
                  throws RemoteException;
}
```

can have the following *ejbCreate(...)* methods:

```
public class AccountBean... {
        public void ejbCreate(Foo foo) {...}
        public void ejbCreate(Foo foo, Bar bar) {...}
                  ...
}
```

Each *ejbCreate(...)* method must include a call to the *setPrimaryKey(primaryKey)* method on the entity context. The *setPrimaryKey(primaryKey)* method allows the container to know the value of the primary key to embed into the reference of the created EJB object.

The following example illustrates the calls to the *setPrimaryKey(primaryKey)* method. The “PrimaryKeyType” type in the example can be any Java class that implements the *java.io.Serializable* interface.

```
public class AccountBean... {
        EntityContext entityContext;

        public void ejbCreate(Foo foo) {
                  PrimaryKeyType primaryKey;
                            ...
                  primaryKey = ...;
                  entityContext.setPrimaryKey(primaryKey);
                            ...
        }
        public void ejbCreate(Foo foo, Bar bar) {
                  PrimaryKeyType primaryKey;
                            ...
                  primaryKey = ...;
                  entityContext.setPrimaryKey(primaryKey);
                            ...
        }
}
```

9.6.2 Container provider's responsibility

The container is responsible for generating a class that provides the implementation of the methods of the factory interface. Each *create(...)* functions invokes a matching *ejbCreate(...)* method.

The container provider is responsible for catching any exception thrown from an *ejbCreate(...)* method, and must perform as follows:

- If the exception thrown by the *ejbCreate(...)* method is an instance of an exception declared in the *throws* clause of the matching *create(...)* method, the exception should be re-thrown to the client.
- All other exceptions should be converted to a *java.rmi.RemoteException,* or an exception of a subclass of *java.rmi.RemoteException*.

The following is a skeleton of the class that a container provider would generate to implement the factory interface:

