



# Tehnologii Java

*Curs -*

Cristian Frăsinaru

`acf@infoiasi.ro`

Facultatea de Informatică

**Universitatea "Al. I. Cuza" Iași**





# Enterprise Java Beans



# Cuprins

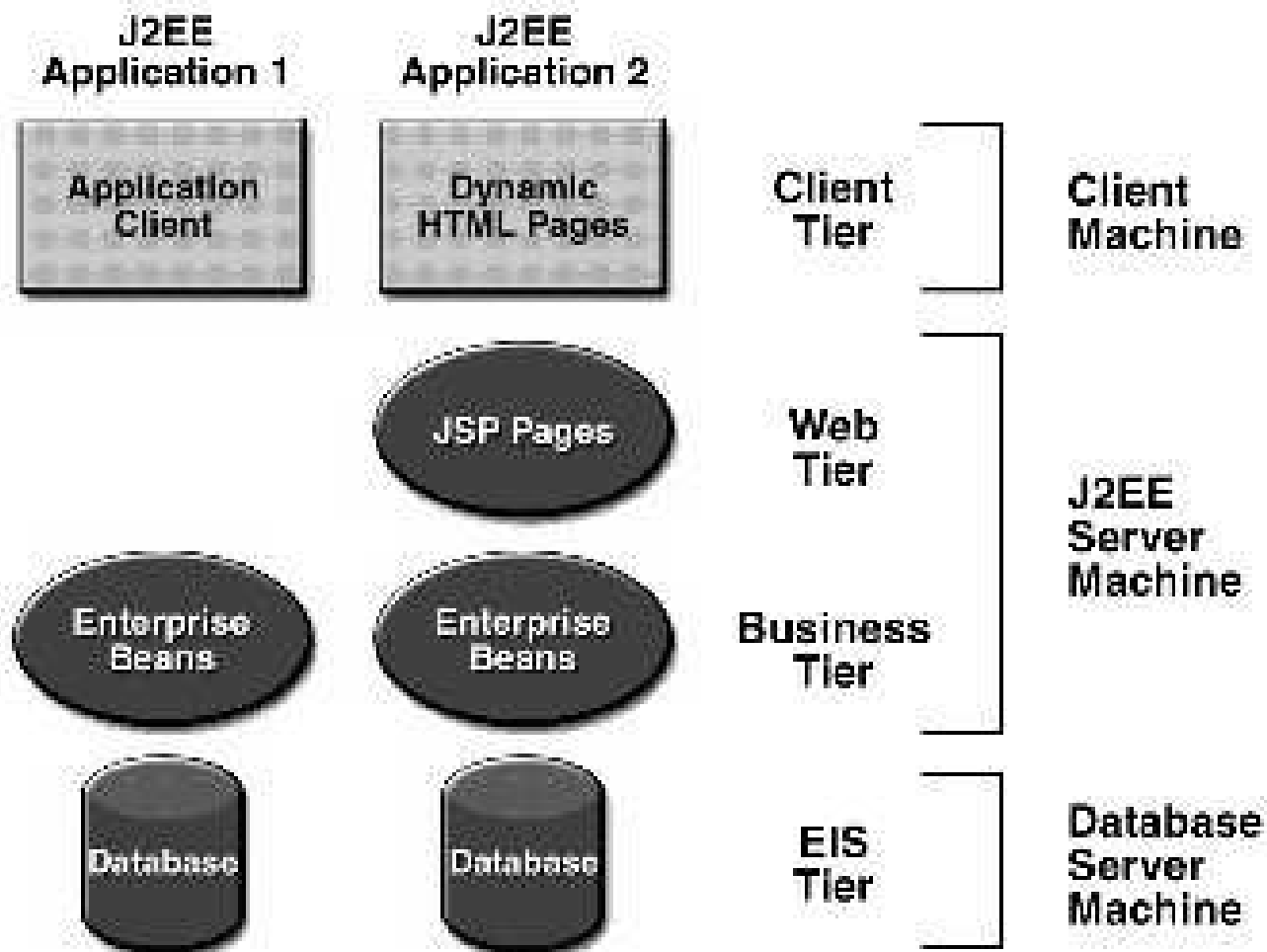


- Introducere
- Nivelul de logică a aplicației
- EJB în arhitectura Java EE
- Tipuri de bean-uri
  - Session Beans
  - Message-Driven Beans
- Inversion of Control și Dependency Injection
- Modele de organizare și încapsulare a logicii



# Introducere

# Aplicații pe mai multe niveluri



# Nivelul de logică

- Scalabilitate
- Tranzacții
- Controlul concurenței
- Persistență
- Autentificare și autorizare
- Comunicare standard (cu diferite tipuri de clienți)
  - sincron (apeluri la distanță)
  - asincron (mesaje)

# Scalabilitate

*Scalabilitatea* indică gradul în care capabilitățile unui sistem cresc pe măsură ce acesta este extins prin adăugarea de noi resurse (hard).

**Sistem scalabil** - sistem în care performanțele cresc proporțional cu resursele adăugate.

**Obținerea scalabilității:**

- **vertical** - adăugarea de resurse unui singur nod (procesor, memorie, etc)
- **orizontal** - adăugarea de noi noduri în sistem  
→ *Clustering, Load-balancing*

Legea lui Amdahl

# Tranzacții

*Tranzacție* - unificarea unui set de operațiuni într-o unitate de execuție atomică.

```
begin transaction
    extrage(cont1, suma);
    depune (cont2, suma);
    actualizareIstoric();
commit transaction
```

Dacă toate operațiunile reușesc  $\mapsto$  COMMIT

Dacă una din operațiuni nu reușește  $\mapsto$  ROLLBACK

# Modele tranzacționale

- **Simple (Flat)**  
O tranzacție este o unitate indivizibilă de lucru.
- **Imbricate (Nested)**  
O tranzacție poate conține la rândul ei alte tranzacții mai fine.
- **Distribuite**  
În tranzacții sunt implicate mai multe servere și resurse aflate în rețea (Protocolul *two-phase commit*)

Un sistem enterprise trebuie să ofere un *serviciu de coordonare a tranzacțiilor*. În Java EE: JTS (Java Transaction Service), JTA (Java Transaction API)

# Controlul concurenței

**Concurență** - execuția controlată a tranzacțiilor, fără a afecta consistența / integritatea datelor.

- **Coliziune** - Două sau mai multe tranzacții încearcă modificarea / accesarea aceleiași entități.
  - Blind write / Lost update (Write-Write)
  - Dirty read (Write-Read)
  - Non Repeatable read / Phantom read (Read-Write)
- **Mecanisme de control**
  - Optimistic Locking
  - Pessimistic Locking

# Enterprise Java Beans

# Tehnologia EJB

*Tehnologia EJB* = Set de specificații inclus în arhitectura Java EE pentru dezvoltarea de *componente server-side* la nivelul de logică a aplicațiilor enterprise.

