Conception of Information Systems
Part 6: Distributed Objects and EJB
PART VI - Distributed Objects and EJB

1. Distributed Objects and CORBA
2. Server-side Object Implementation
3. Object Transaction Monitors and Components
4. Enterprise Java Beans
6.1 Distributed Objects

- Goal: invoke from an application (=client) methods of a remote object accessible through the network (=server object)

- Main issues
  - global identification of an object
  - processing of distributed requests (communication, RPC)
  - implementation of server objects (shared by many applications)

- Main (current) approaches
  - CORBA: language independent standard
  - RMI: Java
  - DCOM: Microsoft

Distributed object computing is based on the idea that applications and objects should be enabled to invoke operations (methods) on any object that is accessible over the network as transparently as possible. It extends the mechanism of invoking remote functions through RPC, which is based on a procedural programming model, to the object programming model. When dealing with objects a first issue is object identification, which should be possible globally across the network. The processing of distributed requests on objects is analogous to RPC. As server objects on which methods are invoked typically are shared by many clients, the efficient implementation of server objects requires particular attention. The questions encountered to that extend are similar to those that are considered in the development of classical transaction monitors.

CORBA was the first standardization effort in order to enable interoperability among distributed objects and to support developers in developing distributed object applications. A lot of hope was put into CORBA to become a general basis distributed programming and distributed information management. In particular industry was investing substantially into large-scale CORBA projects (with mixed success). Nowadays CORBA is still a valid architectural alternative for distributed information systems development, but it appears that its main merits lie in the fact that it paved the way and introduced concepts for the distributed component architectures, such as EJBs (Enterprise Java Beans), which provide functionalities at higher level abstractions and in a more reusable form and Web services that are built around the Web infrastructure.

RMI is Java's distributed object programming model. Besides being bound to a specific programming language it is much less complex (and powerful) than CORBA. DCOM is the Microsoft competitor to CORBA.
We have earlier studied the problem of accessing multiple databases at the same time. Two solutions have been discussed:

- Access through a federated DBMS. With respect to transactions the problem may occur that within the same transaction multiple databases are accessed at the same time, therefore we have what is called a "distributed transaction"

- Access through a user interface and application programs that are web-enabled. Here no distributed transactions occur since every database is accessed by a separate program.

However, we could easily imagine to use exactly the same mechanism that we used to Web-enable databases, i.e. Web-accessible JAVA applications, that access the DBMS via JDBC, for accessing multiple databases from within the same Java application. Thus we have found a different way to access multiple DBs, by integrating them at the application level. Now we can again have distributed transactions, and they need to be dealt with somewhere. The platform that will be doing this job we call an **application server**. This application server also can run (and in many cases has to run) on a computer (server) that is different from the database servers (remember: JAVA allows DB access over the network).
Since 3-tier (and more generally n-tier) architectures are distributed by definition, it is natural to consider their implementation using distributed objects. It is important to keep in mind that the scope of distributed object technology goes beyond the support of distributed, heterogeneous information systems, but that they are one of the main applications of distributed object technology. For example, many real-time applications are developed based on CORBA. One can derive the importance of distributed objects for information systems from the fact that many of the services in CORBA, for example, are related to information management (e.g., transactions, persistence).

Thus one can say that distributed object standards are principally a distributed application development platform that in particular are used as architecture for information systems integration.

The example of a banking application shown demonstrates of how a 3-tier architecture can be mapped into an implementation that is based on distributed objects. For each of the three tiers, there exist distributed objects representing the corresponding functionality and accessing each other via (remote) method invocations. So, we have an object encapsulating the code for supporting the user interface of this application, that invokes the object that encapsulates the business logic. This object in turn invokes the (persistent) objects that represent the data stored in a database. Notice also, that the object that controls the business logic is invoking a transaction management service, which is implemented as another distributed object.
OMA (object management architecture) is a standard architecture for distributed programming developed by the industry consortium OMG (Object Management Group). It is available on many system platforms.

The main constituents of OMA are

- standard programming support through CORBA (common object request broker architecture): CORBA provides an object bus for making distributed objects interoperable over the network. The necessary functionality is provided by a so-called ORB (Object Request Broker). Since ORBs from different vendors are not using necessarily the same communication protocols there exists in addition a standardized protocol IIOP (Internet-Inter-ORB) for ORB to ORB communication over the Internet.
- standard service interfaces (horizontal): the most interesting for our purposes are the basic information system services (transactions, persistence)
- standard domain interfaces (vertical): these are interfaces for special application domains such as Finance, Healthcare, Manufacturing, Telecom, Electronic Commerce, and Transportation.

We can view the function of the OMA in two ways (according to our three-dimensional taxonomy on information systems integration)

- OMA hides distribution by supporting remote method invocation and global object identification. Thus it can be considered as an implementation platform for distributed information systems and in particular for multi-tier architectures (3-tier, n-tier).
- OMA provides a homogeneous application development layer. It allows to hide heterogeneity of underlying (legacy information) systems by providing a standard interface. This is done by encapsulating the legacy systems into distributed objects (comparable to wrappers) that provide their functionality through the standardized interfaces which are specified using the OMA interface definition language (IDL).
First we give an overview of the core of the OMA architecture, which is the Common Object Request Broker Architecture CORBA. This architecture consists of the following components, that are specified as part of the OMA standard document "Common Object Request Broker Architecture: Architecture and Specification" v 2.0, 1995.

- The distributed object model and its execution semantics: this part describes the model of how distributed objects are identified, of how they are invoked over the network, and of how their implementation is accessed upon invocation.
- The architecture of the distributed object bus: this part describes which are the functional components of an ORB and how they interact.
- The interfaces among the different architectural components of an ORB. These interfaces are in particular important for developers that use or provide services to certain of these components.
- The interface for ORB interoperability that is needed to enable the interoperability of ORBs from different vendors.
- The interface definition language that is a syntax for specifying object interfaces (both methods and attributes). It borrows from C++, however has mappings to various standard programming languages.
- The mappings to various programming languages that are called "language bindings".

We will look now in more detail into each of these aspects of CORBA.
The basic model of operation in CORBA is the following: a client that wants to invoke a service on an object through the network needs first of all the object's identifier, which is called the object reference. There exist different ways of how it can obtain such a reference, typically another CORBA service will be providing them. The invocation of a service (i.e. a method of the object) is transformed by the ORB into a request (marshalling), then transmitted by the ORB from the client node to the server node that has the object's implementation, and then transformed by the ORB (unmarshalling) into a method invocation on the server side representation of the object, which is a CORBA object. The ORB is a system layer (possibly distributed over the network) that performs the forwarding of the requests transparently for the client. The CORBA object has access to another object the servant object, which provides the actual implementation of the service. Thus the CORBA objects works like a proxy for the service provided by the servant object. The interface of the CORBA object is specified by means of IDL.
This figure depicts the functional components an ORB needs in order to forward a request from the client to the server. The client has access to three interfaces of the ORB:

- The ORB interface, which it needs to establish connections to the ORB.
- The static stubs, which are invocations to CORBA objects, that are statically compiled into the application code.
- The dynamic invocation interface (DII), which is used to dynamically invoke objects (i.e. decide in runtime which is the object that is invoked and its type).