```
public class AcmeAccountFactory implements AccountFactory {
                ...
        public Account create(Foo foo)
                throws RemoteException{
                AccountBean instance;
                 ...
                instance = ...;
                instance.ejbCreate(foo);
                //
                // if this is EJB with container-managed
                // persistence, extract values and insert
                // a record into database
                 ...
        }
        public Account create(Foo foo, Bar bar)
                throws RemoteException {
                AccountBean instance;
                           ...
                instance = ...;
                instance.ejbCreate(foo, bar);
                //
                // if this is EJB with container-managed
                // persistence, extract values and insert
                // a record into database
                 ...
        }
}
```

If an enterprise bean uses container-managed persistence, the implementation of a *create(...)* function must invoke a matching *ejbCreate(...)* method on the enterprise bean instance before it creates a representation of the entity in the database.

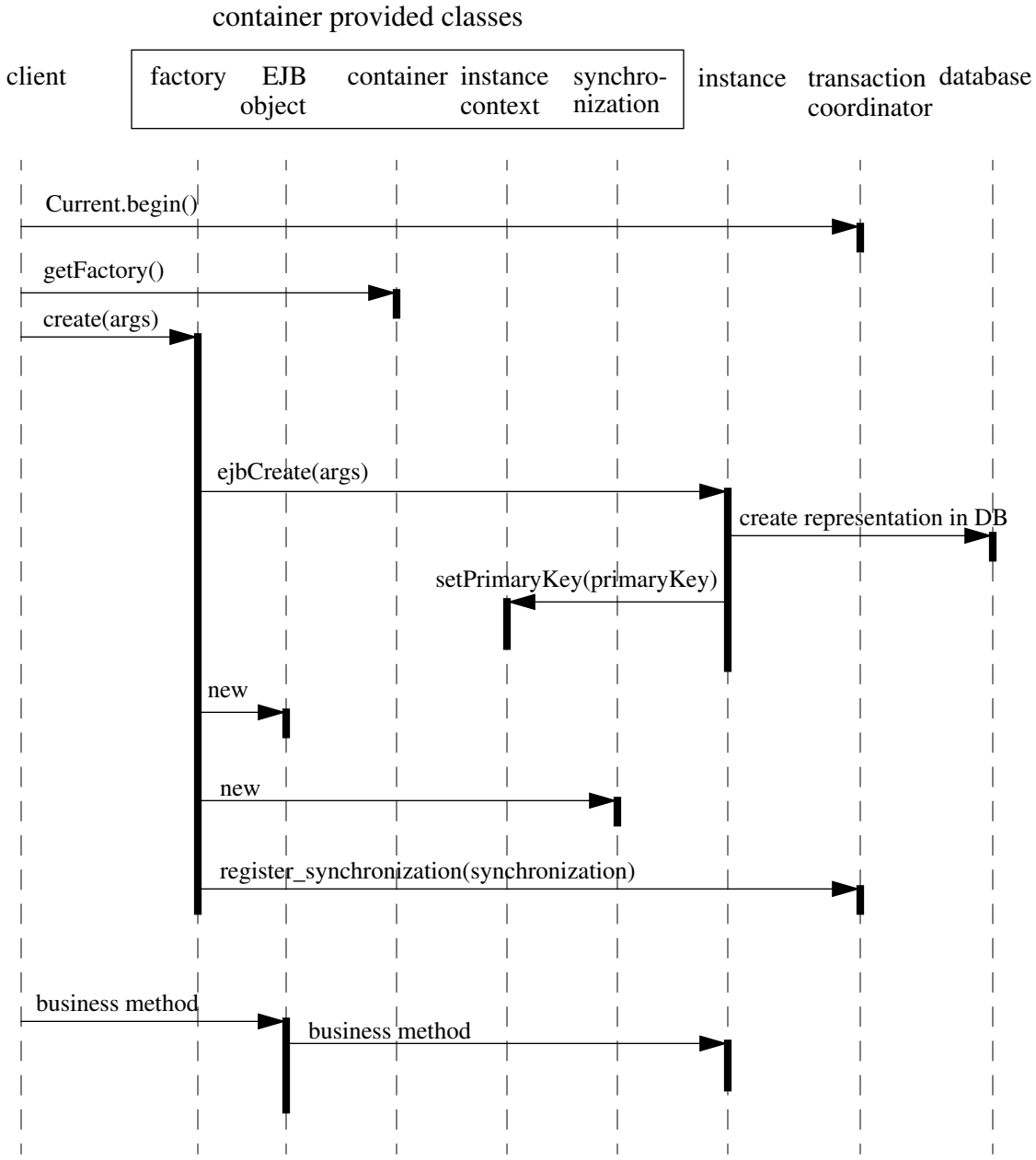
9.7 The design pattern for the finder interface

TODO:

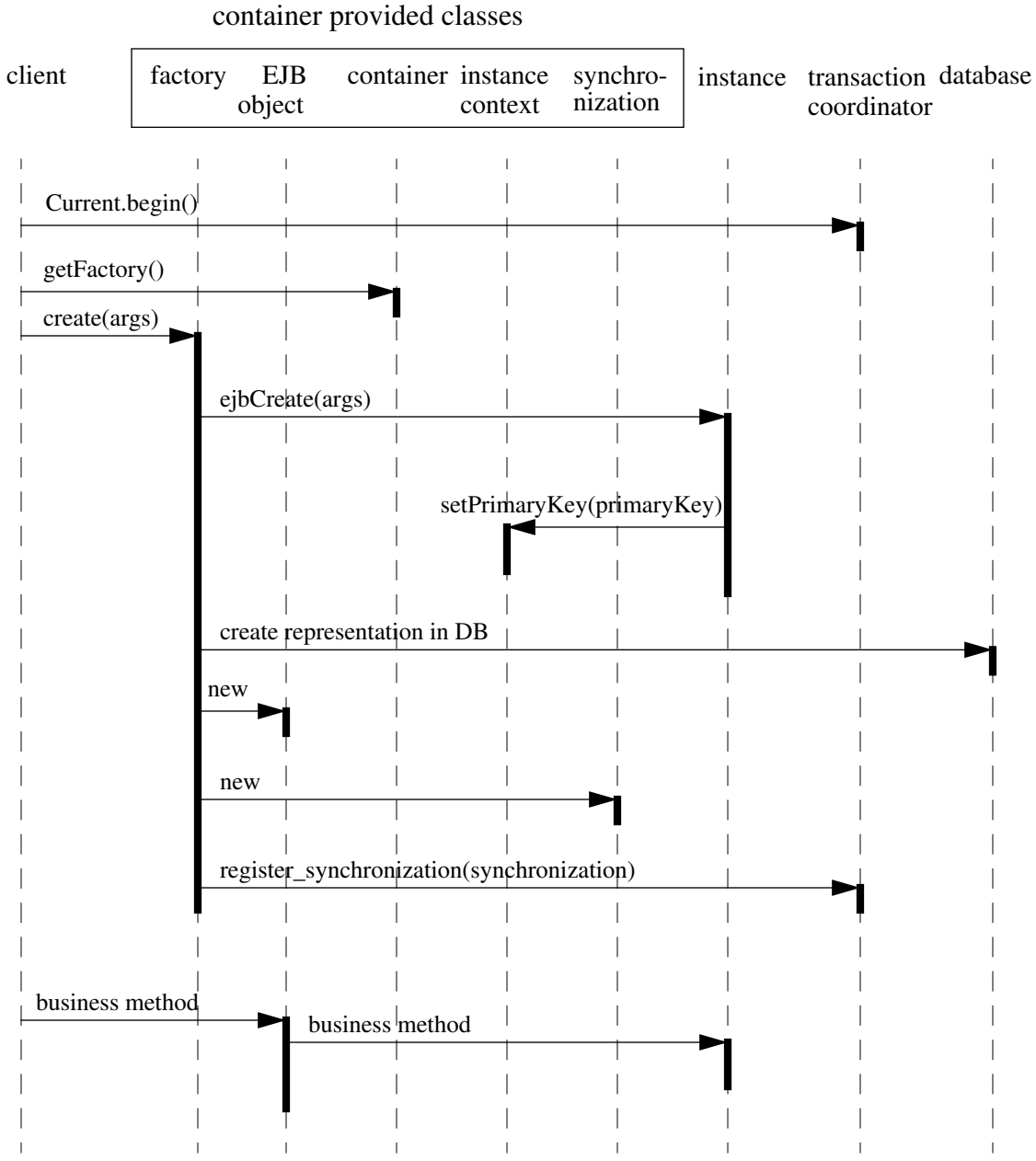
9.8 Sequence diagrams

9.8.1 Creating an entity object

The following diagram illustrates the creation of an enterprise bean with bean-managed persistence.

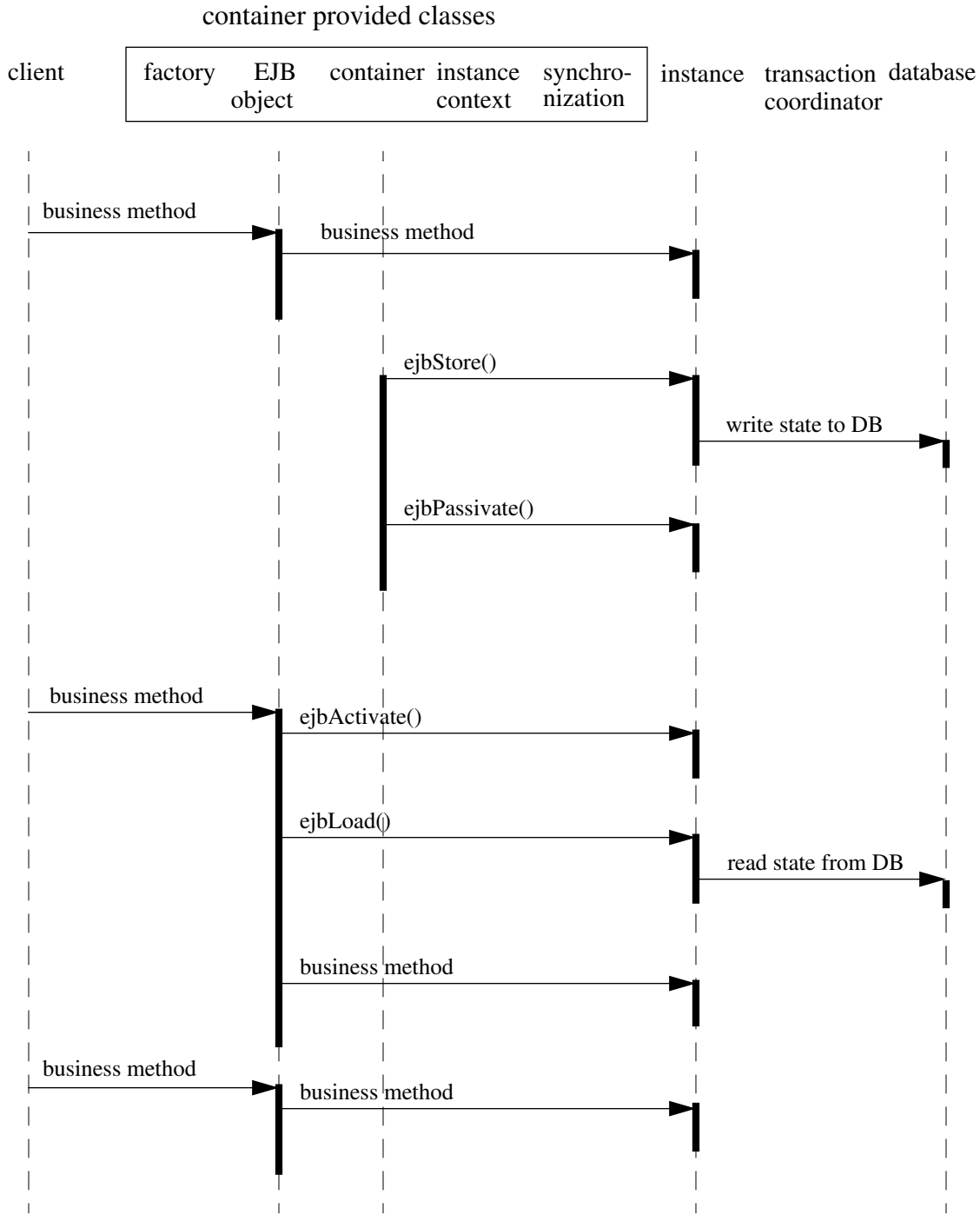


The following diagram illustrates the creation of an enterprise bean with container-managed persistence:

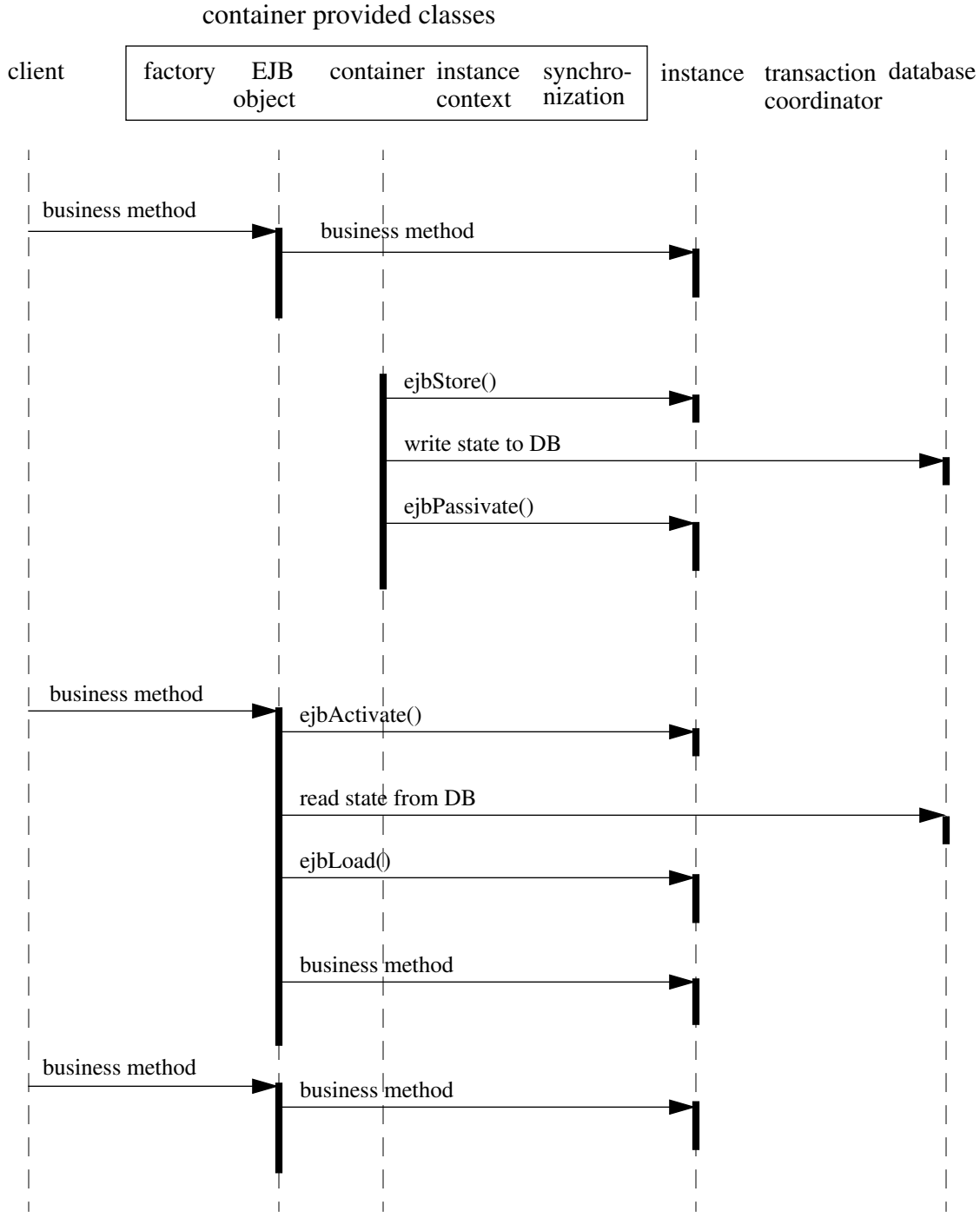


9.8.2 Passivating and activating an instance in a transaction

The following diagram illustrates the passivation and reactivation of an enterprise bean instance with bean-managed persistence.



The following diagram illustrates the passivation and reactivation of an enterprise bean instance with container-managed persistence.



9.8.3 Committing a transaction

This section describes the sequence during transaction commit.

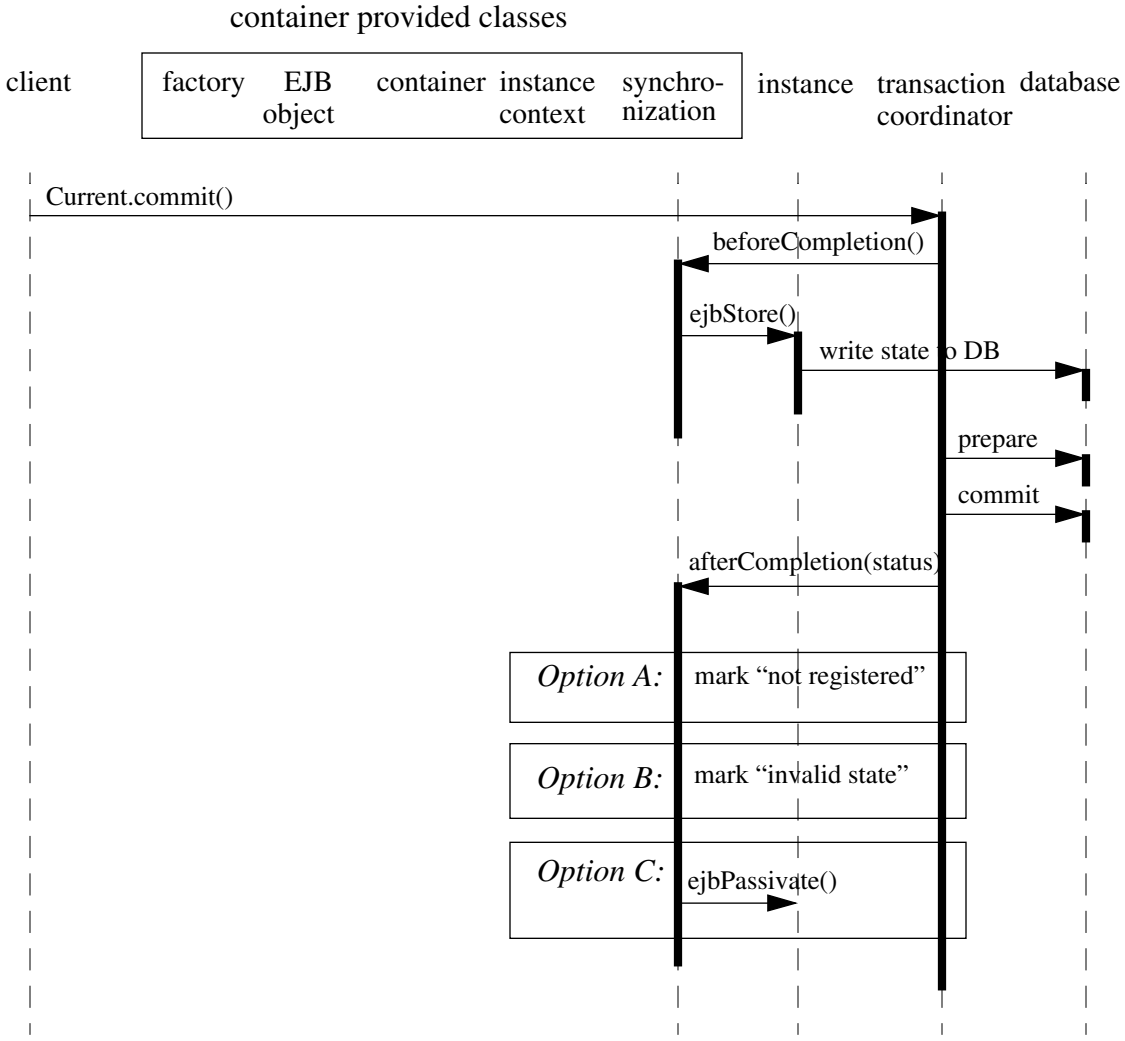
Different containers may provide difference disposition of an instance after transaction commit. The examples below illustrate three different options (these options are illustrative rather than prescriptive).

- Option A:
  - Cache during a transaction and between transactions.
  - Exclusive persistent storage—no refresh of instance state is needed.
- Option B:
  - Cache during a transaction and between transactions.
  - Shared persistent storage—refresh of instance state is needed.
- Option C:
  - Cache only during a transaction.
  - Shared persistent storage.

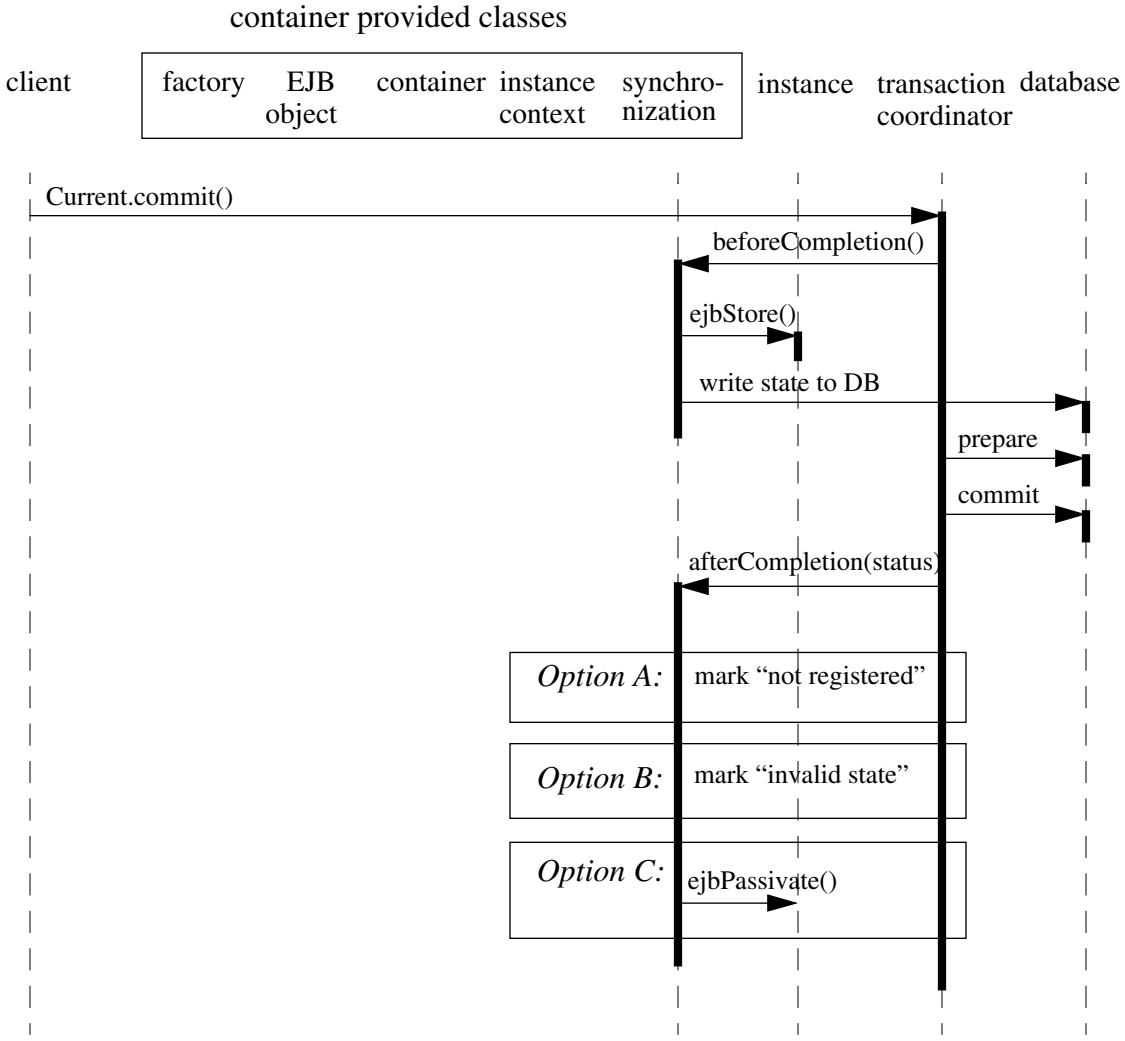
Table 1: Example commit-time options

	Write instance state to database	Instance stays active	Instance state remains valid
Option A	Yes	Yes	Yes
Option B	Yes	Yes	No
Option C	Yes	No	No

The following diagram illustrates the transaction commit protocol that involves an enterprise bean instance with bean-managed persistence.

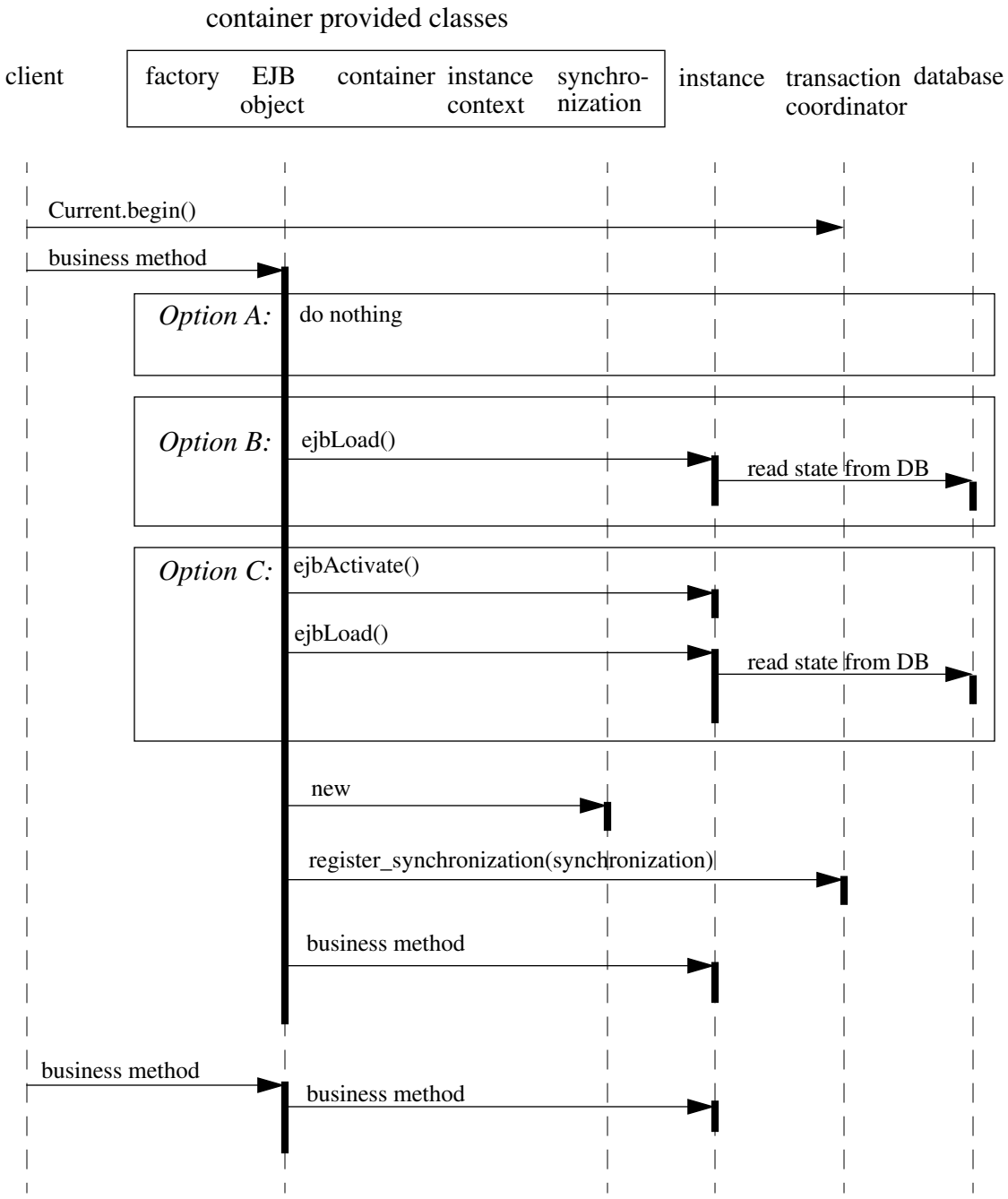


The following diagram illustrates the transaction commit protocol for an enterprise bean instance with container-managed persistence.

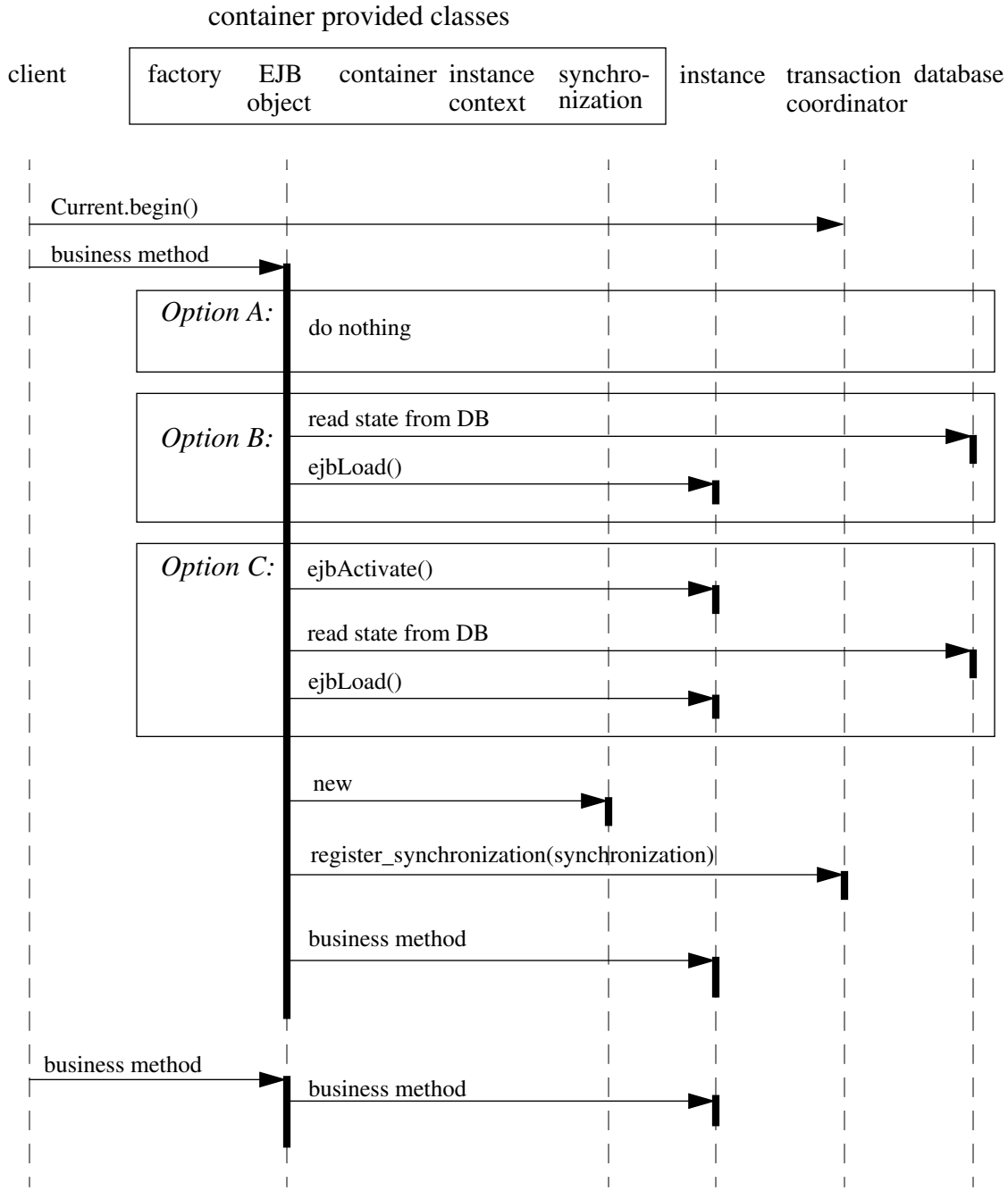


9.8.4 Starting the next transaction

The following diagram illustrates the protocol performed for a bean with bean-managed persistence at the beginning of a newtransaction. The three options illustrated in the diagram correspond to the three commit options in the previous subsection.

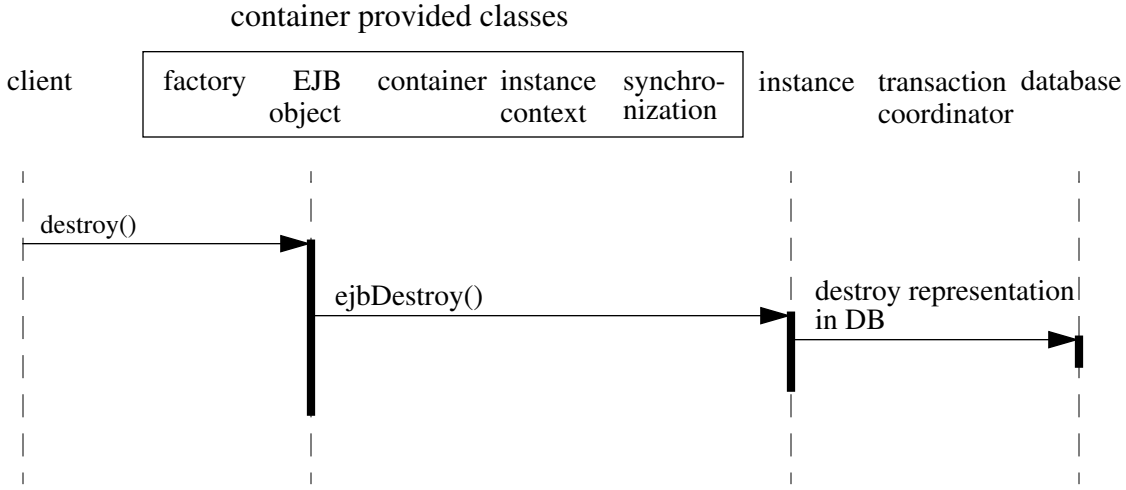


The following diagram illustrates the protocol performed for a bean with container-managed persistence at the beginning of a new transaction.

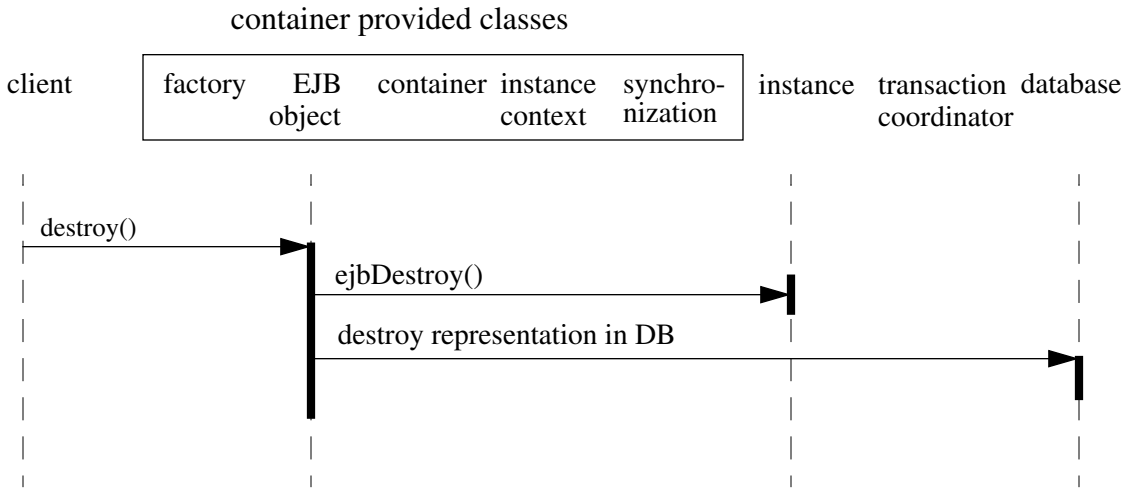


9.8.5 Destroying an entity object

The following diagram illustrates the destruction of an entity enterprise bean with bean-managed persistence.



The following diagram illustrates the destruction of an entity enterprise bean with container-managed persistence.



9.8.6 Finding an object

TODO

10 Example entity scenario

Note: Container support for entity enterprise beans is an optional feature for EJB 1.0 compliance. Container support for entity enterprise beans will become mandatory in EJB 2.0.

This chapter describes an example development and deployment scenario for an entity enterprise bean. We use the scenario to explain the responsibilities of the enterprise bean provider and those of the container provider.

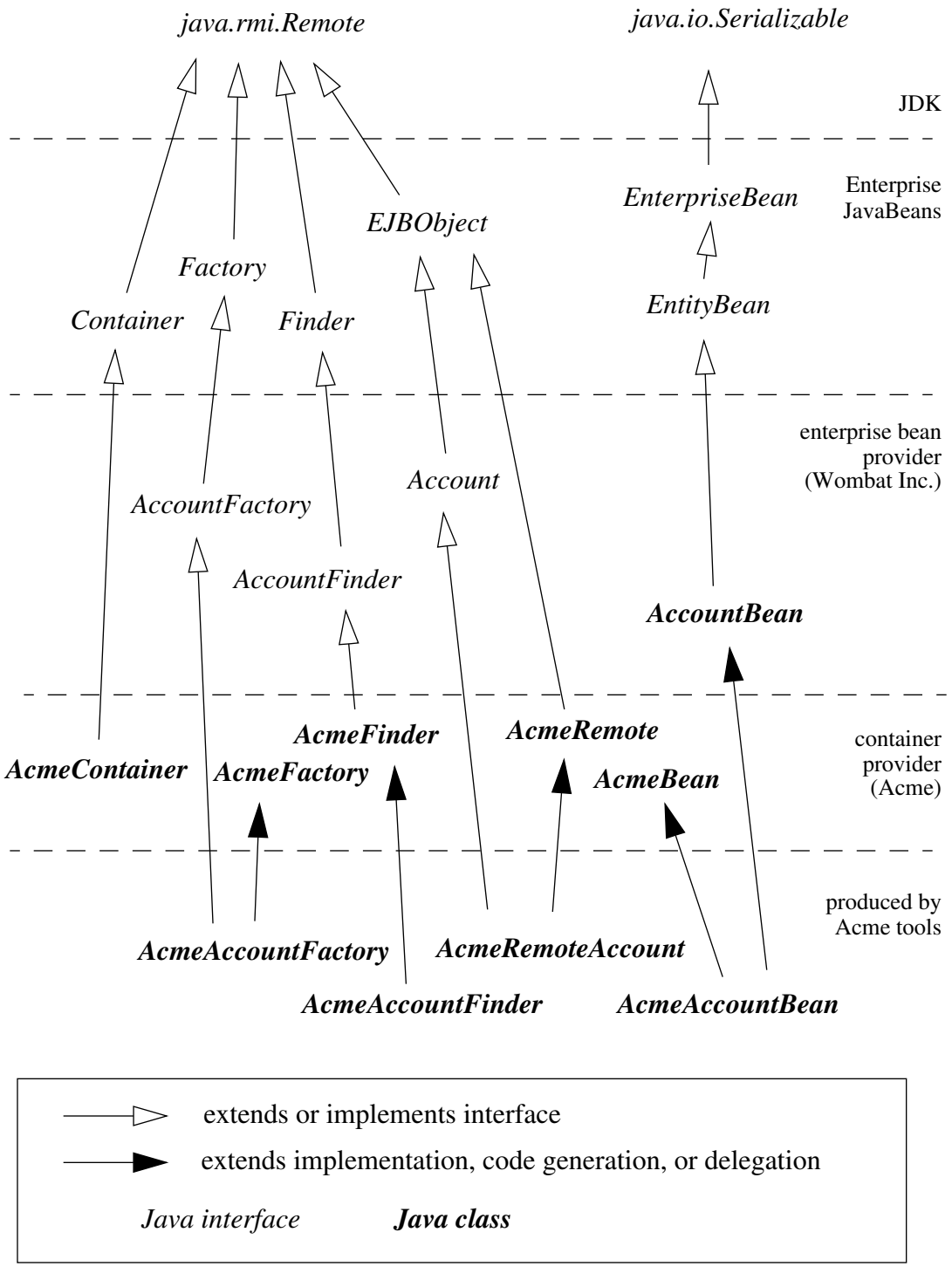
The classes generated by the container provider's tools in this scenario should be considered illustrative rather than prescriptive. Container providers are free to implement the contract between an enterprise bean and its container in a different way that achieves an equivalent effect (from the perspectives of the enterprise bean provider and the client-side programmer).

10.1 Overview

Wombat Inc. has developed the *AccountBean* enterprise bean. The AccountBean enterprise bean is deployed in a container provided by the Acme Corporation.

10.2 Inheritance relationship

An example of the inheritance relationship between the interfaces and classes is illustrated in the following diagram:



10.2.1 What the enterprise bean provider is responsible for

Wombat Inc. is responsible for providing the following:

- Define the enterprise bean's remote interface (Account). The remote interface defines the business methods callable by a client. The remote interface must extend the *java.ejb.EJBObject* interface, and follow the standard rules for a Java RMI remote interface. The remote interface must be defined as *public*.
- Write the business logic in the enterprise bean class (AccountBean). The enterprise bean class must not implement the enterprise bean's remote interface (Account). The enterprise bean must implement the *java.ejb.EntityBean* interface, and define the *ejbCreate(...)* methods invoked at an EJB object creation. The *ejbCreate(...)* methods must follow the factory design pattern described in Section 9.6.
- Define a factory interface (AccountFactory) for the enterprise bean. The signatures of the methods of the factory interface must follow the factory design pattern described in Section 9.6. The factory interface must be defined as *public*, extend the *java.ejb.Factory* interface, and follow the standard rules for Java RMI remote interfaces.
- TODO: describe AccountFinder
- Specify the environment properties that an enterprise bean requires at runtime. The environment properties is a standard *java.util.Properties* file.
- Define a deployment descriptor that specifies any declarative metadata that the enterprise bean provider wishes to pass with the enterprise bean to the next stage of the development/deployment workflow.

10.2.2 Classes supplied by container provider

The following classes are supplied by the container provider, Acme Corp:

- The AcmeContainer class provides the Acme implementation of the *java.ejb.Container* interface.
- The AcmeFactory class provides the Acme implementation of a factory base class.
- The AcmeFinder class provides the Acme implementation of a finder base class.
- The AcmeRemote class provides the Acme implementation of the *java.ejb.EJBObject* methods.
- The AcmeBean class provides additional state and methods to allow Acme's container to manage its enterprise bean instances. For example, if Acme's container uses an LRU algorithm, then AcmeBean may include the clock count and methods to use it.

10.2.3 What the container provider is responsible for

The tools provided by Acme Corporation are responsible for the following:

- Generate the remote bean class (AcmeRemoteAccount) for the enterprise bean. The remote bean class is a “wrapper” class for the enterprise bean and provides the client’s view of the enterprise bean. The tools also generate the classes that implement the communication stub and skeleton for the remote bean class.
- Generate the implementation of the enterprise bean class suitable for the Acme container (AcmeAccountBean). AcmeAccountBean includes the business logic from the AccountBean class mixed with the services defined in the AcmeBean class. Acme tools can use inheritance, delegation, and code generation to achieve mix-in of the two classes.
- Generate the class for the enterprise bean’s factory interface (AcmeAccountFactory). The tools also generate the classes that implement the communication stub and skeleton for the factory class.
- Generate the class for the enterprise bean’s finder interface (AcmeAccountFinder). The tools also generate the classes that implement the communication stub and skeleton for the finder class.

Many of the above classes and tools are container-specific (i.e., they reflect the way Acme Corp implemented them). Other container providers may use different mechanisms to produce their runtime classes, and the generated classes most likely will be different from those generated by Acme’s tools.

11 Support for transactions

One of the key features of Enterprise JavaBeans is support for distributed transactions. Enterprise JavaBeans allows an application developer to write an application that atomically updates data in multiple databases which are possibly distributed across multiple sites. The sites may use EJB servers and containers from different vendors.

No distinction is made between session and entity beans in this section. This section applies equally to both.

An enterprise bean developer or client programmer does not have to deal with the complexity of distributed transactions. The burden of managing transactions is shifted to the container and EJB server providers. A container implements the declarative transaction scopes defined later in this chapter. The EJB server implements the necessary low-level transaction protocols, such as the two-phase commit protocol between a transaction manager and a database system, transaction context propagation, and distributed two-phase commit.

11.1 Transaction model

Enterprise JavaBeans supports flat transactions, modeled after the OMG Object Transaction Service 1.1 (OTS). An enterprise bean object that is *transaction-enabled* corresponds to the *TransactionalObject* described in OTS (a future release may allow an enterprise bean to act as a recoverable object).

> *Note: The decision not to support nested transactions was to allow vendors of existing transaction processing and database management systems to incorporate support for Enterprise JavaBeans. If these vendors provide support for nested transactions in the future, Enterprise JavaBeans may be enhanced to take advantage of nested transactions.*

11.2 Relationship to JTS

Enterprise JavaBeans is high-level component framework that attempts to hide system complexity from the application developer. Therefore, most enterprise beans do not directly access transaction management.

JTS is a lower-level API, part of which must be implemented by the EJB server and used by the container.