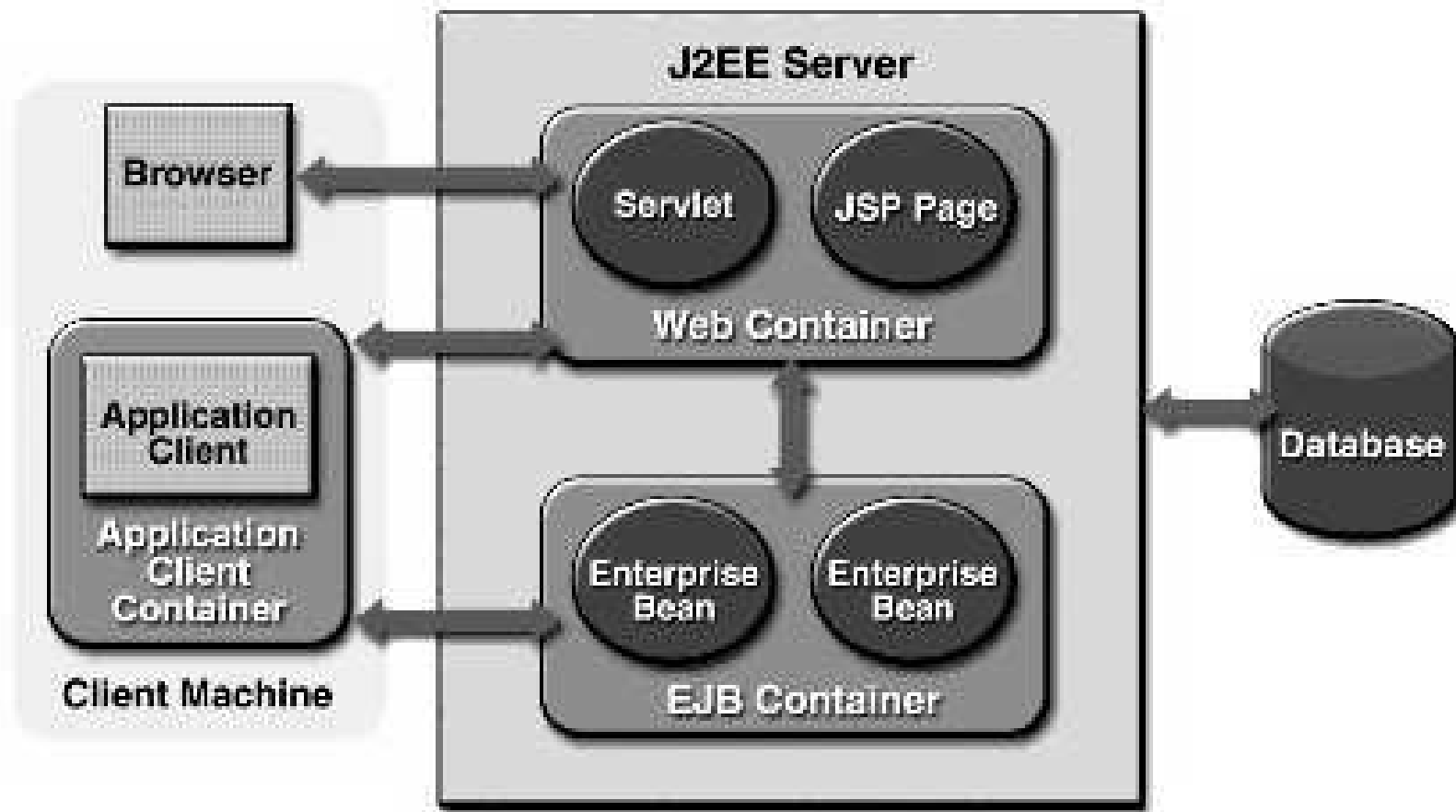
- **Standard agreat de industrie**
- **Portabilitate** la nivelul serverelor de aplicații
- **Dezvoltarea rapidă de aplicații**
- **Crearea unei piețe de componente**

# Ce sunt EJB-urile ?

*Componente EJB* = componente *server-side* ce oferă o anumită funcționalitate și pot fi instalate într-un server de aplicații.

- Simplificarea procesului de creare a aplicațiilor
  - focalizare asupra logicii propriu-zise
  - decuplare / modularizare
  - localizare și accesare standard
- Integrare cu mecanismele de persistență
- Integrare cu serviciile de mesagerie
- Integrare cu arhitectura serviciilor Web

# Containere EJB



# Identificarea *bean-urilor*

## *Location Transparency*

Locația precisă a unei componente Bean ce va executa o anumită cerere nu este cunoscută de apelant. Accesul la componentă presupune:

- **Identificarea** ei pe baza unui nume  
Localizarea se va face folosind un **serviciu de nume** și **JNDI**
- **Apelul** propriu-zis  
Va fi realizat **local** sau prin **RMI**

**Avantaje:** continuitate în exploatare, scalabilitate.

# Bean-uri *Remote* / *Local*

În funcție de locul de unde vor fi apelate, bean-urile pot fi:

- **Remote** - în situația când bean-ul respectiv trebuie să poată fi accesat de oriunde din rețea  
*client* ↔ *stub* ↔ (*marshalling*) ↔ *skeleton* ↔ *EJB*  
Transmiterea parametrilor implică serializarea acestora.
- **Local** - când bean-ul este de "uz intern", fiind apelat de alte componente aflate în aceeași JVM.  
*client* ↔ *EJB*  
Transmiterea parametrilor se face prin intermediul referinței.

# Reutilizarea *bean-urilor*

## *Instance Pooling*

*Pool* = A pune în comun; fond comun.

Containerul EJB este responsabil cu:

- instanțierea,
- distrugerea și
- **reutilizarea** componentelor EJB.

**Avantaj:** economisirea resurselor

# Tipuri de bean-uri

- **Session**

Modelează **acțiuni**: autentificare și autorizare, accesarea unui sistem extern sau a unei baze de date, apelarea altor componente, etc. Opțional, pot implementa **servicii web**.

- **Message-driven**

Beanuri de tip *acțiune* ce pot fi apelate prin **mesaje**.

- **Entity (deprecated)**

Modelează **date**, fiind obiecte ce conțin informații dintr-un mediu persistent.



# Bean-uri de tip *Session*



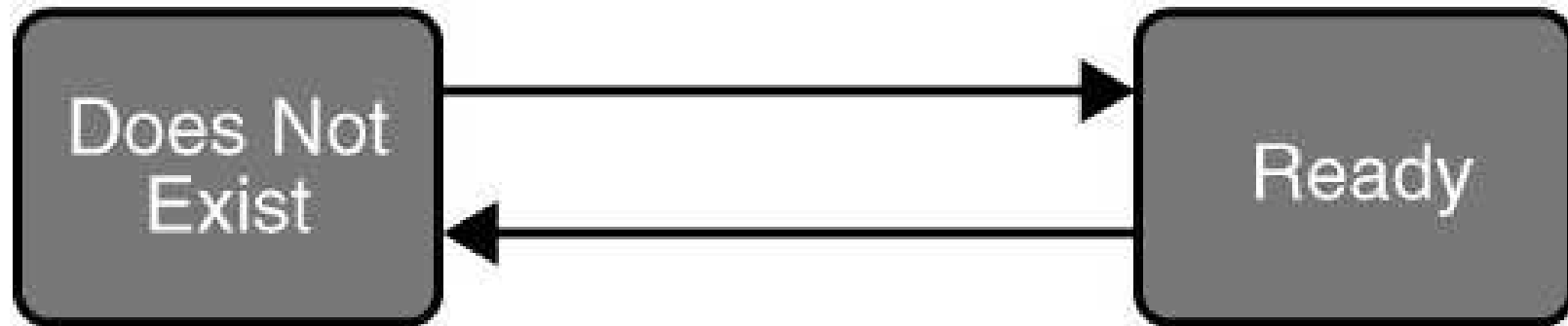
# Ce este un *Session Bean* ?

*Session Beans* = Obiecte ce implementează o anumită **acțiune**: conexiune la o bază de date, tranzacție bancară, compresie audio/video, etc.

- Au un singur client
- Nu sunt persistente (au o durată limitată de viață)
- Sunt de trei tipuri
  - **Stateful** - mențin o *stare conversațională* între apeluri ale aceluiași client
  - **Stateless** - nu mențin nici o stare între apeluri; pot implementa servicii Web
  - **Singleton** - partajate la nivelul aplicației