The server application has on the other side the four following interfaces at its disposal:

- The ORB interface, again for establishing the connection with an ORB.
- The object adaptor which allows the application to register the implementation of a CORBA object at the ORB and provides to the ORB access to the object. We will discuss this interface later in much more detail.
- The skeletons, which are statically compiled (CORBA) objects providing access to the object implementation (servant).
- The Dynamic Skeleton Interface, which allows the server to serve untyped requests to objects (i.e. decide in runtime of which type the request is and how to access its implementation).

For processing a request, the client in step (1) either invokes a static stub method or uses the dynamic invocation interface to dynamically construct a request from an object interface definition. The Client ORB core forwards the request in step (2) to the server ORB core. There, depending on the object implementation (not on the way the request was constructed at the client) either the request is forwarded to the skeleton or it is dynamically associated with an object implementation using the DSI. Then in step (4) the request is sent to the server application (i.e. the servant) that implements the service.
Inter-ORB Protocols

- Different ORB processes communicate over proprietary protocols
  - Vendor specific

- GIOP - General Inter-ORB Protocol
  - defines transfer syntax
  - standard set of messages (request, reply, ...)

- IIOP - Internet Inter-ORB Protocol
  - is the implementation of GIOP over TCP/IP

The fact that vendors are using proprietary protocols for having their ORBs to communicate turned out to be a major paradox when using CORBA for implementing distributed applications, presumably following a "standard". Since ORBs of different vendors could not communicate with each other, in spite of following a standard the software was generally not interoperable. Therefore a protocol was specified that standardizes also the interfaces for request interchange between ORBs and thus allows different ORB implementations to communicate. More specifically it standardizes the transfer syntax for requests and the messages that can be used. This is the GIOP, the general Inter-ORB protocol. A number of assumptions are made by the GIOP, such as connection-oriented transport (i.e. synchronous communication) and that connections are bidirectional, symmetric, reliable, and support byte-streams. The IIOP is the Internet-compliant implementation of GIOP, and is the one that is of relevance in practice.
Object Adaptors

- Interface between the object implementation (servant) and the ORB

- Functionality
  - Register and manage object implementations
  - Manage object references which allow clients to address objects
  - Ensure that target objects are instantiated by the server application
  - Dispatch requests from server-side ORBs to the servants of target objects

- Original design: BOA (Basic Object Adaptor)
  - C-oriented
  - Limited functionality
  - replaced by POA

The object adaptors provide the interface between the object implementation (servant) and the ORB. One might ask why this functionality is not integral part of the server ORB core. The reason is that different types of object adaptors can be provided for different styles of programming and support different behavior of the implementation objects (e.g. related to performance). Putting all these options into the ORB kernel would overload it or unnecessarily limit the flexibility in interfacing to object implementations.

What are the functions of the object adaptor?

- Applications need to be able to register implementations of CORBA objects (i.e. servants) at the ORB. To that end the object adaptors provide interfaces for registration and they access an implementation repository, where the information about the registered implementations is stored.

- The object adaptor generates object references for CORBA objects and thus make the objects accessible through an ORB. To that end it provides interface functions to the server application for providing the object key.

- When an object is requested by a client, the object adaptor needs to make sure that the CORBA object exists and an actual implementation object (servant) is instantiated at the server application. The server application has for that reason to implement callback functions in order to support the object adaptor interface in activating (resp. deactivating) the object implementations.

- Finally the central function of the object adaptor is to dispatch requests that arrive at the server-side ORB through the skeleton (in the static case) to the servant object that is provided by the server application.

The simplest form of object adaptor is the so-called BOA (basic object adaptor). It is generally provided with every ORB implementation but it is very limited in its functionality and it does not support an object-oriented style of programming. A more advanced type of object adaptors are the so-called portable object adaptors (POA) which provide in particular for the implementation of scalable applications, such as necessary of information management, substantial flexibility to tune and optimize the implementation. They constitute also a predecessor to the component concept, that we will introduce later.
Each CORBA object has at least one identifier, called *object reference*, that is used by the client in order to invoke methods on the object. A reference is uniquely associated with a CORBA object, but the same CORBA object may have multiple references. References may be NIL or they may point to CORBA objects that do not exist. The clients cannot see the content of the reference (opaque references) because it is the task of the ORB to use the object references in order to locate the CORBA objects and to transmit the requests. References are strongly typed, that is they are associated with a specific IDL object type. However, they can be bound to objects that have a subtype of the specified IDL type, and the binding to the subtype can be performed in runtime (late binding). References are also not necessarily bound to the lifetime of an application, i.e. they can be made persistent (together with the objects they identify).

An object reference consists of three main parts

1. An identifier of the IDL which determines the type of the object (an IDL repository). For a specific ORB implementation the representation of this part of the identifier needs to be standardized, such that the ORB can determine the type of the object.

2. An identifier of the location in the network where the object resides. For the same reason as before this part of the object reference needs to be standardized for a specific ORB implementation.

3. The object key, identifying the object at a specific server. Part of this object key may depend on the implementation of the server object, and thus it is not standardized for an ORB implementation.

The endpoint info (network location) together with the object key determine uniquely the object, whereas the type information is needed to determine whether an invocation (static or dynamic) on an object is admissible.
The Naming Service is a specific CORBA service providing to applications bindings among interpretable names and object identifiers. The name server maintains these bindings and may organize the names into different contexts (called directories). A server that is providing an object under a specific name has to register the name at the naming service (1). A client that wants to access the object has to know its' name. Using the name it can access the name server and resolve the name to an object identifier. Using the object identifier it can then invoke the service on the CORBA object.

The Naming Service is, as any other CORBA service, defined through the interfaces of CORBA objects,
This is an example of an IDL specification for a CORBA object, the naming service. One can see that IDL allows to specify both attributes and methods for the objects, supports strong typing of attributes and method parameters, and supports standard built-in types and type constructors. We will discuss IDL not in more detail as it is not substantially different from other OO interface definition languages.
This figure illustrates the basic development process that the implementation of each CORBA application has to follow. Notice that after defining the IDLs (i.e. the common interface definitions), the implementation of the clients and servers are performed independently. For the server side a substantial part of the development effort is dedicated to the binding of server-side implementations to the server-side CORBA objects. In addition, a server program needs to be defined that can listen to incoming requests and serve them.
6.2 Server Side Objects

- Implementation of Servers using POA (Portable Object Adaptor): Interface between ORB and programming environment
  - Creation of objects
  - Activation of objects
  - Dispatching of requests
- Multiple portable object adaptors can be used at the same time
  - Different characteristics
  - POA Manager dispatches the requests
- POA does not maintain persistent state for objects
  - Task of the application

The Portable Object Adaptor (POA) establishes the interface between the ORB and the programming environment: it manages the life-cycle of CORBA objects, consisting of creation of objects and their references, activation of objects, such that they are connected to a servant, dispatching of requests, and eventually their destruction. In order to obtain a larger degree of flexibility multiple portable object adaptors can be used at the same time, each of which can exhibit different characteristics related to the type of object management and the resulting performance desired. Since multiple POAs are used at the same time, there exists a POA manager that knows them and knows of how to dispatch the incoming requests to the right POAs such that the requests arrive at the right CORBA objects and servants. An important design decision, that has been made with respect to POAs, is whether they should manage a persistent state for the objects. This would be in principle reasonable since this aspect is closely related to the object lifecycle management. However, this is not the case in order not to hamper the POA's generality and is left as a task to the application implementing the object.
Life Cycles of CORBA Objects and Servant Objects