In Release 1.0, an enterprise bean does not use the JTS interfaces directly. The only exception to this rule is that an enterprise bean with the *BEAN_MANAGED* transaction attribute is allowed to use the *java.jts.CurrentTransaction* interface to demarcate transaction boundaries.

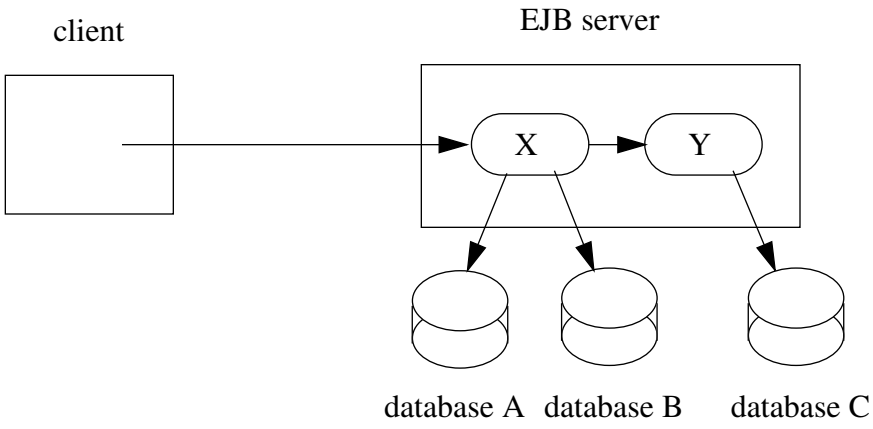
11.3 Scenarios

This section describes several scenarios that illustrate the distributed transaction capabilities of Enterprise JavaBeans.

11.3.1 Update of multiple databases

Enterprise JavaBeans makes it possible for an application program to update data in multiple databases in a single transaction.

In the following figure, a client invokes the enterprise bean X. X updates data in two databases, A and B. Then X calls another enterprise bean Y. Y updates data in database C. The EJB server ensures that the updates to databases A, B, and C are either all committed, or all rolled back.



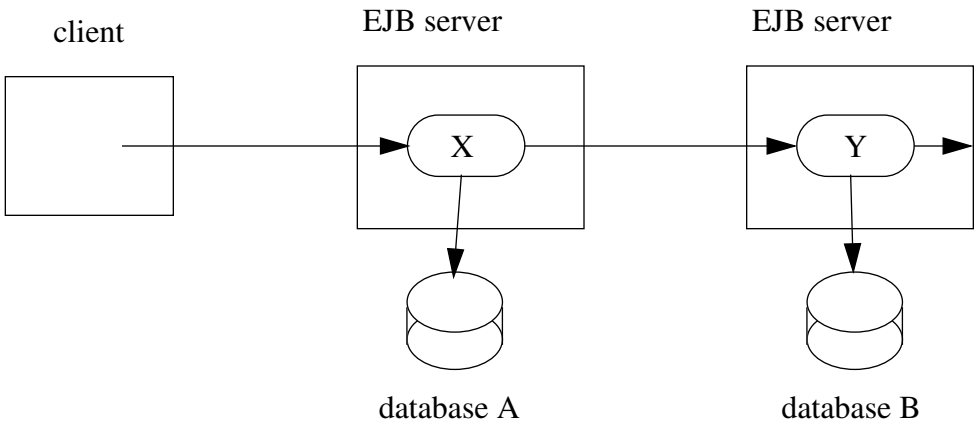
The application programmer does not have to do anything to handle transaction semantics. The enterprise beans X and Y perform the database updates using the standard JDBC API. Behind the scenes, the EJB server enlists the database connections as part of the transaction. When the transaction commits, the EJB server and the database systems perform a two-phase commit protocol to ensure atomic updates across all the three databases.

11.3.2 Update of databases via multiple EJB servers

Enterprise JavaBeans allows updates of data at multiple sites to be performed in a single transaction.

In the following figure, a client invokes the enterprise bean X. X updates data in database A, and then calls another enterprise bean Y that is installed in a remote EJB server.

Y updates data in database B. Enterprise JavaBeans makes it possible to perform the updates to databases A and B as a single transaction.



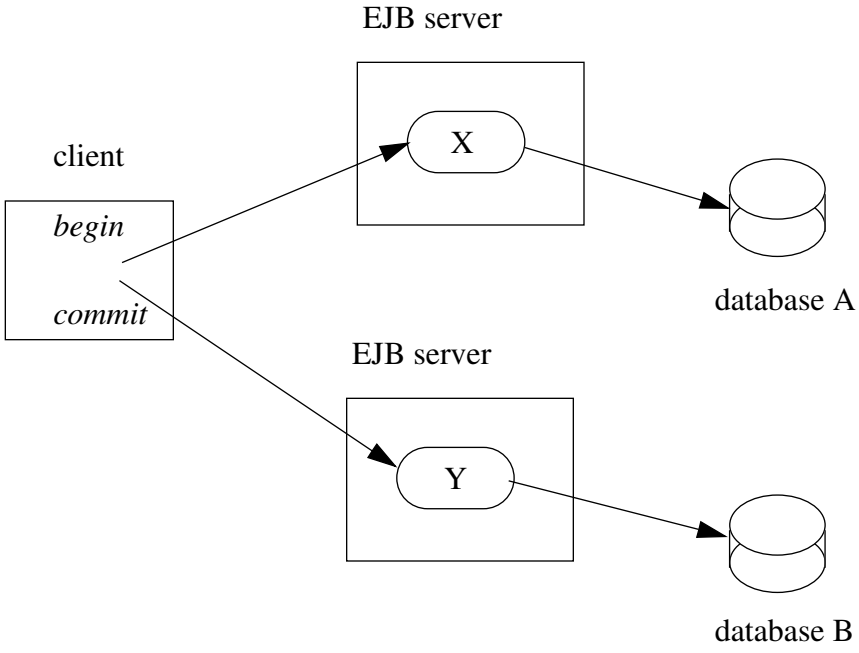
When X invokes Y, the two EJB servers cooperate to propagate the transaction context from X to Y. This transaction context propagation is transparent to the application-level code.

At transaction commit time, the two EJB servers use a distributed two-phase commit protocol (if the capability exists) to ensure the atomicity of the database updates.

11.3.3 Client-managed demarcation

A client or a non-transaction enterprise bean object can use the *java.jts.CurrentTransaction* interface to explicitly demarcate transaction boundaries.

A client program using explicit transaction demarcation may perform atomic updates across multiple databases residing at multiple transaction servers, as illustrated in the following figure.



The application programmer does not have to do anything to make the updates to databases A and B performed by enterprise beans X and Y atomic other than demarcate the transaction by the *begin* and *commit* calls. A proxy of a transaction service on the client automatically propagates the transaction context to the two EJB servers. When the client program calls commit, the two EJB servers perform the two-phase commit protocol.

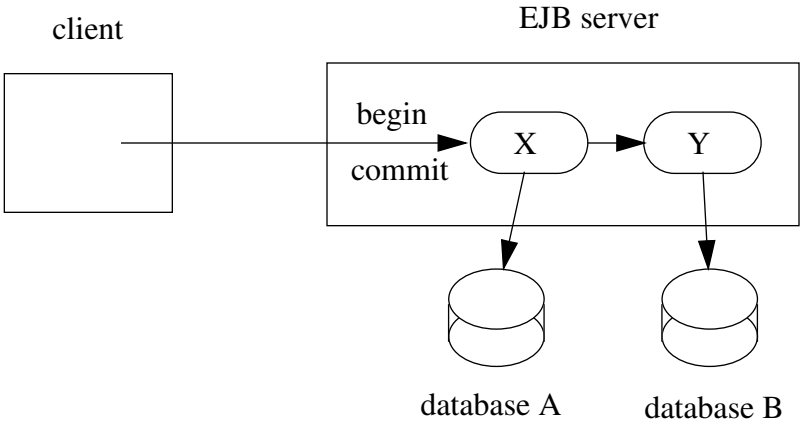
11.3.4 Container-managed demarcation

Whenever a client invokes an enterprise bean, the container interposes on the method invocation. The interposition allows the container to control transaction demarcation declaratively through the *transaction attribute*.

For example, if an enterprise bean is deployed with the *REQUIRES* transaction attribute, the container automatically initiates a transaction whenever a client invokes a transaction-enabled enterprise bean while the client is not associated with a transaction context.

The following figure illustrates such a scenario. A non-transactional client invokes the enterprise bean X. Since the message from the client does not include a transaction context, the container starts a new transaction before dispatching the remote method on X. X's work is performed in the context of the transaction. When X calls other enterprise beans (Y in our example), the work performed by the other enterprise beans is also au-

tomatically included in the transaction (subject to the transaction attribute of the other enterprise bean).



The container automatically commits the transaction at the time X returns a reply to the client.

11.3.5 Bean-managed demarcation

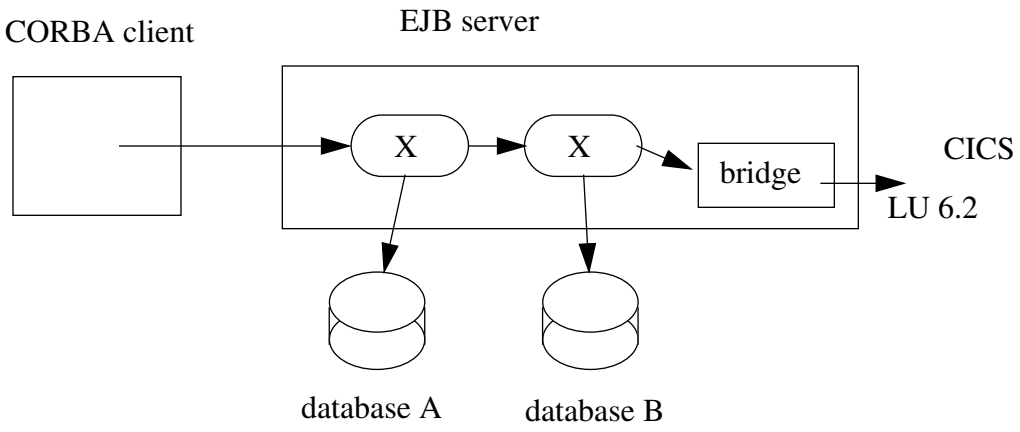
An enterprise bean with the *BEAN_MANAGED* transaction attribute can use the *java.jts.CurrentTransaction* interface to demarcate transactions.

11.3.6 Interoperability with non-Java clients and servers

Although the focus of Enterprise JavaBeans is the Java API for writing distributed enterprise applications in Java, it is desirable that such applications can also interoperate with non-Java clients and servers.

A container can make it possible for an enterprise bean to be invoked from a non-Java client. For example, the CORBA mapping of Enterprise JavaBeans [6] allows any

CORBA client to invoke any enterprise bean object on a CORBA-enabled server using the standard CORBA API.



Providing connectivity to existing server applications is also important. An EJB server may choose to provide access to existing enterprise applications, such as applications running under CICS on a mainframe. For example, an EJB server may provide a bridge that makes existing CICS programs accessible to enterprise beans. The bridge can make the CICS programs visible to the Java developer as if the CICS programs were other enterprise beans installed in some container on the EJB server.

Note: It is beyond the scope of the Enterprise JavaBeans specification to define the bridging protocols that would enable such interoperability. Such bridges will be a value added by some EJB servers.

11.4 Declarative transaction management

Every client method invocation on an enterprise bean object is interposed by the container. The interposition allows for delegating the transaction management responsibilities to the container.

The declarative transaction management is controlled by a *transaction attribute* associated with each enterprise bean's home container. The container provider's tools can be used to set and change the values of transaction attributes.

Enterprise JavaBeans defines the following values for the transaction attribute:

- *NOT_SUPPORTED*
- *BEAN_MANAGED*
- *REQUIRES*
- *SUPPORTS*
- *REQUIRES_NEW*
- *MANDATORY*

The transaction attribute is specified in the enterprise bean's deployment descriptor. A transaction attribute can be associated with the entire bean (to apply to all methods), or it can be associated with an individual method.

11.4.1 NOT_SUPPORTED

A container must always invoke an enterprise bean that has the *NOT_SUPPORTED* transaction attribute without a transaction scope. If a client calls with a transaction scope, the container suspends the association of the transaction scope with the current thread before delegating the method call to the enterprise bean object. The container resumes the suspended association when the method call on the enterprise bean object has completed.

The suspended transaction context of the client is not passed to resources or other enterprise bean objects that are invoked from the enterprise bean object.

11.4.2 BEAN_MANAGED

An enterprise bean with the *BEAN_MANAGED* attribute can use the *java.jts.CurrentTransaction* interface to demarcate transaction boundaries.

TODO: need to described the rules that the enterprise bean must follow when doing transaction demarcation and the rules for the container to deal with suspending and resuming caller's transaction.

11.4.3 REQUIRES

If a client invokes an enterprise bean object that has the *REQUIRES* transaction attribute while the client is associated with a transaction context, the container invokes the enterprise bean's method in the client's transaction context.

If the client invokes the enterprise bean object while the client is not associated with a transaction context, the container automatically starts a new transaction before delegating a method call to the enterprise bean object, and attempts to commit the transaction when the method call on the enterprise bean object has completed. The container performs the commit protocol before the method result is sent to the client.

The transaction context is passed to the resources or other enterprise bean objects that are invoked from the enterprise bean object.

11.4.4 SUPPORTS

An enterprise bean object that has the *SUPPORTS* transaction attribute is invoked in the client's transaction scope. If the client does not have a transaction scope, the enterprise bean is also invoked without a transaction scope.

The transaction context (if any) is passed to the resources or other enterprise bean objects that are invoked from the enterprise bean object.

11.4.5 REQUIRES_NEW

An enterprise bean that has the *REQUIRES_NEW* transaction attribute is always invoked in the scope of a new transaction. The container starts a new transaction before

delegating a method call to the enterprise bean object, and attempts to commit the transaction when the method call on the enterprise bean object has completed. The container performs the commit protocol before the method result is sent to the client.

If the client request is associated with a transaction, the association is suspended before the new transaction is started and is resumed when the new transaction has completed.

The new transaction context is passed to the resources or other enterprise bean objects that are invoked from the enterprise bean object.

11.4.6 MANDATORY

An enterprise bean object that has the *MANDATORY* attribute is always invoked in the scope of the client's transaction. If the client attempts to invoke the enterprise bean without a transaction context, the container throws the *TransactionRequired* exception to the client.

The client's transaction context is passed to the resources or other enterprise bean objects that are invoked from the enterprise bean object.

11.4.7 Transaction attribute summary

The following table provides a summary of the transaction scopes under which a method on an enterprise bean object method executes, as a function of the transaction attribute and client's transaction context. A dash means "no transaction context".

Table 2: Effect of the declarative transaction attribute

Transaction attribute	Client's transaction	Transaction associated with enterprise bean's method
NOT_SUPPORTED	-	-
	T1	-
BEAN_MANAGED	TODO	
REQUIRES	-	T2
	T1	T1
SUPPORTS	-	-
	T1	T1
REQUIRES_NEW	-	T2
	T1	T2
MANDATORY	-	error
	T1	T1

11.5 Bean-managed demarcation

An enterprise bean with the *BEAN_MANAGED* attribute is allowed to use the *java.jts.CurrentTransaction* interface to demarcate transaction boundaries.

The container makes the *java.jts.CurrentTransaction* interface available to the enterprise bean though the *InstanceContext.getCurrentTransaction()* method, as illustrated in the following example.