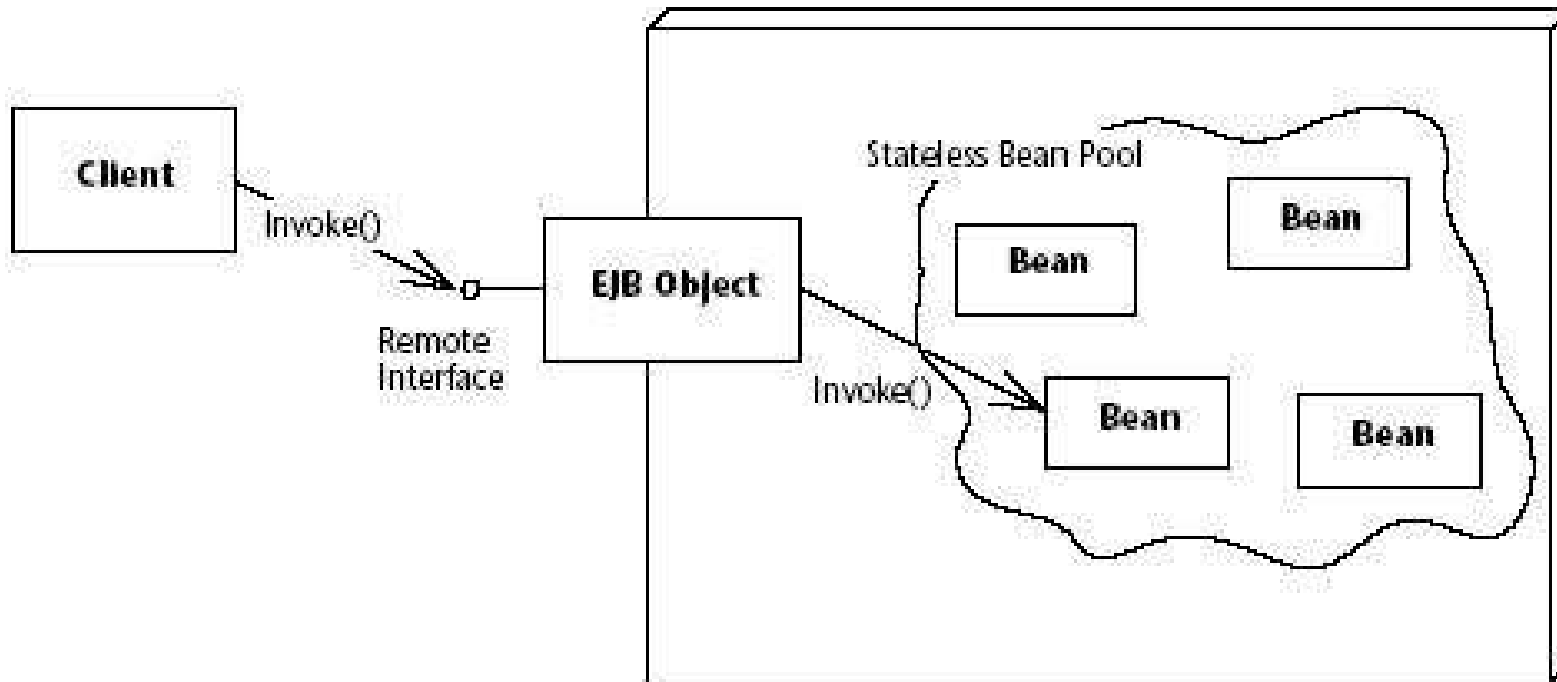
# Stateless - ciclul de viață

1. Dependency injection, if any
2. PostConstruct callbacks, if any



PreDestroy callbacks, if any

# Stateless Bean Pooling



# Crearea unui *Stateless Bean*

- Interfața care descrie bean-ul (opt 3.1)
- Clasa care implementează funcționalitatea
- Aplicația client

# Interfața *Hello*



## Remote

```
import javax.ejb.Remote;
/*
 * Interfața care descrie functionalitatea bean-ului.
 */
@Remote
public interface Hello {
    String sayHello(String name);
}
```

## Local

```
import javax.ejb.Local;

@Local
public interface HelloLocal {
    String sayHello(String name);
}
```



# Clasa *HelloBean*

O clasă poate implementa interfața Local, Remote sau ambele.

```
import javax.ejb.Stateless;

/*
 * Clasa care implementeaza functionalitatea bean-ului.
 */
@Stateless(name="hello")
public class HelloBean implements Hello {

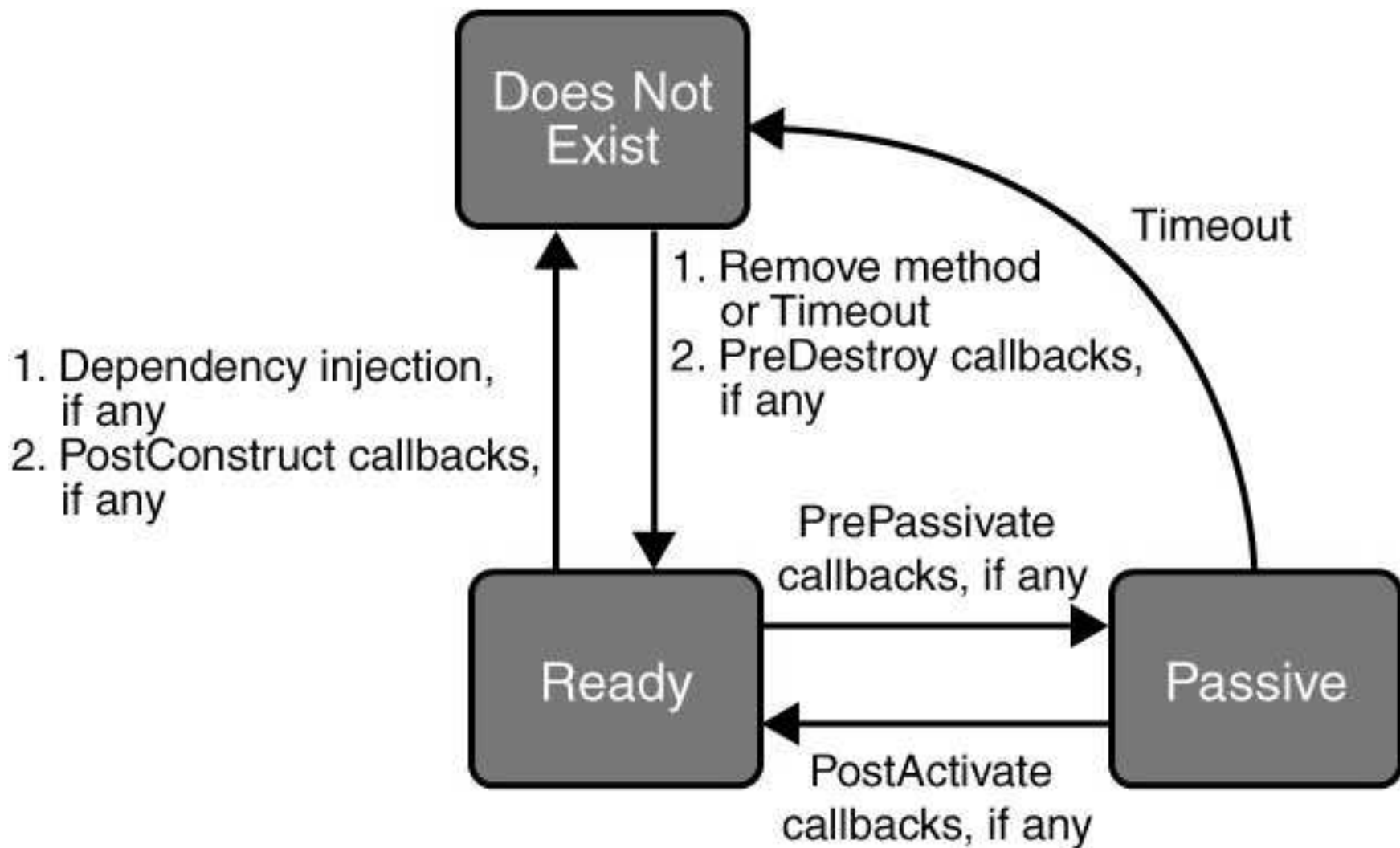
    public HelloBean() {
    }

    public String sayHello(String name) {
        return "Hello " + name + "!";
    }
}
```

# Aplicația client

```
public class Main {  
  
    public static void main(String[] args) {  
        try {  
            InitialContext ctx = new InitialContext();  
            Hello helloBean = (Hello) ctx.lookup("hello/remote");  
  
            String result = helloBean.sayHello("duke");  
            System.out.println(result);  
  
        } catch(NamingException e) {  
            System.err.println(e);  
        }  
    }  
}
```

# Stateful - ciclul de viață



# Interfața *ShoppingCart*

```
import java.util.List;

/**
 * Interfata care descrie functionalitatea bean-ului.
 */
public interface ShoppingCart {

    public void addItem(String name);
    public void removeItem(String name);
    public List<String> getContents();
    public void save();
}
```