- Life cycles of CORBA and servant objects are separated
  - One servant object per CORBA object at a time
  - Different servant objects can incarnate the same CORBA object over time

This diagram shows the life-cycle of CORBA objects and servant objects as they are managed by the POA. One can see that the POA allows to decouple the life-cycles of the CORBA objects and the servant objects. One CORBA object can only be incarnated (=implemented) by one servant object at a time, but servant objects can be shared by multiple CORBA objects, even at the same time. CORBA objects can also be dynamically attached or detached from their servant objects. This is called the activation or deactivation of the CORBA object. A CORBA object that has terminated the connection to its servant is called etherealized. The POA provides the necessary interfaces to the application in order to control the object life-cycle as described.
The POA provides the possibility to develop scalable, high-performance servers by allowing a fine-tuned resource control for server objects. The control strategies are provided by so-called policies. Each policy addresses one aspect of controlling the mapping of CORBA objects to servants. Each POA can use a different policy. The ORB uses the object reference (which contains type information) to determine the right POA for processing a request to a CORBA object. By default there exists a root POA that implements a default policy. Policies themselves are represented as CORBA objects that are accessible through the local ORB. The choice of policies depends on application characteristics, such as space-time tradeoffs (number of objects to be supported, expected rates and durations of requests), the use of underlying persistent storage (need to support user-assigned object identifiers for persistent objects), available resources and services of underlying system, or access to applications that are wrapped as CORBA objects.

A basic decision is whether servants are shared by CORBA objects. Sharing of servants by multiple CORBA objects is analogous to sharing of processes for multiple client requests in transaction monitors. In transaction monitor terminology, a CORBA object corresponds to a service and a servant corresponds to the servers of a server class. Having separate servants for different objects is most appropriate in the case where only a few transient CORBA objects exist, whereas sharing of servants makes sense where CORBA objects are, for example, related to persistent data items stored in a database. Sharing of servants is in that case possible since the state of the object is not kept in the servant process but in the database. As a result not for every data object a separate servant process needs to be instantiated.

For many applications static object creation at startup is sufficient. Objects in information systems correspond frequently to database objects and creating for each of them a separate CORBA object would not be efficient. Therefore the POA allows to create CORBA objects upon actual requests to them only.

Persistent objects exist independently of the runtime of any server process. This is the original model of how objects are managed in CORBA. The transient objects were only introduced for efficiency purposes with the POAs. Transient objects exist only with the lifetime of the POA and are sufficient for many applications.

Retaining object-servant associations allows to optimize performance. If many such associations are kept the active object table holding these associations becomes more memory consuming. Not retaining them increases processing overhead, as the association has to be established for each request anew.
After having introduced the basic working of the CORBA architecture and in particular of the ORB and POAs we turn now our attention to another aspect of the OMA, the horizontal CORBA services. They cover in particular the functions that are found otherwise as components of application or database server architectures, such as transaction monitors or database management systems. Of particular interest for distributed information management is the transaction service, which constitutes the CORBA counterpart to the X/Open architecture, that has been introduced for distributed transaction processing. In the following we introduce the transaction service for two purposes:

- In order to give an example of how a typical horizontal CORBA service looks like.
- In order to understand the specific concepts for implementing distributed transactions that the CORBA transaction service introduces. Some of the technical aspects will be of importance for our subsequent discussion of component technologies (such as Enterprise Java Beans).

The specification of a CORBA service, and in particular of the transaction service, consists essentially of a set of interfaces, that are defined for all the (distributed) objects that participate in the service, i.e. in a transaction. Of course this specification of interfaces is based on an underlying processing model. In the case of the CORBA transaction service the underlying model is compatible with the X/Open Distributed Transaction Processing Model. It supports a number of standard transaction concepts, including flat and distributed transactions, which we have introduced earlier in the part on transaction management. In addition, the transaction service supports the concept of nested transactions, which allow to structure transactions in an hierarchical manner, such that subtransactions can independently commit or rollback. The parent transaction is always committed when all its subtransactions are committed. This allows to do rollbacks of subtransactions only, and thus can offer advantages as compared to flat transactions, because in case of failure not the complete transaction needs to be aborted. The design of the transaction service has been guided by the intention to allow its implementation in a transaction monitor environment.
For understanding the transaction service specification one needs first to understand the relevant entities (objects) that participate in a distributed transactional application. To that end one distinguishes three types of objects: transactional clients, transactional servers, and recoverable servers. One can think of these three type of objects as corresponding to the three layers of a three-tier architecture.

- **Transactional clients** are objects that can invoke calls to transactional objects and begin or start a transaction.
- **Transactional servers** contain (multiple) transactional objects. A transactional object can participate in a transaction and refer to some persistent data indirectly. For executing transactions the transactional context is transferred to them, as with transactional RPC.
- **Recoverable servers** contain (multiple) recoverable objects. A recoverable object registers a persistent resource (e.g. a database). Recoverable objects are transactional objects with additional properties. Since they represent persistent data objects, they must participate in the Transaction Service protocols (commit and rollback).

Both transactional objects and recoverable objects can participate in transactions and force the rollback (abort) of a transaction, but only recoverable objects need to participate in the commit processing. Transactional clients cannot distinguish transactional from recoverable objects.

The scope of a transaction is defined by a **transaction context**: it refers to a transaction (by means of a transaction identifier) and to transaction parents in nested transactions. The scope also contains a timeout value. The transaction context (object) is managed by the transaction service and passed with each invocation of a transactional operation.
Transaction Implementation

• Indirect and implicit transaction implementation

```java
org.omg.CORBA.Object crtRef =
    orb_.resolve_initial_references("TransactionCurrent");
Current txn_crt = CurrentHelper.narrow(crtRef);
txn_crt.begin();
...
// the client issues requests, some of which involve transactional
objects;
BankAccount1->makeDeposit(deposit);
...
txn_crt.commit(false);
```

• Direct and explicit transaction implementation

```java
CosTransactions::Control c = TFactory->create(0);
CosTransactions::Coordinator co = c->get_coordinator();
CosTransactions::Terminator t = c->get_terminator();
...
BankAccount2->makeDeposit(deposit, co);
...
t->commit(false);
```

CORBA offers different implementation styles for transactions: With respect to the invocation of transactions one distinguishes indirect from direct management. The indirect management corresponds to the way of how transactions are invoked in procedural programming languages, by explicitly starting and ending transactions through function calls, which are methods of a pseudo object Current. With direct management the transaction objects implementing the transaction service are explicitly accessed. For example, by creating a transaction control object a transaction is implicitly started. The advantage of this style of transaction management is, that it allows a finer-grained control of transactions, but it is more complex to use.

