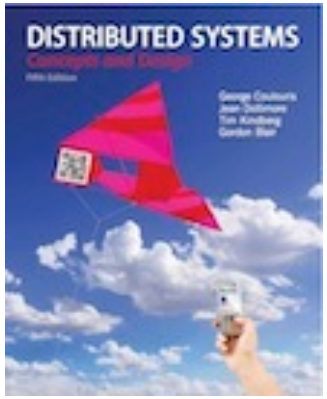
Slides for Chapter 8:
Distributed Objects and Components



From **Coulouris, Dollimore, Kindberg and Blair**

Distributed Systems:

Concepts and Design

Distributed Object-based Systems

- Common paradigm used to organize distributed systems: **distributing objects**!
- Object orientation makes easier to build software by means of independent (→ distributed) components.
- Everything is treated as an object:
  - clients can access to services and resources in the form of objects they can invoke.
- Objects allow to easily hide distribution aspects behind their interface.
- Hence, programmers can concentrate on implementing their functionality in an independent way.
- Moreover, developers benefit from using:
  - richer programming abstractions (using familiar languages);
  - object oriented design principles, tools and techniques.

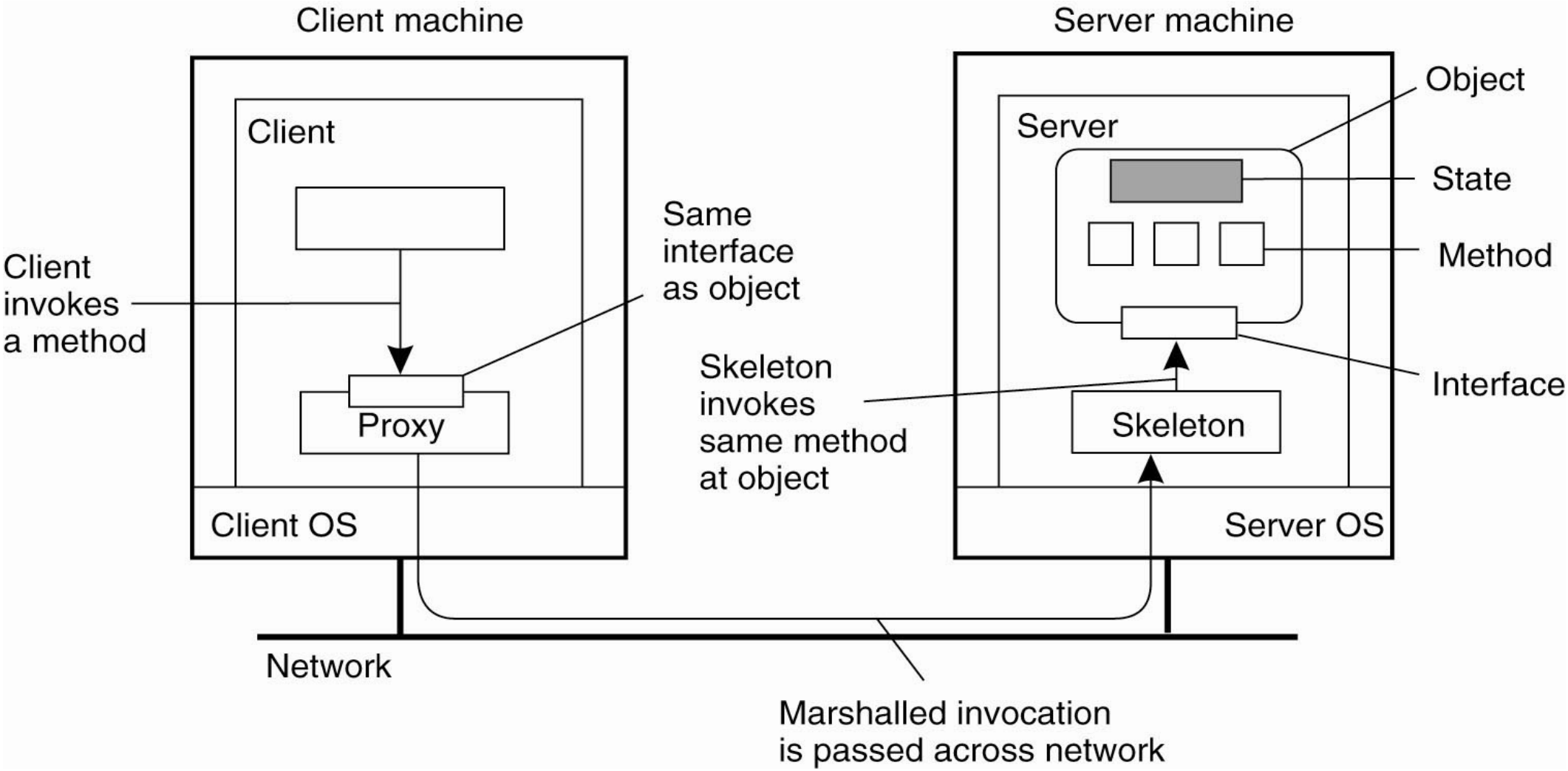
Distributed Objects

- An object encapsulates:
  - data → **state**;
  - operations on data → **methods**.
- Methods are made available through an **interface**:
  - the state can be manipulated only through the interface;
  - an object may implement multiple interfaces;
  - given an interface, there may be several objects implementing it.
- The above mentioned separation between interfaces and objects implementing them is crucial:
  - an interface can be placed at one machine, while a related object resides at another machine → **distributed object**.