# Clasa *ShoppingCartBean*

```
@Stateful(name="shoppingCart")
@Remote(ShoppingCart.class)
public class ShoppingCartBean implements ShoppingCart{
    List<String> contents;
    @PostConstruct
    public void init() {
        contents = new ArrayList<String>();
    }
    public void addItem(String title) {
        contents.add(title);
    }
    public List<String> getContents() {
        return contents;
    }
    @Remove
    public void save() {
        System.out.println("Saving ... \n" + contents);
    }
}
```

# Aplicația client

```
public class Main {  
    public static void main(String[] args) {  
        try {  
            // Obținem o referință la bean  
            InitialContext ctx = new InitialContext();  
            ShoppingCart cart = (ShoppingCart)  
                ctx.lookup(ShoppingCart.class.getName());  
  
            // Apelăm metodele expuse de interfață  
            cart.addItem("Christmas Tree");  
            cart.addItem("Jingle Bells");  
            /*  
             * Intre apeluri este menținută starea obiectului.  
             */  
            System.out.println(cart.getContents());  
            cart.save();  
        } catch (NamingException e) {  
            System.err.println(e);  
        }  
    }  
}
```

# Adnotări ale metodelor

- Generale (javax.annotation)
  - **@PostConstruct**
  - **@PreDestroy**
- Specifice EJB (javax.ejb)
  - **@Remove**
  - **@PrePassivate**
  - **@PostActivate**



# *Inversion of Control și Dependency Injection*

# Inversion of Control

**Inversion of Control (IoC)** = Șablon de proiectare care adresează în principal rezolvarea dependențelor între componente (obiecte) pentru a obține:

- clase mai ușor de reutilizat
- clase mai ușor de testat
- sisteme mai modulare și ușor de configurat

Tehnica generală este de inversare a direcției de accesare a unor resurse, servicii, etc.

**The Hollywood Principle:** "Don't call us we'll call you"

# Exemplu

```
public interface CashRegister {
    public BigDecimal calculateTotalPrice(ShoppingCart cart);
}

public interface PriceMatrix {
    public BigDecimal lookupPrice(Item item);
}

public class PriceMatrixImpl implements PriceMatrix {
    public BigDecimal lookupPrice(Item item) {
        // Acces la o baza de date, etc.
        return new BigDecimal("1");
    }
}

public class CashRegisterImpl implements CashRegister {
    ....
}
```

# CashRegisterImpl

```
public class CashRegisterImpl implements CashRegister {

    //-----
    private PriceMatrix priceMatrix = new PriceMatrixImpl();
    //-----

    public BigDecimal calculateTotalPrice(ShoppingCart cart) {
        BigDecimal total = new BigDecimal("0.0");
        for(Item item : cart.getItems()) {
            total.add(priceMatrix.lookupPrice(item));
        }
    }
    return total;
}
```

**Probleme ?**

**Soluții ?**

# Service Locator

```
public class CashRegisterImpl implements CashRegister {
    private PriceMatrix priceMatrix;

    public CashRegisterImpl() {
        //-----
        priceMatrix = ServiceLocator.getPriceMatrix();
        //-----
    }

    public BigDecimal calculateTotalPrice(ShoppingCart cart) {
        BigDecimal total = new BigDecimal("0.0");
        for(Item item : cart.getItems()) {
            total.add(priceMatrix.lookupPrice(item));
        }
    }
    return total;
}
```

# Dependency Injection

```
public class CashRegisterImpl implements CashRegister {
    private PriceMatrix priceMatrix;

    //-----
    // Setter-based injection
    //-----
    public PriceMatrix setPriceMatrix(PriceMatrix priceMatrix) {
        this.priceMatrix = priceMatrix;
    }

    public BigDecimal calculateTotalPrice(ShoppingCart cart) {
        BigDecimal total = new BigDecimal("0.0");
        for(Item item : cart.getItems()) {
            total.add(priceMatrix.lookupPrice(item));
        }
    }
    return total;
}
```

# Exemplu *Spring*

spring.xml

```
-----  
<beans>  
  <bean id="cashRegister" class="CashRegisterImpl">  
    <property name="priceMatrix">  
      <ref local="priceMatrix"/>  
    </property>  
  </bean>  
  <bean id="priceMatrix" class="PriceMatrixImpl">  
    <property name="filename">  
      <value>prices.txt</value>  
    </property>  
  </bean>  
</beans>
```

aplicatie

```
-----  
ApplicationContext ctx = new FileSystemXmlApplicationContext("spring.xml");  
CashRegister register = (CashRegister) ctx.getBean("cashRegister");
```

# Exemplu *JMock*

## Crearea unităților de testare

```
public void testCalculateTotalPrice() {  
  
    Mock mockPriceMatrix = mock(PriceMatrix.class);  
    PriceMatrix priceMatrix = (PriceMatrix) mockPriceMatrix.proxy();  
  
    CashRegister cashRegister = new CashRegisterImpl();  
    cashRegister.setPriceMatrix(priceMatrix);  
    ShoppingCart cart = new ShoppingCart();  
    cart.addItem("Christmas Tree");  
    cart.addItem("Jingle Bells");  
  
    // testam metoda ce calculeaza pretul total  
    assertEquals(2, cashRegister.calculateTotalPrice(cart));  
}  
return total;
```

# Dependency Injection în EJB (1)

## Obținerea referințelor la bean-uri

```
public class Main {
    @EJB
    private static Hello helloBean;
    @EJB
    private static ShoppingCart cart;

    public static void main(String[] args) {
        System.out.println(helloBean.sayHello("Duke"));

        cart.addItem("Christmas Tree");
        cart.addItem("Jingle Bells");
        System.out.println(cart.getContents());
        cart.save();
    }
}
```

# Dependency Injection în EJB (2)



## Obținerea de referințe la resurse

```
@Resource  
TimerService tms;
```

```
@Resource  
SessionContext ctx;
```

```
@Resource (name="DefaultDS")  
DataSource myDb;
```

```
@Resource (name="ConnectionFactory")  
QueueConnectionFactory factory;
```

```
@Resource (name="queue/A")  
Queue queue;
```



# Bean-uri de tip *Singleton*

Instanțiate o singură dată la nivelul unei aplicații.

```
import javax.ejb.Singleton;

@Singleton
public class CounterBean {
    private int hitCount;

    //Note the use of synchronized keyword
    public synchronized int incrementAndGetHitCount() {
        return hitCount++;
    }
}
```

Accesate concurrent:

- Container Managed Concurrency (CMC)
- Bean Managed Concurrency (BMC)

# Utilizarea unui bean *Singleton*

```
public class SimpleServlet extends HttpServlet {

    @EJB CounterBean counterBean;

    protected void processRequest(
        HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        try {
            out.println("<html><head>");
            out.println("<title>Singleton</title></head>");
            out.println("<body>");
            out.println("<h1>Number of times this servlet is accessed:"
                + counterBean.incrementAndGetHitCount());
            out.println("</body></html>");
        } finally {
            out.close();
        }
    }
}
```

# Invocarea asincronă a metodelor



## Declararea

```
@Asynchronous
public Future<String> processPayment(Order order) {
    ...
    if (SessionContext.wasCancelled()) {
        // clean up
    } else {
        // process the payment
    }
    String status = ...;
    return new AsyncResult<String>(status);
}
```