Similarly for the transfer of the transaction call with a method invocation (corresponding to transactional RPC) the transaction context can be explicitly or implicitly propagated. The implicit propagation corresponds to the classical approach that we have already seen with transactional RPC. It requires that the ORB is able to support this functionality (similarly as a transaction monitor, which handles RPC requests, propagates "implicitly" the transaction context). This also shows that the transaction service is not a service that can be provided independently of the ORB, but requires extensions of the ORB implementation itself. This is a special feature of the CORBA transaction service. With explicit propagation the forwarding of transaction context is delegated from the ORB to the developer. The developer is responsible to extend the IDL interfaces of his transactional objects, such that they allow to include the transaction context as an explicit parameter into the method interface. Of course he has then also to provide the necessary values for these parameters in a method invocation. The trade-offs are similar as for direct/indirect management.

The two aspects of transaction management and context propagation are independent of each other. Both styles can be arbitrarily combined. The example gives an impression of the resulting different implementation styles:

With indirect management and implicit propagation the transaction is started by sending the begin() method to the Current object and invoking the method within the transaction boundaries. This corresponds exactly to the style of programming that is used also in the X/Open model and in procedural languages. With direct management and explicit propagation the application first creates the transaction object (Control) obtains from it the coordinator object and passes it then as a parameter to the makeDeposit method. The transaction is ended by invoking the commit() method that is provided by means of the Terminator object.
Other Services

- **Persistence Service**
  - make the state of CORBA objects persistent
  - support interfaces for efficient access to large numbers of objects
  - reuse existing transaction service

- **Query Service**
  - Provides interfaces for query operations on object collections
  - Operations can return object collections
  - Language independent, but support for SQL92 or OQL required

- **Concurrency Service**
  - Provides locks in order to ensure serializability in concurrent access of multiple clients to the same resource
  - Intentional read, read, upgrade, intentional write and write locks
  - Supports protocols for strict two-phase locking

We mention other horizontal services that are related to the access of persistent data.

The persistence service allows servants to persistently store their objects in a standardized way. Differently to the transaction service the persistence service is not visible at the client, but is just used for the implementation of servants and provides thus interfaces to the servants and POAs. The importance of the persistence service lies in the fact that its specification has influenced the development of persistence mechanisms for component technologies, such as EJB that we will introduced in the following.

The query service is on the other hand intended to be used by the clients and supports the access to large object collections. It is primarily intended to allow the use of declarative query languages, such as SQL or OQL, from within object-oriented applications.

The concurrency service allows to implement locking mechanisms for resources. This is particularly useful for non-database resources (e.g. files). It supports different lock types and the strict 2PL protocol.
Summary

- Distributed object architectures can be used
  - As distributed programming environment
  - Heterogeneous system integration platform

- Scalable CORBA servers require the flexibility of the Portable Object
  Adaptor (POA)
  - Association of CORBA objects with servants

- CORBA Services provide all essential functions of information systems
  - Transactions, persistency
  - Support explicit and implicit programming styles
6.3 Object Transaction Monitors

- Implementing CORBA Server Objects
  - Provide calls to the horizontal services
  - Orchestrate those calls within the implementation
- The problem of server-side distributed object development
  - change of runtime behavior requires reprogramming
  - resource management needs to be implemented by developer
  - complex system-specific code, difficult maintenance
  - repeating implementation patterns

As we have seen in the previous part of the lecture, implementing a server-side CORBA object requires the developer to consider a number of aspects. He needs to establish policies for the object adaptor in order to control the object life-cycle and he needs to use horizontal CORBA services properly. Different parts of the implementation, such as client application, servants or persistent resources need to be combined in a way such that they are compatible with each other (e.g. when declaring an object to be persistent the developer needs to provide the necessary mappings to make objects persistent, has to implement the Resource interfaces etc.). This requires a thorough understanding on the side of the developer in order to coordinate correctly the interaction of the application objects and services.

All of this leads to a number of problems, in particular for developing complex applications:

If a change in runtime behavior is desired, e.g. by changing POA policies or transactional boundaries, reprogramming and recompilation of the applications is required.

The developer has to be aware of the resource requirements, for all potential future uses of the application. Once an implementation is compiled, the runtime behavior can no more be changed.

The coding is, as elaborated before, a complex task and therefore error-prone. Changes can often not be performed in isolation, but require the adaptation of multiple parts of the implementation. Therefore maintenance becomes difficult. It requires substantial skills by the developers, which distracts their attention from the implementation of the business logic.

The implementation patterns for the system specific code are repeating, and it is not economical to repeatedly perform the same system-specific implementation tasks for similar types of applications.
The difficulty of using the different horizontal CORBA services in a coordinated fashion, in particular those related to persistent data management, is illustrated by this figure. Without understanding all the details one can see that for accessing a persistent resource, such as a database, a number of CORBA objects provided by the different services need to be accessed. Some of the services and their corresponding objects, such as transaction service or resource we have introduced earlier. It is easy to imagine that following this complex interplay of multiple services poses to the developers substantial problems. This was one of the main reasons why the original intention to directly use these services for implementing scalable server applications encountered many difficulties and did not prevail in practice.
Accessing Persistent Objects (2)

- Commit transaction
- Release session
- Prepare
- End
- Write modified data
- Prepare
- Commit
- Release session
- No association
- Transaction Coordinator
- Transactional Session
- Resource
- Database
- Session Manager
- Servant
- Transaction Service

©2003, Karl Aberer, EPFL-SSC, Laboratoire de systèmes d'informations répartis
Server-side Objects vs. Transaction Monitors

- **Commonalities between Distributed Object architectures and Transaction Monitors**
  - 3-tier model
  - transparency (heterogeneity and distribution)
  - distributed transaction and communication management for applications with many clients and servers
  - services: transaction, persistence, communication services etc.

- **Difference**
  - programming style: object-oriented vs. procedural
  - Explicit vs. implicit service representation
  - TM provide efficient process, reliable, scalable, performant, secure
  - Parametric resource configuration

Server-side objects play in an information system implemented on top of a distributed object platform the same role as applications running on a transaction monitor in a conventional 3-tier architecture. In both cases services, typically for managing access to persistent resources, are used by many clients in a multi-tier architecture. Both, the ORB and a transaction monitor, provide similar services, like distributed transaction and communication management and support the application developer in dealing with the problems of distribution and heterogeneity. In fact, the distributed transaction processing concept in CORBA is based upon the X/Open standard, that had been developed for transaction monitors.

Historically, there was from a practical perspective however a large gap among TM and CORBA platforms. At the first glance one recognizes a clear difference in the programming style, which is procedural for TMs and object-oriented for CORBA. This results also in the different ways of how services, such as transactions, are represented. In TMs these are provided as integral part of a system architecture and are typically accessed in an implicit manner (e.g. implicit transaction context propagation), whereas in CORBA these services are represented as objects in their own right and enable thus more explicit ways of dealing with them (e.g. explicit transaction context propagation).