Distributed Objects

- When a clients binds to a distributed object:
  - a proxy (i.e., an implementation of the object's interface) is loaded into the client's address space → similar to a client stub in RPC.
- At a server machine where the object resides:
  - the same interface is referred to as a skeleton.
- Most distributed objects do not have a distributed state.
- When the state resides at a single machine, we speak of a remote object.

Distributed Objects



Common organization of a remote object with client-side proxy.

Compile-Time vs. Runtime Objects

- Compile-time objects:
  - directly related to language-level objects (e.g., in Java, C++, C# etc.);
  - an object is an instance of a class;
  - a class is a description of an abstract type.
- Compile-time objects are easy to use for building distributed applications:
  - an object is fully defined by its class and the implemented interfaces;
  - compilation yields code that can be used to instantiate objects;
  - interfaces can be compiled into client-side and server-side stubs;
  - this allows objects to be invoked from a remote machine.
- Main drawback: compile-time objects depend on a particular programming language.

Compile-Time vs. Runtime Objects

- Runtime objects allow to:
  - distribute objects during runtime,
  - build distributed applications from objects written in multiple languages.
- A common approach is to use object adapters (i.e., wrappers) giving the appearance of an object to a given implementation (e.g., a C library).
- Objects are thus defined only in terms of the interfaces they implement.
- Hence, an implementation of an interface can be registered as an adapter.
- Such adapter can make the interface available for (remote) invocations.

Persistent and Transient Objects

- A persistent object continues to exist even if it is not contained in the address space of any server process:
  - it is not dependent on its current server.
- A transient object exists only as long as the server hosting it.
- Most object-based distributed systems support both types.

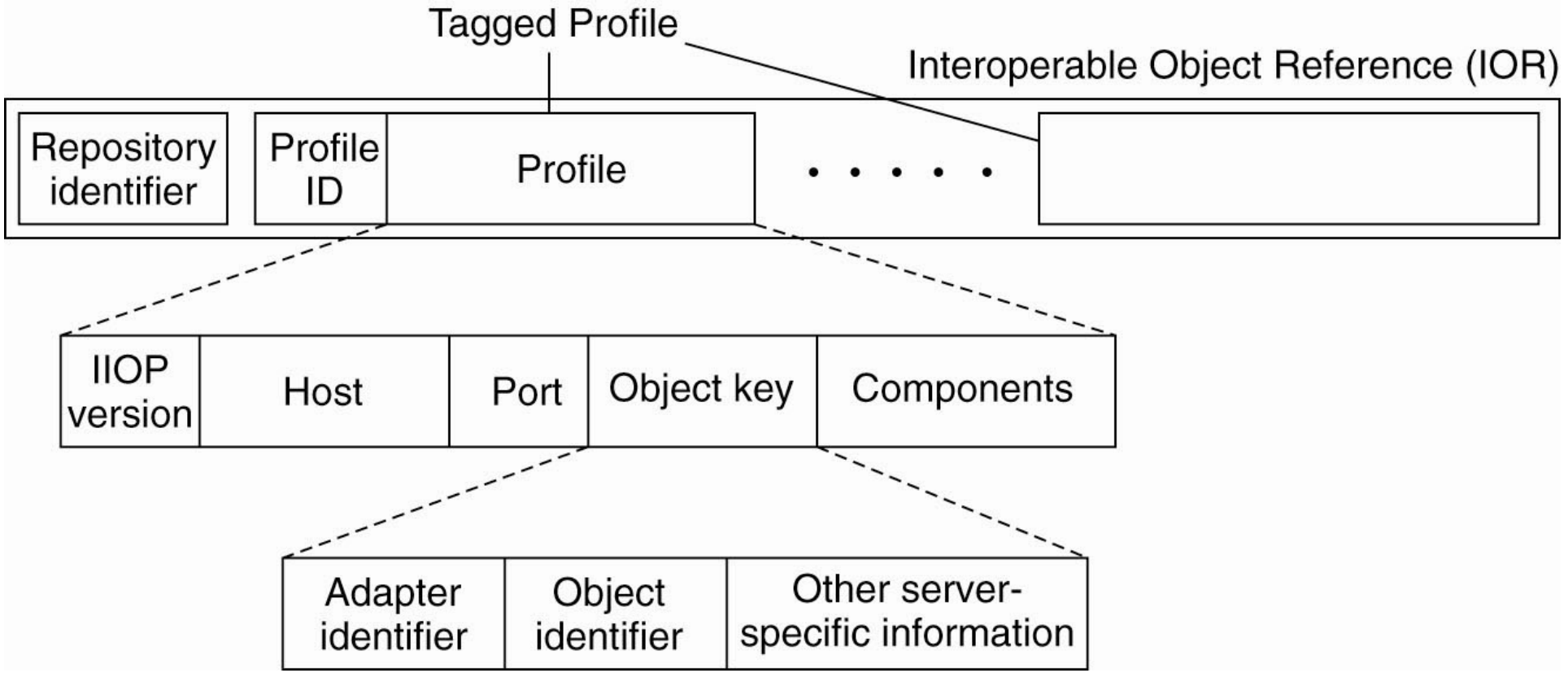
Distributed objects

Objects	*Distributed objects*	*Description of distributed object*
Object references	Remote object references	Globally unique reference for a distributed object; may be passed as a parameter.
Interfaces	Remote interfaces	Provides an abstract specification of the methods that can be invoked on the remote object; specified using an interface definition language (IDL).
Actions	Distributed actions	Initiated by a method invocation, potentially resulting in invocation chains; remote invocations use RMI.
Exceptions	Distributed exceptions	Additional exceptions generated from the distributed nature of the system, including message loss or process failure.
Garbage collection	Distributed garbage collection	Extended scheme to ensure that an object will continue to exist if at least one object reference or remote object reference exists for that object, otherwise, it should be removed. Requires a distributed garbage collection algorithm.