## Invocarea

```
Future<V>.isDone, Future<V>.get, Future<V>.cancel
```





# Bean-uri *Message Driven*

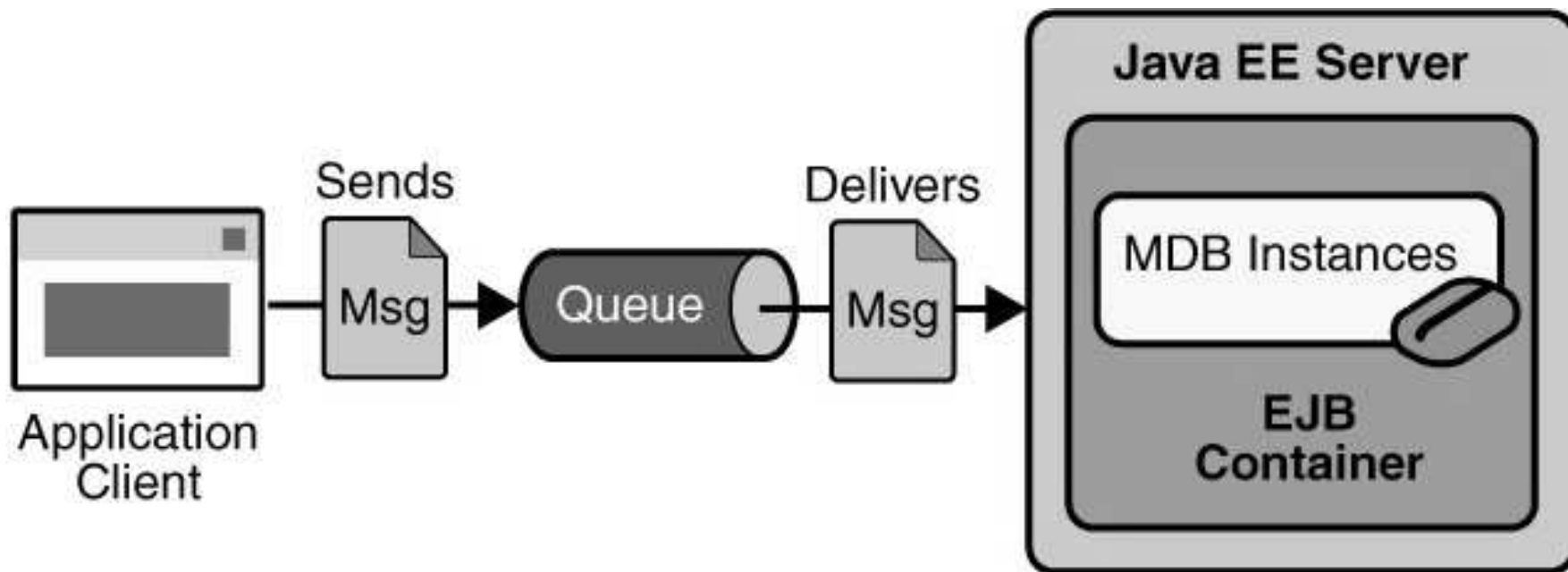


# Ce este un bean *Message driven* ?

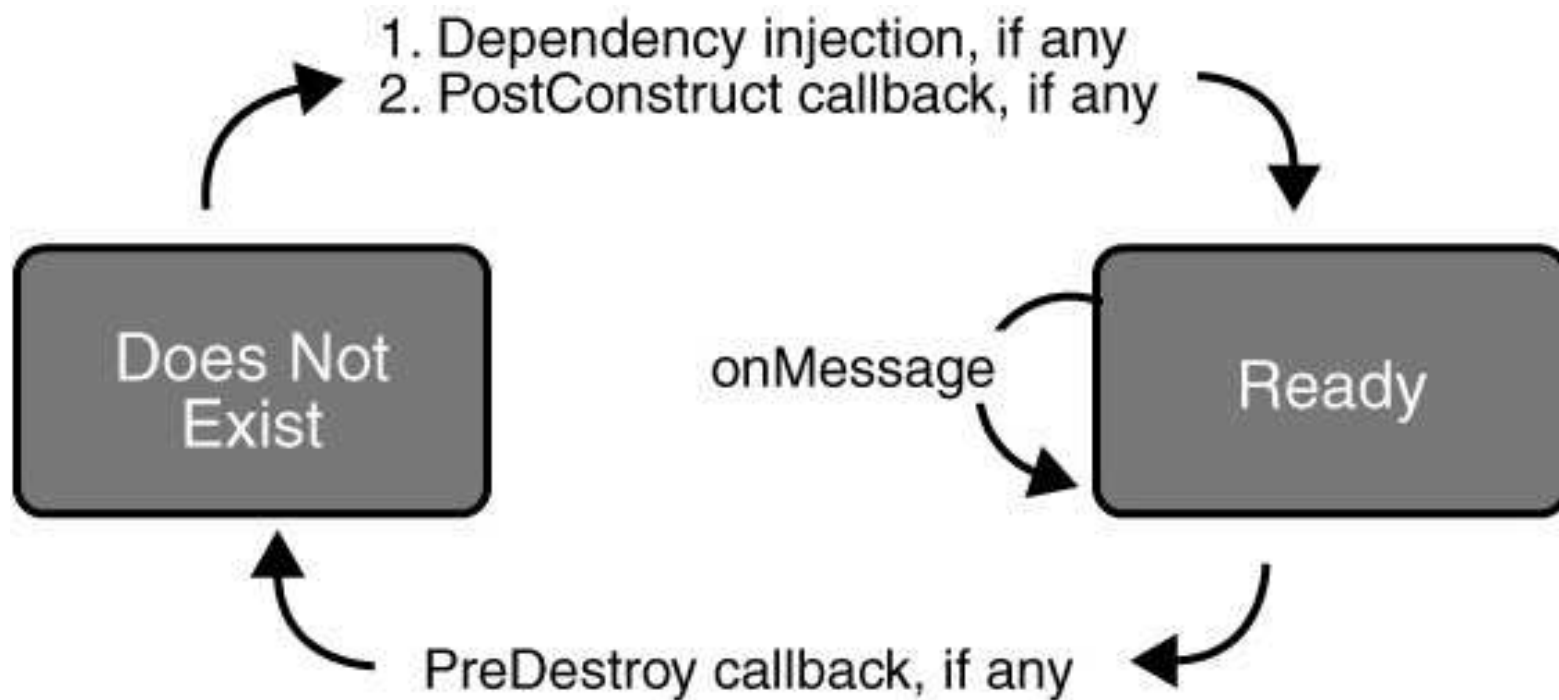
*Message Driven Beans* = Componente EJB capabile să primească și să "consume" mesaje, folosind tehnologia JMS.

- Bean-urile MD sunt decuplate de client
- Comunicarea se face prin intermediul unui furnizor de mesagerie
- Metodele bean-ului sunt *slab tipizate*
- Bean-urile MD pot să nu returneze valori sau excepții către clienți
- Bean-urile MD sunt *stateless*

# Imagine de ansambul MDB



# MDB - ciclul de viață



# Crearea unui MDB

```
@MessageDriven(mappedName="jms/Queue")
public class SimpleMessageBean implements MessageListener {

    public void onMessage(Message inMessage) {
        TextMessage msg = null;
        try {
            if (inMessage instanceof TextMessage) {
                msg = (TextMessage) inMessage;
            } else {
                ...
            }
        } catch (JMSEException e) {
            e.printStackTrace();
        }
    }
}
```

# Crearea unei aplicații client



```
@Resource(mappedName="jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;

@Resource(mappedName="jms/Queue")
private static Queue queue;

connection = connectionFactory.createConnection();
session = connection.createSession(false,
    Session.AUTO_ACKNOWLEDGE);
producer = session.createProducer(queue);

message = session.createTextMessage();
message.setText("Hello");
producer.send(message);
}
```

