



Spring

Università di Bologna
CdS Laurea Magistrale in Ingegneria Informatica
I Ciclo - A.A. 2013/2014

Corso di Sistemi Distribuiti M (6 cfu)
08- Lightweight Container:
Tecnologia Spring

Docente: Paolo Bellavista
paolo.bellavista@unibo.it

<http://lia.deis.unibo.it/Courses/sd1314-info/>
<http://lia.deis.unibo.it/Staff/PaoloBellavista/>



Introduzione a Spring

Che cos'è Spring?

- ❑ **Framework leggero** per la costruzione di applicazioni Java SE e Java EE

Molti dei concetti chiave alla base di Spring sono stati di successo così rilevante da essere diventati linee guida per l'evoluzione di EJB3.0

Funzionalità chiave:

- ❑ ***Inversion of Control (IoC) e Dependency injection***
- ❑ Supporto alla persistenza
- ❑ Integrazione con Web tier
- ❑ ***Aspect Oriented Programming (AOP)***



Funzionalità Chiave: Dependency Injection e Persistenza

Dependency Injection

- ❑ Gestione della configurazione dei componenti applica principi di ***Inversion-of-Control*** e utilizza ***Dependency Injection***
 - Eliminazione della necessità di binding “manuale” fra componenti
- ❑ ***Idea fondamentale di una factory per componenti (BeanFactory)*** utilizzabile globalmente. Si occupa del ritrovamento di oggetti per nome e della ***gestione delle relazioni fra oggetti (configuration management)***

Persistenza

- ❑ ***Livello di astrazione generico*** per la gestione delle transazioni con DB (senza essere forzati a lavorare dentro un EJB container)
- ❑ Strategie generiche e built-in per JTA e l'interazione con una singola sorgente JDBC
 - ***Elimina dipendenza da container J2EE*** per il supporto alle transazioni
- ❑ Integrazione con framework di persistenza come Hibernate, JDO, JPA



Funzionalità Chiave: Web tier e AOP

Integrazione con Web tier

- ❑ **Framework MVC per applicazioni Web**, costruito sulle funzionalità base di Spring, con supporto per diverse tecnologie per la **generazione di viste**, ad es. JSP, FreeMarker, Velocity, Tiles, iText e POI (Java API per l'accesso a file in formato MS)
- ❑ **Web Flow** per navigazione a grana fine

Supporto a Aspect Oriented Programming

- ❑ Framework di supporto a servizi di sistema, come gestione delle transazioni, tramite **tecniche AOP**
 - Miglioramento soprattutto in termini di **modularità**
 - Parzialmente correlata anche la facilità di testing



Yet Another Framework?

No, Spring rappresenta un approccio piuttosto unico

(che ha fortemente influenzato i container successivi, **verso tecnologie a microcontainer** – Spring 1.2 è datato Maggio 2005).

In altre parole, **proprietà originali:**

- ❑ Spring come **framework modulare. Architettura a layer**, possibilità di utilizzare anche solo alcune **parti in isolamento**
 - Anche possibilità di introdurre Spring **incrementalmente in progetti esistenti** e di imparare ad utilizzare la tecnologia “pezzo per pezzo”
- ❑ **Supporto a importanti aree** non coperte da altri framework diffusi, come **il management degli oggetti** di business
- ❑ **Tecnologia di integrazione** di soluzioni esistenti
- ❑ Facilità di **testing**



Quindi, perché usare Spring?

- ❑ Integrazione e cooperazione fra componenti (secondo il semplice modello JavaBean) via **Dependency Injection**
 - **Disaccoppiamento**
- ❑ **Test-Driven Development** (TDD)
 - Possibilità di effettuare testing delle classi (POJO) **senza essere legati al framework**
- ❑ Uso semplificato di tecnologie diffuse e di successo
 - Astrazioni che isolano il codice applicativo, eliminazione di codice ridondante, gestione di comuni condizioni di errore (caso delle **unchecked exception**)
 - **Specificità** delle tecnologie sottostanti sono comunque ancora **accessibili (parziale visibilità)**
- ❑ Progettazione per interfacce
 - Ottimo isolamento delle funzionalità dai dettagli implementativi
- ❑ **Programmazione dichiarativa via AOP**
 - Facile configurazione degli **aspetti**, ad esempio supporto alle transazioni