Object-Based Messaging (CORBA)

- **CORBA** (Common Object Request Broker Architecture) is a specification for distributed systems which aims to language independency.
- The communication layer is built on top of a **messaging** system.
- **Object references** held by processes are **different** from those implemented by the RTS (language-specific vs. process-independent).
- **Interoperable Object Reference (IOR)**: language-independent representation of an object reference.
- Layout of an IOR:
  - repository identifier,
  - one or more tagged profiles:
    - **Internet InterORB Protocol (IIOP)** version,
    - host field,
    - port field,
    - object key field (adapter, object, server-specific info),
    - components field.

CORBA Object References (2)



The organization of an IOR with specific information for IIOP.

CORBA IDL

- A CORBA IDL **interface** specifies
  - a **name**,
  - a set of **methods** that clients can request.
- In more detail, IDL provides facilities for defining modules, interfaces, types, attributes and method signatures.
- The syntax is a subset of ANSI C++ with additional constructs supporting method signatures.

IDL interfaces Shape and ShapeList

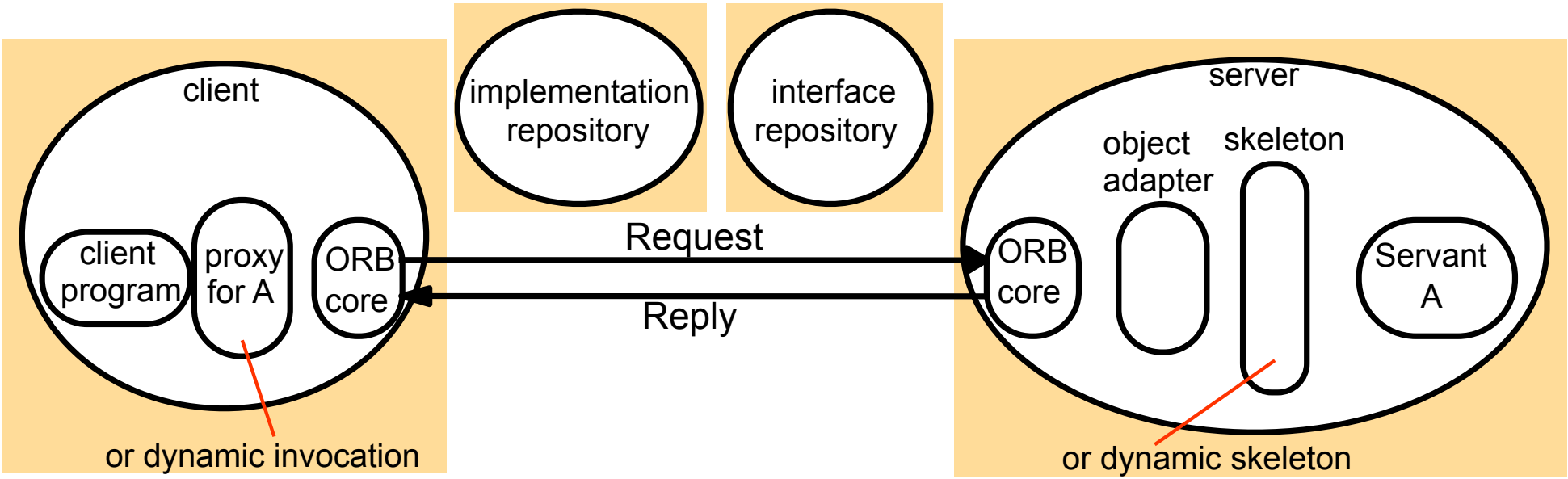
```
struct Rectangle{                                   1
        long width;
        long height;
        long x;
        long y;
} ;
```

```
struct GraphicalObject {                            2
        string type;
        Rectangle enclosing;
        boolean isFilled;
};
```

```
interface Shape {                                                                   3
        long getVersion() ;
        GraphicalObject getAllState() ;          // returns state of the GraphicalObject
};
```

```
typedef sequence <Shape, 100> All;                                                  4
interface ShapeList {                                                               5
        exception FullException{ };                                                 6
        Shape newShape(in GraphicalObject g) raises (FullException);                7
        All allShapes();                         // returns sequence of remote object references   8
        long getVersion() ;
};
```

The main components of the CORBA architecture



CORBA architecture

- The CORBA architecture is designed to support the role of an **object request broker**, enabling **clients** to **invoke methods** in **remote objects**.
- The main components are:
  - **ORB core**: includes all the communication functionalities;
  - **Object adapter**: bridges the gap between CORBA objects with IDL interfaces and the programming languages interfaces;
  - **Portable adapters**: provide standards between different developers;
  - (**Dynamic**) **Skeletons**;
  - **Client stubs/proxies**;
  - **Implementation repositories**: it activates registered servers on demand;
  - **Interface repository**: it provides information about IDL interfaces to clients and servers.

Client/server example

- We use Java as the client and server languages, but the approach is similar for other languages.
- The interface compiler `idlj` takes as input CORBA interfaces and generates the following items:
  - two Java interfaces per IDL interface:
    - the name of the first ends in **Operations** (operations of IDL interface);
    - the name of the second is the **same** of the **IDL interface** (operations of the first interface + CORBA object methods);
  - the server **skeletons** for each IDL interface (class names end in **POA**);
  - **proxy classes/client stubs** (names ending in **Stub**);
  - a Java **class** for each of the **structs** in the IDL interface;
  - **Helper** classes providing the narrow method;
  - **Holder** classes dealing with out and inout parameters which cannot be mapped directly onto Java.