The more important gap between the platforms was however much more related to the different functional features and performance. TMs are by construction high-performance transaction processing platforms, and thus exhibit all the desired functional features, such as efficient process management, high reliability and scalability, high performance and security, which the early CORBA implementations clearly didn't exhibit. Another very important difference is that TMs allow to adapt certain functional attributes, e.g. those related to resource usage, in runtime and therefore to tune the performance of an application in runtime (remember: the administration of a TM is a non-trivial task). This capability was lacking in early CORBA implementations, and any change in runtime required code changes as already pointed out.
Thus there was the quite obvious idea to combine the virtues of the two approaches of distributed object computing and transaction monitor technology by combining the two technologies. One can view this combination in two ways, and both ways reflect actual evolution paths that had been taken.

1. The extension of transaction monitors by the object model (based on CORBA): this approach has been taken by transaction monitor vendors, such as Tuxedo. They took their proven, robust technology and extended the architecture in way that they turned their products into full-fledged ORBs.

2. The extension of distributed object platforms by TM technology: this was the approach taken by CORBA vendors, such as IONA. They integrated into their ORBs increasingly features and technologies that are known from traditional transaction monitors.

In both cases the result were object transaction monitors. Object transaction monitors (OTMs) however introduce a new aspect: in order to combine the approach of having the important services, such as transactions, hard-wired into a system architecture such as an object transaction monitors, the responsibility of making use of those services needed to be moved from the developer to the OTM. Thus the original CORBA services are no longer made directly available to the developer, but they are internally used by the OTM. This is necessary, since the OTM provides implementations of these services that can take advantage of the specific characteristics of their architecture. The result is that an OTM turns into an environment (which is called container) where server-side objects, that implement the business logic, reside and make use of the OTM services via a high-level interface. In particular the OTM will be in charge of managing the object life-cycle. The functional characteristics of the object management, such as the properties that would be specified in POA policies, transaction demarcations or performance-related properties related to the specific OTM architecture, e.g. for process management, are supplied by the developer in a declarative manner, and can be changed in runtime.

In this way the OTM provides the following functions:

Integrated services: 2PC Transactions, Persistency, Resource access, Events, Security, Queued messaging, Object life cycle management

Communication: CORBA, DCOM (the Microsoft distributed object model), stubs, skeletons, object adaptors (as with classical ORBs)

Resource Management: Load Balancing, Message Routing, Multithreading (as with classical TM)
Developing Objects for the OTM

- The OTM provides more abstract interfaces for using the services

- The developer provides
  - Specification and implementation of business objects
    - IDL interfaces and object implementations
  - Support functions for the OTM for life-cycle management
    - Object activation and deactivation methods (e.g. managing persistence)
    - Factory objects: creation of object references
  - Specification of functional properties of the objects
    - Deployment descriptor (metadata)

- The OTM manages
  - object lifecycle (using POA and developer-supplied functions)
  - requests to and from objects
  - functional properties (according to the deployment descriptor using CORBA services)

For the developer an OTM presents itself as a quite different development environment as compared to a distributed object platform. The development process for OTMs differs substantially from developing server objects using directly the POA and CORBA services, since the OTMs provide more abstract interfaces for using these services.

The main focus of the developer is on the business logic of the objects as expressed through IDL interfaces and object implementations. In addition, he has only provide support functions that the OTM needs in order to deal with the application specific aspects of the object life-cycle. Most important this concerns the identification of objects, if it requires application-specific identification schemes. In addition, the developer has for example the possibility to provide application specific operations that should be performed upon activation or deactivation of objects. All the functional properties of the object, that should be guaranteed by the OTM, are provided by a so-called deployment descriptors, which are a declarative specification of the functional properties and can be changed in runtime.

Based on these specifications the OTM takes care of all the rest.

- It performs the object management: creates/destroys objects, automatically invokes activate and deactivate methods, allocates the required resources (all of this for example using a POA).
- It intercepts calls from/to other objects, forwards them.
- And it guarantees the functional properties. For example, it automatically invokes transaction services when accessing persistent objects within transactions.
Example: Deployment Descriptor

```xml
<?xml version="1.0"?>
<!DOCTYPE M3-SERVER SYSTEM "m3.dtd">
<M3-SERVER server-implementation="com.beasys.samples.BankAppServerImpl" server-descriptor-name="BankApp.ser">
  <MODULE name="com.beasys.samples">
    <IMPLEMENTATION name="TellerFactoryImpl" activation="process" transaction="never"/>
    <IMPLEMENTATION name="TellerImpl" activation="method" transaction="optional"/>
    <IMPLEMENTATION name="DBAccessImpl" activation="method" transaction="optional"/>
  </MODULE>
</M3-SERVER>
```

For illustration, we give just one example of how in such an OTM environment the developer can specify, by using a deployment descriptor, the functional properties of an object in runtime, after he has specified its business logic. In this example, taken from a banking application, the developer has supplied implementations for certain business methods for accessing a teller machine. The TellerFactoryImpl is used to create a teller machine object that receives these methods, the TellerImpl is the implementation of the teller object machine interface, and the DBAccessImpl provides an application specific way of making the teller machine object persistent in a database.

There are two functional properties that are specified, which are related to the object activation policy and the transactional properties. With respect to the object activation policy for the TellerFactoryImpl object it is declared that it should stay activated, once it has been activated, till the process activating the object ends. The other two methods should be deactivated after the execution of a method on them (in order to free the resources for other objects). A third possible object activation policy would be to bind the activation time to a transaction (Note: activation of objects corresponds to the notion that we have introduced for the POA earlier).

The transactional properties relate to the different ways of how methods on the objects can be executed as part of transactions: The possibilities are

- Always (either the method is invoked within a transaction or a transaction is started)
- optional (only executed as part of a transaction if invoked within transaction)
- never (not invokable within transaction)
- ignore (not taking part in a transaction even if invoked within the transaction)

One can also see that the deployment descriptor is packaged as XML document.
From a software engineering perspective the server-side objects that are developed for an OTM and that are taking advantage of the environment of the OTM are what is called *components*.

Components are characterized by the fact that they are self-contained pieces of software that can be deployed (without further implementation effort) in a specific environment, which is the container. The idea is that such an approach will allow to assemble complex software constructs from smaller pieces - the components -, which have proven functionality, can be reused in many different application contexts, and therefore also can be easily adapted to different application environments. This is also a reason why the deployment descriptor plays such a central role. The components interact with the container through a standardized interface. They support introspection, i.e. they allow tools to discover their interfaces, and configurability, for customizing their properties for the applications. A packaging mechanism is provided, that allows to bind together all the necessary parts of a component into one file, that can then be *deployed* on a container.

A *component model* describes the interface between components and the container. We may consider it as the language in which contracts between components and containers are established. The difficulty is to find for the component model a tradeoff between being sufficiently flexible and easy of use. For example, the generic CORBA model was arbitrarily flexible but very difficult to use.

