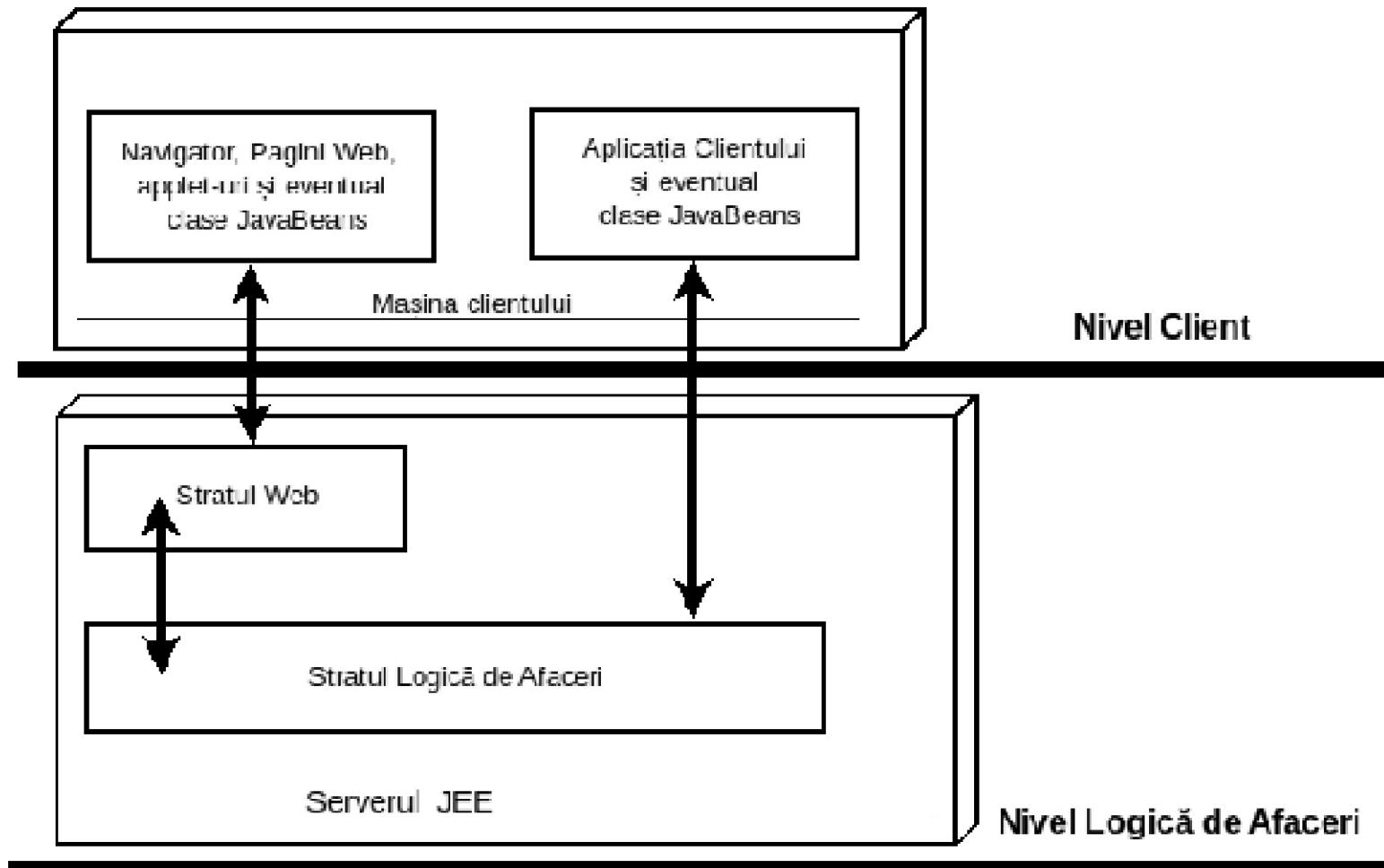




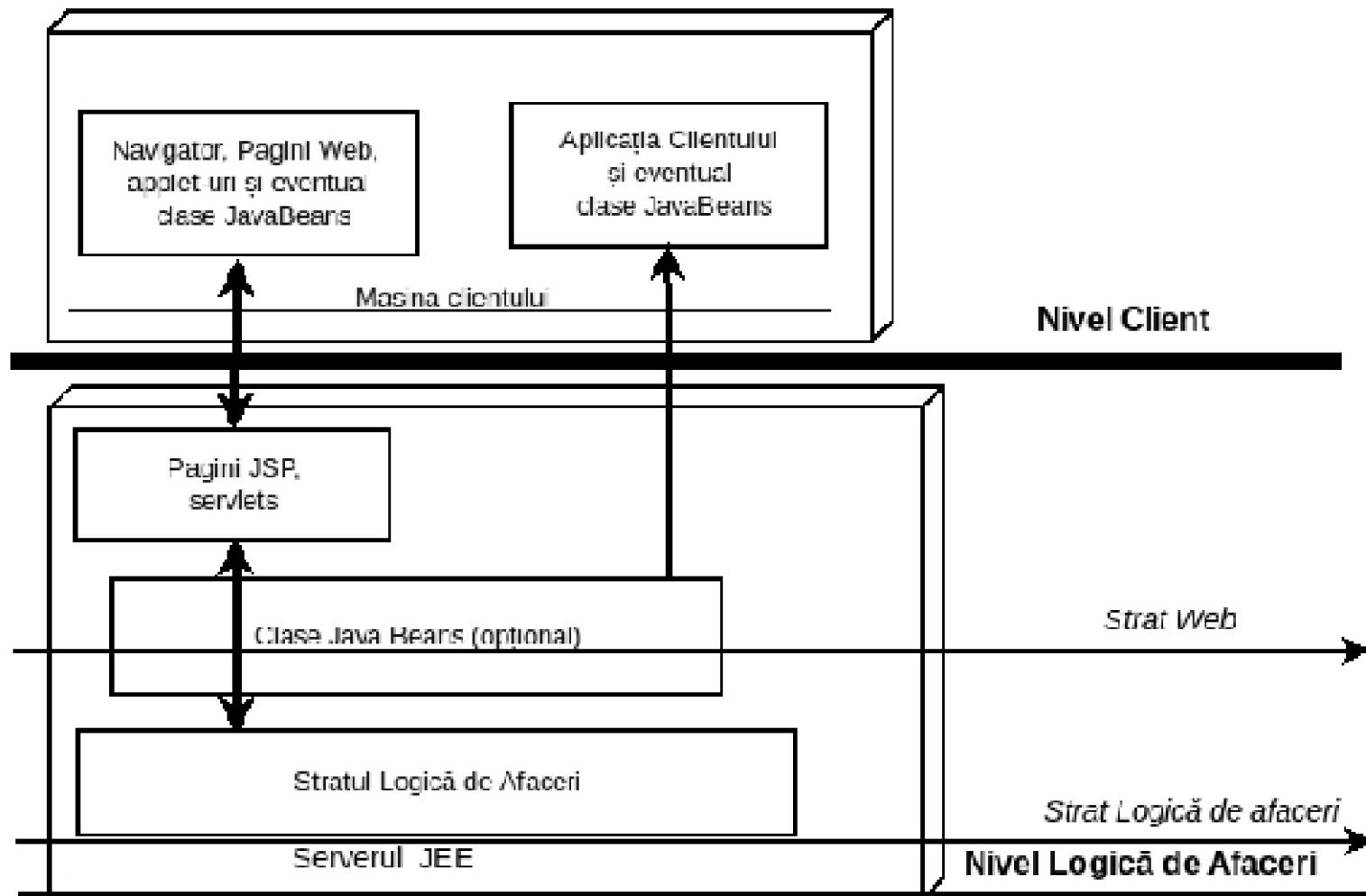
# Sisteme Distribuite

Mihai Zaharia  
Cursul 2

# Comunicarea cu serverul Java EE

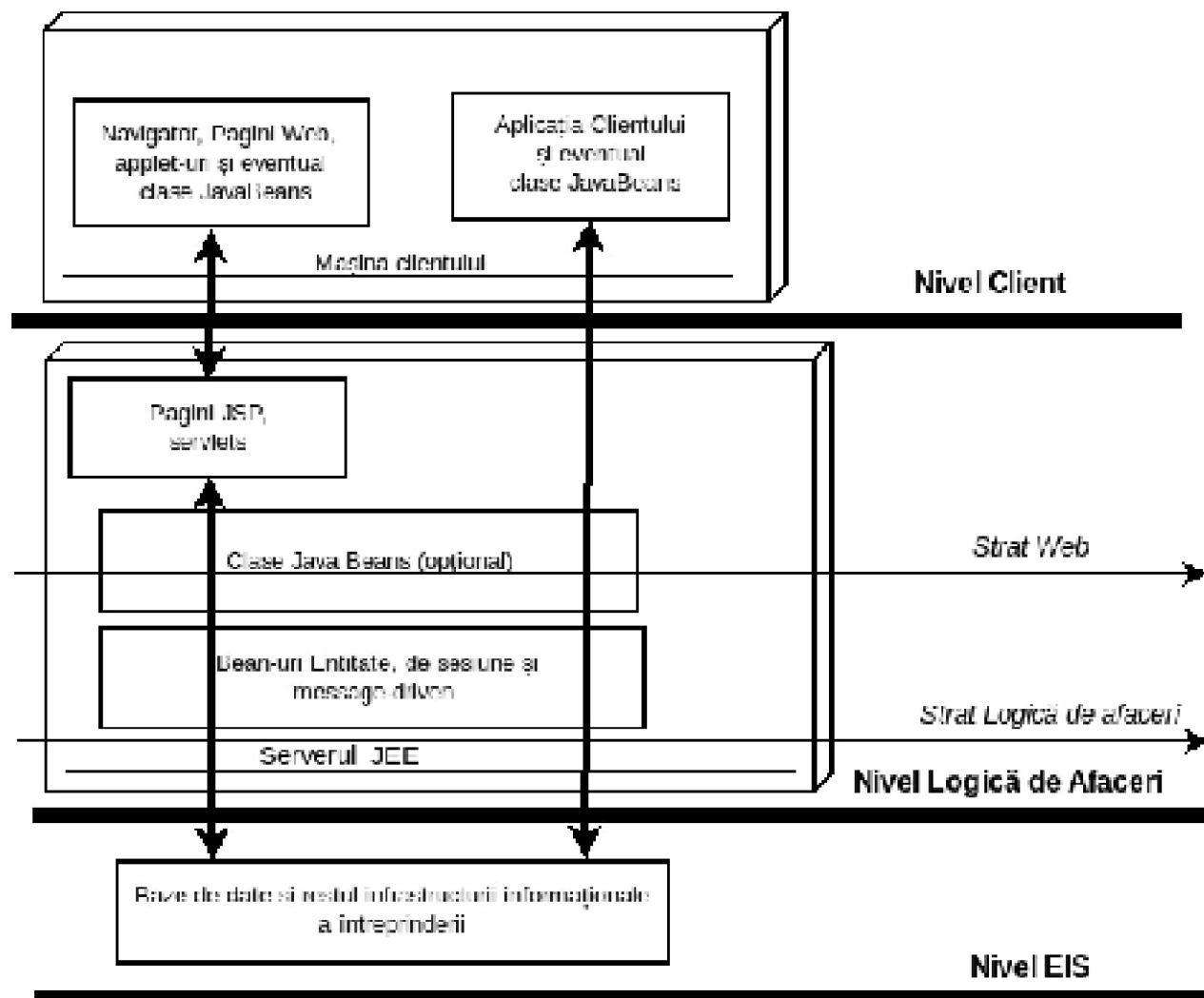


# Stratul de Web al aplicației JEE

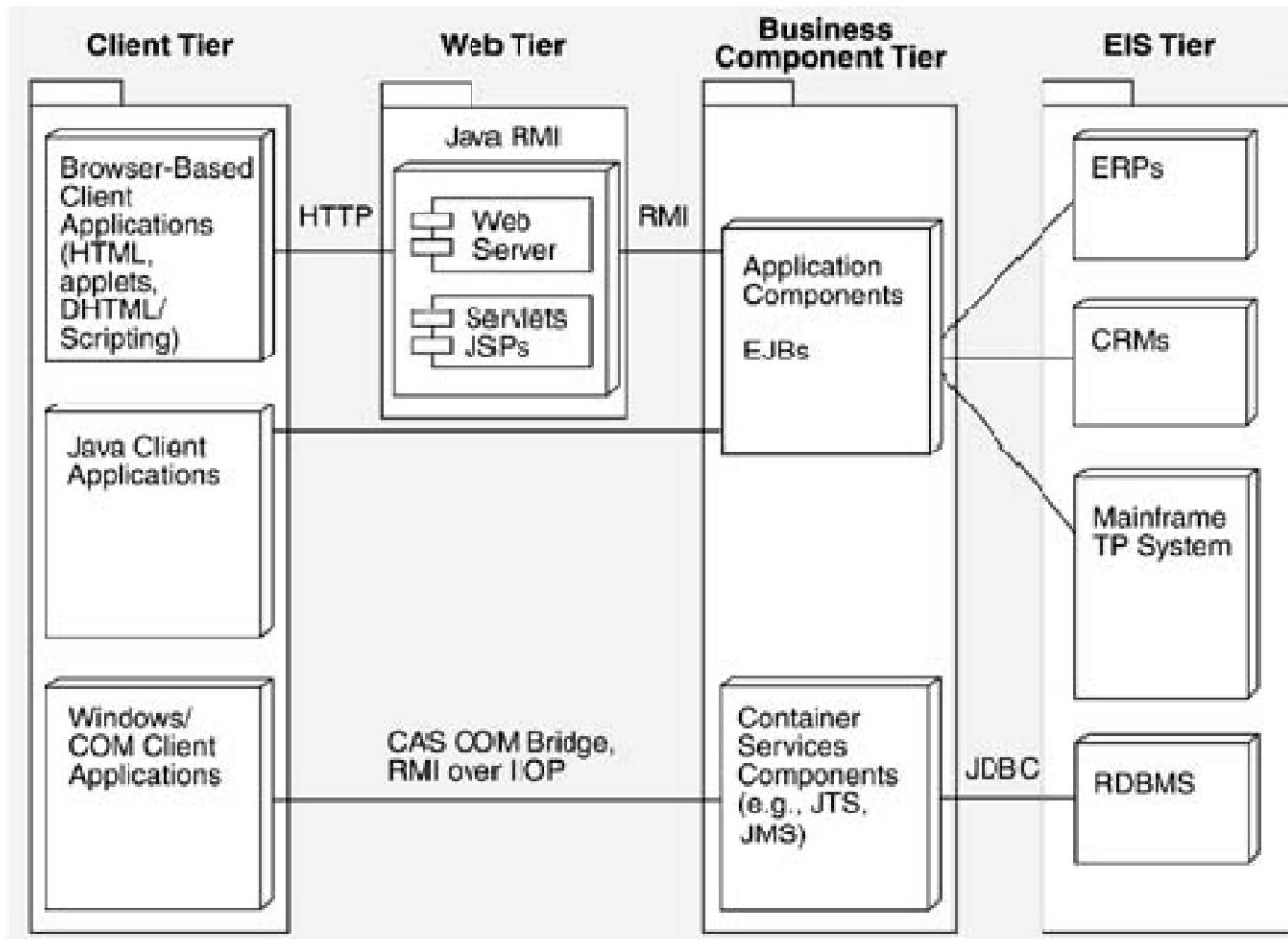


Clienți minimali “Thin” și Componentele Web

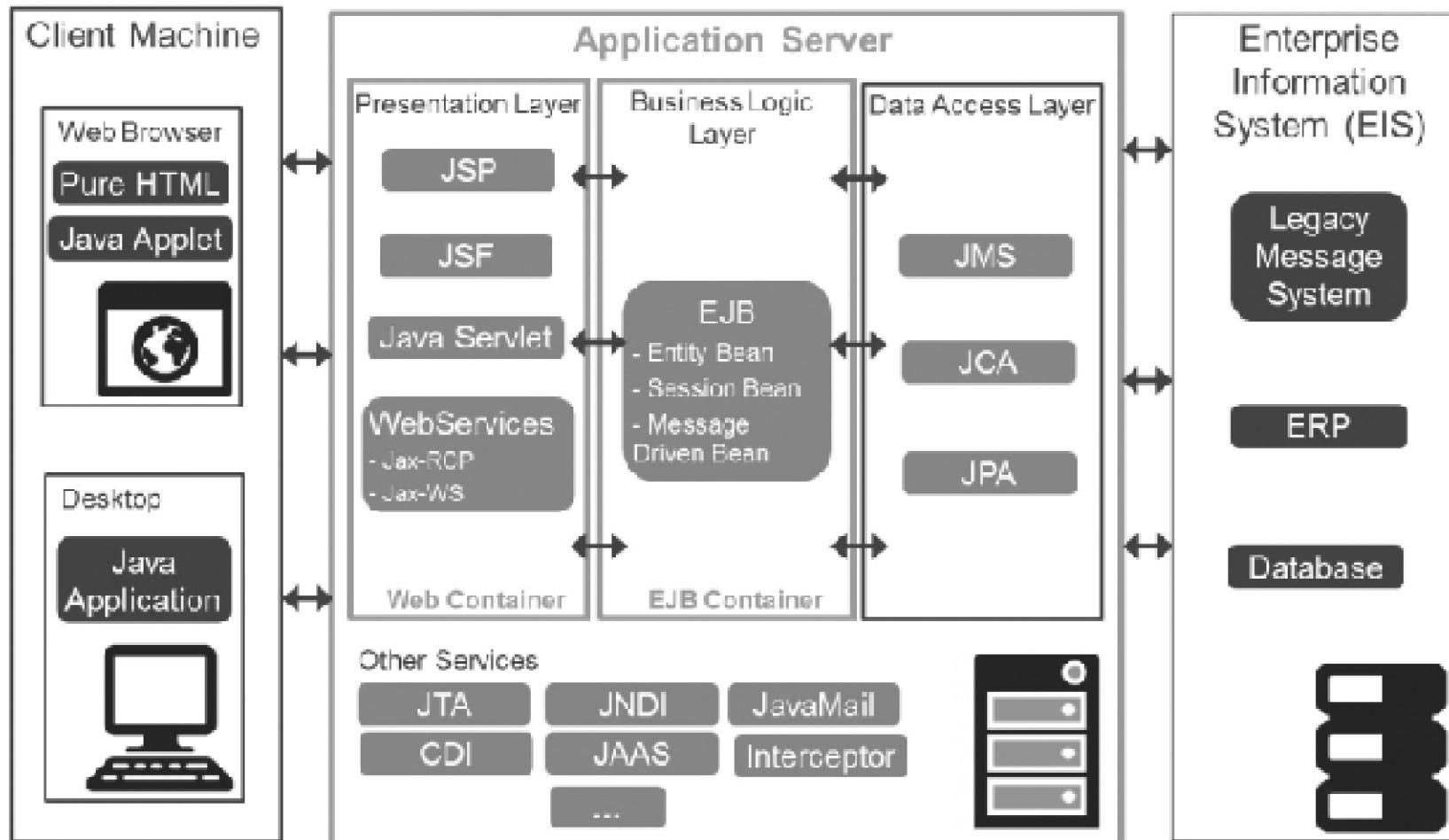
# Componentele de afaceri



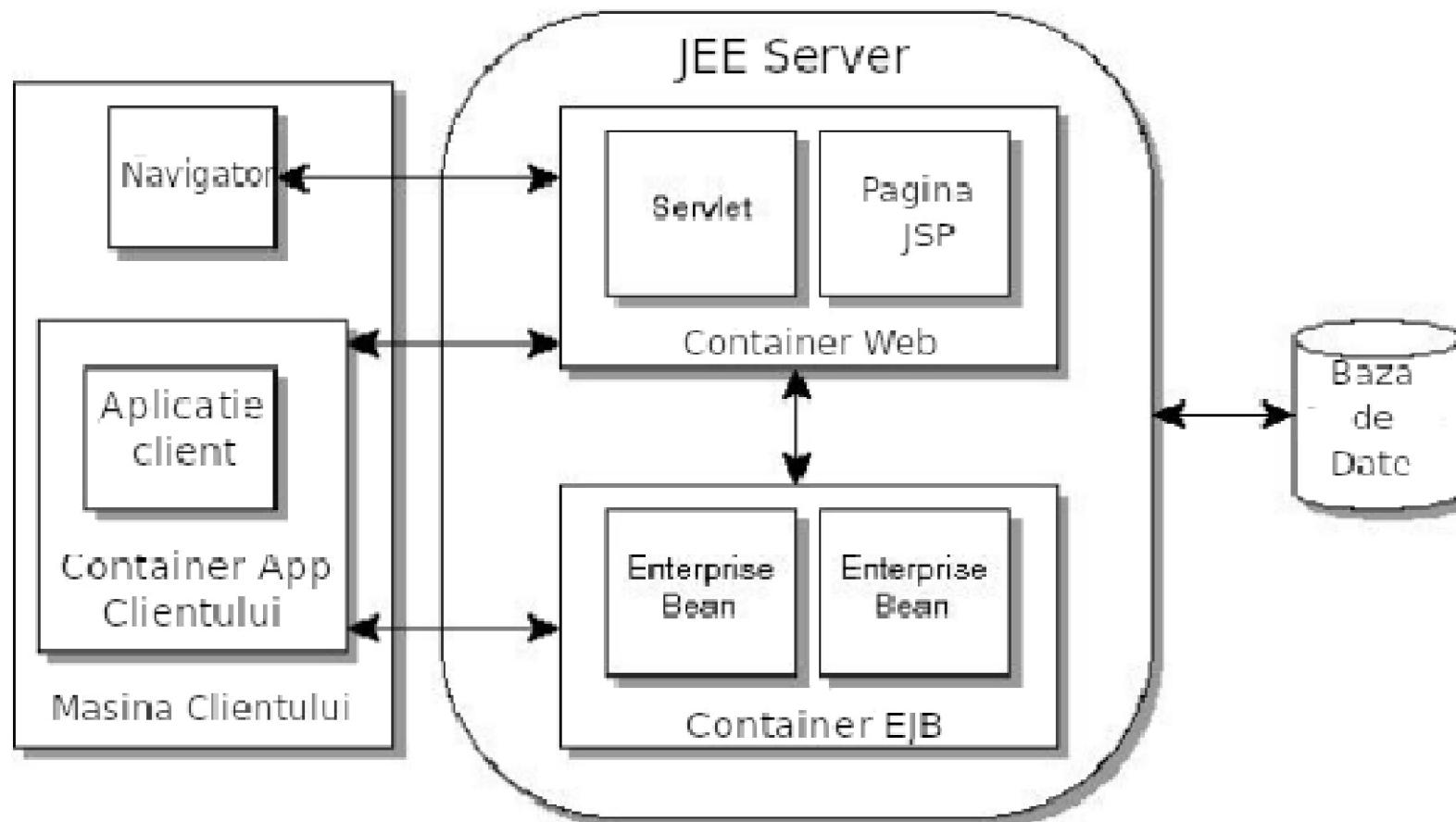
# Sistemul informational al întreprinderii (Enterprise Information System)