Using idlj

- Download the code from http://www.cdk5.net/corba/Java%201.4/
- In the directory where you downloaded the source files, type the following command:
  - **`idlj -fall Shape.idl`**
- Then a bunch of Java files is generated:
  - **`ShapeOperations.java`**, **`ShapeListOperations.java`**, **`Shape.java`** and **`ShapeList.java`**;
  - **`ShapePOA.java`** and **`ShapeListPOA.java`**
  - **`_ShapeStub.java`** and **`_ShapeListStub.java`**
  - **`Rectangle.java`**, **`GraphicalObject.java`** and **`ShapeListPackage/FullException.java`**
  - **`*Helper.java`**, **`*Holder.java`** and **`ShapeListPackage/FullExceptionH*.java`**

Java interfaces generated by *idlj* from CORBA interface *ShapeList*

```
public interface ShapeListOperations {
    Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException;
    Shape[] allShapes();
    int getVersion();
}

public interface ShapeList extends ShapeListOperations, org.omg.CORBA.Object,
            org.omg.CORBA.portable.IDLEntity { }
```

ShapeListServant class of the Java server program for CORBA interface *ShapeList*

```
import org.omg.CORBA.*;
import org.omg.PortableServer.POA;
class ShapeListServant extends ShapeListPOA {
    private POA theRootpoa;
    private Shape theList[];
    private int version;
    private static int n=0;
    public ShapeListServant(POA rootpoa){
        theRootpoa = rootpoa;
        //  initialize the other instance variables
    }
```

// continued on the next slide

... continued

```
public Shape newShape(GraphicalObject g)
        throws ShapeListPackage.FullException {                    1
        version++;
        Shape s = null;
        ShapeServant shapeRef = new ShapeServant(g, version);
        try {
            org.omg.CORBA.Object ref =
                theRoopoa.servant_to_reference(shapeRef);          2
            s = ShapeHelper.narrow(ref);
        } catch (Exception e) {}
         if(n >=100) throw new ShapeListPackage.FullException();
        theList[n++] = s;
        return s;
    }
    public  Shape[] allShapes(){ ... }
    public int getVersion() { ... }
}
```

Java class *ShapeListServer*

```
import org.omg.CosNaming.*;   import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;       import org.omg.PortableServer.*;
public class ShapeListServer {
      public static void main(String args[]) {
         try{
            ORB orb = ORB.init(args, null);                                          1
            POA rootpoa =  POAHelper.narrow(orb.resolve_initial_references("RootPOA"));2
            rootpoa.the_POAManager().activate();                                     3
            ShapeListServant SLSRef = new ShapeListServant(rootpoa);                 4
            org.omg.CORBA.Object ref = rootpoa.servant_to_reference(SLSRef);         5
            ShapeList SLRef = ShapeListHelper.narrow(ref);
            org.omg.CORBA.Object objRef =orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);                6
            NameComponent nc = new NameComponent("ShapeList", "");                   7
            NameComponent path[] = {nc};                                             8
            ncRef.rebind(path, SLRef);                                               9
            orb.run();                                                               10
         } catch (Exception e) { ... }
      }}
```

Java client program for CORBA interfaces *Shape* and *ShapeList*

```
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
public class ShapeListClient{
    public static void main(String args[]) {
        try{
            ORB orb = ORB.init(args, null);                                    1
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            NameComponent nc = new NameComponent("ShapeList", "");
            NameComponent path [] = { nc };
            ShapeList shapeListRef =
                ShapeListHelper.narrow(ncRef.resolve(path));                   2
            Shape[] sList = shapeListRef.allShapes();                          3
            GraphicalObject g = sList[0].getAllState();                        4
        } catch(org.omg.CORBA.SystemException e) {...}
    }
```

Running the example

- Launch the ORB daemon orbd:
  - **`orbd -ORBInitialPort 1050 -ORBInitialHost localhost&`**
- Launch the server:
  - **`java ShapeListServer -ORBInitialPort 1050 -ORBInitialHost localhost&`**
  - **`ShapeList Server ready and waiting...`**
- Launch the client:
  - **`java ShapeListClient -ORBInitialPort 1050 -ORBInitialHost localhost&`**
  - **`Shape type = 1050`**
  - **`Added a shape`**

Distributed object middleware vs. component-based middleware

- **Distributed object** middleware:
  - OO programming model (encapsulation, data abstraction) hiding the complexity of distributed programming;
  - objects communicate using remote method invocation or distributed events.
- **Component-based** middleware is a natural evolution of the OO-based approach overcoming the following limitations:
  - implicit dependencies;
  - programming complexity;
  - lack of separation of distribution concerns;
  - no support for deployment.

Implicit dependencies

- A distributed object offers a **contract** to the outside world in terms of the interface provided.
- However, such **contract** is **not complete**
  - the encapulated behaviour is **hidden**;
  - use and **deployment** of the object may depend on **other objects** or **services** (e.g., in CORBA server and client communicate with the naming service).
- Hence, substitution, composition and implementation of objects is **difficult** and **error-prone**.

Interaction with the middleware

- The programmer is still exposed to **low-level details** associated with the middleware architecture:
  - e.g., in CORBA many essential calls are related to **naming** purposes, Portable Object Adapter (**POA)** and **ORB core** interaction.
- Hence, the programmer is **distracted** from the main purpose of the code.