Components are different from other paradigms for increasing reusability from software engineering. *Objects* are encapsulated pieces of software that make up part of components, but are not self-contained. *Patterns* are proven solution methods that are described *textually* according to a specific method. *Frameworks* are partially implemented solutions of which the code must be completed.
Server-side Components

- Client-side (e.g. Java beans) and server-side components

- Server-Side Components
  - serving multiple clients on a middle-tier server
  - transactional, persistent, secure, high-performing
  - Components once developed can be deployed on every application server (portability)

- Object Transaction Monitors
  - Are containers for server side components
  - Orchestrate the component interactions with services
  - Support declarative specification of functional properties

On the client side the component concept has been very successful for user interface development e.g. with the introduction of JavaBeans. The server-side objects running on OTMs are, following the characterization given for components before, clearly also components. Thus we will in the following no longer call them server-side objects, but server-side components. Their main properties are summarized on this slide.
Summary

• Object Transaction Monitors integrate the following three concepts

1. Distributed Object Computing (CORBA)
2. Transaction Monitor Technology
3. Software Components

in a non-standard fashion!
6.4 Enterprise Java Beans

- Motivation
  - From ad-hoc and vendor specific component models (like WebLogic Enterprise, which is CORBA based) to standard and portable components
  - Integration of Web technologies with Object Transaction Monitor technology to provide scalable Web servers and services

- Java component models
  - Enterprise Java Beans
    - server-side component models: interprocess components
  - Java Beans
    - client-side component models: intraprocess components
  - The two models share nothing but the programming language and name!

Server-side components appeared first in the context of CORBA-based OTMs. The difficulty is that the component models used by different vendors were not standardized, and thus components would not be reusable across platforms. The standardization of a CORBA based component model, as attempted in CORBA 3.0, lagged behind development. In parallel, Sun defined a Java-based component model, that was heavily borrowing from the CORBA model. This component model, which is called Enterprise Java Beans (EJBs), and is in the meanwhile the de-facto standard. Basing the component model on the Java programming language had the additional advantage that a strong integration with other Web-technologies was made possible.
The EJB model is the core of the J2EE platform, the Java 2 Enterprise Edition platform of Sun. This figure shows the different parts of J2EE. As we can see we are already familiar with most of its parts. So in the following we want to complete the picture by introducing the EJB model in detail.
The core of the EJB architecture is the component model, which is illustrated here. A component always distinguishes between an EJB home (object) and an EJB (remote) object. The home interface consists of methods, that are used to handle the bean's life-cycle, whereas the remote interface are the "business" methods, that a client can invoke on the EJB object (often called the "bean"). Other equivalent terms for "home" would be "factory" or "class object". The home is implemented by the container, whereas the implementations of the "business" methods the EJB object, are supplied by the developer through a so-called bean class. The home object together with the EJB object make up the implementation of the EJB that is running in the EJB container. The properties of the bean can be controlled by a deployment descriptor. The objects (home and EJB objects) are accessed from the client through stubs (exactly as in CORBA). The stubs are generated by the platform. In the figure those parts that are supplied by the developer are darker, whereas the system-generated parts of the EJB are kept lighter.

The Deployment Descriptors contain the specifications for runtime management of beans (transactions etc.) and are packaged as XML file.

In CORBA the EJB object would correspond to the CORBA object and the bean class would correspond to the servant.
Steps in Developing an EJB

- **Write a home interface**
  - extends `javax.ejb.EJBHome`
  - lists one or more create methods that can be used to create an instance of this enterprise bean
  - will be implemented by the container

- **Write a remote interface**
  - extends `javax.ejb.EJBObject`
  - describes the business methods of the bean

- **Write an enterprise bean class**
  - implements `javax.ejb.SessionBean` or `javax.ejb.EntityBean`
  - the method names and signatures must exactly match the method names and signatures of the remote interface

- **Write a deployment descriptor**
  - specifies functional properties of the bean

For developing a bean the "dark grey" boxes on the previous slide need to be supplied: This includes the interfaces, namely the home interface derived from EJBHome, declaring the methods to be used for creating beans and the remote interfaces derived from EJBOBJECT for declaring the business methods that are supported by the beans.

The implementations for the remote interface are then supplied as implementations of classes SessionBean or EntityBean (the difference will be explained later). For the home interface methods no implementations need to be supplied, but for persistent beans (entity beans) a primary key class needs to be implemented to support the container in managing the objects. Finally the deployment descriptor is supplied.
We illustrate the various steps by means of an example for implementing an "author" bean:

First the home interface is defined. One can see that it contains two methods: a create method, which returns a bean object (The fact that the method returns an object of an application specific type is the reason why the developer has to declare, though not to implement, this method), and the findByPrimaryKey method, which allows to find an object based on its persistent identifier. The type of the identifier is authorId, which is the primary key class. It we will described on the next slide and is also supplied by the developer. A method for destroying the object is inherited from the EJBHome interface. It needs not to be declared since it contains no application-specific types.

The remote interface declares the application specific methods of the bean.
EJB Example: Primary Key Class

- The primary key class

```java
public class authorId implements java.io.Serializable {
    public int id;

    public int hashCode() {
        return id;
    }

    public boolean equals(Object obj) {
        if (obj instanceof authorId) {
            if (((authorId)obj).id==id)
                return true;
        }
        return false;
    }
}
```

The primary key class implements the functions that are needed in order to support the container in identifying objects that are stored in stable storage. The methods for hashing an identifier and for testing equality of identifiers are required by the container in order to implement methods making use of the primary key, such as findByPrimaryKey.
EJB Example: Bean Class

- Bean class

```java
import javax.ejb.*;

public abstract class authorBean implements EntityBean {
    private EntityContext ctx;
    
    public abstract String getAuthId();
    public abstract void setAuthId(authorId aId);
    public abstract String getName();
    public abstract void setName(String nam);
    
    public void setEntityContext(EntityContext ctx) {
        this.ctx = ctx;
    }
    public void unsetEntityContext() {
        this.ctx = null;
    }
    
    ...
```

The bean class itself implements the application-specific methods as well as the life-cycle management methods, that are required by the container. Even if the life-cycle methods are not performing any actions, they need to be provided with empty implementations. The only life cycle method that performs an activity for the author bean is ejbCreate. It is used to set the identifier of the object to the ID value supplied by the container.

An entity bean with container-managed persistence has persistent and relationship fields. These fields are virtual, they do not need to be coded in the class as instance variables. Instead, they are specified in the bean's deployment descriptor. To permit access to the fields, abstract get and set methods must be defined in the entity bean class.

As can be seen from the remote interface, the only business methods implemented in the example are one get and one set.
Bean Class - Continuation

... public void ejbActivate() { } public void ejbPassivate() { } public void ejbLoad() { } public void ejbStore() { } public void ejbRemove() throws RemoveException { } public String ejbCreate(authorId aId, String nam) throws CreateException {
    setAuthId(aId);
    setName(nam);
    return null;
}

public void ejbPostCreate(authorId aId, String nam) { }
Finally a deployment descriptor is supplied. It provides the necessary information in order to locate all parts of the implementation and declares some functional properties of the bean. In this example it is declared, that the persistent state of the bean should be managed by the container and the bean is non-reentrant, which essentially forbids cyclic invocations of the bean by other beans. Once all the parts of the bean implementation are available, they are packaged, and can be deployed on a container.
Session and Entity Beans

- **Entity beans**: persistent, used to model data
  - Container-managed (implicit)
  - Bean-managed (explicit)

- **Session beans**: non-persistent, used to model processes
  - Stateful vs. stateless
  - Can implement the interaction among entity beans

- **Stateless session beans**
  - No state is maintained in between calls

- **Stateful session beans**
  - Sessions: maintain a state from one invocation to the next for a particular client
    (state not shared and not persistent)
  - Allows to implement conversations with the client
  - Are dedicated to a single client and are not pooled
  - Have a preset timeout period

We have already mentioned the fact that there exist different kinds of beans. The differences are related to whether beans have a persistent state and whether they support sessions with clients.