# Arhitectura Java EE



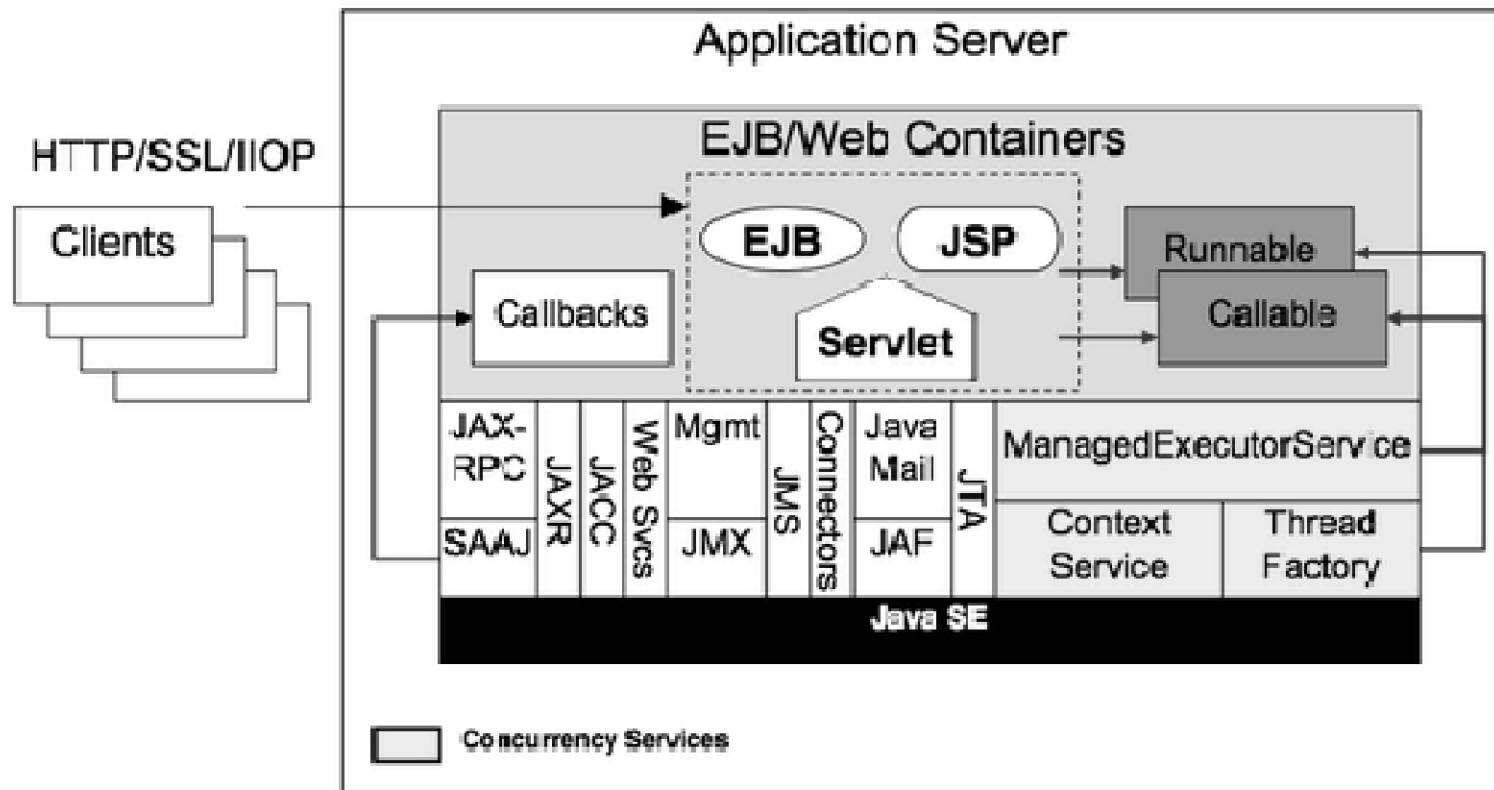
# Containere și Servicii



# **Configurări la nivel de container**

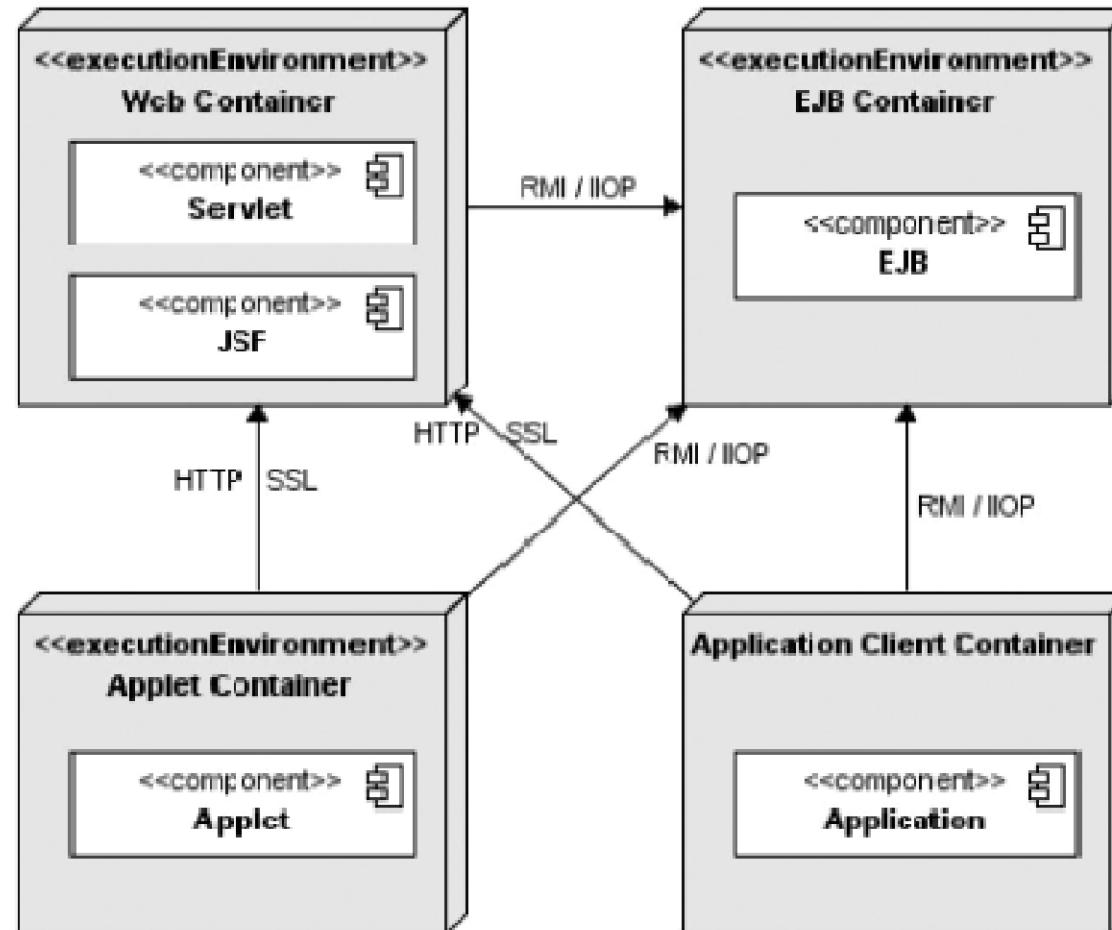
- **Modelul de securitate Java EE**
- **Modelul tranzactional al JEE**
- **Serviciile de acces JNDI**
- **Modelul de accesare la distanta al JEE**