Lack of separation of distribution concerns

- In distributed object middleware programmers have to deal **explicitly** with **non-functional concerns**:
  - security, transactions, coordination and replication.
- This is achieved with **ad-hoc calls**.
- Hence, programmers must have a **detailed knowledge** of the **internals** of the middleware services.
- The **complexity** of programming **increases**.

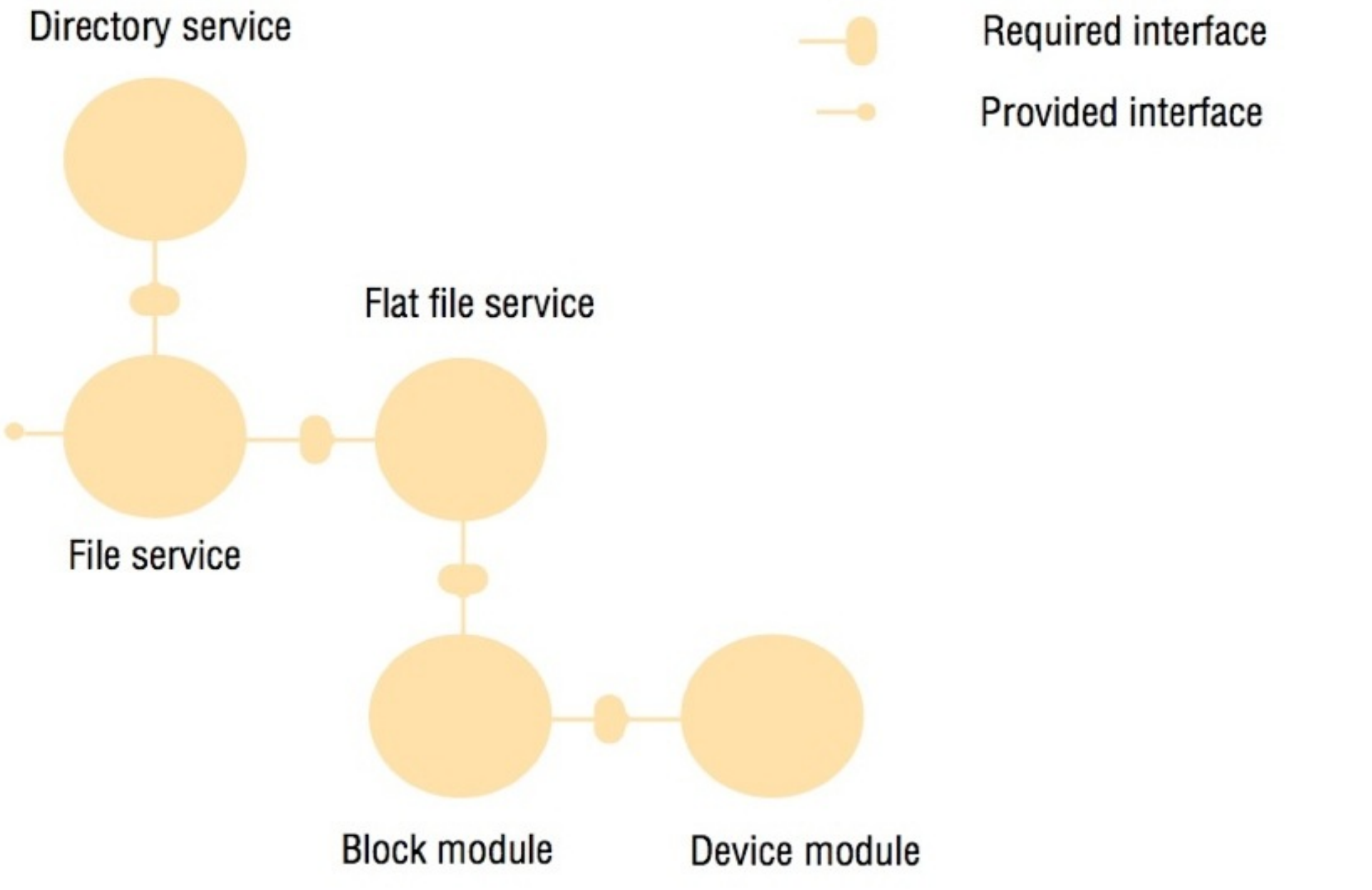
No support for deployment

- In distributed object middleware, objects must be deployed **manually** on individual machines.
- Thus, large deployments are **error-prone**.
- Moreover, programmers must resort to **ad-hoc techniques** which are **not portable** to other environments.

Essence of components

- **Components** definition [Szyperski, 2002]:
  - a software component is a unit of composition with contractually specified interfaces and explicit context dependencies **only**.
- The **contract** of a component is specified in terms of
  - a set of **provided** interfaces, i.e., services offered to other components;
  - a set of **required** interfaces, i.e., its dependencies.