Entity beans are the beans that have a persistent state, that are thus used to model data and that survive the lifetime of any process using the bean. For entity beans one has two options of implementation: explicit or implicit. With implicit management of persistent state, the container is taking over the task to make the bean's state persistent. That means it implements a mapping to the storage system. With bean-managed persistence the developer is responsible to manage the persistent state. This is the approach that typically would be taken if the persistent state of the beans is stored in existing databases, that would for example be accessed via JDBC. The developer has in that case to supply implementations of methods, that are used by the container to read and write object state to and from the persistent storage.

Session beans maintain no persistent state. That means, they exist only throughout the lifetime of a server process. However a distinction is made between beans that maintain a session with a client or not. Stateful session beans allow multiple invocations on the same bean for a particular client (similarly as we have already seen earlier for servlets). Note that, maintaining the state for a session is not related to the problem of having a persistent state, the state is only kept within the transient memory of the server process. As for servlets, the advantage is that complex conversations with clients can be supported, as they are for example required in ecommerce interactions. The sessions are always exclusive to one client and are terminated after a timeout is exceeded.

The distinction between entity and session beans is comparable to the distinction between transactional and recoverable objects in the CORBA transaction service model.
Resource Management

- EJB objects can be dynamically associated with Bean Instances
  - Several clients access the same bean instance through different EJB objects
  - Like CORBA objects with servants
- Advantages
  - Fewer resources needed (memory)
  - Reusing instances more efficiently (fewer object creations and deletions)

Similarly as with a POA also the association of EJB objects (the equivalent of a CORBA object) and Bean Instances (the equivalent of a servant) can be managed in a flexible manner, such that different EJB objects can share different bean instances from a pool of bean instances. In fact, it is the normal modus operandi of an EJB container. This reduces the amount of resources consumed (memory) and improves performance as fewer objects need to be created and deleted. The basic life cycle of a bean, as perceived from the client side, consists of invoking a create() method on the EJB home which creates a new instance of the EJB object class, which is then dynamically associated with a bean instance from a bean instance pool. Then the reference to the EJB object is returned to the application.

Depending on the bean type (entity, stateless session, statefull session) the life cycles of the beans as seen from a system perspective (or better: the container perspective) differ substantially.
Here we see the life cycle of an entity bean. We discuss it step by step passing through the different phases of the bean life:

Initially the bean does not exist. When the server is started and the container is initiated, the first thing it does, is to instantiate bean instances (objects of the bean implementation class, e.g. authorBean) and put them into the pool. How many of them are initiated is system-specific. Before the bean object is put into the pool (and thus into the pooled state) the setEntityContext() method is called. It gives the bean instance a handle on the EntityContext object, which in turn gives the bean instance the possibility to access throughout its lifetime relevant information from the container on the EJBObjects it is associated with (and that are changing).

The next step is the creation of the EJBObject which is initiated by the user. As part of the creation the EJB object the container invokes the ejbCreate() and ejbPostCreate() methods from the bean interface. ejbCreate() creates typically a primary key and ejbPostCreate() can already make use of this primary key. Before methods can be invoked on beans with bean-managed persistence, the ejbLoad() method is invoked in order to retrieve the bean's persistent state from the stable storage. Then the bean is in the ready state and the business methods are called by the client. There exist now two ways of how the bean can return from the ready state back to the pooled state, i.e. of how the bean instance can be dissociated from the EJBObject. Either initiated by the container after a timeout (that is system-specific), i.e. when the bean has not received methods for a while, or by the user when he removes the bean with remove(), which implies that actually the object is deleted and also removed from the persistent storage. With container-initiated passivation, the ejbStore() method is used to save the beans state (if the state is bean managed) and then the ejbPassivate method is invoked. Passivated beans still are connected to the clients by means of the EJBObject, and as soon as methods are invoked they are again activated, which implies that the container invokes the ejbActivate() and ejbLoad() methods. When the bean is destructed the ejbRemove() method is invoked. The life cycle of the bean instance ends when the container decides to do so, either because it wants to free the resources for other purposes or the server is shut down.
The life cycle of stateless session beans is substantially simpler than the one for entity beans. All the method invocations from the side of the container are self-explaining. The beans in the method-ready pool answer incoming request and then immediately return into the pool. The container controls the number of bean instances that are kept in the pool, depending on the request load. Note that since the EJBOBJECTs are not persistent, no identifiers need to be supplied by the application, and the ejbCreate() method is executed when the bean instance is created, not when the client invokes the create() method for the EJBOBJECT from the home interface, when it wants to access the bean.
The life cycle of a stateful session bean is again quite different from the two previous ones. The difference lies in the fact that the stateful session beans are not using session pooling. In order to save resources the only thing that can occur, is that the bean is passivated. This implies that the bean instance is removed from the memory and the conversational state of the session is preserved.

Therefore the life cycle of the bean starts differently: upon the invocation of the create() method on the home interface by the client, a EJB object is created AND a bean instance is put in the method-ready pool and associated with the EJBOBJECT after executing the setSessionContext() and ejbCreate() methods. The assignment of the SessionContext reference allows the bean instance to access information on the session of the EJBOBJECT, to which it is associated with.

Then the business methods are executed. After a timeout the bean is passivated by the container and its conversational state is conserved. Upon the invocation of other business methods the bean is again activated. If it is not accessed a too long period the bean is removed (this also happens when the bean instance throws an exception). It can also be the case that beans are directly removed from the method-ready state. Also the client can remove the bean by calling the remove() method on the EJBOBJECT.
Transactions

- Managed explicitly or implicitly
  - Explicitly: Java transaction API (JTA)
  - Implicitly (normal): EJB transaction attributes
    - Set for the whole bean or particular methods
    - Specified in the deployment descriptor

- EJB transaction attributes for methods
  - NotSupported: the method does not take part in the transaction of the caller
  - Supports: the method will be executed within the callers transaction if it exists
  - Required: the method must be executed within a transaction, either it is called as part of a transaction and takes part in that transaction or it initiates a new transaction
  - RequiresNew: always a new transaction is started when the method is called
  - Mandatory: the caller must call the method from within a transaction
  - Never: the method never must be called from within a transaction

Transactions for bean methods can be handled in two ways: explicitly and implicitly. Explicit transaction management is what we would call in the CORBA transaction service indirect management with implicit propagation, and corresponds to the procedural style of transaction invocation as known already from X/Open, and as provided in the Java world by means of JTA. Implicit management of transactions is however the standard way of how beans should be used: it takes fully advantage of the possibility of declaring transactional properties of methods in the deployment descriptors. The transactional properties can there be declared either for the whole bean or for specific methods of the bean. The different possibilities of how methods participate (or not participate) in transactions are listed.
Example Specification

```xml
<ejb-jar>
  <enterprise-beans>
    <entity>
      <ejb-name>authorEJB</ejb-name>...
    </entity>
    <assembly-descriptor>
      <container-transaction>
        <method>
          <ejb-name>authorBean</ejb-name>
          <method-name>setName</method-name>
        </method>
        <trans-attribute>Required</trans-attribute>
      </container-transaction>
    </assembly-descriptor>
  </enterprise-beans>
</ejb-jar>
```