```
import java.jts.CurrentTransaction;
...
InstanceContext ic = ...;
...
CurrentTransaction tx = ic.getCurrentTransaction();
tx.begin();
...
tx.commit();
```

Enterprise beans deployed with a transaction attribute other than *BEAN_MANAGED* are not allowed to access directly the underlying transaction manager. This means that the container makes the JTS API unavailable to the enterprise bean.

11.6 Transaction management exceptions

The container throws the Transaction*RollbackException*, *TransactionRequiredException*, and *InvalidTransactionException* exceptions in the situations defined in the JTS specification.

TODO - here we need to define rules for how a container shall handle transaction management exceptions

12 Support for distribution

12.1 Overview

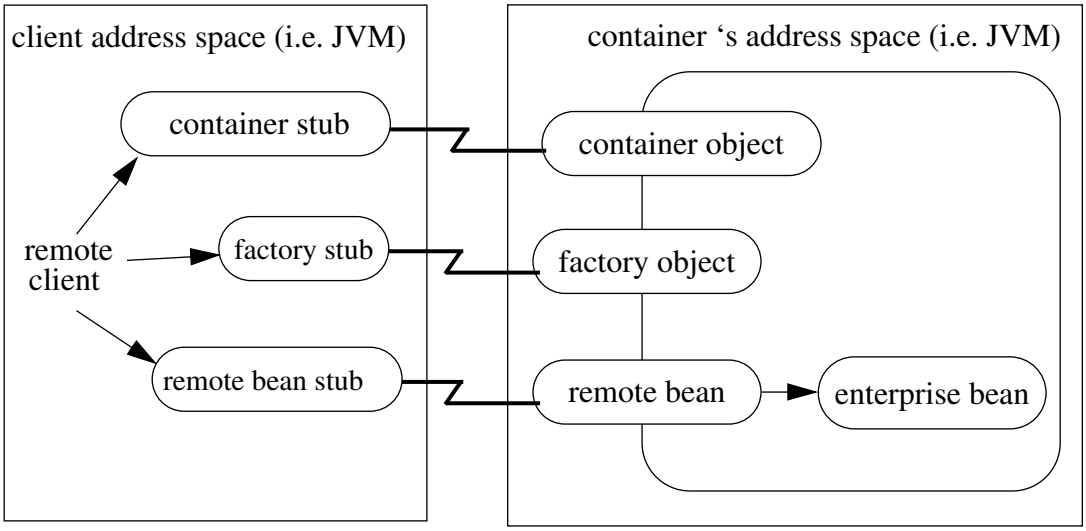
Support for remote client access to an enterprise bean object is through the standard Java API for remote method invocation (Java RMI) [3]. This API allows a client to invoke an enterprise bean object using the industry standard IIOP protocol, as defined in the OMG Java to IDL Mapping specification [5].

The Java RMI API makes access to an enterprise bean object *location transparent* to a client programmer.

12.2 Client-side objects

The following objects are present in the client's JVM:

- A local container object.
- A stub for the enterprise bean's factory object.
- A stub for the EJB object.



The factory object, container, and the EJB object are remote objects in the sense of Java RMI. The Java RMI specification [3] and the OMG Java to IDL Mapping specification [5] define the stubs for the factory, container, and EJB objects, and the communication between the stubs and the objects on the server.

12.3 Interoperability via network protocol

12.3.1 Mapping to CORBA

The standard mapping of Enterprise JavaBeans to CORBA is defined in [6].

The mapping enables the following interoperability:

- Any non-Java CORBA client can access any enterprise bean object.
- A client using an ORB from one vendor can access enterprise beans residing on a CORBA-based EJB server provided by another vendor.
- Enterprise beans in one CORBA-based EJB server can access enterprise beans in another CORBA-based EJB server.

12.3.2 Support for other protocols

Other forms of distributions are possible. For example, a client may use HTTP to invoke a servlet that invokes an enterprise bean object via its remote bean. Similarly, an ActiveX client may invoke an enterprise bean via its remote bean through a DCOM/IIOP bridge. These forms of distribution are not covered by this specification at this time.

13 Support for security

Support for security in Enterprise JavaBeans includes the following components:

- Use of the existing Java security APIs defined in the core package *java.security.*
- Security-related methods in *InstanceContext.*
- Security-related attributes in the deployment descriptor.

The following sections describe support for security in more detail.

TODO: this chapter needs more examples

13.1 Package *java.security*

The package *java.security* provides the generic Java security-related interfaces. Enterprise JavaBeans uses the applicable existing Java security APIs. This section describes the parts of the java.security API that are relevant to Enterprise JavaBeans.

13.1.1 class *java.security.Identity*

The *java.security.Identity* class encapsulates the concept of "user identity" for security purposes. Please refer to the reference page of *java.security.Identity* for the description.

13.2 Security-related methods in *InstanceContext*

The InstanceContext interface contains the following security-related method:

- *getCallerIdentity*
- *hasRole*

Please refer to reference page of *java.ejb.InstanceContext* for the description of this method.

13.3 Security-related deployment descriptor properties

An enterprise bean's deployment descriptor allows a container to perform security management outside of the enterprise bean code. The security management of an enterprise bean is determined by the bean's security descriptor. Please see the reference page for *java.ejb.deployment.SecurityDescriptor*.

13.4 Examples

The following example illustrates programmatic access to the security information.

13.4.1 Obtain client's Identity