An example software architecture



Components: interfaces

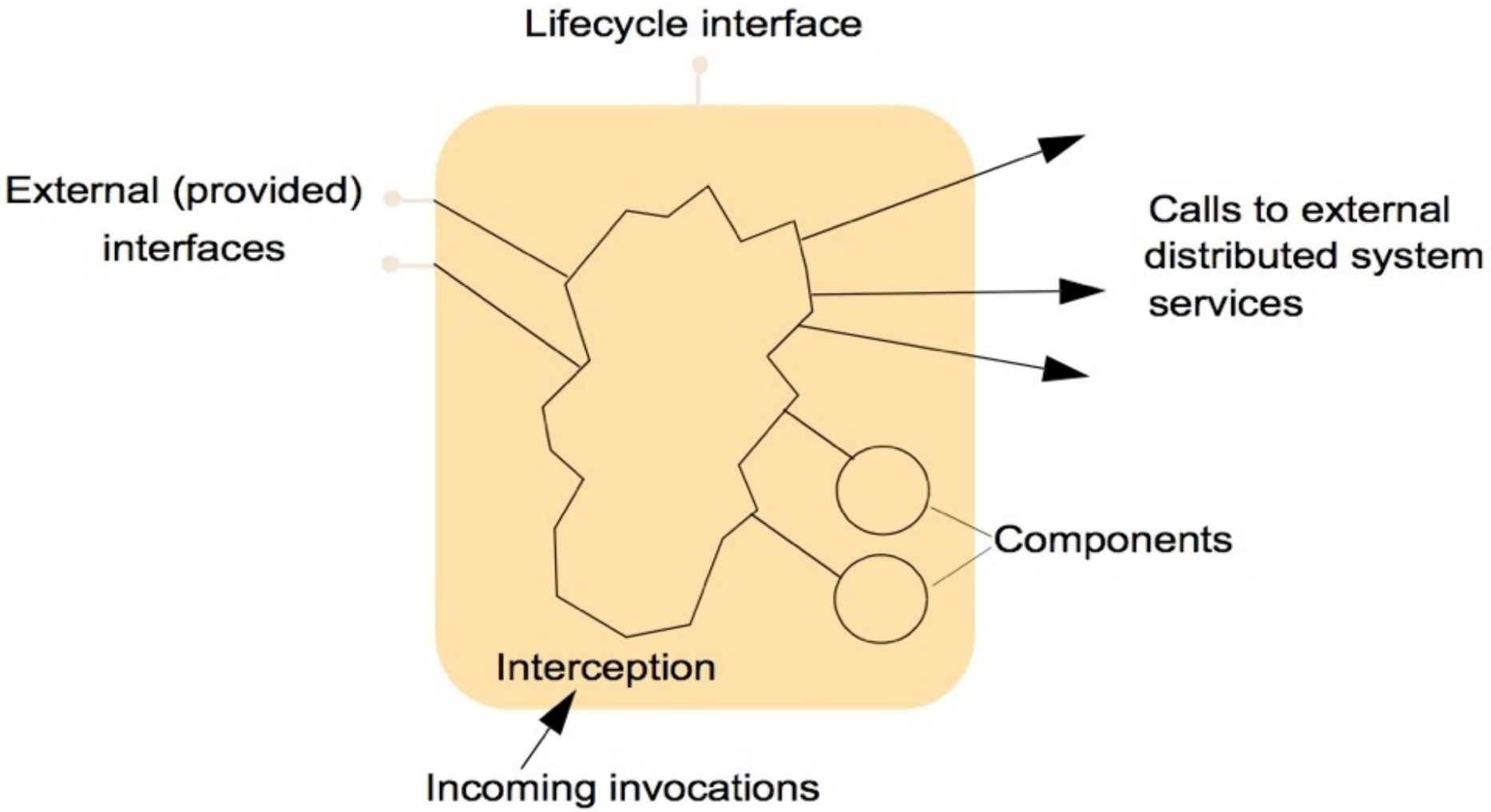
- Many component-based approaches offer two styles of interfaces:
  - interfaces supporting **RMI**;
  - interfaces supporting **distributed events**.
- Programming becomes a task of **composition**:
  - **software development** is replaced by **software assembly**;
  - **inheritance** is replaced by **composition**.
    - inheritance creates implicit dependencies between classes

- **Component-based** middleware has emerged as a promising **evolution** of distributed object systems.
- The key notions behind their success are:
  - **containers**;
  - support for **deployment**.

Containers

- **Containers** support a common pattern, consisting in:
  - a front-end (perhaps web-based) client;
  - a container holding one or more components implementing the application or business logic;
  - system services managing the associated data in persistent storage.
  - (Similar to three-tier architectures)
- Containers provide a **managed server-side hosting** environment:
  - **components** deal with **application** concerns;
  - **containers** deal with **distribution**, **middleware** issues etc.

The structure of a container



Support for deployment

- Middleware supporting the container pattern is known as an **application server**.
- An application server supports the deployment of **component configurations**:
  - software releases are packaged as software architectures (components + interconnections);
  - **deployment descriptors** (XML files) ensure that:
    - components are correctly connected;
    - the underlying middleware is properly configured;
    - the associated distributed system services are set up in order to provide the right level of security, transaction support, etc.

Application servers

Technology	*Developed by*	*Further details*
WebSphere Application Server	IBM	[www.ibm.com]
Enterprise JavaBeans	SUN	[java.sun.com XII]
Spring Framework	SpringSource (a division of VMware)	[www.springsource.org]
JBoss	JBoss Community	[www.jboss.org]
CORBA Component Model	OMG	[Wang *et al.* 2001]
JOnAS	OW2 Consortium	[jonas.ow2.org]
GlassFish	SUN	[glassfish.dev.java.net]