This example shows how in an EJB deployment descriptor the transactional properties are set for our example bean authorBean. For the method setName of the bean authorBean (which is an entity bean), the transaction attribute is set to "required". This means that either it must be invoked within a transaction or it starts itself a new transaction. This makes of course sense for a method that updates the state of the entity bean.
Persistence

- Two possibilities
  - Container-managed persistence
  - Bean-managed persistence

- Container-managed persistence
  - Depends on underlying storage systems
  - Usually object-to-relational persistence: mapping to relational DB
  - Based on the mapping definition the persistence is automatically managed by the container
  - Mapping can be complex and costly

- Bean-managed persistence
  - Typically required when persistent data is stored in legacy database
  - Application developer has to provide the code to implement persistent storage of bean objects (e.g. using JDBC)

We have already mentioned the two possibilities of managing the persistent state of entity beans: either by the container or by the bean (resp. the application).

The implementation of container-managed persistence depends clearly on the EJB platform and the available storage systems. The standard solution is that the container uses an object-to-relation mapping and stores the beans state in a relational database. The mapping itself must be provided by the developer, for example, in a form of an extended deployment descriptor. As we also know from the problem of mapping XML to a relational representation, this can be a non-trivial task. Care has to be taken that the mapping does not introduces substantial inefficiencies.

The second alternative is to store the beans state in a database and access the database through a programming language API, in the Java world naturally through JDBC. Then the application developer has to provide the methods in order to store and retrieve the state from the database. These methods are made available to the container through the ejbLoad() and ejbStore() methods.
Example: Container-Managed Persistence

- Definition of a relational table
  
  ```sql
  CREATE TABLE AUTHOR(ID INT PRIMARY KEY, NAME CHAR(3))
  ```

- Primary key class property must be public
  
  ```java
  public class authorId implements java.io.Serializable {
      public int id;
      ...
  }
  ```

- Deployment descriptor lists the (potentially) persistent fields
  
  ```xml
  <ejb-jar>
    <enterprise-beans>
      <entity>
        ...
        <prim-key-class>epfl.lib.author.authorId</prim-key-class>
        <persistence-type>Container</persistence-type>
        ...
        <cmp-field><field-name>authId</field-name><cmp-field>
        <cmp-field><field-name>name</field-name><cmp-field>
      </entity>
    </enterprise-beans>
  </ejb-jar>
  ```

- The home interface supports the `findByPrimaryKey()` method
  
  - implemented by the container
  - Other "finder" methods can be provided

In the following we give a more detailed description of how a mechanism for container-managed persistence works. Let us assume that we have defined a table author that should be used in order to store the state of the author bean, that we have used in the earlier examples. One has to take care, that the primary key class is declared public (which we did already, wisely enough). The deployment descriptor now contains information on

- the primary key of the entity bean (tag `<prim-key-class>`)  
- The type of persistence management, namely container-managed persistence (tag `<persistence-type>`)  
- And a list of the fields that should be made persistent (tag `<cmp-field>`, `cmp=container-managed-persistent`)  

This part of the descriptor is according to the EJB standard. Given this information the container will, for example, by default provide an implementation for the `findByPrimaryKey()` method, in order to locate and access entity beans from the stable storage. In order to be able to do this the container needs however further information on the mapping to the database. The specification of this mapping is vendor-specific. We look at it next.
There exist different solutions for specifying the mapping, including graphical interfaces or command line interface. We look at the solution that is taken by the Weblogic Server which is particularly elegant, since it builds on the already existing concept of deployment descriptors.

We assume, that in addition to the relational tables, also application specific methods for locating objects have been included in the home interface as extensions of the standard ejbHome finder methods, such as a method findAuthor, that locates an author based on his/her name and the ID. The implementation of such a method will be given in the mapping specification.
Example: WebLogic RDBMS Deployment Descriptor

```xml
<!DOCTYPE weblogic-rdbms-bean $
PUBLIC "-//BEA Systems, Inc.//DTD WebLogic 5.1.0 EJB RDBMS Persistence//EN" 
'http://www.bea.com/servers/wls510/dtd/weblogic-rdbms-persistence.dtd'>
<weblogic-rdbms-bean>
<pool-name>demoPool</pool-name>
<table-name>authors</table-name>
<attribute-map>
<object-link>
  <bean-field>authId</bean-field>
  <dbms-column>id</dbms-column>
</object-link>
<object-link>
  <bean-field>name</bean-field>
  <dbms-column>name</dbms-column>
</object-link>
</attribute-map>
<finder-list>
  <finder>
    <method-name>findAuthor</method-name>
    <method-params>
      <method-param>string</method-param>
    </method-params>
    <finder-query><![CDATA[(= name $0)]]></finder-query>
  </finder>
</finder-list>
</weblogic-rdbms-bean>
```

Here we see now of how the mapping from the relational table to the EJBOBJECT is given.

- First the table name is given (tag `<table-name>`)  
- Then an attribute map, consisting of the mappings of different attributes, is given (tag `<attribute-map>`)  
- Within the attribute map a mapping between object attributes and table attributes is established (tags `<object-link>`). For example, the bean attribute authId is mapped to the table attribute id.
- Then a finder list is given, i.e. methods used to locate objects in the database (tag `<finder-list>`). For each finder method the method name, the method parameters and a query (actually an awkwardly encoded relational query) implementing it are given.

One can in particular notice that the expressiveness of the mapping is rather limited. It allows only to map single attributes to single columns. Thus, this approach can only be used in order to map to databases that have been specifically designed for storing the bean objects' state. More recent versions of Weblogic provide for that purpose tools to generated the corresponding tables automatically.
Example: Storage Mapping Through GUI

©2003, Karl Aberer, EPFL-SSC, Laboratoire de systèmes d'informations répartis

Part 5 - 54
Summary

- **Object Transaction Monitors**
  - Combine the worlds of distributed object computing and transaction monitors
  - Provide runtime environments (containers) for components
- **Components**
  - Ready-to-run software packages that support an object-oriented client interface and support a component interface (contract) with the container
  - EJBs are the Java-based component model
- **Web Application Servers**
  - Are forming the backbone of the Internet computing infrastructure

The presence of declarative container services does not absolve the bean developer (and deployer) from understanding how transactions, communications, and exception handling work in the context of EJB technology.

References

- **Books**

- **Websites**
  - www.omg.org
  - OMG standard documents on services
  - WebLogic EJB OTM: [http://www.weblogic.com/docs51/resources.html](http://www.weblogic.com/docs51/resources.html)
Integration of Distributed Objects Checklist

• Task: integrate distributed, heterogeneous (and probably autonomous) transactional resources (like databases)

• Abstract Model ✓
  - Object model, IDL, EJB

• Embedding ✓
  - Object adaptors, Containers

• Architectures and Systems ✓
  - Object Management Architecture (OMA)
  - J2EE

• Methodology