# Containere



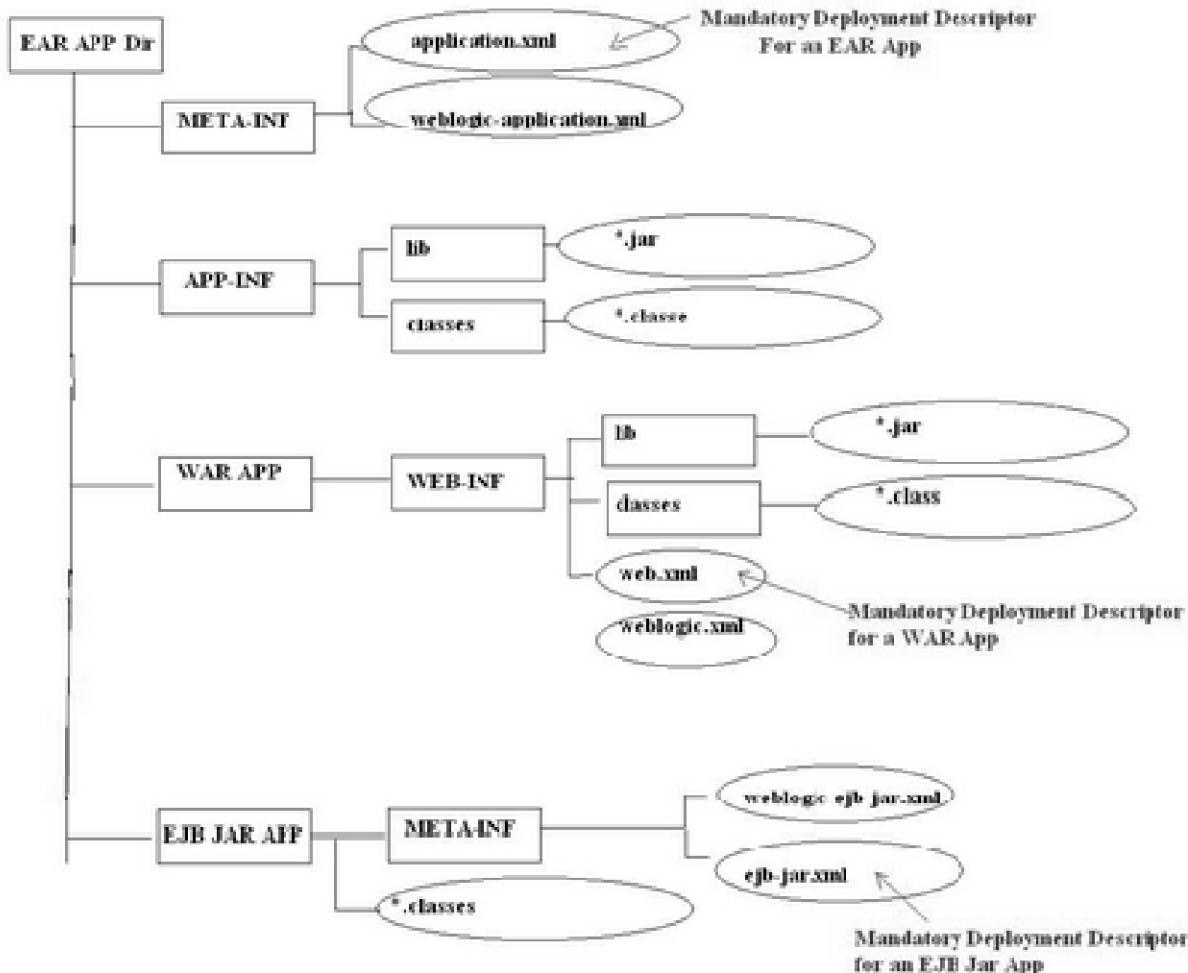
utilizând concurență

# Fu&#259;ncti&#259;onare container

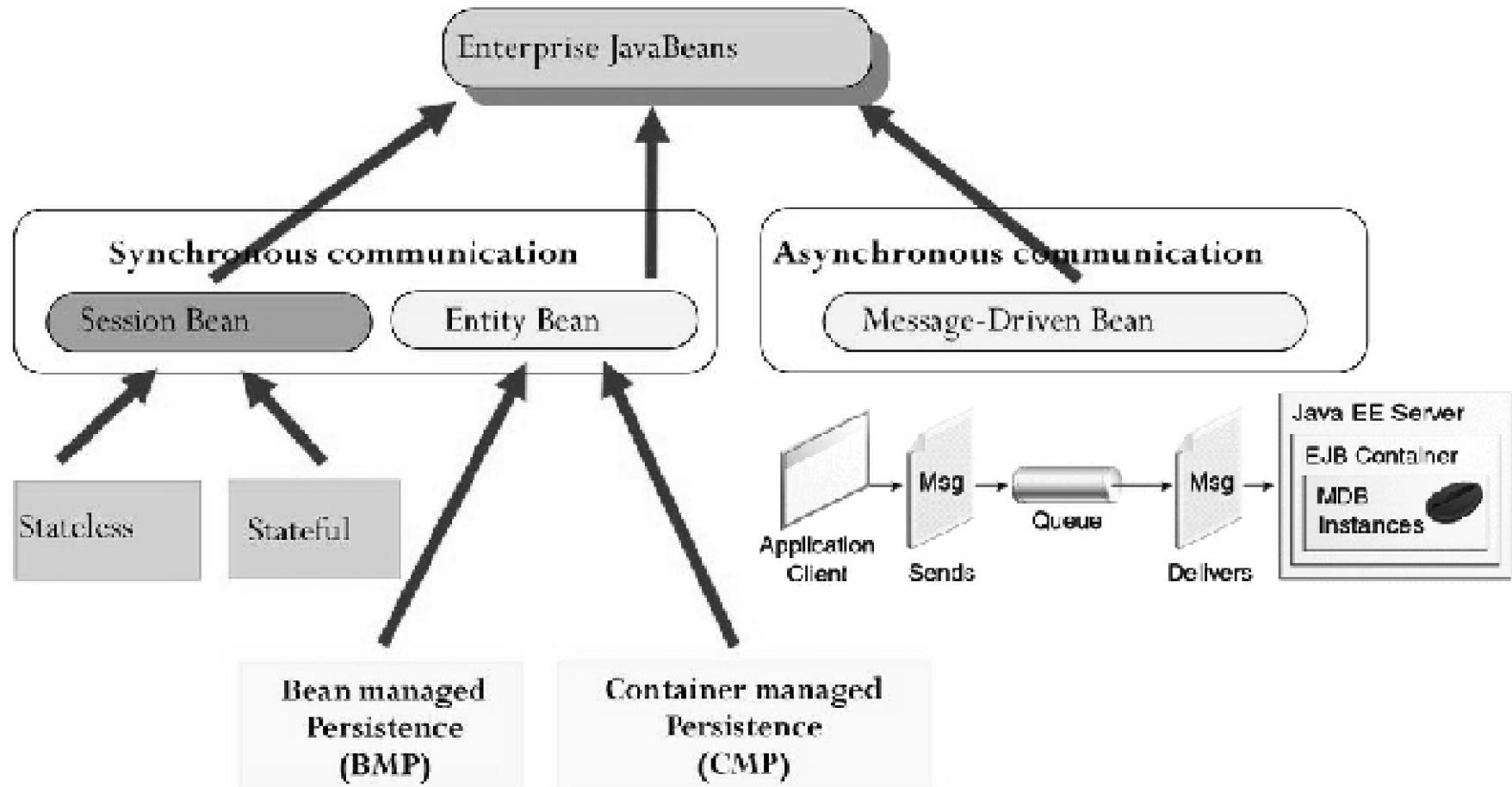


# Împachetarea componentelor

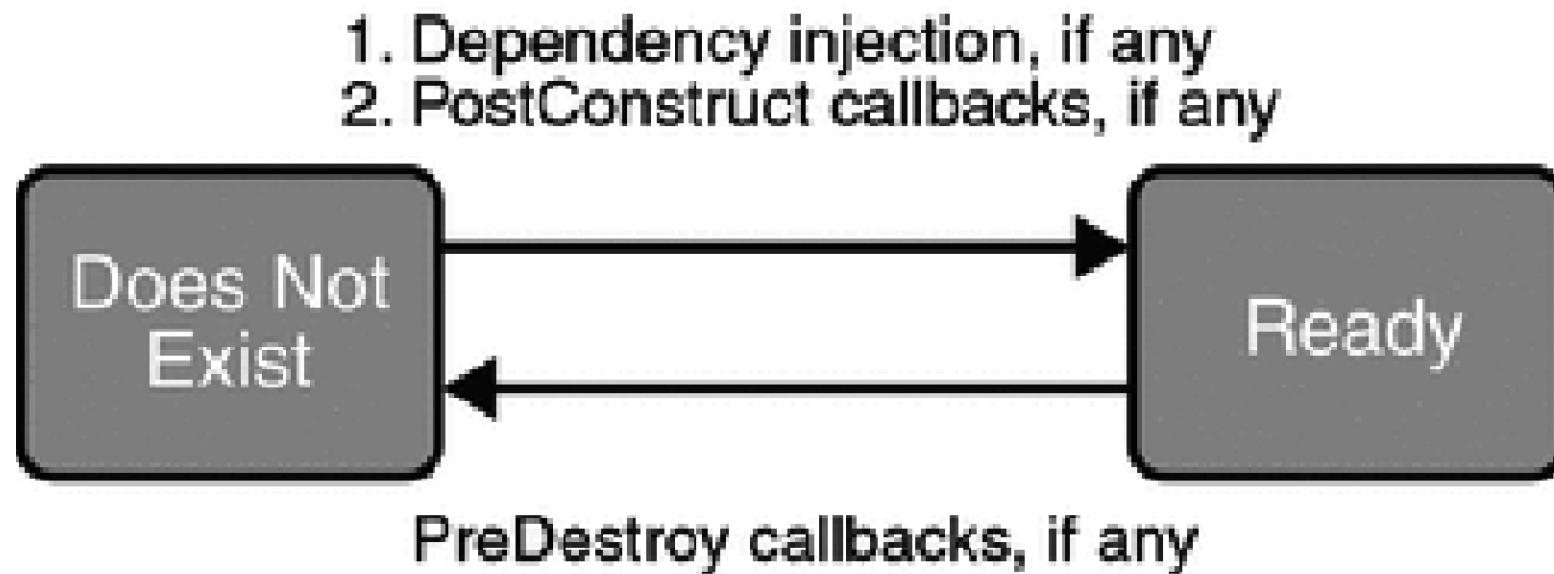
- Jar
- Ear
- War



# Tipuri de ejb

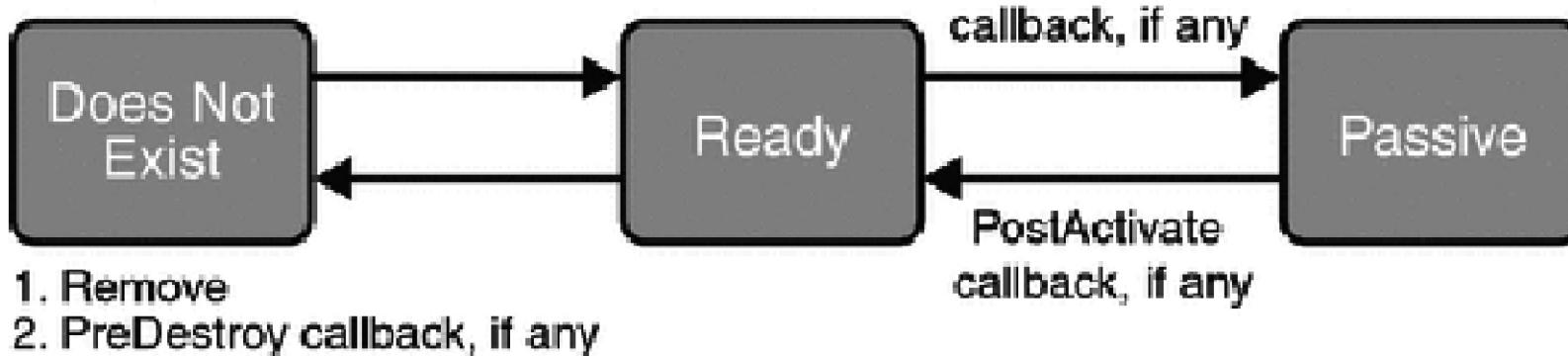


# Ciclul de viață al unui bean de sesiune fără stare



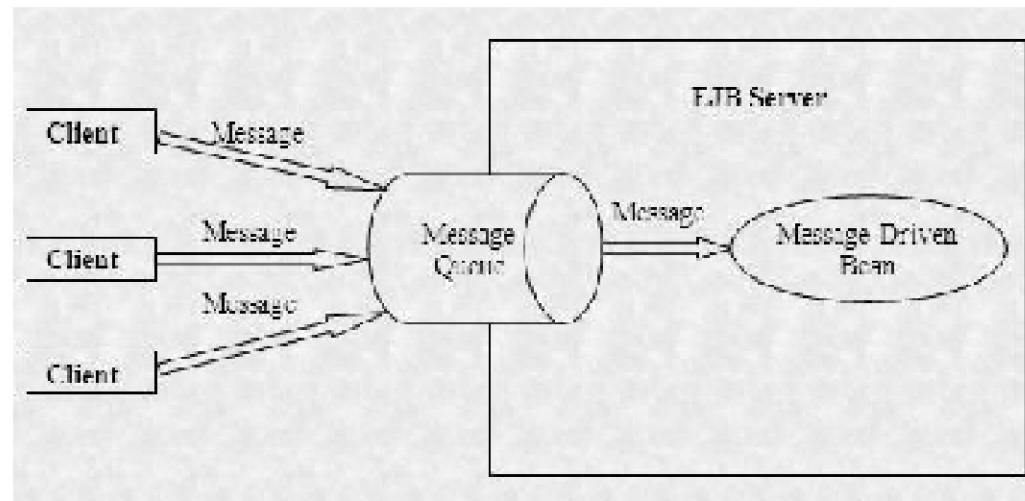
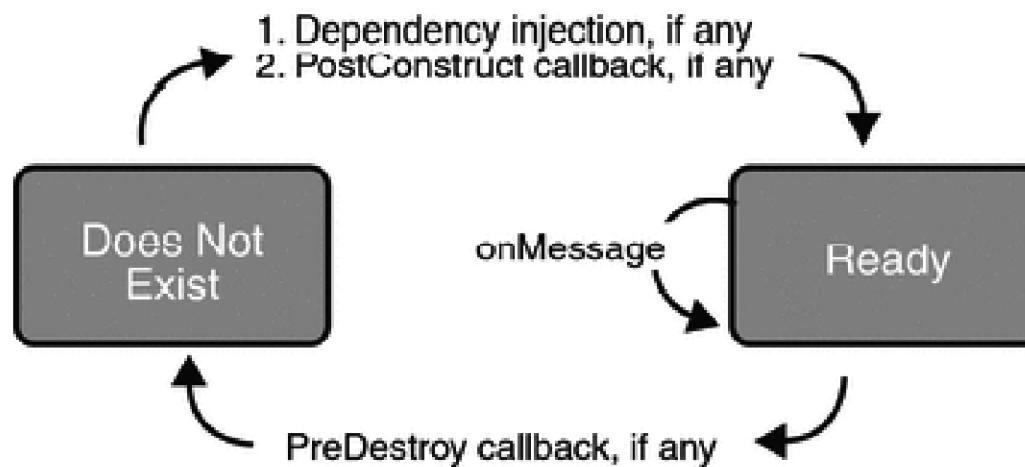