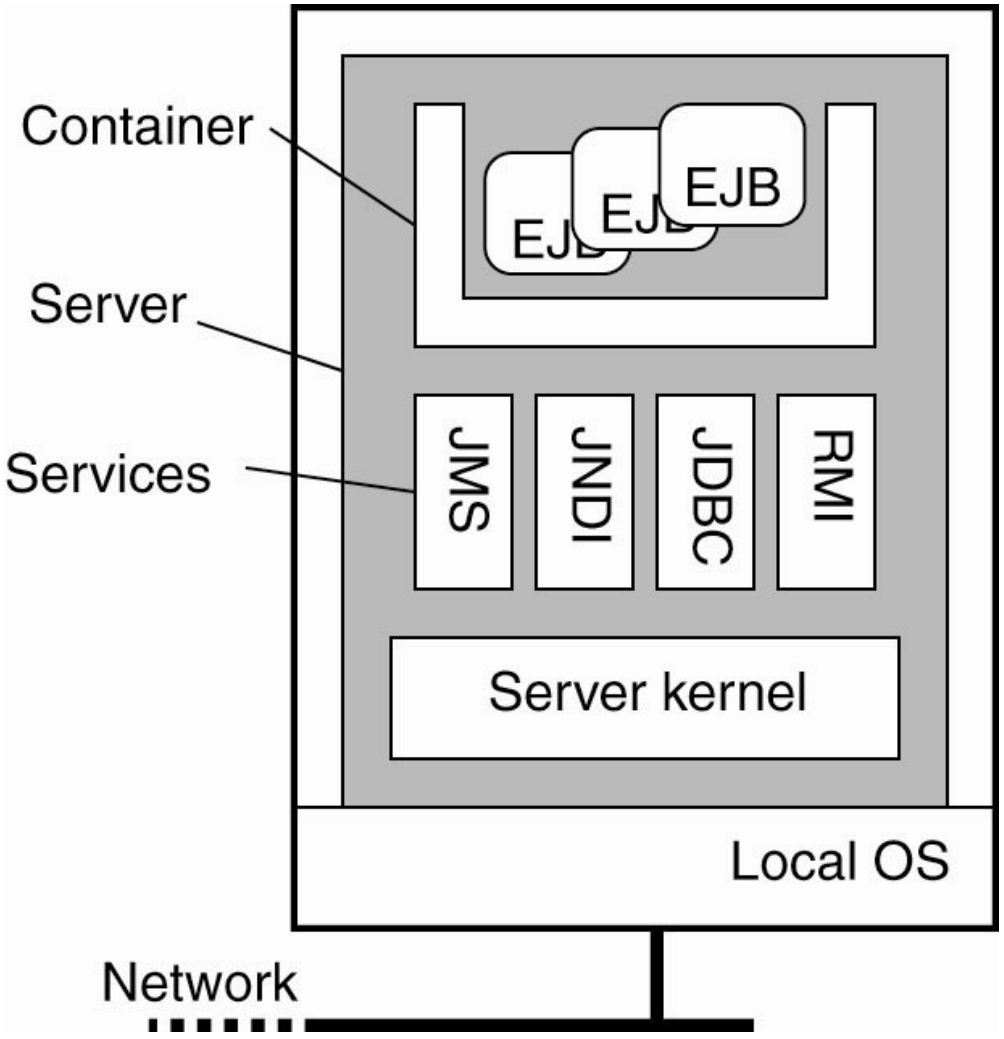
- Application servers provide support for the three-tier approach to distributed programming
- Pro: most complexities of distributed programming are hidden from the user/programmer
- Cons: this approach is
  - Prescriptive: it mandates the particular style of system architecture
  - Heavyweight: application servers are large and complex software systems, with considerable overheads
- This general approach is suited for high-end servers
  - Example: Enterprise Java Beans
- A *lightweight component model* can be adapted in simpler cases
  - Examples: Fractal, Apache Tomcat

EJB: Enterprise Java Beans (1)

- The popularity of Java together with its support for
  - object orientation
  - and remote method invocation

  formed the foundation for several distributed systems.
- Besides language support, there is the need of a **runtime environment** supporting (**three-tiered**) client-server architectures.
- Such support is provided by (Enterprise) Java Beans (EJB).
- EJBs are objects hosted by a special server allowing remote clients to invoke them.
- The server must separate:
  - **application** functionality,
  - **system-oriented** functionality.

EJB: Enterprise Java Beans (2) - general architecture of a EJB server



- EJB: **server-side** component model.
- Beans capture the **application** (business) **logic**.
- EJB supports the **separation** between the logic and its persistent storage in a back-end database.
- EJB is **managed** by default: the **container injects** appropriate calls to services (container-managed).
- However, it is **possible** for the bean developer to **control** in a **direct way** the access to services (bean-managed).

EJB: Enterprise Java Beans (3)

- EJB maintains a **strong separation** between the different **roles** involved in distributed applications:
  - bean provider;
  - application assembler;
  - deployer;
  - service provider;
  - persistence provider;
  - container provider;
  - system administrator.
- EJB is a **heavyweight** component architecture.

Four Types of EJBs

- **Stateless** session beans: the server maintains a variable amount of instances in a pool. Each time a client requests a method of a stateless bean, a random instance is chosen to serve that request. The same instance can serve different clients in different moments.
  - Best for beans without history, e.g., selection queries to a database.
- **Stateful** session beans: closely connected to the client. Each instance is created and bounded to a single client and serves only requests from that particular client.
  - Best for beans with history and a conversational state between the requests; e.g., electronic shopping cart.
- **Entity** beans: manages persistent data, performs complex business logic, and can be uniquely identified by a primary key
  - e.g., handling customer information.
- **Message-driven** beans: stateless, transaction-aware component, driven by a Java message. It is invoked by the EJB Container when a message is received from a JMS Queue or Topic.
  - e.g., publish-subscribe beans.

POJOs and annotations

- EJB **programming** is rather **simple** since it builds upon:
  - Enterprise **JavaBeanPOJOs** (Plain Old Java Object);
  - Enterprise JavaBean **annotations**.
- Essentially, a bean is a plain old Java object **decorated** with annotations ensuring its **correct behaviour** in the EJB context.