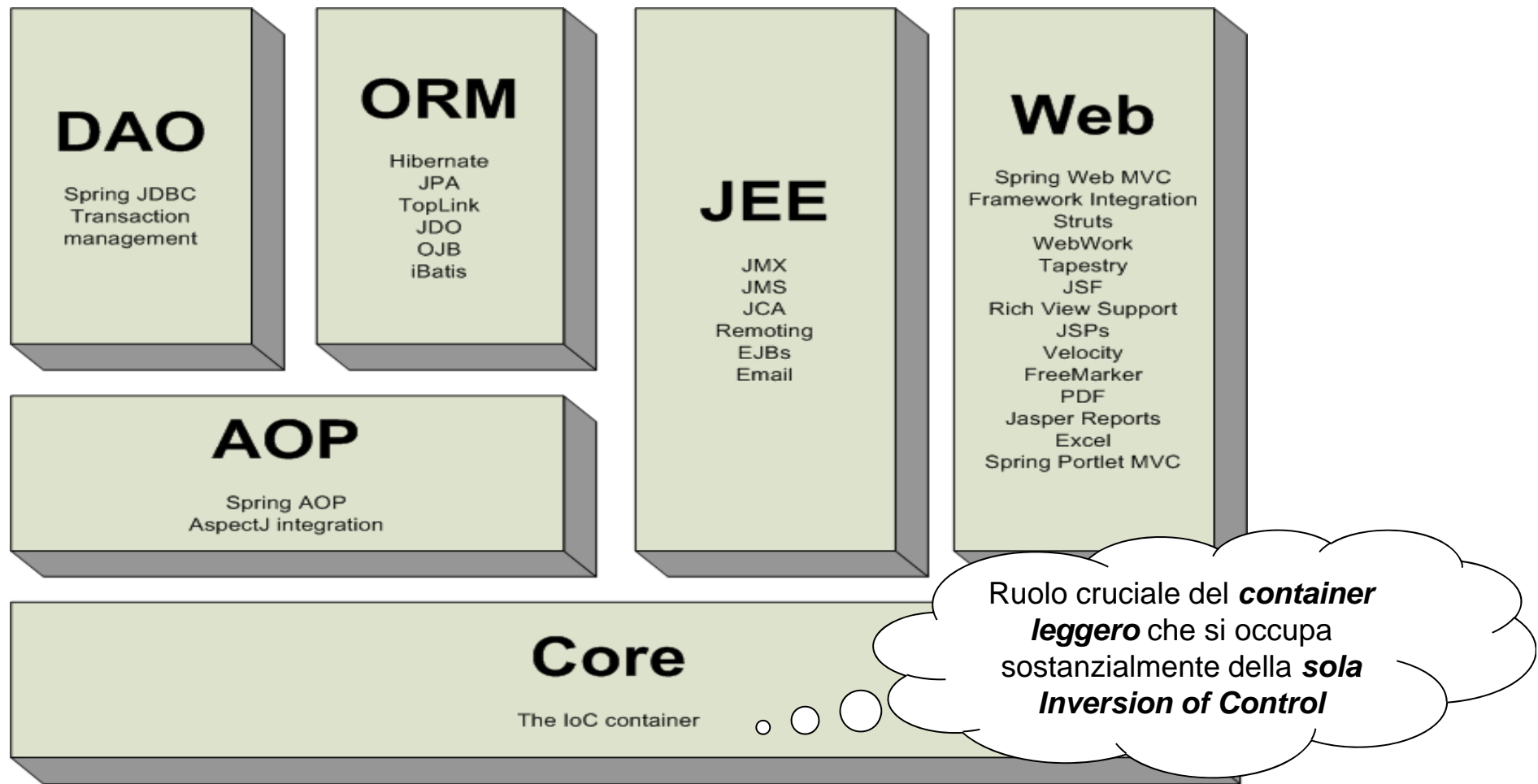


Quindi, perché usare Spring?

- ❑ ***NON è una soluzione “all-or-nothing”***
 - Estrema ***modularità e flessibilità***
 - Progettata per essere ***facile da estendere*** e con molte classi riutilizzabili
- ❑ ***Integrazione*** con altre tecnologie
 - *EJB* per J2EE
 - *Hibernate, iBates, JDBC* per l'accesso a dati e O/RM
 - *Java Persistence API* per persistenza
 - *Struts e WebWork* per Web tier
 - ...