# Ciclul de viață al unui bean de sesiune cu stare

1. Create
2. Dependency injection, if any
3. PostConstruct callback, if any
4. Init method, or ejbCreate<METHOD>, if any

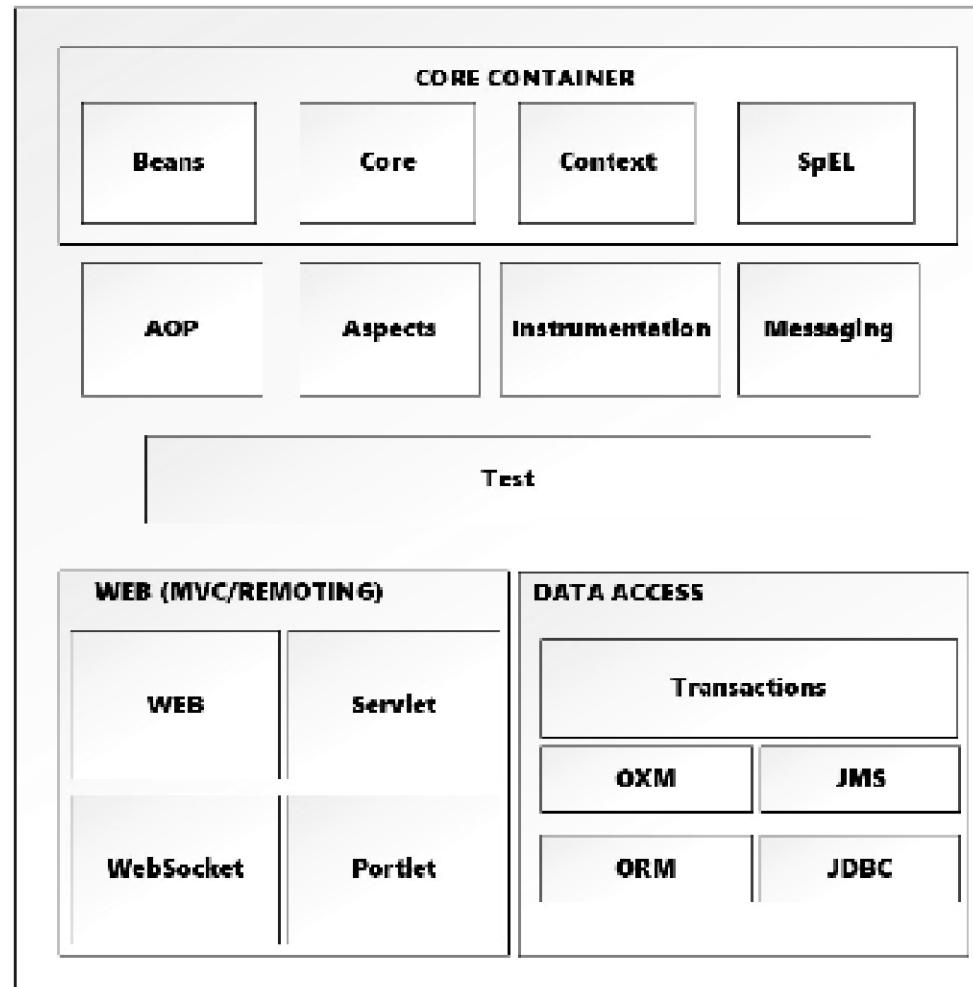


1. Remove
2. PreDestroy callback, if any

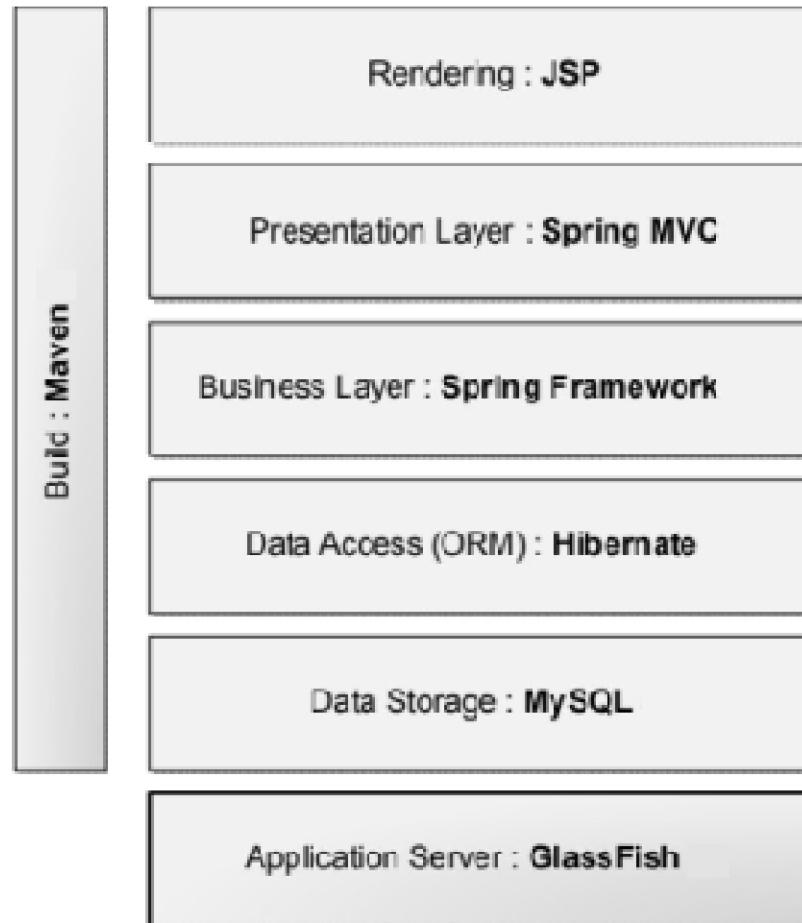
# Ciclul de viață al unui bean orientat pe mesaj



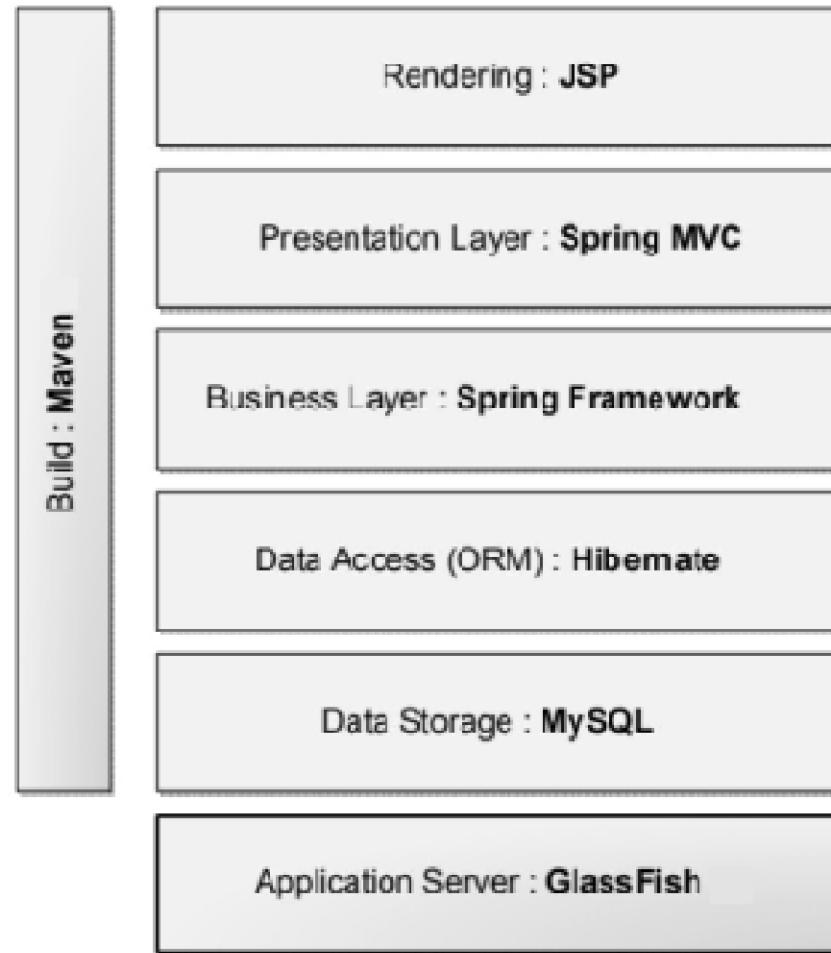
# Framework-ul Spring



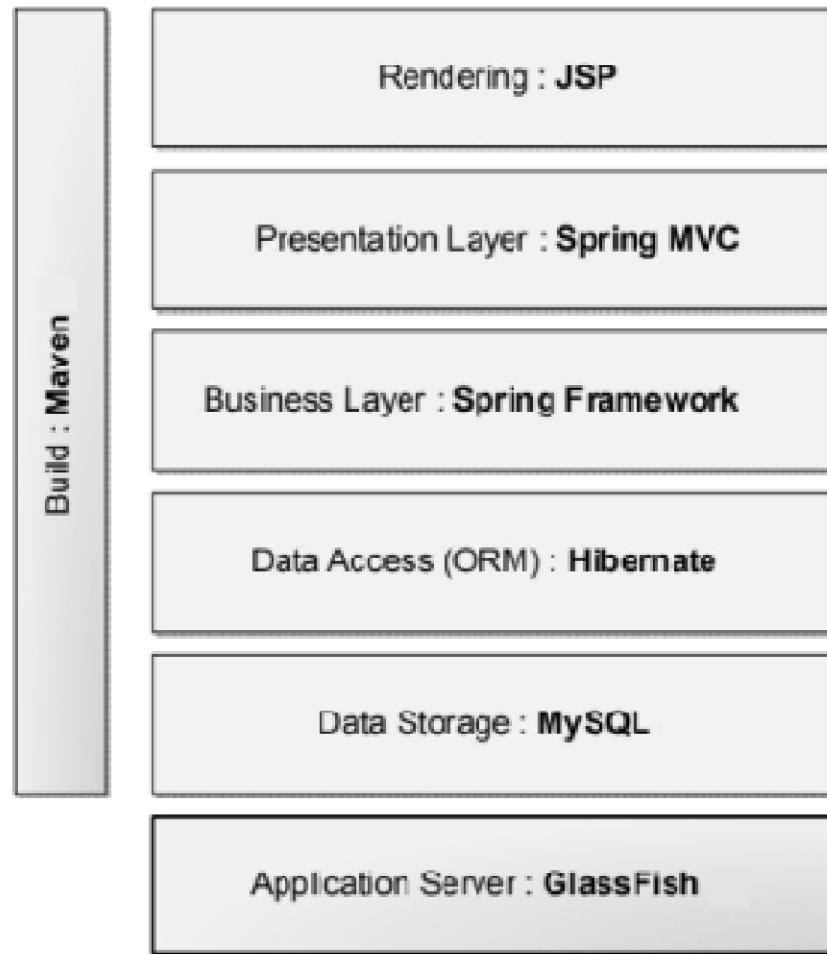
# Ce-i cu Stack-urile astea ?