```
/* Obtain the security identity of the client. */
Identity caller = instanceContext.getCallerIdentity();

/* getName returns a printable representation of identity. */
String clientAccount = caller.getName();
```

13.4.2 Check client's role

```
/*
* Check if the client has the "vip-account" role
*/
Identity vipAccount = new Identity("vip-account");

if (instanceContext.isCallerInRole(vipAccount)) {
        do something;
} else {
        do something else;
}
```

14 Ejb-jar file

Enterprise JavaBeans defines the format for packaging of enterprise beans. The packaging format can be used both for distribution of individual enterprise beans as components, and for distribution of an entire server-side application built of multiple enterprise beans.

14.1 ejb-jar file

Enterprise beans are packaged for deployment in a standard Java Archive File called an *ejb-jar* file.

An ejb-jar file contains the enterprise beans' class files and their deployment descriptors. The ejb-jar file's manifest file identifies the enterprise beans that are included in the file.

14.2 Deployment descriptor

An enterprise bean provider must include a deployment descriptor for each enterprise bean. A deployment descriptor is a serialized instance of a *java.ejb.deployment.EntityDescriptor* or *java.ejb.deployment.SessionDescriptor* object. Please refer to the reference pages for information on deployment descriptors.

14.3 ejb-jar Manifest

An ejb-jar file must include a *manifest file*. The manifest file identifies the enterprise beans included in the ejb-jar file.

The manifest file must be named "META-INF/MANIFEST.MF".

The manifest file is organized as a sequence of *sections*. Sections are separated by empty lines. Each section contains one or more *headers*, each of the form *<tag>: <value>*. The sections that provide information on enterprise beans in the archive use headers with the following *<tags>*:

- **Name**, whose *<value>* is the relative name of the enterprise bean's serialized deployment descriptor.
- **Enterprise-Bean**, whose *<value>* is **True**.

Every enterprise bean must have a section in the manifest file. The headers with the **Name** and **Enterprise-Bean** *<tags>* are mandatory for all enterprise beans.

For example, two relevant sections of an ejb-jar manifest might be:

```
Name: bank/AccountDeployment.ser
Enterprise-Bean: True

Name: quotes/QuoteServerDeployment.ser
Enterprise-Bean: True
```

15 Enterprise bean provider responsibilities

15.1 Classes and interfaces

The enterprise bean provider is responsible for providing the class files for the following classes and interface:

- Enterprise bean class.
- Enterprise bean's remote interface.
- Enterprise bean's factory interface.
- Enterprise bean's finder interface.

The enterprise bean provider must provide the enterprise bean class and the enterprise bean's remote interface for every enterprise bean.

The enterprise bean provider must also provide the factory and finder interface for every enterprise bean, with several exceptions that are noted below.

15.1.1 Enterprise bean class

The enterprise bean provider must provide an enterprise bean class for every enterprise bean. An enterprise bean class implements the business logic.

The following are the requirements for an enterprise bean class:

- An enterprise bean class must provide an implementation of a session or entity enterprise bean.
- A session enterprise bean class must implement the *java.ejb.SessionBean* interface. The implementation of the *java.ejb.SessionBean* methods must follow the rules described in Subsection 6.6.
- A session enterprise bean class can optionally implement the *java.ejb.SessionSynchronization* interface. The implementation of the *java.ejb.SessionSynchronization* methods must follow the rules described in Subsection 6.6.
- An entity enterprise bean class must implement the *java.ejb.EntityBean* interface. The implementation of the *java.ejb.EntityBean* methods must follow the rules described in Subsection 9.6.1.
- An enterprise bean must not be an abstract class.
- An enterprise bean class must define zero or more *ejbCreate(...)* methods, as described in Section 9.6.
- An enterprise bean class must not implement its associated remote interface.
- An enterprise bean class must define the implementation of the business methods defined in the enterprise bean's remote interface, as described in Section 9.5.

- The implementation of the enterprise bean's methods must follow the programming restrictions defined in Section 15.4.

15.1.2 Enterprise bean's remote interface

The enterprise bean provider must provide a remote interface for every enterprise bean. An enterprise bean's remote interface defines the business methods that are callable by clients.

The following are the requirements for an enterprise bean's remote interface:

- An enterprise bean remote interface must extend the *java.ejb.EJBObject* interface.
- An enterprise bean remote interface must follow all the standard rules for Java remote interfaces (e.g. *throws* clause of each enterprise bean remote interface method must include the *java.rmi.RemoteException*).
- The types used for the arguments and results of the enterprise bean's remote interface methods must be restricted to the subset of Java RMI described in [5].

15.1.3 Enterprise bean's factory interface

An enterprise bean's factory interface defines the *create(...)* methods used by clients to create new EJB objects.

The enterprise bean provider must provide a factory interface for every enterprise bean, with one exception. It is legal not to provide a factory interface for an entity enterprise bean if the enterprise bean provider wishes to restrict the clients from creating new entity EJB objects (for example, when the entities are stored in a read-only data source).

The following are the requirements for an enterprise bean's remote interface:

- An enterprise bean factory interface must extend the *java.ejb.Factory* interface.
- All the factory interface methods must be named *create(...)*.
- The *create(...)* methods must match the enterprise bean's *ejbCreate(...)* methods using the design pattern described in Section 9.6.
- An enterprise bean factory interface must follow all the standard rules for Java remote interfaces (e.g. *throws* clause of each enterprise bean factory interface method must include the *java.rmi.RemoteException*).
- The types used for the arguments and results of the enterprise bean's factory interface methods must be restricted to the subset of Java RMI described in [5].

15.1.4 Enterprise bean's finder interface

TODO:

15.2 Environment properties

If the enterprise bean depends on some environment properties, the enterprise bean provider must provide the environment properties for the bean. Environment properties are defined as a standard *java.util.Properties* object.

The enterprise bean provider must define file all the *key:value* pairs that the enterprise bean's instances will require at runtime. The values are typically edited at deployment time.

15.3 Deployment descriptor

The enterprise bean provider must provide a deployment descriptor for every enterprise bean. The format of a deployment descriptor is described in Section 15.3.

15.4 Programming restrictions

NOTE: this is still only a partial list

The following rules must be followed by a programmer developing an enterprise bean class:

- An enterprise bean is not allowed to start new threads or attempt to terminate the running thread.
- An enterprise bean is not allowed to use read/write *static* fields. Using read-only *static* fields is allowed. Therefore, all *static* fields must be declared as *final*.
- An enterprise bean is not allowed to use thread synchronization primitives.
- An enterprise bean is not allowed to use the JTS interfaces directly. The only exception are enterprise beans with the *BEAN_MANAGED* transaction attribute which are allowed to use the *java.jts.CurrentTransaction* interface to demarcate transactions.
- An enterprise bean is not allowed to change its *java.security.Identity*. Any such attempt will result in the *java.security.SecurityException* being thrown.
- A transaction-enabled enterprise bean using JDBC is not allowed to use the *commit* and *rollback* methods. An enterprise bean that is not transaction-enabled is allowed to use the *commit* and *rollback* methods.

15.5 Component packaging responsibilities

The enterprise bean provider is responsible for putting the following classes and files in the ejb-jar file:

- The enterprise bean class with any classes that the enterprise bean depends on.
- The deployment descriptor file that contains the deployment attributes for the enterprise bean.
- The factory and finder interfaces if they are required by the enterprise bean.
- Enterprise bean's environment properties.
- The Manifest file that identifies the deployment descriptors of all the enterprise beans in the ejb-jar file.

16 Container provider responsibilities

XXX - this chapter is still work in progress

17 Enterprise JavaBeans API Reference

The following interfaces and classes comprise the Enterprise JavaBeans API:

package *java.ejb*:

Interfaces:

```
public interface Container
public interface ContainerMetaData
public interface EJBObject
public interface EnterpriseBean
public interface EntityBean
public interface EntityContext
public interface Factory
public interface Finder
public interface Handle
public interface InstanceContext
public interface SessionBean
public interface SessionContext
public interface SessionSynchronization
```

Classes:

```
public class BeanPermission
public class InvalidKeyException
public class NoObjectWithKeyException
public class NotDestroyableException
```

package *java.ejb.deployment*:

Classes:

```
public class DeploymentDescriptor
public class EntityDescriptor
public class MethodDescriptor
public class MethodDescriptor
public class SessionDescriptor
public class TransactionAttribute
```

Interface Container

```
public interface java.ejb.Container
    extends java.rmi.Remote
{
    public abstract void
        destroy(Handle handle);
    public abstract void
        destroy(Object primaryKey);
    public abstract ContainerMetaData
        getContainerMetaData();
    public abstract Factory getFactory();
    public abstract Finder getFinder();
}
```

The Container interface provides client's access to the EJB object's lifecycle operations. All EJB containers must implement this interface.

The Container interface allows a client to obtain the enterprise bean's factory and finder interfaces, and to destroy an existing EJB object.

Methods

- **destroy**

```
public abstract void destroy(Handle handle)
     throws RemoteException, NotDestroyableException
```

Destroy an EJB object identified by its handle.

Throws: NotDestroyableException
Thrown if the container does not allow the client to destroy the object.

Throws: RemoteException
Thrown when the method failed due to a system-level failure.

- **destroy**

```
public abstract void destroy(Object primaryKey)
     throws RemoteException, NotDestroyableException
```

Destroy an EJB object identified by its primary key.

Throws: NotDestroyableException
Thrown if the container does not allow the client to destroy the object.

Throws: RemoteException
Thrown when the method failed due to a system-level failure.

- **getContainerMetaData**

```
public abstract ContainerMetaData
getContainerMetaData()
     throws RemoteException
```

Obtain the ContainerMetaData interface that allows the client to get the various metadata associated with the container.

Returns:
The ContainerMetaData interface for the container.

Throws: RemoteException
Thrown when the method failed due to a system-level failure.

- **`getFactory`**

```
public abstract Factory getFactory()
     throws RemoteException
```

Get the enterprise bean factory associated with the container.

Returns:
The enterprise bean factory interface, or null if the container does not provide a factory for its enterprise beans.

- **`getFinder`**

```
public abstract Finder getFinder()
     throws RemoteException
```

Get enterprise bean finder associated with the container.

Returns:
The enterprise bean finder interface, or null if the container does not provide a finder for its enterprise beans.

Throws: RemoteException
Thrown when the method failed due to a system-level failure.

Interface ContainerMetaData

```
public interface java.ejb.ContainerMetaData
    extends java.rmi.Remote
{
    public abstract String getClassName();
    public abstract Container getContainer();
    public abstract Class
        getPrimaryKeyClass();
}
```

The ContainerMetaData interface allows a client to obtain the various metadata associated with a container.

TODO: the definition of this interface is still in-progress

Methods

- **getClassName**

```
public abstract String getClassName()
     throws RemoteException
```

Obtain the class name of the enterprise bean that this factory interface is associated with.

Throws: RemoteException
Thrown when the method failed due to a system-level failure.

- **getContainer**

```
public abstract Container getContainer()
     throws RemoteException
```

Obtain the container associated with this factory.

- **getPrimaryKeyClass**

```
public abstract Class getPrimaryKeyClass()
     throws RemoteException
```

Obtain the Class object for the primary key class.

Throws: RemoteException
Thrown when the method failed due to a system-level failure.

Interface EJBObject

```
public interface java.ejb.EJBObject
    extends java.rmi.Remote
{
    public abstract void destroy();
    public abstract String getClassName();
    public abstract Container getContainer();
    public abstract Handle getHandle();
    public abstract Object getPrimaryKey();
    public abstract boolean
        isIdentical(EJBObject bean);
}
```

The EJBObject interface provides the client view of an EJB object.

Methods

- **destroy**

  ```
  public abstract void destroy()
       throws RemoteException, NotDestroyableException
  ```

  Destroy the EJB object.

  Throws: NotDestroyableException
  The container does not allow the client to destroy the EJB object.

  Throws: RemoteException
  Thrown when the method failed due to a system-level failure.

- **getClassName**

  ```
  public abstract String getClassName()
       throws RemoteException
  ```

  Obtain the class name of the enterprise bean that provides the implementation of this EJB object.

  Returns:
  The class name of the enterprise bean that provides the implementation of this EJB object.

  Throws: RemoteException
  Thrown when the method failed due to a system-level failure.

- **getContainer**

  ```
  public abstract Container getContainer()
       throws RemoteException
  ```

  Obtain a reference to the EJB object's container.

  Returns:
  A reference to this object's container.

  Throws: RemoteException
  Thrown when the method failed due to a system-level failure.

- **`getHandle`**

  ```
  public abstract Handle getHandle()
       throws RemoteException
  ```

  Obtain a handle for this EJB object. A handle can be used at later time to re-obtain a reference to the EJB object.

  Returns:
  A handle for the EJB object.

  Throws: RemoteException
  Thrown when the method failed due to a system-level failure.

- **`getPrimaryKey`**

  ```
  public abstract Object getPrimaryKey()
       throws RemoteException
  ```

  Obtain the primary key of the EJB object.

- **`isIdentical`**

  ```
  public abstract boolean isIdentical(EJBObject bean)
       throws RemoteException
  ```

  Test if a given EJB object is identical to the invoked EJB object.

  Returns:
  True if the given EJB object is identical to the invoked object, false otherwise.

  Throws: RemoteException
  Thrown when the method failed due to a system-level failure.

Interface EnterpriseBean

```
public interface java.ejb.EnterpriseBean
    extends java.io.Serializable
{
}
```

The EnterpriseBean interface is an interface that every enterprise bean class must implement.

Interface EntityBean

```
public interface java.ejb.EntityBean
    extends java.ejb.EnterpriseBean
{
    public abstract void ejbActivate();
    public abstract void ejbDestroy();
    public abstract void ejbLoad();
    public abstract void ejbPassivate();
    public abstract void ejbStore();
    public abstract void
        setEntityContext(EntityContext ctx);
    public abstract void
        unsetEntityContext();
}
```

The EntityBean interface is implemented by every enterprise bean class that provides the implementation for an entity EJB object. The methods defined in the EntityBean interface are invoked by the container to allow an enterprise bean instance to participate in its lifecycle management.

Note: Support for entity enterprise beans is optional for EJB 1.0 compliant containers. Support for entities will become mandatory for EJB 2.0 compliant containers.

Methods

- **ejbActivate**

```
public abstract void ejbActivate()
     throws Exception
```

A container invokes this method on the instance when the instance is taken out of the pool of available instances to become associated with a specific EJB object. This method transitions the instance to the ready state.

This method executes in an unspecified transaction context.

Throws: Exception
The implementation of this method can throw an arbitrary exception. The container is responsible for catching and handling the exception.

- **ejbDestroy**

```
public abstract void ejbDestroy()
     throws Exception
```

A container invokes this method before it ends the life of the EJB object that is currrently associated with the instance. This method is invoked when a client invokes a destroy operation. This method transitions the instance from the ready state to the pool of available instances.

This method is called in the proper transaction context.

Throws: Exception
The implementation of this method can throw an arbitrary exception. The container is responsible for catching and handling the exception.

- **ejbLoad**

```
public abstract void ejbLoad()
     throws Exception
```

A container invokes this method on the instance to instruct the instance to synchronize its state by loading it state from the underlying database.

This method executes in the proper transaction context.

Throws: Exception
The implementation of this method can throw an arbitrary exception. The container is responsible for catching and handling the exception.

- **ejbPassivate**

```
public abstract void ejbPassivate()
     throws Exception
```

A container invokes this method on instance before the instance becomes disassociated with a specific EJB object. After this method completes, the container will place the instance into the pool of available instances.

This method executes in an unspecified transaction context.

Throws: Exception
The implementation of this method can throw an arbitrary exception. The container is responsible for catching and handling the exception.

- **ejbStore**

```
public abstract void ejbStore()
     throws Exception
```

A container invokes this method on the instance to instruct the instance to synchronize its state by storing it to the underlying database.

This method executes in the proper transaction context.

Throws: Exception
The implementation of this method can throw an arbitrary exception. The container is responsible for catching and handling the exception.

- **setEntityContext**

```
public abstract void
setEntityContext(EntityContext ctx)
     throws Exception
```

Set the associated entity context. The container calls this method on the instance after the instance creation before the instance is is entered into the pool of available instances.

The EJB instance should store the reference to the context object in an instance variable.

This method is called in unspecified or no transaction context.

Parameters:
`ctx`
An EntityContext interface for the instance.

Throws: Exception
The implementation of this method can throw an arbitrary exception. The container is responsible for catching and handling the exception.

- **`unsetEntityContext`**