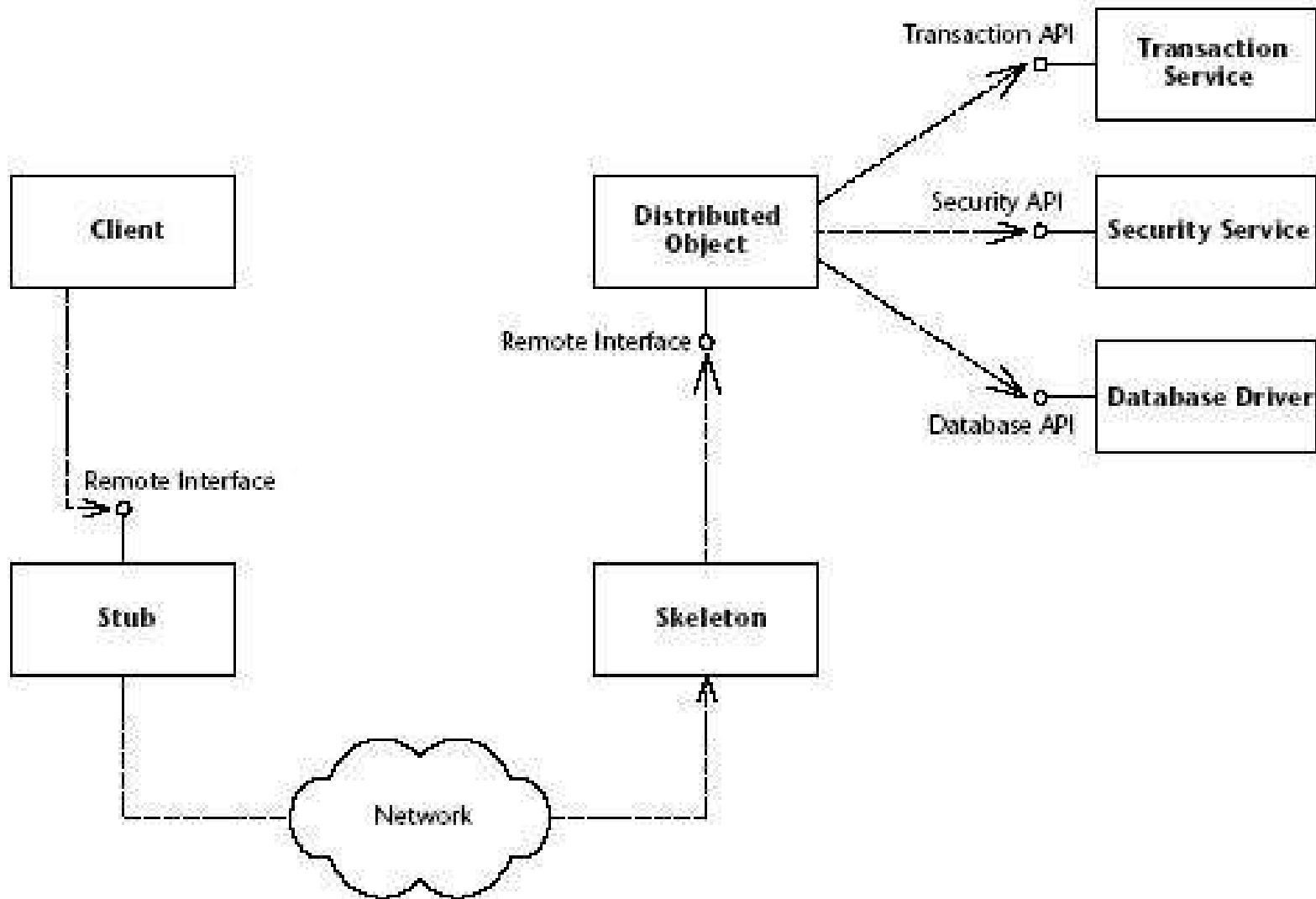




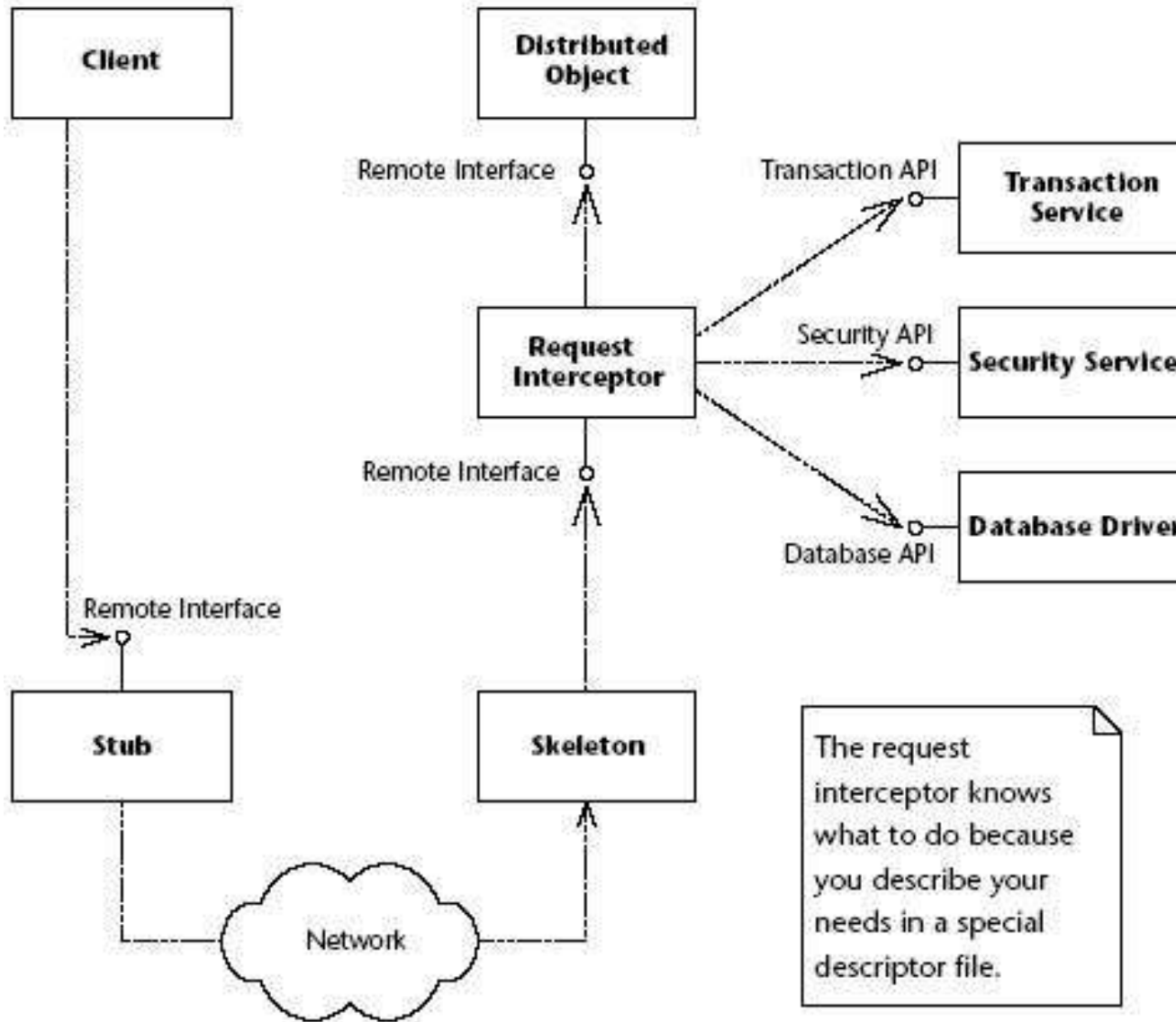
# **Servicii oferite de containerele EJB**



# Explicit Middleware



# Implicit Middleware



# Servicii pentru tranzacții

```
@Statefull
public class ShoppingCartBean implements ShoppingCart {

    // ... ...

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void save() throws Exception {
        // Secventa de instructiuni care actualizeaza baza de date
        // ... ...
        // Toate operatiile trebuie sa se termine cu succes
        // altfel baza de date nu va fi actualizata
    }
}
```

# Servicii pentru securitate

```
@Stateless
@SecurityDomain("other")
public class ShoppingBean implements Shopping {

    @RolesAllowed({"AdminUser"})
    public void addProduct (Product product, int quantity) {
        // ... ..
    }

    @RolesAllowed({"RegularUser"})
    public float getDiscount (Product product) {
        // ... ..
    }

    @PermitAll
    public List<Product> getProducts () {
        // ... ..
    }
}
```

# TimerService



```
@Stateless
public class TimerSessionBean implements TimerSession {

    @Resource
    TimerService timerService;

    public void createTimer(long milliseconds) {
        Date timeout = new Date(new Date().getTime() + milliseconds);
        timerService.createTimer(timeout, "Hello World!");
    }

    @Timeout
    public void timeoutHandler(Timer timer) {
        logger.info("Timer event: " + timer.getInfo());
    }
}
```