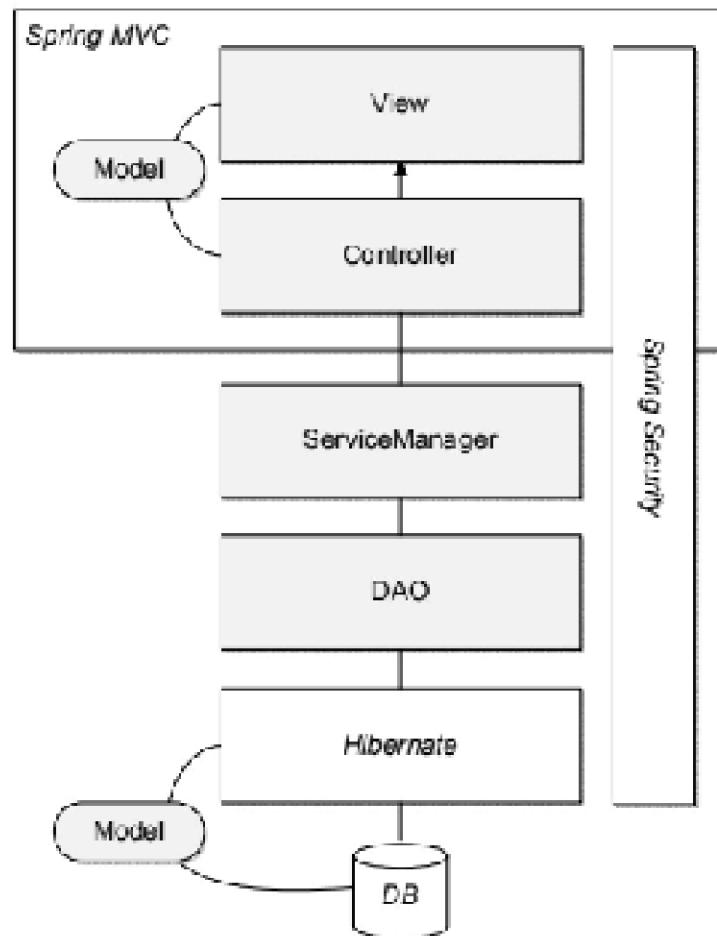
# Alte observații nivel persistență



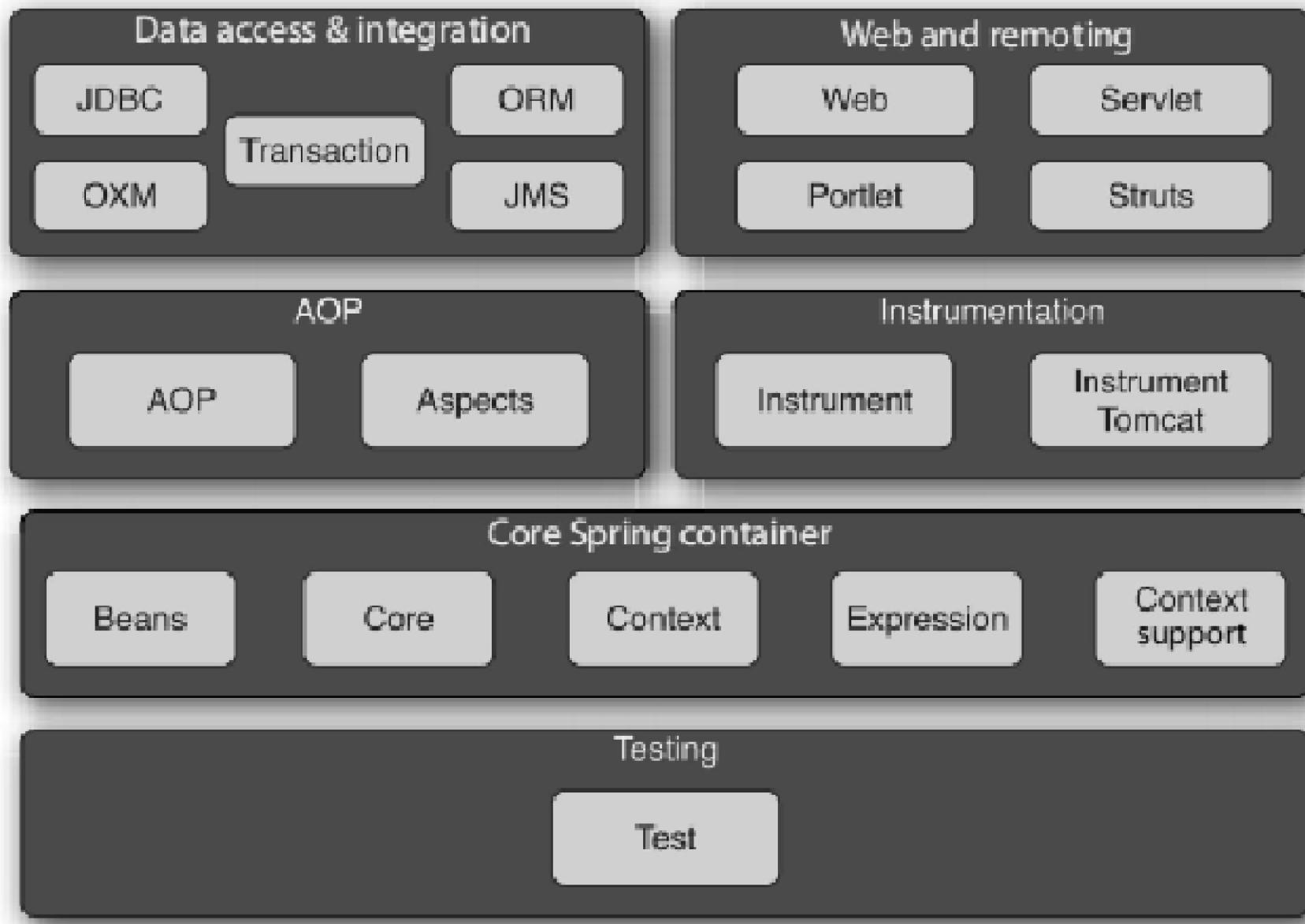
# Observații nivel prezentare



# Arhitectura care a condus la stive tehnologice specifice Spring



# Modulele framework-ului Spring



# **Container-ul principal (core)**



# **Suportul AOP**



# Acesul la, și integrarea datelor



# **WEB și accesul la distanță (REMOTING)**



# Testare

Testing

Test

# **Ce mai oferă?**

- **SPRING WEB SERVICES**
- **SPRING SECURITY**
- **SPRING INTEGRATION**
- **SPRING BATCH**

# **Ce mai oferă?**

- **SPRING SOCIAL**
- **SPRING MOBILE**
- **SPRING DYNAMIC MODULES**
- **SPRING LDAP**

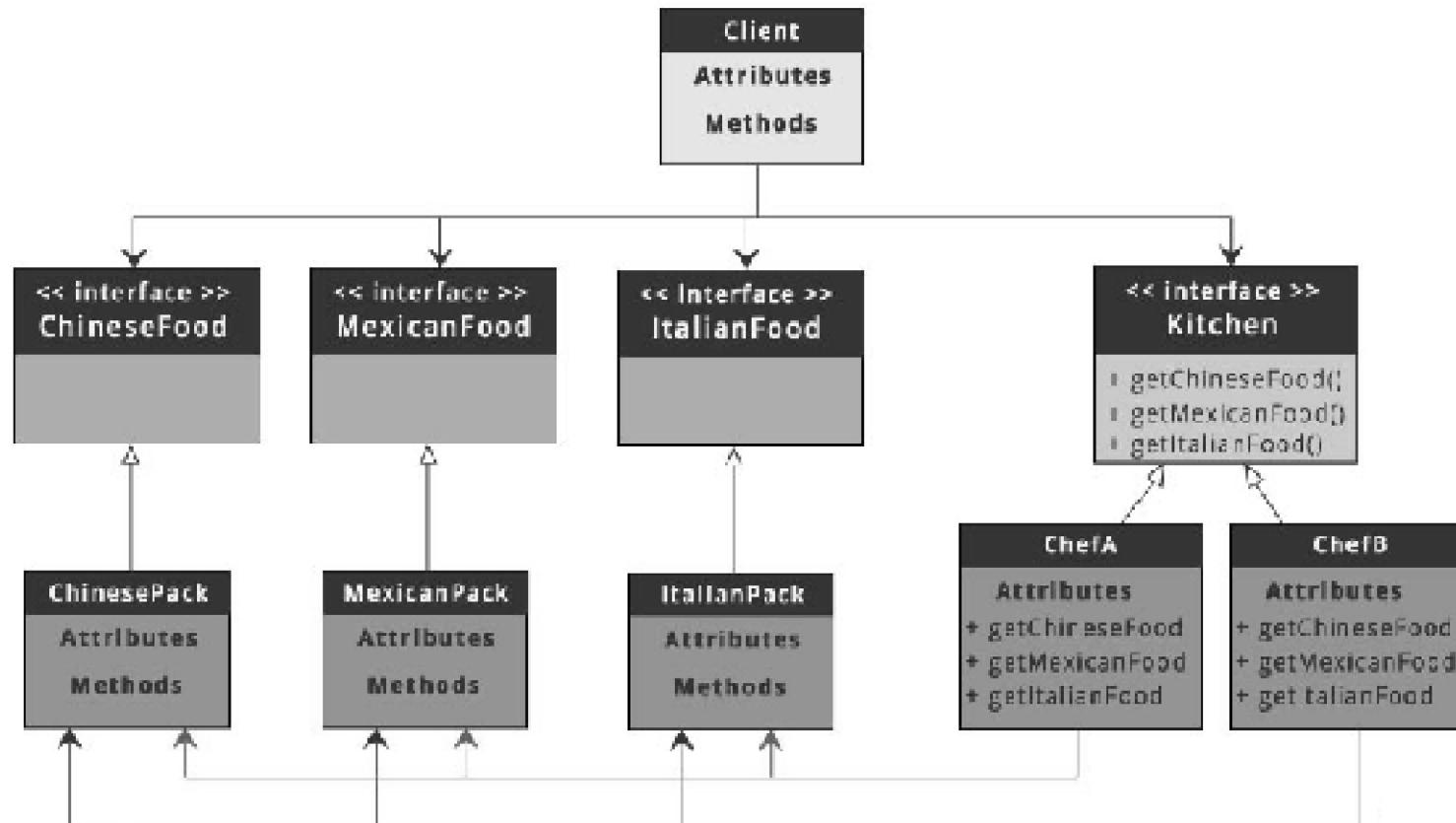
# **Ce mai oferă?**

- **SPRING RICH CLIENT**
- **SPRING.NET**
- **SPRING-FLEX**
- **SPRING ROO**

# **SPRING EXTENSIONS**

- o implementare a Spring pentru Python
- stocare blob (specifica Azure)
- persistenta bazata pe db4o si CouchDB
- o biblioteca pentru workflow management
- extensii ale Spring Security pentru Kerberos si SAML