  ```
  public abstract void unsetEntityContext()
       throws Exception
  ```

  Unset the associated entity context. The container calls this method after the instance is removed from the pool of available instances. The container will then discard the instance.

  This is the last method that the container invokes on the instance. The Java garbage collector will eventually invoke the finalize() method on the instance.

  This method is called in unspecified or no transaction context.

  Throws: Exception
  The implementation of this method can throw an arbitrary exception. The container is responsible for catching and handling the exception.

Interface EntityContext

```
public interface java.ejb.EntityContext
    extends java.ejb.InstanceContext
{
    public abstract EJBObject getEJBObject();
    public abstract EJBObject
        getEJBObject(Object primaryKey);
    public abstract Object getPrimaryKey();
    public abstract void
        setPrimaryKey(Object primaryKey);
}
```

The EntityContext interface provides access to the runtime context that the container provides for an entity enterprise bean instance. The container passes the EntityContext interface to an entity enterprise bean instance after the instance has been created. The entity context remains associated with the instance for the lifetime of the instance.

Methods

- **getEJBObject**

```
public abstract EJBObject getEJBObject()
     throws IllegalStateException
```

Obtain a reference to the EJB object that is currently associated with the instance.

An instance of an entity enterprise bean can call this method only when the instance is in the ready state; from the ejbActivate(), ejbPassivate(), and ejbDestroy() methods; and from the ejbCreate(...) methods after the instance has called the set primary key method on the InstanceContext interface.

An instance can use this method, for example, when it wants to pass a reference to itself in a method argument or result.

Returns:
The EJB object currently associated with the instance.

Throws: IllegalStateException
Thrown if the instance invokes this method while the instance is in a state that does not allow the instance to invoke this method.

- **getEJBObject**

```
public abstract EJBObject
getEJBObject(Object primaryKey)
```

Create an EJB object reference for a given primary key. An instance uses this method, for example, when it needs to create an EJB object reference to another entity that lives in the same container, in order to pass the EJB object reference as a method argument or result.

Parameters:
`primaryKey`
The primary key for which an EJB object reference should be created.

Returns:
An EJB object reference for the given primary key.

- **`getPrimaryKey`**

```
public abstract Object getPrimaryKey()
     throws IllegalStateException
```

Obtain the primary key of the EJB object that is currently associated with this instance.

An entity enterprise bean instance can call this method only when the instance is in the ready state; from the ejbActivate(), ejbPassivate(), and ejbDestroy() methods; and from the ejbCreate(...) methods after the instance has called the set primary key method on the InstanceContext interface.

Note: The result of this method is that same as the the result of getEJBObject().getPrimaryKey().

Returns:
The EJB object currently associated with the instance.

Throws: IllegalStateException
Thrown if the instance invokes this method while the instance is in a state that does not allow the instance to invoke this method.

- **`setPrimaryKey`**

```
public abstract void
setPrimaryKey(Object primaryKey)
     throws IllegalStateException
```

Associate the primary key with the EJB object that is being currently created. This method is callable only from the ejbCreate(...) methods of an entity enterprise bean class.

Throws: IllegalStateException
Thrown if the instance invokes this method from a method other than ejbCreate(...).

Interface Factory

```
public interface java.ejb.Factory
    extends java.rmi.Remote
{
    public abstract String getClassName();
    public abstract Container getContainer();
}
```

The Factory interface is a base interface of all EJB factory interfaces.

Methods

- **getClassName**

```
public abstract String getClassName()
     throws RemoteException
```

Obtain the class name of the enterprise bean that provides the implementation for the EJB objects created by this factory.

Returns:
The class name of the enterprise bean that provides the implementation of the EJB object created by this factory.

Throws: RemoteException
Thrown when the method failed due to a system-level failure.

- **getContainer**

```
public abstract Container getContainer()
     throws RemoteException
```

Obtain the container associated with this factory.

Returns:
A reference to the container associated with this factory.

Throws: RemoteException
Thrown when the method failed due to a system-level failure.

Interface Finder

```
public interface java.ejb.Finder
    extends java.rmi.Remote
{
    public abstract EJBObject
        findByPrimaryKey(Object primaryKey);
    public abstract String getClassName();
    public abstract Container getContainer();
}
```

The Finder interface is a base interface of all EJB finder interfaces.

Methods

- **findByPrimaryKey**

```
public abstract EJBObject
findByPrimaryKey(Object primaryKey)
     throws RemoteException, InvalidKeyException, NoObjectWithKeyEx-
ception
```

Find an EJB object by its primary key.

Parameters:

primaryKey
The primary key by which to look up the EJB object.

Returns:
The looked up EJB object.

Throws: InvalidKeyException
The specified primary key had invalid format.

Throws: NoObjectWithKeyException
The EJB object with the given primary key does not exist.

Throws: RemoteException
Thrown when the method failed due to a system-level failure.

- **getClassName**

```
public abstract String getClassName()
     throws RemoteException
```

Obtain the class name of the enterprise bean that provides the implementation for the EJB objects found by this finder.

Returns:
The class name of the enterprise bean that provides the implementation of the EJB object found by this finder.

Throws: RemoteException
Thrown when the method failed due to a system-level failure.

- **getContainer**

```
public abstract Container getContainer()
```

```
    throws RemoteException
```

Obtain the container associated with this finder.

Returns:
A reference to the container associated with this finder.

Throws: RemoteException
Thrown when the method failed due to a system-level failure.

Interface Handle

```
public interface java.ejb.Handle
    extends java.io.Serializable
{
    public abstract EJBObject getEJBObject();
}
```

The Handle is an abstraction of a network reference to an EJB object. It should be used as a "robust" persistent reference to an EJB object.

Methods

- **`getEJBObject`**

  ```
  public abstract EJBObject getEJBObject()
  ```

  Obtain the EJB object represented by this handle.

Interface InstanceContext

```
public interface java.ejb.InstanceContext
{
    public abstract Identity
        getCallerIdentity();
    public abstract Container getContainer();
    public abstract CurrentTransaction
        getCurrentTransaction();
    public abstract Properties
        getEnvironment();
    public abstract boolean
        isCallerInRole(Identity role);
}
```

The InstanceContext interface provides access to the runtime context that a container provides for an enterprise bean instance. A container passes the InstanceContext interface to an instance after the instance has been created. The instance context remains associated with the instance for the lifetime of the instance.

This interface is extended by the SessionContext and EntityContext interface which provide additional methods specific to the enterprise bean type.

Methods

- **getCallerIdentity**

  ```
  public abstract Identity getCallerIdentity()
  ```

  Obtain the security Identity of the immediate caller.

  Returns:
  The Identity object that identifies the immediate caller.

- **getContainer**

  ```
  public abstract Container getContainer()
  ```

  Obtain a reference to the enterprise bean's container.

  Returns:
  A reference to the enterprise bean's container.

- **getCurrentTransaction**

  ```
  public abstract CurrentTransaction
  getCurrentTransaction()
       throws IllegalStateException
  ```

  Obtain the transaction demaraction interface.

  Returns:
  The CurrentTransaction interface that the enterprise bean instance can use for transaction demarcation.

  Throws: IllegalStateException
  Thrown if the instance container does not make the CurrentTransaction interface available to the instance because the instance is not deployed with the NotSupported transaction attribute.

- **`getEnvironment`**

  ```
  public abstract Properties getEnvironment()
  ```

  Obtain the ServerBean's environment Properties.

  The method returns the enterprise bean's environment properties.

  If no environment properties were specified for the enterprise bean, this method returns an empty Properties object. This method never returns null.

  Returns:

  The environment properties for the enterprise bean.

- **`isCallerInRole`**

  ```
  public abstract boolean
  isCallerInRole(Identity role)
  ```

  Test if the caller has a given role.

  Parameters:

  `role`

  The Identity of the role to be tested.

  Returns:

  True if the caller has the specified role.

Interface SessionBean

```
public interface java.ejb.SessionBean
    extends java.ejb.EnterpriseBean
{
    public abstract void ejbActivate();
    public abstract void ejbDestroy();
    public abstract void ejbPassivate();
    public abstract void
        setSessionContext(SessionContext ctx);
}
```

The SessionBean interface is implemented by every enterprise bean class that provides the implementation for a session EJB object. The methods defined in the SessionBean interface are invoked by the container to allow an enterprise bean instance to participate in its lifecycle and transaction management.

Methods

- **ejbActivate**

```
public abstract void ejbActivate()
     throws Exception
```

The activate method is called when the instance is activated from its "passive" state. The instance should acquire any resource that it has released earlier in the ejbPassivate() method.

Throws: Exception
The implementation of this method can throw an arbitrary exception. The container is responsible for catching and handling the exception.

- **ejbDestroy**

```
public abstract void ejbDestroy()
     throws Exception
```

A container invokes this method before it ends the life of the session object. This happens as a result of a client's invoking a destroy operation, or when a container decides to terminate the session object after a timeout.

This method is called in the proper transaction context.

Throws: Exception
The implementation of this method can throw an arbitrary exception. The container is responsible for catching and handling the exception.

- **ejbPassivate**

```
public abstract void ejbPassivate()
     throws Exception
```

The passivate method is called before the instance enters the "passive" state. The instance should release any resources that that it can re-acquire later in the ejbActivate() method.

After the passivate method completes, the instance must be in a state that allows the container to use the Java Serialization protocol to externalize and store away the instance's state.

Throws: Exception

The implementation of this method can throw an arbitrary exception. The container is responsible for catching and handling the exception.

- **`setSessionContext`**

```
public abstract void
setSessionContext(SessionContext ctx)
     throws Exception
```

Set the associated session context. The container calls this method after the instance creation.

The enterprise bean instance should store the reference to the context object in an instance variable.

This method is called in unspecified or no transaction context.

Parameters:

`ctx`
An SessionContext interface for the instance.

Throws: Exception
The implementation of this method can throw an arbitrary exception. The container is responsible for catching and handling the exception.

Interface SessionContext

```
public interface java.ejb.SessionContext
    extends java.ejb.InstanceContext
{
    public abstract EJBObject getEJBObject();
}
```

The SessionContext interface provides access to the runtime session context that the container provides for a session enterprise bean instance. The container passes the SessionContext interface to an instance after the instance has been created. The session context remains associated with the instance for the lifetime of the instance.

Methods

- **getEJBObject**

```
public abstract EJBObject getEJBObject()
     throws IllegalStateException
```

Obtain a reference to the EJB object that is currently associated with the instance.

An instance of a session enterprise bean can call this method at anytime between the ejbCreate() and ejbDestroy() methods, including from within the ejbCreate() and ejbDestroy() methods.

An instance can use this method, for example, when it wants to pass a reference to itself in a method argument or result.

Returns:
The EJB object currently associated with the instance.

Throws: IllegalStateException
Thrown if the instance invokes this method while the instance is in a state that does not allow the instance to invoke this method.

Interface SessionSynchronization

```
public interface java.ejb.SessionSynchronization
{
    public abstract void
        afterCompletion(boolean committed);
    public abstract void beforeCompletion();
    public abstract void beginTransaction();
}
```

The SessionSynchronization interface allows a session bean to be notified by its container of transaction boundaries.

An session bean class is not required to implement this interface. A session bean class should implement this interface only if it wishes to synchronize its state with the transactions.

Methods

- **afterCompletion**

```
public abstract void
afterCompletion(boolean committed)
     throws Exception
```

- **beforeCompletion**

```
public abstract void beforeCompletion()
     throws Exception
```

The beforeCompletions method notifies a session bean instance that a transaction is about to be committed. The instance can use this method, for example, to write any cached data to a database.

This method executes in the proper transaction context.

Throws: Exception
The implementation of this method can throw an arbitrary exception. The container is responsible for catching and handling the exception.

- **beginTransaction**

```
public abstract void beginTransaction()
     throws Exception
```

The beginTransaction method notifies a session bean instance that the next method invocation will be in the context of a new transaction.

The instance can use this method, for example, to read data from a database and cache the data in the instance fields.

This method executes in the proper transaction context.

Throws: Exception
The implementation of this method can throw an arbitrary exception. The container is responsible for catching and handling the exception.

Class BeanPermission

```
public final class java.ejb.BeanPermission
    extends java.security.BasicPermission
{
    public BeanPermission(String actions);
}
```

The class BeanPermission represent a permission to perform specified operations on an enterprise bean. A BeanPermission consists of one or more comma-separated actions. The possible actions are "create", "destroy", "invoke", "invoke:method_name", and "manage". Their meaning is defined as follows:

The "create" permission controls the Identities that can create a enterprise bean object.

The "destroy" permission controls the Identities that can destroy a enterprise bean object.

The "invoke" permission controls the Identities that can invoke all methods on a enterprise bean object.

The "invoke:method_name" permission controls the Identities that can invoke a specific method on a enterprise bean object. The string *method_name* is the name of the enterprise bean method.

The "manage" permission controls the Identities that can modify a enterprise bean's deployment descriptor. A BeanPermission permission is intended to be used in the ACList of the deployment descriptor of an enterprise bean.

Constructors

- **BeanPermission**

  ```
  public BeanPermission(String actions)
  ```

Class InvalidKeyException

```
public class java.ejb.InvalidKeyException
    extends java.lang.Exception
{
    public InvalidKeyException();
    public
        InvalidKeyException(String message);
}
```

The InvalidKeyException exception is thrown when an EJB object is being looked up by a key, and the specified key is invalid. A key can be invalid because its class does not match, or the fields of the key have invalid values.

Constructors

- **InvalidKeyException**

  ```
  public InvalidKeyException()
  ```

  Constructs an InvalidKeyException with no detail message.

- **InvalidKeyException**

  ```
  public InvalidKeyException(String message)
  ```

  Constructs an InvalidKeyException with the specified detail message.

Class NoObjectWithKeyException

```
public class java.ejb.NoObjectWithKeyException
    extends java.lang.Exception
{
    public NoObjectWithKeyException();
    public
        NoObjectWithKeyException(String message);
}
```

The NoObjectWithKeyException exception is thrown when an object is being looked up by a key, and the object with the specified key does not exist.

Constructors

- **NoObjectWithKeyException**

  ```
  public NoObjectWithKeyException()
  ```

  Constructs an NoObjectWithKeyException with no detail message.

- **NoObjectWithKeyException**

  ```
  public NoObjectWithKeyException(String message)
  ```

  Constructs an NoObjectWithKeyException with the specified detail message.

Class NotDestroyableException

```
public class java.ejb.NotDestroyableException
    extends java.lang.Exception
{
    public NotDestroyableException();
    public
        NotDestroyableException(String message);
}
```

The NotDestroyableException exception is thrown at an attempt to destroy an EJB object that is in a state that does not allow destruction.

Constructors

- **NotDestroyableException**

  ```
  public NotDestroyableException()
  ```

  Constructs an NotDestroyableException with no detail message.

- **NotDestroyableException**

  ```
  public NotDestroyableException(String message)
  ```

  Constructs an NotDestroyableException with the specified detail message.

Class DeploymentDescriptor

```
public class java.ejb.deployment.DeploymentDescriptor
    extends java.lang.Object
    implements java.io.Serializable
{
    public DeploymentDescriptor();
    public Name getContainerName();
    public Class getEnterpriseBeanClass();
    public Properties
        getEnvironmentProperties();
    public Class getFactoryInterface();
    public MethodDescriptor[]
        getMethodDescriptors();
    public MethodDescriptor
        getMethodDescriptors(int index);
    public Class getRemoteInterface();
    public SecurityDescriptor
        getSecurityDescriptor();
    public int getTransactionAttribute();
    public void setContainerName(Name value);
    public void
        setEnterpriseBeanClass(Class value);
    public void
        setEnvironmentProperties(Properties value);
    public void
        setFactoryInterface(Class value);
    public void
        setMethodDescriptors(int index,
                             MethodDescriptor value);
    public void
        setMethodDescriptors(MethodDescriptor value[]);
    public void
        setRemoteInterface(Class value);
    public void
        setSecurityDescriptor(SecurityDescriptor value);
    public void
        setTransactionAttribute(int value);
}
```

The DeploymentDescriptor class is the common baseclass for the SessionDescriptor and EntityDescriptor deployment descriptor classes.

A serialized instance of a deployment descriptor class is used as the standard way for passing an enterprise bean's declarative attributes in the ejb-jar file. The deployment descriptor setter functions are used by the enterprise bean provider's tools to create the deployment descriptor before it is put into the ejb-jar file, and the getter functions are used by the container provider's tools to read the deployment descriptor from the ejb-jar file when the enterprise bean is installed into the container.

Constructors

- **DeploymentDescriptor**

  ```
  public DeploymentDescriptor()
  ```

Create an instance of the deployment descriptor.

Methods

- **`getContainerName`**

  ```
  public Name getContainerName()
  ```

  Get the name to associate with the enterprise bean's container in the JNDI name space.

  Returns:
  The container's name in the JNDI name space.

- **`getEnterpriseBeanClass`**

  ```
  public Class getEnterpriseBeanClass()
  ```

  Get the enterprise bean's class.

  Returns:
  The Class object for the enterprise bean's class.

- **`getEnvironmentProperties`**