# Interceptori generici

```
@Stateful
public class ShoppingBean implements Shopping {

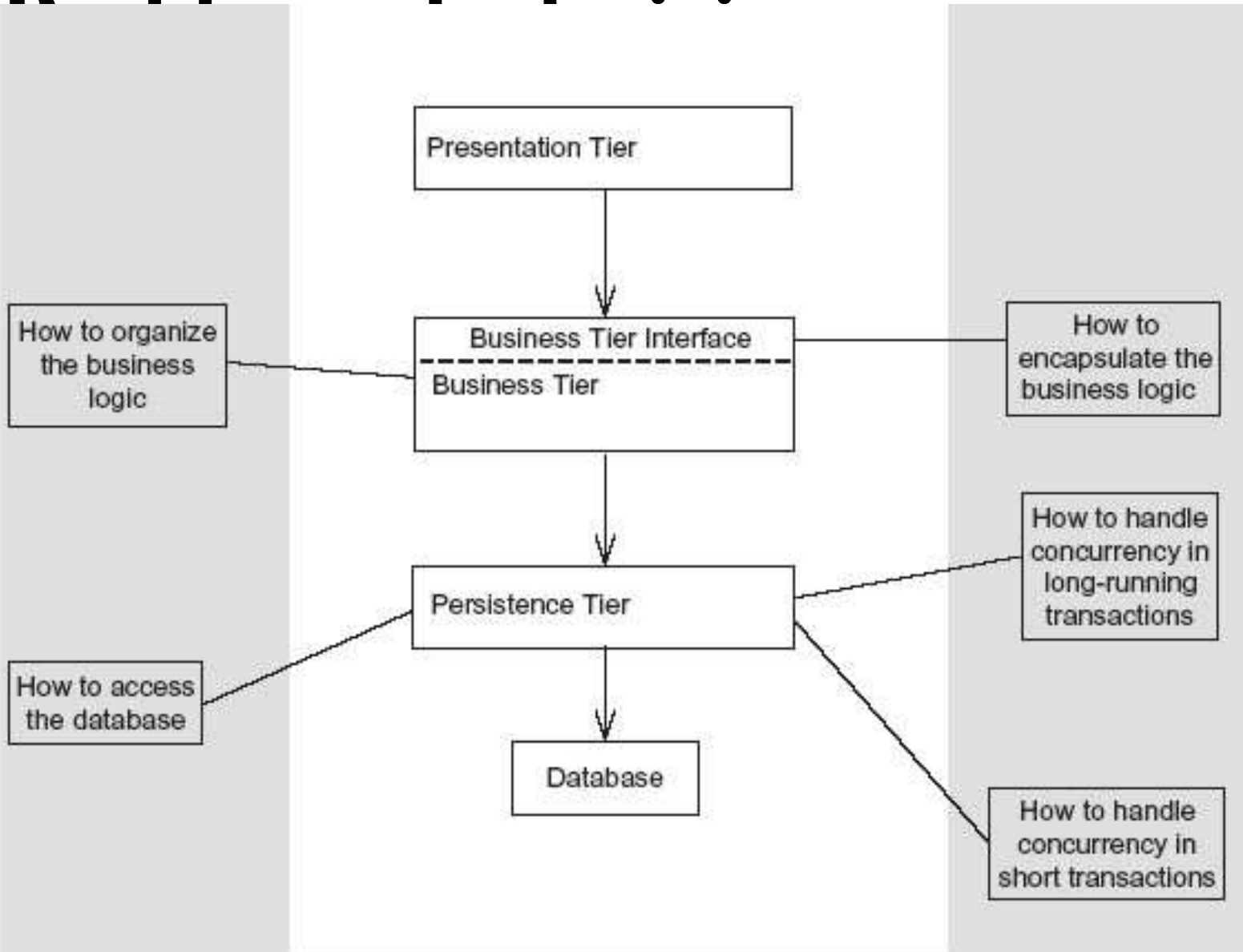
    // Toate metodele bean-ului vor fi interceptate de metoda "log()"
    @AroundInvoke
    public Object log (InvocationContext ctx) throws Exception {
        String className = ctx.getBean().getClass().getName();
        String methodName = ctx.getMethod().getName();
        String target = className + "." + methodName + "()";
        long t1 = System.currentTimeMillis();
        try {
            return ctx.proceed();
        } catch (Exception e) {
            throw e;
        } finally {
            long t2 = System.currentTimeMillis();
            System.out.println(target + " took " + (t2-t1) + "ms to execute");
        }
    }
}
```