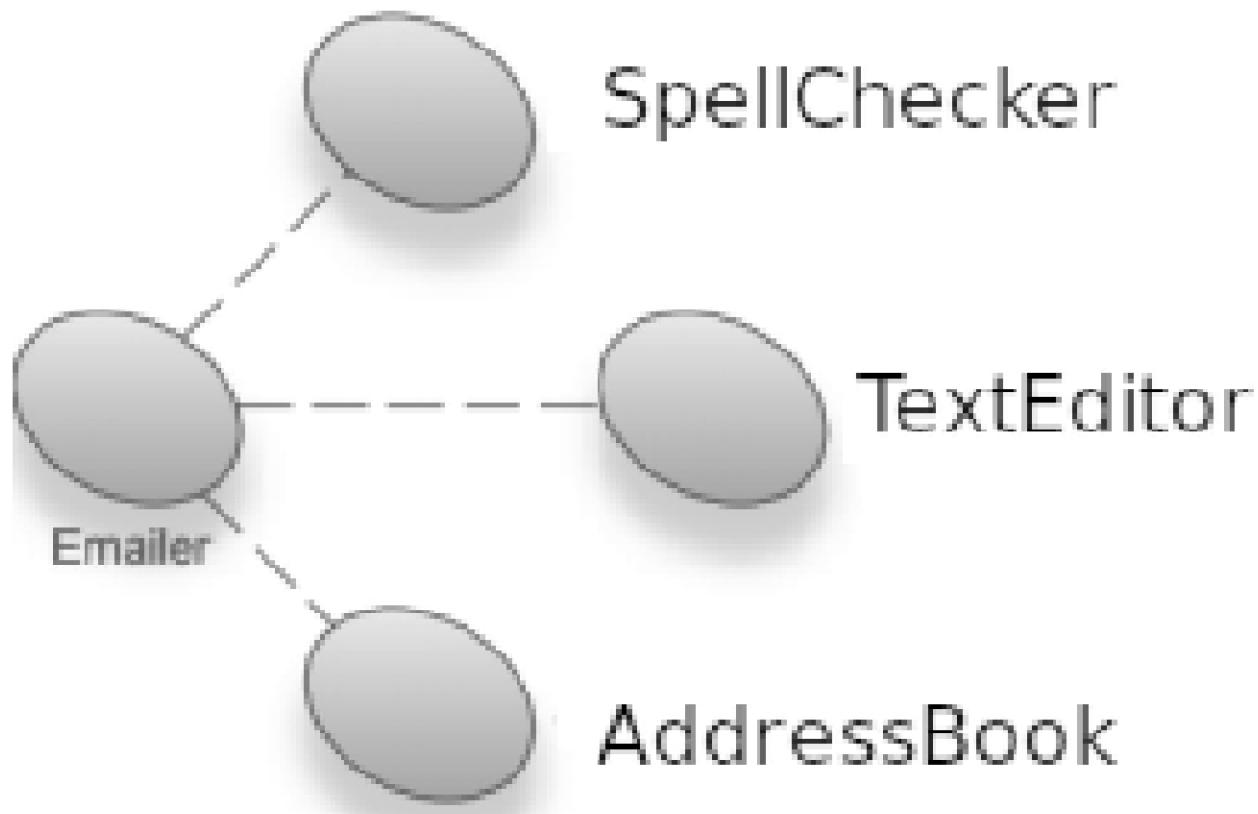
# Decuplarea prin interfețe



# Injectarea dependințelor



# Injectarea dependințelor



# Injectarea dependințelor

- **Dependință**
- **Dependent**

```
public class Emailer {  
    private SpellChecker spellChecker;  
    public void setSpellChecker(SpellChecker  
        spellChecker) {  
        this.spellChecker = spellChecker;  
    }  
    ...  
}
```

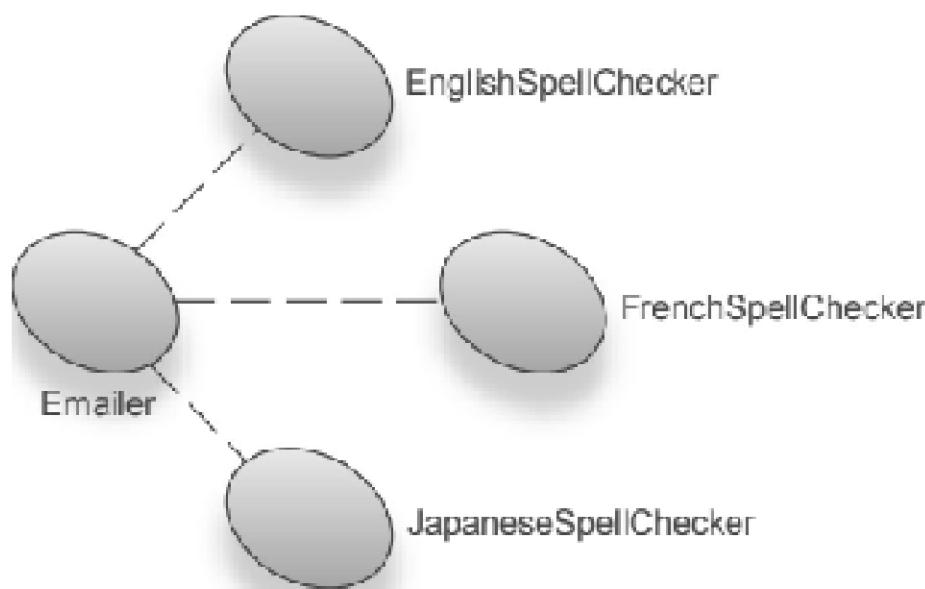
# Injectarea dependințelor



```
@Test
public void ensureEmailerChecksSpelling() {
    MockSpellChecker mock = new MockSpellChecker();
    Emailer emailer = new Emailer();
    emailer.setSpellChecker(mock); //trimiterea
    dependenței inexistente pentru testare
    emailer.send("Hello there!");
    assert mock.verifyDidCheckSpelling(); //verificare că
    dependența a fost utilizată corect
}
```

# Injectarea dependințelor

```
Emailer service = new Emailer();  
service.setSpellChecker(new FrenchSpellChecker());
```



# Injectarea dependințelor

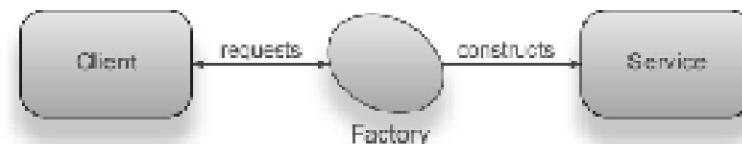
- Tot în acest caz se poate utiliza un constructor în loc de un setter:

```
public class Emailer {  
    private SpellChecker spellChecker;  
    public Emailer(SpellChecker spellChecker) {  
        this.spellChecker = spellChecker;  
    }  
    ...  
}
```

- În acest caz creare obiectului emailer se simplifică:  
`Emailer service = new Emailer(new JapaneseSpellChecker());`

# Injectarea dependințelor

```
public class Emailer {  
    private SpellChecker spellChecker;  
    public Emailer(SpellChecker spellChecker) {  
        this.spellChecker = spellChecker;  
    }  
    ...  
}
```



Deci pasăm sarcina creării către alta clasă:

```
public class EmailerFactory {  
    public Emailer newFrenchEmailer() {  
        return new Emailer(new FrenchSpellChecker());  
    }  
}
```

Astfel injectia manuală devine:

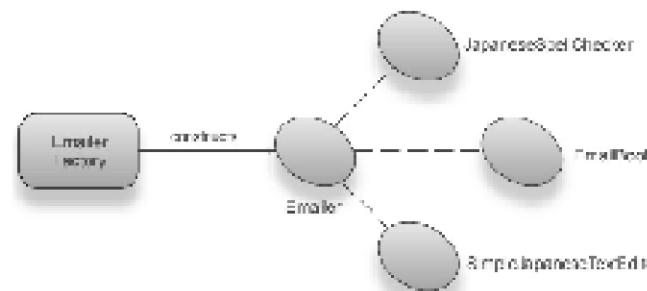
```
Emailer service = new EmailerFactory().newFrenchEmailer();
```

# Injectarea dependințelor

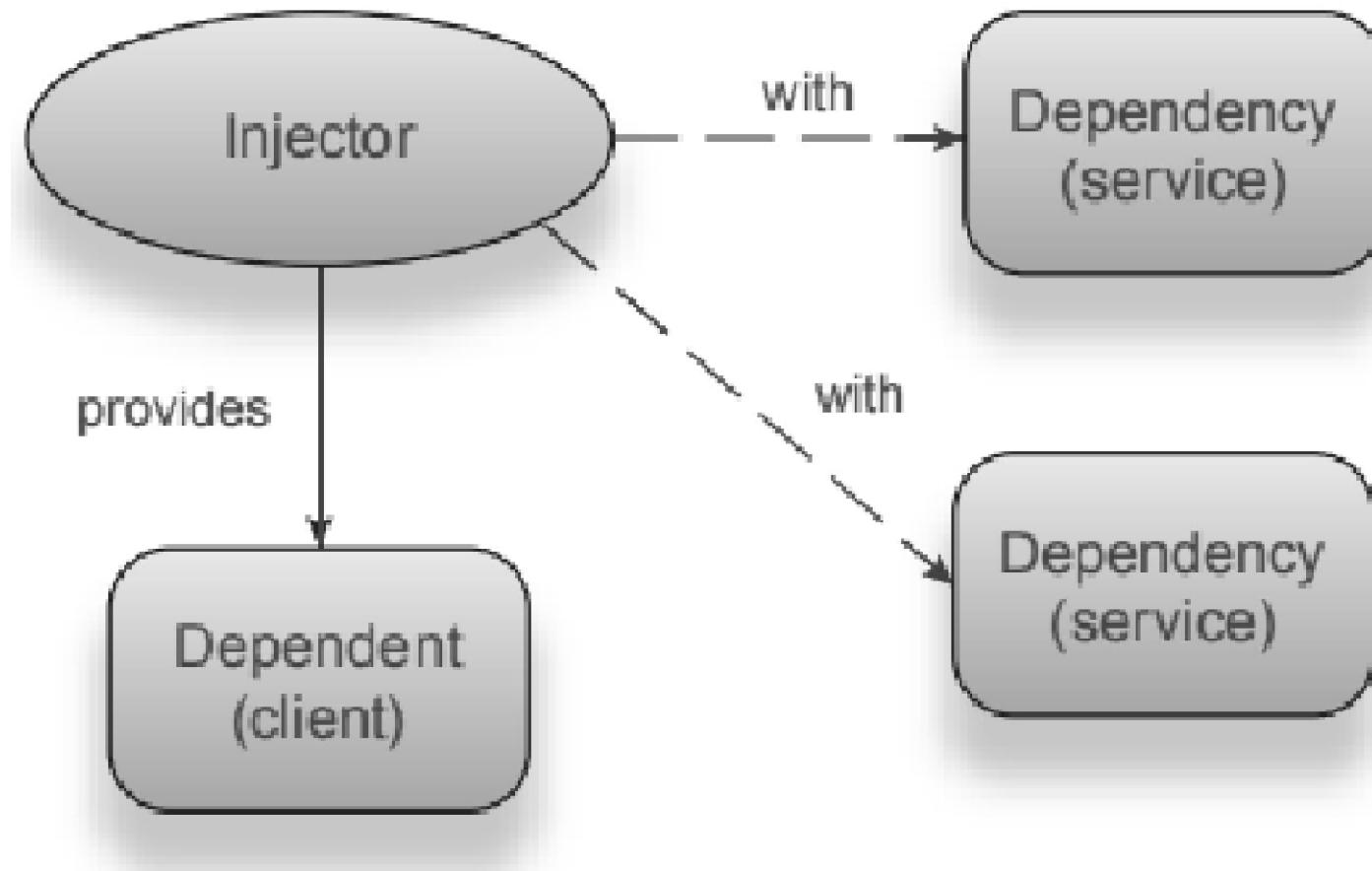
```
public class EmailerFactory {  
    public Emailer newJapaneseEmailer() {  
        Emailer service = new Emailer();  
        service.setSpellChecker(new JapaneseSpellChecker());  
        service.setAddressBook(new EmailBook());  
        service.setTextEditor(new SimpleJapaneseTextEditor());  
        return service;  
    }  
}
```

- Si astfel apelul devine

```
Emailer emailer = new EmailerFactory().newJapaneseEmailer();
```



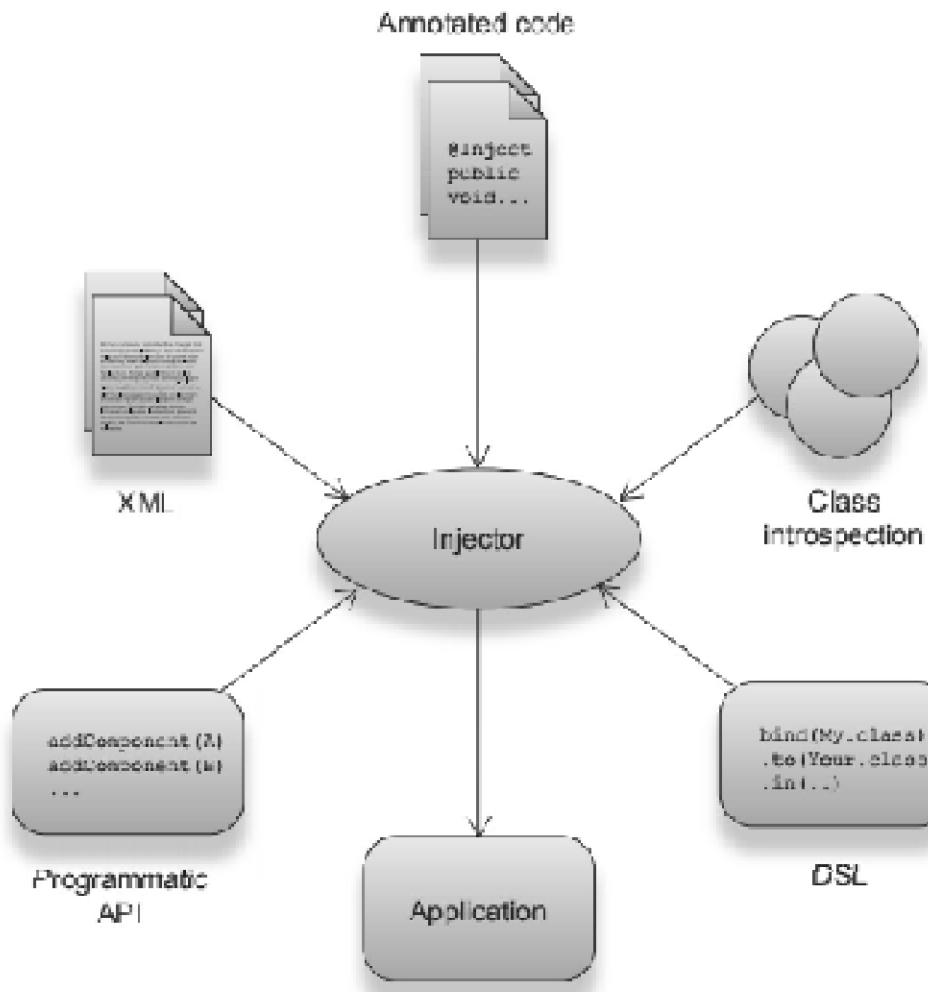
# Injectarea dependințelor - Spring



## **Controlul Invers versus injectarea dependintelor**

- un modul într-un server aplicații JEE
- un obiect legat de un injector de dependințe
- o metoda apelată automat într-un framework
- un gestionar de evenimente apelat de un eveniment (de ex. apăsare buton mouse)