Examples of annotations

- Annotated bean definitions:
  - @Stateful public class eShop implements Orders {...}
  - @Stateless public class CalculatorBean implements Calculator {...}
  - @MessageDriven public class SharePrice implements MessageListener {...}
- Annotated interfaces:
  - @Remote public interface Orders {...}
  - @Local public interface Calculator {...}

EJB: transaction support

- Bean-managed transactions:

```
@Stateful
@TransactionManagement(BEAN)
public class eShop implements Orders
{
  @Resource
javax.transaction.UserTransation ut;

  public void MakeOrder(...) {
    ut.begin();
    ...
    ut.commit();
  }
}
```

- Container-managed transactions:

```
@Stateful
@TransactionManagement(CONTAINER)
public class eShop implements
Orders {
  ...
  @TransactionAttribute(REQUIRED)
  public void MakeOrder(...) {
    ...
  }
}
```

Transaction attributes in EJB.

Attribute	*Policy*
REQUIRED	If the client has an associated transaction running, execute within this transaction; otherwise, start a new transaction.
REQUIRES_NEW	Always start a new transaction for this invocation.
SUPPORTS	If the client has an associated transaction, execute the method within the context of this transaction; if not, the call proceeds without any transaction support.
NOT_SUPPORTED	If the client calls the method from within a transaction, then this transaction is suspended before calling the method and resumed afterwards – that is, the invoked method is excluded from the transaction.
MANDATORY	The associated method must be called from within a client transaction; if not, an exception is thrown.
NEVER	The associated methods must not be called from within a client transaction; if this is attempted, an exception is thrown.

EJB: interceptions

- The programmer is allowed to **intercept** two kinds of operations on beans:
  - **method calls** associated with a business interface;
  - **lifecycle events**.
- This mechanism can be used whenever it is needed to **associate** a particular set of **actions** with **incoming calls** on a business interface or with **events** regarding the lifecycle of a bean.

Interceptions example: logging

```
@Stateful
public class eShop implements Orders {
  public void MakeOrder() {
    ...
  }
  @AroundInvoke
  public Object log(InvocationContext ctx) throws Exception
  {
    System.out.println("The following method was invoked: "+
                ctx.getMethod().getName());
    return ctx.proceed();
  }
}
```

Invocation contexts in EJB

Signature	*Use*
public Object getTarget()	Returns the bean instance associated with the incoming invocation or event
public Method getMethod()	Returns the method being invoked
public Object[] getParameters()	Returns the set of parameters associated with the intercepted business method
public void setParameters(Object[] params)	Allows the parameter set to be altered by the interceptor, assuming type correctness is maintained
public Object proceed() throws Exception	Execution proceeds to next interceptor in the chain (if any) or the method that has been intercepted

Example: Hello World in EJB (1)

- Hello.java (remote interface):

```
package org.acme;
import javax.ejb.Remote;
@Remote
public interface Hello {
        public String sayHello();
}
```

- HelloBean.java (bean class):

```
package org.acme;
import javax.ejb.Stateless;
@Stateless
public class HelloBean implements Hello {
        public String sayHello(){
                return "Hello World!!!!";
        }
}
```

- Compiling:

```
$ javac -classpath $OPENEJB_HOME/lib/javaee-api-5.0-3.jar *.java
```

- Packaging:

```
$ jar cvf hello.jar org
```

Example: Hello World in EJB (2)

- The EJB client:

```
package org.acme;
import java.util.Properties;
import javax.naming.InitialContext;
import javax.naming.Context;
import javax.rmi.PortableRemoteObject;
public class HelloClient{
        public static void main(String[] args) throws Exception{
                Properties props = new Properties();

props.put(Context.INITIAL_CONTEXT_FACTORY,"org.apache.openejb.client.RemoteInitialContextFactor
y");
                props.put(Context.PROVIDER_URL,"ejbd://127.0.0.1:4201");
                Context ctx = new InitialContext(props);
                Object ref = ctx.lookup("HelloBeanRemote");
                Hello h = (Hello)PortableRemoteObject.narrow(ref,Hello.class);
                String result = h.sayHello();
                System.out.println(result);
        }
}
```

- Compiling:

```
$javac  HelloClient.java
```

Example: Hello World in EJB (3)

- Starting the server (OpenEJB - openejb.apache.org):

```
$ openejb start
Apache OpenEJB 3.1.4    build: 20101112-03:32
http://openejb.apache.org/
log4j:WARN No appenders could be found for logger (org.apache.openejb.resource.activemq.ActiveMQResourceAdapter).
log4j:WARN Please initialize the log4j system properly.
[init] OpenEJB Remote Server
  ** Starting Services **
  NAME                 IP              PORT
  httpejbd             127.0.0.1       4204
  admin thread         127.0.0.1       4200
  ejbd                 127.0.0.1       4201
  ejbd                 127.0.0.1       4203
  hsql                 127.0.0.1       9001
  telnet               127.0.0.1       4202
-------
Ready!
```

- Deploying the object:

```
$ openejb deploy hello.jar
Application deployed successfully at "hello.jar"
App(id=/Users/ivan/openejb-3.1.4/apps/hello.jar)
    EjbJar(id=hello.jar, path=/Users/ivan/openejb-3.1.4/apps/hello.jar)
        Ejb(ejb-name=HelloBean, id=HelloBean)
            Jndi(name=HelloBeanRemote)
```

- Running the client:

```
$ java -cp /Users/ivan/openejb-3.1.4/lib/openejb-client-3.1.4.jar:/Users/ivan/openejb-3.1.4/lib/javaee-
      api-5.0-3.jar:. org.acme.HelloClient
Hello World!!!!
```