# Șabloane de proiectare a nivelului de logică

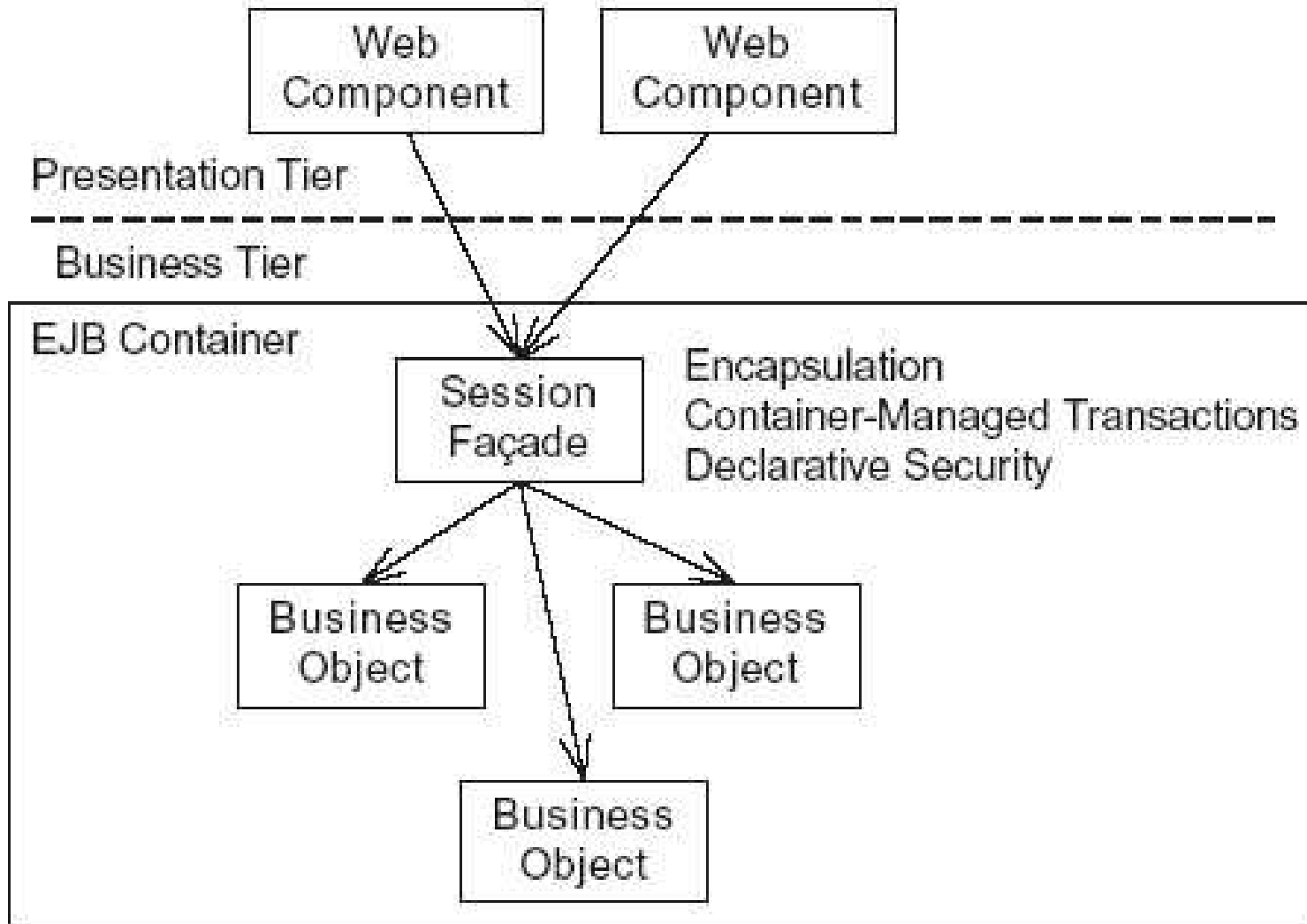


# Design Decisions

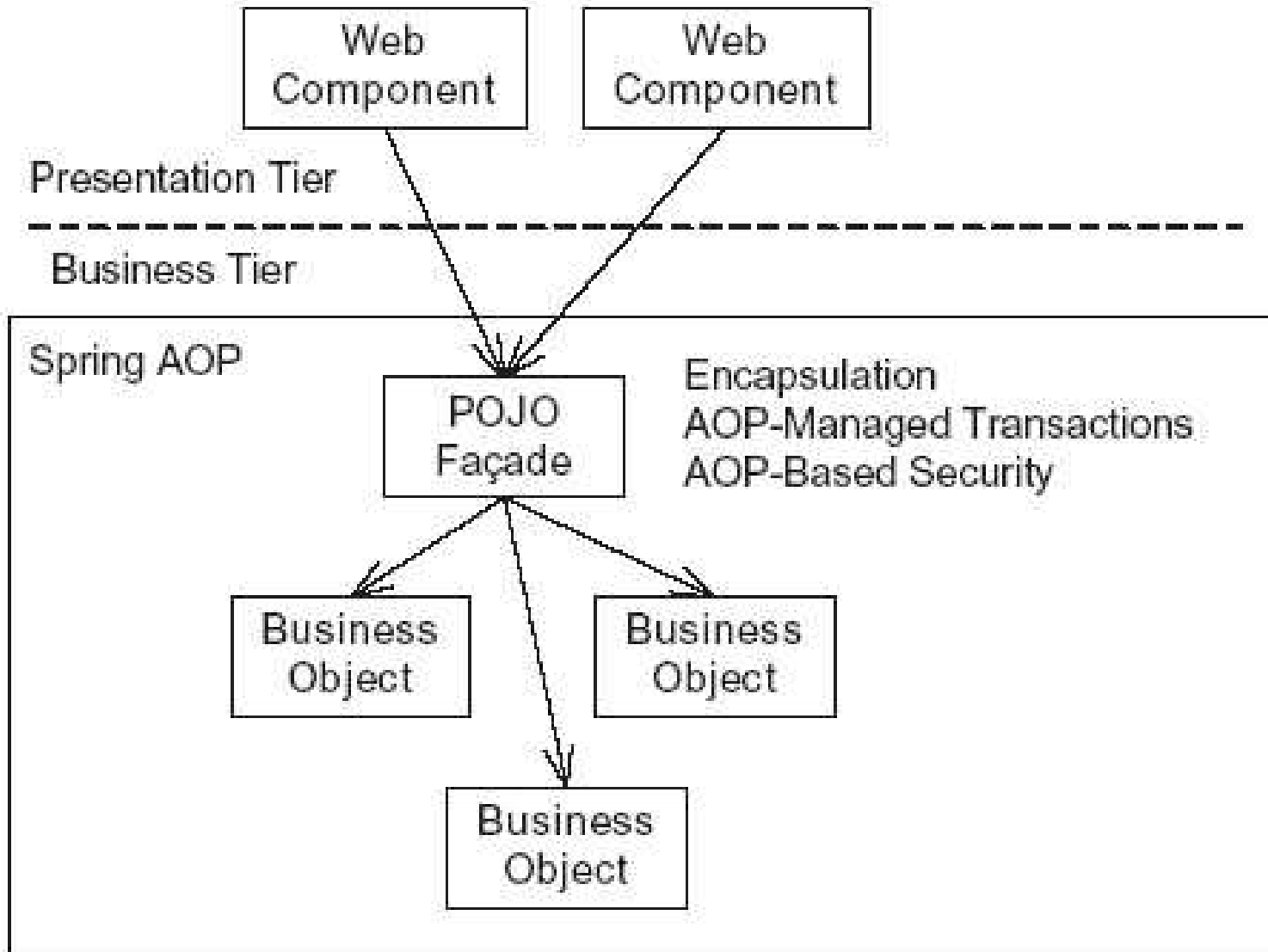


Design Decisions

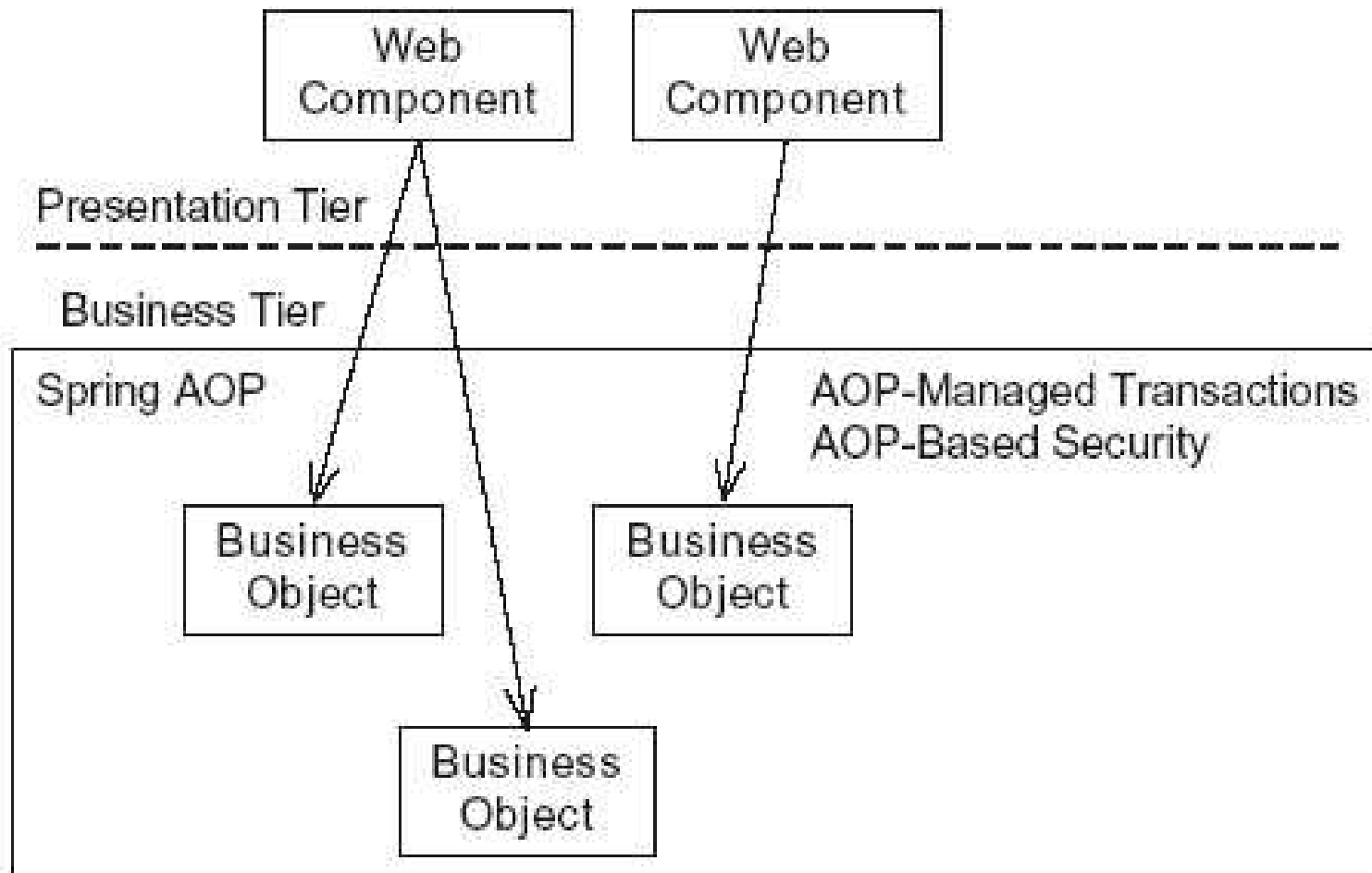
# EJB Session Facade



# POJO Session Facade



# Exposed Domain



# Când folosim EJB ?

- Aplicația poate fi distribuită pe mai multe mașini
- Sunt necesare tranzacții distribuite
- Este necesară securitate la nivel de componentă
- Este necesară asigurarea persistenței obiectelor
- Trebuie integrate sisteme existente deja
- Scalabilitatea este un factor cheie  
*...In this way, Java applications running on Solaris machines can scale up to more than 100 CPUs, 10,000 threads, and 100 GB of RAM per virtual machine (on the right hardware)*

# Beanuri session vs. Servleturi

Folosim servlet atunci când:

- logica poate/trebuie să existe în nivelul Web al aplicației
- sincronizarea firelor de execuție corespunzătoare servletului nu este o problemă
- scalabilitatea nu este o problemă
- aplicația nu este suficient de complexă astfel încât să necesite componente EJB