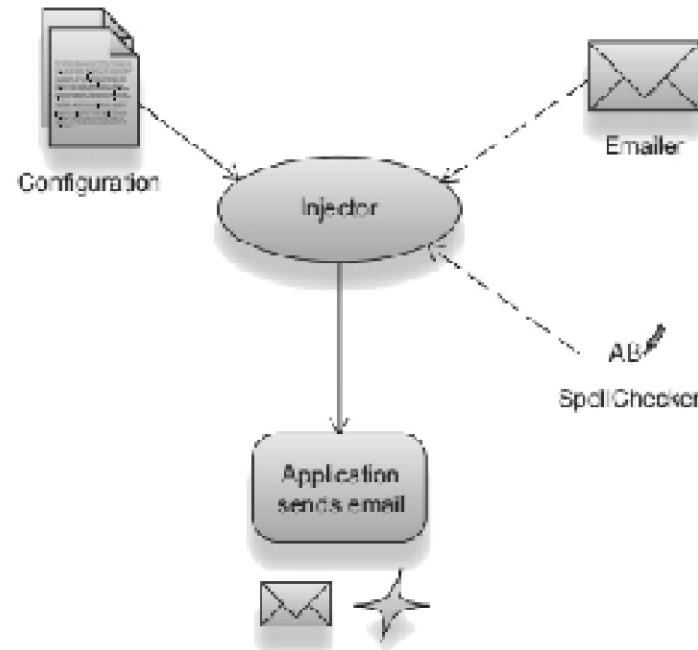
# Metadatele și configurarea injectorului



**@Autowired și @Component**

# configurarea injectorului - cu XML

```
BeanFactory injector = new  
FileSystemApplicationContext("email.xml");  
Emailer emailer = (Emailer) injector.getBean("emailer");  
emailer.send("Hello!");
```



# **configurarea injectorului - cu XML**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/b
eans
http://www.springframework.org/schema/beans/spring-beans-
x.x.xsd" >
    <bean id="emailer" class="Emailer"> //declaram instancele lui
Emailer
        <constructor-arg ref="spellChecker"/> //injectia
    </bean>
    <bean id="spellChecker" class="SpellChecker"/> // instancele lui
SpellChecker
</beans>
```

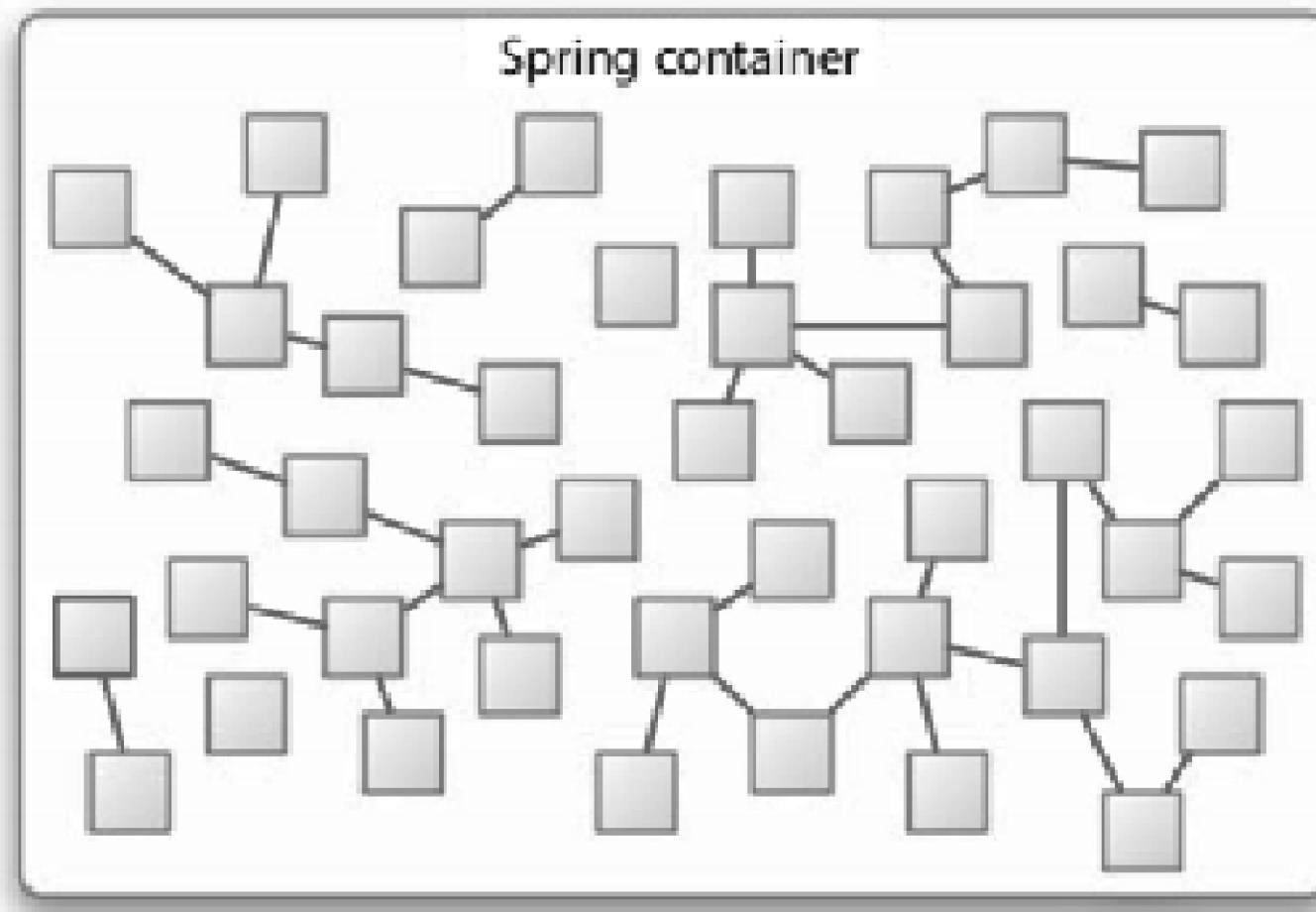
# **configurarea injectorului - automată**

```
public class Emailer {  
    private SpellChecker spellChecker;  
    public Emailer(SpellChecker spellChecker) {  
        this.spellChecker = spellChecker; }  
    public void send(String text) {  
        spellChecker.check(text);  
        // trimite dacă totul este corect... }  
    }  
<beans ...>  
    <bean id="spellChecker" class="SpellChecker"/>  
    <bean id="emailer" class="Emailer" autowire="constructor"/>  
</beans>
```

# configurarea injectorului - automată

```
@Component
public class Emailer {
    private SpellChecker spellChecker;
    public Emailer() {
        this.spellChecker = null; }
    @Autowired
    public Emailer(SpellChecker spellChecker) {
        this.spellChecker = spellChecker; }
    public void send(String text) {
        spellChecker.check(text); }
}
<beans ...>
    <bean id="spellChecker" class="SpellChecker"/>
    <bean id="emailer" class="Emailer"/>
</beans>
```

# Containere Spring



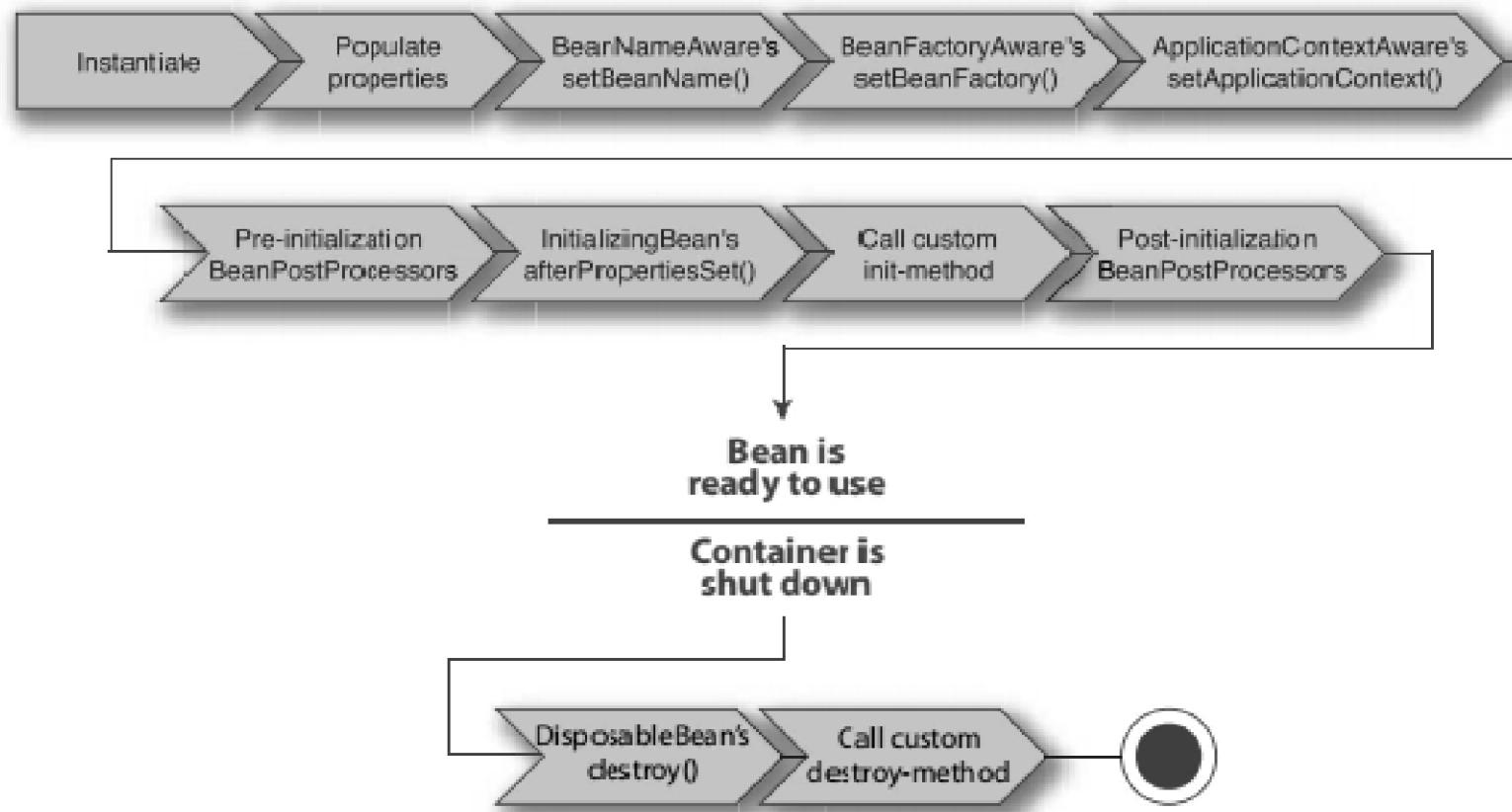
# **Contexte predefinite pentru aplicații**

- ClassPathXmlApplicationContext
- FileSystemXmlApplicationContext
- XmlWebApplicationContext

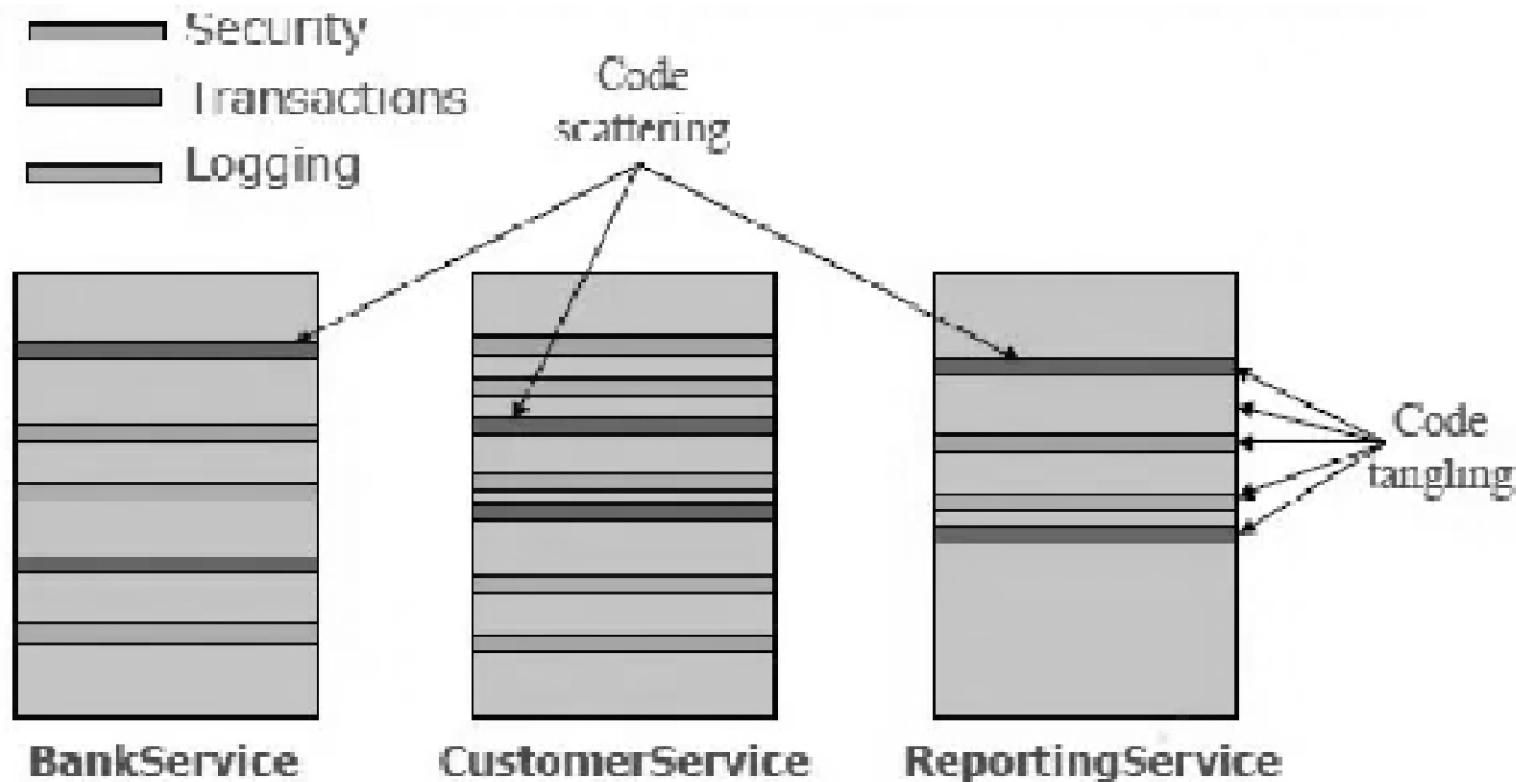
```
ApplicationContext context = new  
FileSystemXmlApplicationContext("c:/foo.xml");
```