  ```
  public Properties getEnvironmentProperties()
  ```

  Get enterprise bean's environment properties.

  Returns:
  Enterprise bean's environment properties.

- **`getFactoryInterface`**

  ```
  public Class getFactoryInterface()
  ```

  Get the enterprise bean's remote factory.

  Returns:
  The Class object for the enterprise bean's factory.

- **`getMethodDescriptors`**

  ```
  public MethodDescriptor[] getMethodDescriptors()
  ```

  Get the array of the enterprise bean's method descriptors.

  Returns:
  An array of enterprise bean's method descriptors, or null of the enterprise bean does not provide method-level descriptors.

- **`getMethodDescriptors`**

  ```
  public MethodDescriptor
  getMethodDescriptors(int index)
  ```

  Get the method descriptor at a given index.

  Parameters:
  `index`
  The index of the method descriptor.

Returns:
The method descriptor at the specified index.

- **`getRemoteInterface`**

  `public Class getRemoteInterface()`

  Get the enterprise bean's remote interface.

  Returns:
  The Class object for the enterprise bean's remote interface.

- **`getSecurityDescriptor`**

  `public SecurityDescriptor getSecurityDescriptor()`

  Get the enterprise bean's security descriptor.

  Returns:
  The enterprise bean's security descriptor.

- **`getTransactionAttribute`**

  `public int getTransactionAttribute()`

  Get the enterprise bean's transaction attribute. The values of the transaction attribute are defined in the class TransactionAttribute.

  Returns:
  The enterprise bean's transaction attribute.

- **`setContainerName`**

  `public void setContainerName(Name value)`

  Set the name to associate with the enterprise bean's container in the JNDI name space.

  Parameters:
  `value`
  The container's name in the JNDI name space.

- **`setEnterpriseBeanClass`**

  `public void setEnterpriseBeanClass(Class value)`

  Set the enterprise bean's class.

  Parameters:
  `value`
  The Class object for the enterprise bean's class.

- **`setEnvironmentProperties`**

  ```
  public void
  setEnvironmentProperties(Properties value)
  ```

  Set enterprise bean's environment properties.

  Parameters:
  `value`
  Enterprise bean's environment properties.

- **setFactoryInterface**

```
public void setFactoryInterface(Class value)
```

Set the enterprise bean's remote factory.

Parameters:

`value`
The Class object for the enterprise beans factory.

- **setMethodDescriptors**

```
public void
setMethodDescriptors(int index,
                     MethodDescriptor value)
```

Set the method descriptor at a given index.

Parameters:

`index`
The index of the method descriptor.
`value`
The method descriptor to be set at the specified index.

- **setMethodDescriptors**

```
public void
setMethodDescriptors(MethodDescriptor value[])
```

Set the array of the enterprise bean's method descriptors.

A deployment descriptor may define zero or several MethodDescriptors. If a MethodDescriptor is defined for a method, the MethodDescriptor overrides the values of the transactionAttribute and securityDescriptor set at the bean level.

A MethodDescriptor can be provided for the methods defined in the enterprise bean's remote interface, factory interface, finder interface, and the java.ejb.Container interface.

Parameters:

`value`
An array of the enterprise bean's method descriptors.

- **setRemoteInterface**

```
public void setRemoteInterface(Class value)
```

Set the enterprise bean's remote interface.

Parameters:

`value`
The Class object for the enterprise bean's remote interface.

- **setSecurityDescriptor**

```
public void
setSecurityDescriptor(SecurityDescriptor value)
```

Set the enterprise bean's security descriptor.

The SecurityDescriptor applies to all methods of the enterprise bean, unless it is overridden by a MethodDescriptor.

Parameters:

`value`

The enterprise bean's security descriptor.

- **`setTransactionAttribute`**

  `public void setTransactionAttribute(int value)`

  Set the enterprise bean's transaction attribute. The values of the transaction attribute are defined in the class TransactionAttribute.

  The transaction attribute applies to all methods of the enterprise bean, unless it is overridden by a Method-Descriptor.

  Parameters:

  `value`

  The enterprise bean's transaction attribute.

Class EntityDescriptor

```
public class java.ejb.deployment.EntityDescriptor
    extends java.ejb.deployment.DeploymentDescriptor
{
    public EntityDescriptor();
    public Field[]
        getContainerManagedFields();
    public Field
        getContainerManagedFields(int index);
    public Class getFinderInterface();
    public Class getPrimaryKeyClass();
    public void
        setContainerManagedFields(Field values[]);
    public void
        setContainerManagedFields(int index,
                                  Field value);
    public void
        setFinderInterface(Class value);
    public void
        setPrimaryKeyClass(Class value);
}
```

The SessionDescriptor class defines the deployment descriptor for an entity enterprise bean.

A serialized instance of a deployment descriptor class is used as the standard format for passing an enterprise bean's declarative attributes in the ejb-jar file. The deployment descriptor setter functions are used by the enterprise bean provider's tools to create the deployment descriptor before it is put into the ejb-jar file, and the getter functions are used by the container provider's tools to read the deployment descriptor from the ejb-jar file.

Constructors

- **EntityDescriptor**

  `public EntityDescriptor()`

  Create an instance of the deployment descriptor.

Methods

- **getContainerManagedFields**

  `public Field[] getContainerManagedFields()`

  Get the array of container-managed enterprise bean's fields.

  Returns:
  The array of container-managed fields.

- **getContainerManagedFields**

  `public Field getContainerManagedFields(int index)`

  Get the container-managed field at the given index.

Parameters:

`index`
The index of the field.

Returns:
The Field of the specified container-managed field.

- **`getFinderInterface`**

```
public Class getFinderInterface()
```

Get the enterprise bean's finder interface.

Returns:
The Class object for the finder interface.

- **`getPrimaryKeyClass`**

```
public Class getPrimaryKeyClass()
```

Get the enterprise bean's primary key class.

Returns:
The Class object for the primary key.

- **`setContainerManagedFields`**

```
public void
setContainerManagedFields(Field values[])
```

Set the array of container-managed enterprise bean's fields.

Parameters:

`value`
The array of container-managed fields.

- **`setContainerManagedFields`**

```
public void
setContainerManagedFields(int index, Field value)
```

Set the container-managed field at the given index.

Parameters:

`index`
The index of the field.
`value`
The Field of the specified container-managed field.

- **`setFinderInterface`**

```
public void setFinderInterface(Class value)
```

Set the enterprise bean's finder interface.

Parameters:

`value`
The Class object for the finder interface.

- **`setPrimaryKeyClass`**

```
public void setPrimaryKeyClass(Class value)
```

Get the enterprise bean's primary key class.

Parameters:

`value`
The Class object for the primary key.

Class MethodDescriptor

```
public class java.ejb.deployment.MethodDescriptor
    extends java.lang.Object
{
    public MethodDescriptor(Method method);
    public Method getMethod();
    public SecurityDescriptor
        getSecurityDescriptor();
    public int getTransactionAttribute();
    public void
        setSecurityDescriptor(SecurityDescriptor value);
    public void
        setTransactionAttribute(int value);
}
```

The MethodDescriptor is used to provide deployment information specific to a single method. The method-level information overrides any information set at the bean-level.

A MethodDescriptor can be provided for any method defined in the enterprise bean's remote interface, factory interface, finder interface, and java.ejb.Container interface. A Method descriptor shall not be provided for a method defined in the enterprise bean class (use a MethodDescriptor for the corresponding method of the enterprise bean's remote interface instead).

Constructors

- **MethodDescriptor**

  `public MethodDescriptor(Method method)`

  Construct a MethodDescriptor for a given Method.

Methods

- **getMethod**

  `public Method getMethod()`

  Get the Method to which this MethodDescriptor applies.

  Returns:
  The Method associated with this MethodDescriptor.

- **getSecurityDescriptor**

  `public SecurityDescriptor getSecurityDescriptor()`

  Get the SecurityDescriptor for the method.

  Returns:
  The SecurityDescriptor for the method.

- **getTransactionAttribute**

  `public int getTransactionAttribute()`

  Get the transaction attribute for the method.

Returns:
The transaction attribute for the method.

- **`setSecurityDescriptor`**

```
public void
setSecurityDescriptor(SecurityDescriptor value)
```

Set the SecurityDescriptor for the method.

Parameters:
`value`
The SecurityDescriptore for the method.

- **`setTransactionAttribute`**

```
public void setTransactionAttribute(int value)
```

Set the transaction attribute for the method.

Parameters:
`value`
The transaction attribute for the method.

Class MethodDescriptor

```
public class java.ejb.deployment.MethodDescriptor
    extends java.lang.Object
{
    public MethodDescriptor(Method method);
    public Method getMethod();
    public SecurityDescriptor
        getSecurityDescriptor();
    public int getTransactionAttribute();
    public void
        setSecurityDescriptor(SecurityDescriptor value);
    public void
        setTransactionAttribute(int value);
}
```

The MethodDescriptor is used to provide deployment information specific to a single method. The method-level information overrides any information set at the bean-level.

A MethodDescriptor can be provided for any method defined in the enterprise bean's remote interface, factory interface, finder interface, and java.ejb.Container interface. A Method descriptor shall not be provided for a method defined in the enterprise bean class (use a MethodDescriptor for the corresponding method of the enterprise bean's remote interface instead).

Constructors

- **MethodDescriptor**

  `public MethodDescriptor(Method method)`

  Construct a MethodDescriptor for a given Method.

Methods

- **getMethod**

  `public Method getMethod()`

  Get the Method to which this MethodDescriptor applies.

  Returns:
  The Method associated with this MethodDescriptor.

- **getSecurityDescriptor**

  `public SecurityDescriptor getSecurityDescriptor()`

  Get the SecurityDescriptor for the method.

  Returns:
  The SecurityDescriptor for the method.

- **getTransactionAttribute**

  `public int getTransactionAttribute()`

  Get the transaction attribute for the method.

Returns:
The transaction attribute for the method.

- **`setSecurityDescriptor`**

```
public void
setSecurityDescriptor(SecurityDescriptor value)
```

Set the SecurityDescriptor for the method.

Parameters:
`value`
The SecurityDescriptore for the method.

- **`setTransactionAttribute`**

```
public void setTransactionAttribute(int value)
```

Set the transaction attribute for the method.

Parameters:
`value`
The transaction attribute for the method.

Class SecurityDescriptor

```
public class java.ejb.deployment.SecurityDescriptor
    extends java.lang.Object
{
    public final static int Client;
    public final static int SpecifiedUser;
    public final static int System;
    public SecurityDescriptor();
    public Identity[] getGrantAccess();
    public Identity
        getGrantAccess(int index);
    public int getRunAsAttribute();
    public Identity getRunAsIdentity();
    public void
        setGrantAccess(Identity values[]);
    public void setGrantAccess(int index,
                               Identity value);
    public void setRunAsAttribute(int value);
    public void
        setRunAsIdentity(Identity value);
}
```

The SecurityDescriptor descriptor defines security-related attributes for enterprise bean. A SecurityDescriptor can be used both at the level of the entire bean, or the level of individual methods.

Variables

- **Client**

  ```
  public final static int Client
  ```

  Run the enterprise bean method with the client's Identity.

- **SpecifiedUser**

  ```
  public final static int SpecifiedUser
  ```

  Run the enterprise bean method with the Identity of a specified user account.

- **System**

  ```
  public final static int System
  ```

  Run the enterprise bean method with the Identity of a "privileged account". The container maps the abstract notion of a "privileged account" to some privileged account on the target system, such as the database administrator, or the operating system administrator account.

Constructors

- **SecurityDescriptor**

  ```
  public SecurityDescriptor()
  ```

  Constructor.

Methods

- **`getGrantAccess`**

  ```
  public Identity[] getGrantAccess()
  ```

  Get the array of Identities that are granted access to execute the enterprise bean method.

  Returns:
  An array of Identities.

- **`getGrantAccess`**

  ```
  public Identity getGrantAccess(int index)
  ```

  Get the Identity at the specified index in the array of Identities that are granted access to execute the enterprise bean method.

  Parameters:
  `index`
  Index of the Identity to be obtained.

  Returns:
  The Identity at the specified index.

- **`getRunAsAttribute`**

  ```
  public int getRunAsAttribute()
  ```

  Get the "runAsAttribute" security attribute.

  Returns:
  The runAs attribute. The value must be one of Client, SpecifiedUser, and System.

- **`getRunAsIdentity`**

  ```
  public Identity getRunAsIdentity()
  ```

  Get the Identity of the specified user account with whose credentials to run the enterprise bean method. This attribute is consulted only if the value of the runAsAttribute is SpecifiedUser.

  Returns:
  The Identity to associate with the execution of the enterprise bean method.

- **`setGrantAccess`**

  ```
  public void setGrantAccess(Identity values[])
  ```

  Set the array of Identities that are granted access to execute the enterprise bean method. param value An array of Identities.

- **`setGrantAccess`**

  ```
  public void
  setGrantAccess(int index, Identity value)
  ```

  Set the Identity at the specified index in the array of Identities that are granted access to execute the enterprise bean method.

  Parameters:
  `index`

Index of the Identity to be set.

`value`
The Identity to be set.

- **`setRunAsAttribute`**

  `public void setRunAsAttribute(int value)`

  Set the "runAsAttribute" security attribute.

  Parameters:

  `value`
  The runAs attribute. The value must be one of Client, SpecifiedUser, and System.

- **`setRunAsIdentity`**

  `public void setRunAsIdentity(Identity value)`

  Set the Identity of the specified user account with whose credentials to run the enterprise bean method. This attribute is consulted only if the value of the runAsAttribute is set to SpecifiedUser.

  Parameters:

  `value`
  The Identity to associate with the execution of the enterprise bean method.

Class SessionDescriptor

```
public class java.ejb.deployment.SessionDescriptor
    extends java.ejb.deployment.DeploymentDescriptor
{
    public final static int PINNED;
    public final static int stateful;
    public final static int STATELESS;
    public SessionDescriptor();
    public int getSessionTimeout();
    public int getStateManagement();
    public void setSessionTimeout(int value);
    public void setStateManagement();
}
```

The EntityDescriptor class defines the deployment descriptor for a session enterprise bean.

A serialized instance of a deployment descriptor class is used as the standard format for passing an enterprise bean's declarative attributes in the ejb-jar file. The deployment descriptor setter functions are used by the enterprise bean provider's tools to create the deployment descriptor before it is put into the ejb-jar file, and the getter functions are used by the container provider's tools to read the deployment descriptor from the ejb-jar file.

Variables

- **PINNED**

  ```
  public final static int PINNED
  ```

  The session bean is stateful and can never be passivated.

- **stateful**

  ```
  public final static int STATEFUL
  ```

  The session bean is stateful and can be passivated between transactions.

- **STATELESS**

  ```
  public final static int STATELESS
  ```

  The session bean is stateless. A stateless bean can be reused for multiple session objects.

Constructors

- **SessionDescriptor**

  ```
  public SessionDescriptor()
  ```

  Create an instance of the deployment descriptor.

Methods

- **getSessionTimeout**

  ```
  public int getSessionTimeout()
  ```

Get the session timeout value in seconds. A zero value means that the container should use a default.

Returns:
The timeout value in seconds.

- **`getStateManagement`**

```
public int getStateManagement()
     throws IllegalStateException
```

Get the session bean's state management attribute.

Returns:
The session bean's state management attribute. It must be one of STATELESS, stateful, and PINNED.

Throws: IllegalStateException
Thrown if the attribute's value has not yet been set.

- **`setSessionTimeout`**

```
public void setSessionTimeout(int value)
```

Set the session timeout value in seconds. A zero value means that the container should use a default.

Parameters:
`value`
The timeout value in seconds.

- **`setStateManagement`**

```
public void setStateManagement()
```

Set the session bean's state management attribute.

Parameters:
`value`
The session bean's state management attribute. It must be one of STATELESS, stateful, and PINNED.

Class TransactionAttribute

```
public class java.ejb.deployment.TransactionAttribute
    extends java.lang.Object
{
    public final static int BEAN_MANAGED;
    public final static int MANDATORY;
    public final static int NOT_SUPPORTED;
    public final static int REQUIRED;
    public final static int REQUIRES_NEW;
    public final static int SUPPORTS;
}
```

The TransactionAttribute class defines the value of the enterprise bean's transaction attribute.

Variables

- **BEAN_MANAGED**

  ```
  public final static int BEAN_MANAGED
  ```

  The enterprise bean manages transaction boundaries itself using the java.jts.CurrentTransaction interface. The container does not perform any transaction management on the bean's behalf.

- **MANDATORY**

  ```
  public final static int MANDATORY
  ```

  The container is responsible for managing transaction boundaries for the enterprise bean as follow.

  If the caller is associated with a transaction, the execution of the enterprise bean method will be associated with the caller's transaction.

  If the caller is not associated with a transaction, the container throws the java.jts.TransactionRequiredException to the caller.

- **NOT_SUPPORTED**

  ```
  public final static int NOT_SUPPORTED
  ```

  The enterprise bean does not support a transaction. The container must not invoke the enterprise bean's method in the scope of a transaction.

- **REQUIRED**