```
ApplicationContext context = new  
ClassPathXmlApplicationContext("foo.xml");
```

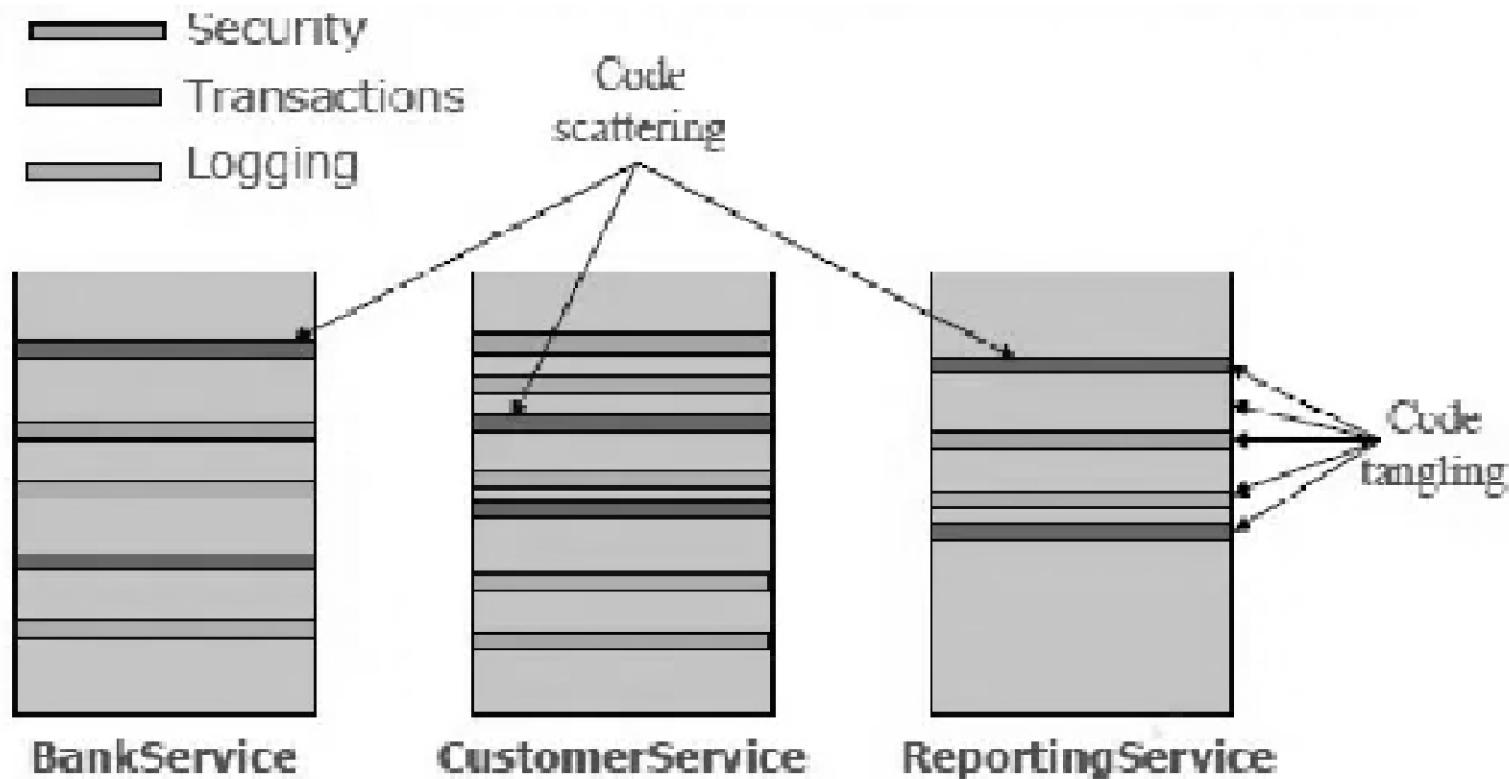
# Ciclul de viata al unui bean Spring



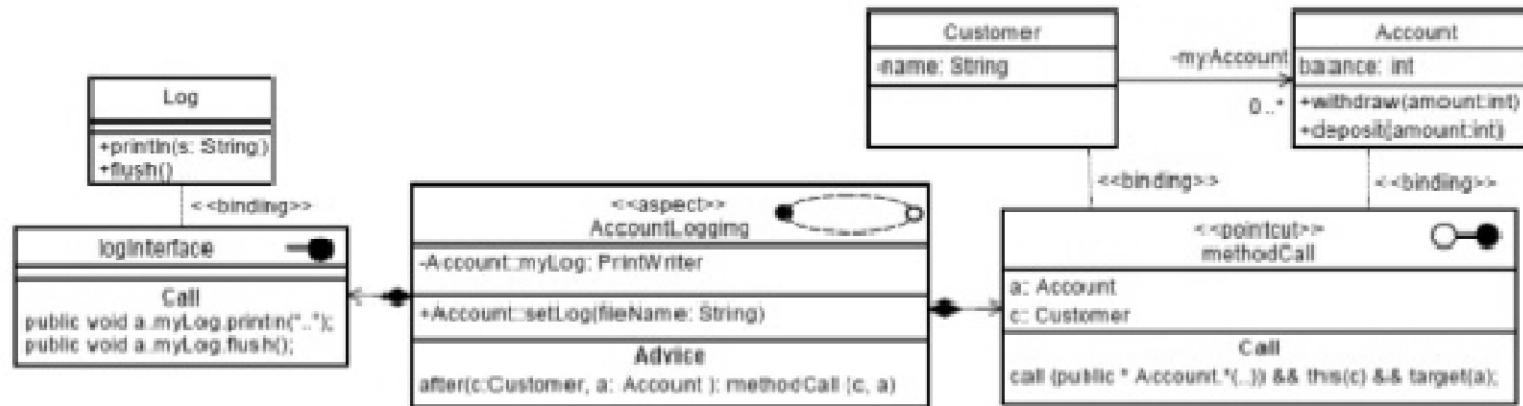
# AOP



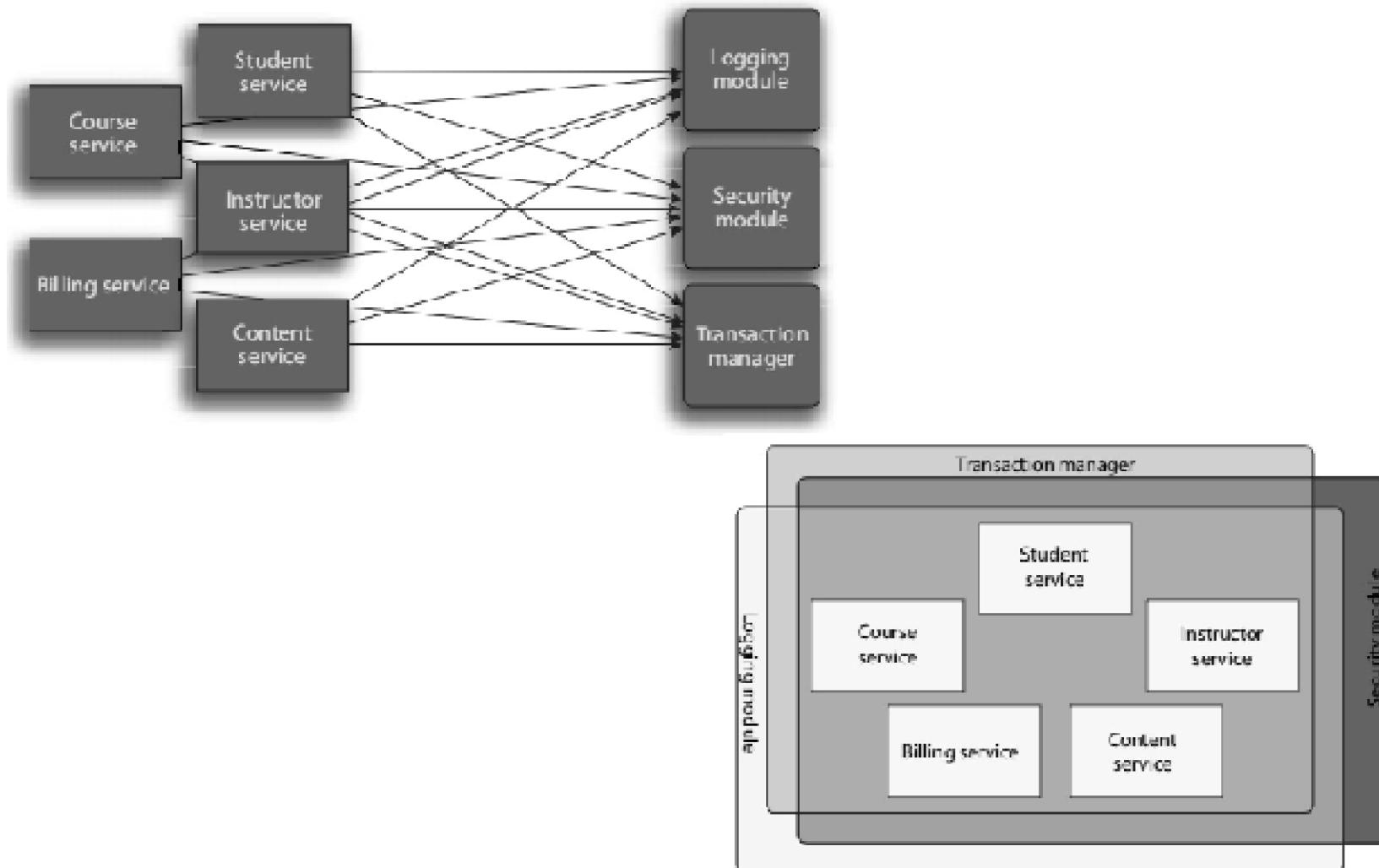
# AOP



# Aspectul



# Identificarea aspectelor periferice (crosscutting concerns)



# **Pașii în proiectarea AOP**

- Descompunere: Analiza cerințelor va identifica două tipuri de funcționalități:
  - common concerns (funcționalități primare ale produsul de bază)
  - crosscutting concerns (logare, securitate, transactii etc)
- Implementare: se va face folosind instrumente specifice metodologiei folosite:
  - clase, bean-uri, servicii (Produs)
  - aspecte (logare, securitate, transactii etc)
- Recomponere: se va face folosind un integrator (weaver) pentru care trebuie definite reguli de compunere

# **Termeni specifici**

- **Aspect**
- **Weaving**
- **Join point**
- **Target object:**
- **Target method:**

# **Termeni specifici**

- **Advice**
- **Before advice**
- **After returning advice**
- **After throwing advice**
- **After (finally) advice:**
- **Around advice**

# **Termini specifici**

- **Pointcut**
- **Introduction - @Declare\***
- **AOP proxy**

# Diferențe între Spring AOP și AspectJ

Joinpoint	Spring AOP	AspectJ
Apel Metodă	Nu	Da
Executare metodă	Da	Da
Apel Constructor	Nu	Da
Execuție Constructor	Nu	Da
Execuție inițializare statică	Nu	Da
Inițializare obiect	Nu	Da
Referință la câmp date	Nu	Da
Inițializare câmp de date	Nu	Da
Execuție handler	Nu	Da
Execuție advice	Nu	Da

# Principalele diferențe

## Spring AOP

Implementat în Java

Nu necesită compilare suplimentară

Este disponibilă numai întrețeserea codului numai în timpul execuției

Simplifică suportă numai întrețesere la nivel de metodă

Poate fi utilizat numai sănbean-urile controlate de containerul Spring

Suportă punct de unire numai la nivelul execuției metodei

Se crează intermediari pentru obiectele unde se aplică aspectele iar acestea sunt aplicate asupra acestora

Muult mai lent decât AspectJ

Ușor de înțeles și de aplicat

## AspectJ

Implementat utilizând extensii ale limbajului Java

Necesită compilatorul de AspectJ compiler (ajc) - cu ocazii anterioare

întrețeserea codului în timpul execuției nu este disponibilă în schimb se poate întreține codul, în timpul, după compilare sau la încărcare

Poate întreține cod cu acțiuni la nivel de câmpuri de date, metode, constructori, inițializatori statici sau dinamici, clase sau metode finale etc

Poate fi implementat pentru orice obiect din aplicație

Acceptă toate tipurile de puncte de unire

Aspectele sunt introduse în cod înaintea execuției aplicației

Performanță ridicată

Muult mai complicat trebuie înțeles conceptul de proiectare aop care trebuie aplicat peste arhitectura curentă a aplicației sau din primele faze de proiectare