  ```
  public final static int REQUIRED
  ```

  The container is responsible for managing transaction boundaries for the enterprise bean as follow.

  If the caller is associated with a transaction, the execution of the enterprise bean method will be associated with the caller's transaction.

  If the caller is not associated with a transaction, the container starts a new transaction, executes the enterprise bean's method in the scope of the transaction, and commits the transaction when the enterprise bean's method has completed.

- **REQUIRES_NEW**

  ```
  public final static int REQUIRES_NEW
  ```

The container is responsible for managing transaction boundaries for the enterprise bean as follow.

The container starts a new transaction, executes the enterprise bean's method in the scope of the new transaction, and commits the new transaction when the enterprise bean's method has completed.

If the caller is associated with a transaction, the association of the current thread with the caller's transaction is suspended during the execution of the enterprise bean's method, and resumed when the enterprise bean's method has completed.

- **`SUPPORTS`**

  ```
  public final static int SUPPORTS
  ```

  The container is responsible for managing transaction boundaries for the enterprise bean as follow.

  If the caller is associated with a transaction, the execution of the enterprise bean method will be associated with the caller's transaction.

  If the caller is not associated with a transaction, the container executes the enterprise bean's method without a transaction.

18 Related documents

[1] JavaBeans. *http://www.javasoft.com/beans*.

[2] Java Naming and Directory Interface (JNDI). *http://www.javasoft.com/products/jndi*.

[3] Java Remote Method Invocation (RMI). *http://www.javasoft.com/products/rmi.*

[4] Java Security. *http://www.javasoft.com/security.*

[5] Java to IDL Mapping. Joint Initial Submission. OMG TC Document TC orbos/97-08-06.

[6] Enterprise JavaBeans to CORBA Mapping. Unpublished JavaSoft document available to the Enterprise JavaBeans reviewers.

[7] OMG Object Transaction Service. *http://www.omg.org/corba/sectrans.htm#trans*.

[8] ORB Portability Submission, OMG document orbos/97-04-14.

Appendix A: Glossary of terms

Appendix B: Example application

TODO

Appendix C: Features deferred to future releases

The focus of Release 1.0 is to define the basic component model for session and entity enterprise beans. The model includes: the distributed object model; enterprise bean application programming model; state and transaction management protocols.

Given the broad scope of the Enterprise JavaBeans specification, we defer to future releases the features that introduce an advanced programming style. This conservative approach reduces the chance of our having to make a backward incompatible change in a future release.

Examples of the features that we would like to consider for a later release are listed below.

C.1 Programmatic access to security

We would like to allow expert-level enterprise beans to temporarily change their effective security Identity.

C.2 Enterprise beans with extended transactional semantics

Some enterprise beans may need to themselves be resource managers. One example is an enterprise bean that tries to provide transactional semantics across resource managers that do not participate in two-phase commit. The access to legacy data or business logic may be transaction monitor requests or a database stored procedures, each of which is itself a transaction with a corresponding compensating transaction. The enterprise bean would keep track of these transactions and call the compensating transaction if rollback occurred. This allows the same simple transactional programming model to be used externally on the enterprise bean (i.e., transaction begin, commit, rollback), while not holding locks on the real resources.

These enterprise beans need a much richer contract with the transaction coordinator that allows full participation in two-phase commit.

Appendix D: Issues and dependencies

D.1 Pending issues

We track the important pending issues here on which we would like to receive input from the reviewers.

D.1.1 Argument passing semantics

Enterprise JavaBeans should define the rules for the semantics of parameter passing in calls between two enterprise beans. We believe that the arguments that implement the *java.rmi.Remote* interface should be passed by-reference, and that all other arguments should be passed by-value even if the target enterprise bean is in the same container and JVM.

We believe that this needs to be made part of the architecture to prevent unpredictable behaviour of applications built from enterprise beans when deployed in different containers.

These proposed semantics are consistent with those used by RMI and the Java to IDL Mapping [5] and therefore would be familiar to Java programmers.

D.1.2 Runtime significance of ejb-jar file

In this revision of the specification, an ejb-jar file has no significance at runtime. This means that a container treats a method invocation between two enterprise bean objects in the same way, independent from whether the two enterprise beans are part of the same or different ejb-jar files.

Some reviewers suggested that it would make sense to treat the interactions between the enterprise beans that are part of the same ejb-jar *special*, as follows.

- The *runAsIdentity* applies to the whole ejb-jar file, not to the individual enterprise beans.
- All arguments are passed by-reference (i.e. using the Java semantics, not the Java RMI semantics).

This would imply that an ejb-jar should treated as a single application with multiple entry points (each enterprise bean is an entry point) rather than a collection of independent components (independent at least as far as the transaction, state, and security management is concerned). For a given transaction, a container would have to execute all the enterprise beans that are part of an ejb-jar in the same JVM process in order to preserve the by-reference semantics of argument passing.

This proposal would effectively lead to a two-level component architecture (ejb-jar files and individual enterprise beans) with more complex rules than in the current single-level component architecture. We have currently no plans to introduce such a two-level component architecture.

D.1.3 Declarative transaction attribute at method-level

Many reviewers gave us input that enterprise beans are likely to be coarse-grained objects potentially with a large number of client callable methods. A single declarative transaction attribute at the object-level may often become a problem because different methods may require different transaction attributes.

The suggested solution is to allow a per-method transaction attribute to override the one at object-level. This would be analogous to the security permissions defined on per-method basis.

D.1.4 *NOT_SUPPORTED* transaction attribute

It was pointed out that the name of this transaction attribute might be confusing if the enterprise bean in fact uses JTS to perform explicit transaction demarcation. A suggestion was made that we add a new attribute (e.g. *UsesCurrentTransaction*) that distinguishes an enterprise bean that uses the *CurrentTransaction* interface from one that does not support transactions. The container would not make the *CurrentTransaction* interface available to an enterprise bean that does not support transaction.

We have added the BEAN_MANAGED transaction attribute to the 0.796 specification.

D.1.5 Deployment descriptor format

Many reviewers suggested that using a serialized bean as a deployment descriptor would be a better alternative to using a *java.util.Properties* file.

We have changed the format of the deployment descriptor to a serialized Java object in the 0.796 release.

Appendix E: package java.jts

This Appendix provides the documentation of the classes and interfaces that are part of the package *java.jts* that are relevant to Enterprise JavaBeans. Note that the package *java*.jts may include other classes and interfaces that are not shown here.

```
interface CurrentTransaction
interface TransactionControl

class Status

class HeuristicCommitException
class HeuristicException
class HeuristicMixedException
class HeuristicRollbackException
class TransactionRequiredException
class TransactionRolledbackException
class InvalidTransactionException
```

Interface CurrentTransaction

```
public interface java.jts.CurrentTransaction
{
    public abstract void begin();
    public abstract void commit();
    public abstract TransactionControl
        getControl();
    public abstract int getStatus();
    public abstract void
        resume(TransactionControl suspended);
    public abstract void rollback();
    public abstract void rollbackOnly();
    public abstract void
        setTransactionTimeout(int seconds);
    public abstract TransactionControl
        suspend();
}
```

The CurrentTransaction interface defines the methods that allow an application to to explicitly manage transaction boundaries and control the association of transactions and threads.

Methods

- **begin**

```
public abstract void begin()
     throws IllegalStateException
```

Create a new transaction and associate it with the current thread.

Throws: IllegalStateException
Thrown if the thread is already associated with a transaction.

- **commit**

```
public abstract void commit()
     throws TransactionRolledbackException, HeuristicMixedException,
HeuristicRollbackException, SecurityException, IllegalStateException
```

Complete the transaction associated with the current thread. When this method completes, the thread becomes associated with no transaction.

Throws: TransactionRolledbackException
Thrown to indicate that the transaction has been rolled back rather than committed.

Throws: HeuristicMixedException
Thrown to indicate that a heuristic decision was made and that some relevant updates have been committed while others have been rolled back.

Throws: HeuristicRollbackException
Thrown to indicate that a heuristic decision was made and that some relevant updates have been rolled back.

Throws: SecurityException
Thrown to indicate that the thread is not allowed to commit the transaction.

Throws: IllegalStateException
Thrown if the current thread is not associated with a transaction.

- **`getControl`**

```
public abstract TransactionControl getControl()
```

Obtain the TransactionControl for the transaction that is associated associated with the current thread.

The returned TransactionControl can be used later by the thread as an argument to the resume method.

Returns:
The TransactionControl control for the transaction that is associated with the thread, or null if no transaction is associated with the thread.

- **`getStatus`**

```
public abstract int getStatus()
```

Obtain the status of the transaction associated with the current thread.

Returns:
The transaction status. The values of the transaction status are defined in the java.jts.Status class. If no transaction is associated with the current thread, this method returns the Status.NoTransaction value.

- **`resume`**

```
public abstract void
resume(TransactionControl suspended)
     throws IllegalArgumentException
```

Resume association of a transaction with the current thread.

Parameters:
`suspended`
A TransactionControl obtained previously by the current thread via the suspend or getControl method.

Throws: IllegalArgumentException
Thrown if the TransactionControl is invalid for the current thread (i.e. it was not obtained via the suspend or getControl method.

- **`rollback`**

```
public abstract void rollback()
     throws IllegalStateException, SecurityException
```

Roll back the transaction associated with the current thread. When this method completes, the thread becomes associated with no transaction.

Throws: SecurityException
Thrown to indicate that the thread is not allowed to roll back the transaction.

Throws: IllegalStateException
Thrown if the current thread is not associated with a transaction.

- **`rollbackOnly`**

```
public abstract void rollbackOnly()
     throws IllegalStateException
```

Modify the transaction associated with the current thread such that the only possible outcome of the transaction is to roll back the transaction.

Throws: IllegalStateException
Thrown if the current thread is not associated with a transaction.

- **`setTransactionTimeout`**

```
public abstract void
setTransactionTimeout(int seconds)
```

Modify the value of the timeout value that is associated with the transactions started by the current thread with the begin method.

If an application has not called this method, the transaction service uses some default value for the transaction timeout.

Parameters:
`seconds`
The value of the timeout in seconds. If the value is zero, the transaction service restores the default value.

- **`suspend`**

```
public abstract TransactionControl suspend()
```

Suspend the association of the current thread with a transaction. When this method completes, the thread becomes associated with no transaction.

The returned TransactionControl can be used later by the thread as an argument to the resume method.

Returns:
The TransactionControl control for the transaction that was associated with the thread, or null if no transaction was associated with the thread.

Interface TransactionControl

```
public interface java.jts.TransactionControl
{
}
```

The TransactionControl interface represents a transactions. It does not define any methods that would allow the application to directly manipulate the transaction.

Class Status

```
public class java.jts.Status
    extends java.lang.Object
{
    public final static int Active;
    public final static int Committed;
    public final static int Committing;
    public final static int MarkedRollback;
    public final static int NoTransaction;
    public final static int Prepared;
    public final static int Preparing;
    public final static int RolledBack;
    public final static int RollingBack;
    public final static int Unknown;
}
```

The class Status defines the values of a transaction status.

Variables

- **Active**

  ```
  public final static int Active
  ```

  A transaction is associated with the target object and it is in the active state. An implementation returns this status after a transaction has been started and prior to a Coordinator issuing any prepares unless the transaction has been marked for rollback.

- **Committed**

  ```
  public final static int Committed
  ```

  A transaction is associated with the target object and it has been committed. It is likely that heuristics exists, otherwise the transaction would have been destroyed and NoTransaction returned.

- **Committing**

  ```
  public final static int Committing
  ```

  A transaction is associated with the target object and it is in the process of committing. An implementation returns this status if it has decided to commit, but has not yet completed the process, probably because it is waiting for responses from one or more Resources.

- **MarkedRollback**

  ```
  public final static int MarkedRollback
  ```

  A transaction is associated with the target object and it has been marked for rollback, perhaps as a result of a rollback_only operation.

- **NoTransaction**

  ```
  public final static int NoTransaction
  ```

  No transaction is currently associated with the target object. This will occur after a transaction has completed.

- **Prepared**

  ```
  public final static int Prepared
  ```

  A transaction is associated with the target object and has been prepared, i.e. all subordinates have responded Vote.Commit. The target object may be waiting for a superior's instruction as how to proceed.

- **Preparing**

  ```
  public final static int Preparing
  ```

  A transaction is associated with the target object and it is in the process of preparing. An implementation returns this status if it has started preparing, but has not yet completed the process, probably because it is waiting for responses to prepare from one or more Resources.

- **RolledBack**

  ```
  public final static int RolledBack
  ```

  A transaction is associated with the target object and the outcome has been determined as rollback. It is likely that heuristics exist, otherwise the transaction would have been destroyed and NoTransaction returned.

- **RollingBack**

  ```
  public final static int RollingBack
  ```

  A transaction is associated with the target object and it is in the process of rolling back. An implementation returns this status if it has decided to rollback, but has not yet completed the process, probably because it is waiting for responses from one or more Resources.

- **Unknown**

  ```
  public final static int Unknown
  ```

  A transaction is associated with the target object and, but its current status cannot be determined. This is a transient condition and a subsequent invocation will ultimately return a different status.

Class HeuristicCommitException

```
public class java.jts.HeuristicCommitException
    extends java.rmi.RemoteException
{
    public HeuristicCommitException();
    public
        HeuristicCommitException(String msg);
}
```

This exception is thrown by the rollback operation on a resource to report that a heuristic decision was made and that all relevant updates have been committed.

Constructors

- **HeuristicCommitException**

  ```
  public HeuristicCommitException()
  ```

- **HeuristicCommitException**

  ```
  public HeuristicCommitException(String msg)
  ```

Class HeuristicException

```
public class java.jts.HeuristicException
    extends java.rmi.RemoteException
{
    public HeuristicException();
    public HeuristicException(String msg);
}
```

This exception indicates that indicates that one or more participants in a transaction has made a unilateral decision to commit or roll back updates without first obtaining the outcome determined by the transaction service.

Heuristic decisions are normally made only in unusual circumstances, such as communication failures, that prevent normal processing. When a participant makes a heuristic decision, there is a risk that the decision will differ from the consensus outcome, potentially resulting in loss of data integrity.

The subclasses of this exception provide more specific reporting of the incorrect heuristic decision or the possibility of incorrect heuristic decision.

Constructors

- **HeuristicException**

  ```
  public HeuristicException()
  ```

- **HeuristicException**

  ```
  public HeuristicException(String msg)
  ```

Class HeuristicMixedException

```
public class java.jts.HeuristicMixedException
    extends java.rmi.RemoteException
{
    public HeuristicMixedException();
    public
        HeuristicMixedException(String msg);
}
```

This exception is thrown to report that a heuristic decision was made and that some relevant updates have been committed and others have been rolled back.

Constructors

- **HeuristicMixedException**

  ```
  public HeuristicMixedException()
  ```

- **HeuristicMixedException**

  ```
  public HeuristicMixedException(String msg)
  ```

Class HeuristicRollbackException

```
public class java.jts.HeuristicRollbackException
    extends java.rmi.RemoteException
{
    public HeuristicRollbackException();
    public
        HeuristicRollbackException(String msg);
}
```

This exception is thrown by the commit operation to report that a heuristic decision was made and that all relevant updates have been rolled back.

Constructors

- **HeuristicRollbackException**

  ```
  public HeuristicRollbackException()
  ```

- **HeuristicRollbackException**

  ```
  public HeuristicRollbackException(String msg)
  ```

Class TransactionRequiredException

```
public class java.jts.TransactionRequiredException
    extends java.rmi.RemoteException
{
    public TransactionRequiredException();
    public
        TransactionRequiredException(String msg);
}
```

This exception indicates that a request carried a null transaction context, but the target object requires an activate transaction.

Constructors

- **TransactionRequiredException**

  ```
  public TransactionRequiredException()
  ```

- **TransactionRequiredException**

  ```
  public TransactionRequiredException(String msg)
  ```

Class TransactionRolledbackException

```
public class java.jts.TransactionRolledbackException
    extends java.rmi.RemoteException
{
    public TransactionRolledbackException();
    public
        TransactionRolledbackException(String msg);
}
```

This exception indicates that the transaction associated with processing of the request has been rolled back, or marked to roll back. Thus the requested operation either could not be performed or was not performed because further computation on behalf of the transaction would be fruitless

Constructors

- **TransactionRolledbackException**

  ```
  public TransactionRolledbackException()
  ```

- **TransactionRolledbackException**

  ```
  public TransactionRolledbackException(String msg)
  ```

Class InvalidTransactionException

```
public class java.jts.InvalidTransactionException
    extends java.rmi.RemoteException
{
    public InvalidTransactionException();
    public
        InvalidTransactionException(String msg);
}
```

This exception indicates that the request carried an invalid transaction context. For example, this exception could be raised if an error occured when trying to register a resource.

Constructors

- **InvalidTransactionException**

  ```
  public InvalidTransactionException()
  ```

- **InvalidTransactionException**

  ```
  public InvalidTransactionException(String msg)
  ```