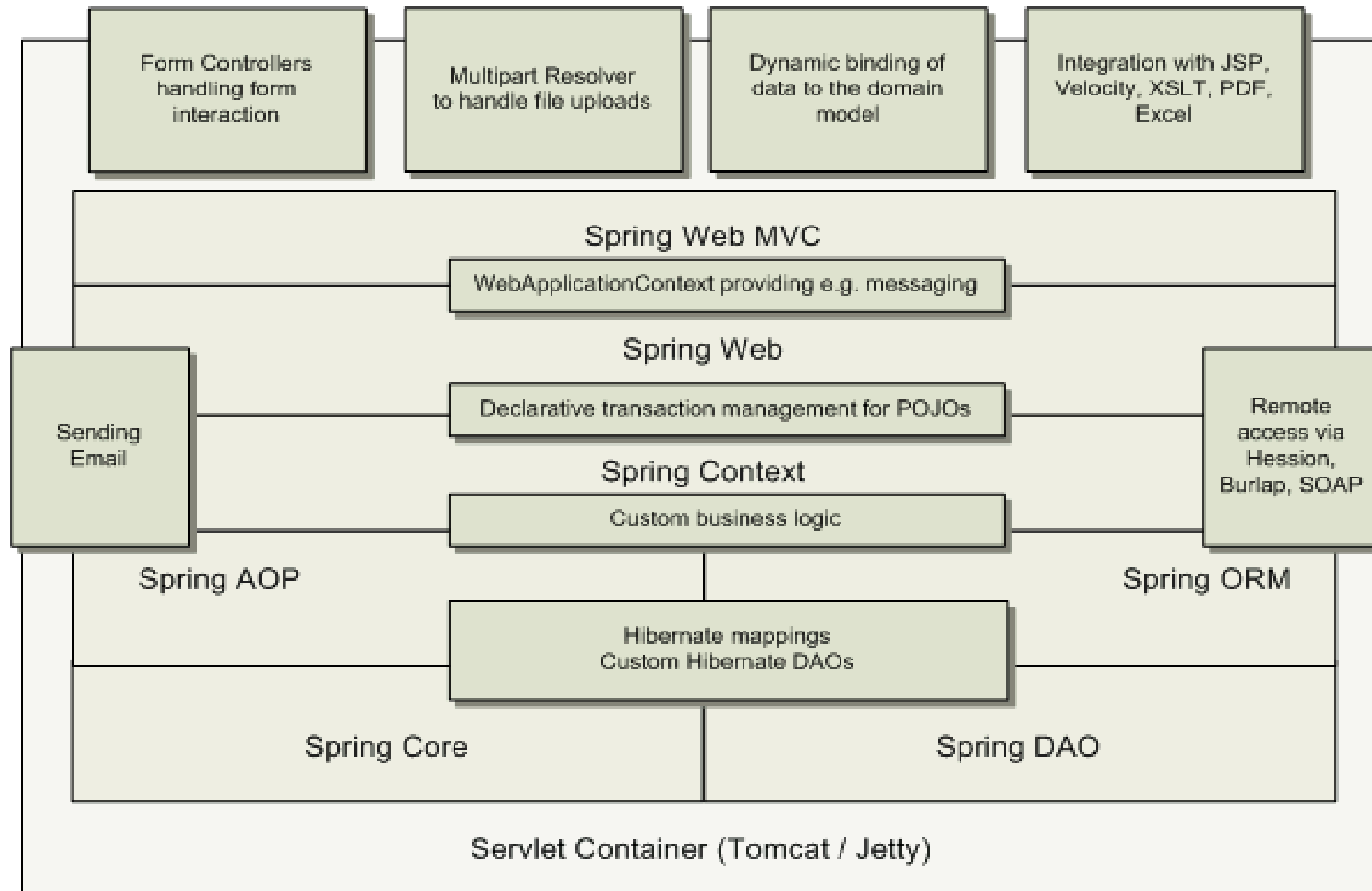


Architettura di Spring





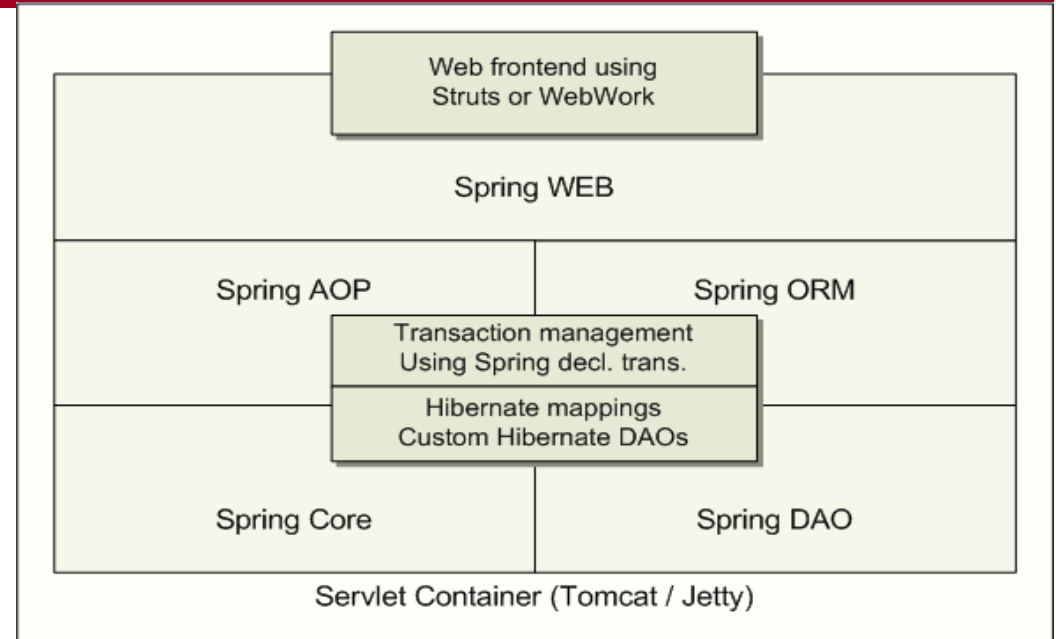
Spring e Flessibilità di Utilizzo in Scenari Differenti



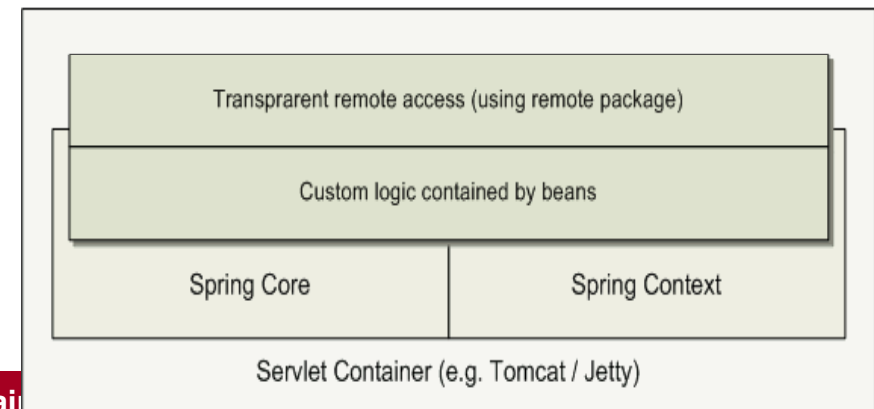


Spring e vari Scenari di Utilizzo

Ad esempio, solo middle tier e integrazione con Web



Ad esempio, solo remoting





Architettura di Spring

Core Package

- ❑ **Parte fondamentale del framework.** Consiste in un **container leggero** che si occupa di **Inversion of Control** o, per dirla alla Fowler, **Dependency Injection**
- ❑ L'elemento fondamentale è **BeanFactory**, che fornisce una implementazione estesa del pattern factory ed **elimina la necessità di gestione di singleton** a livello di programmazione, permettendo di disaccoppiare configurazione e dipendenze dalla logica applicativa

Vi ricordate bene,
vero, che cos'è un
singleton?

DAO Package

- ❑ Livello di astrazione che non rende più necessario boilerplate code per JDBC, né parsing di codici di errore database-specific
- ❑ Gestione delle transazioni sia da codice che in modo dichiarativo, **non solo per classi che implementano interfacce speciali** (possibilità aperta a tutti i POJO)



Architettura di Spring

ORM Package

- ❑ Livello di integrazione con soluzioni diffuse per OR/M, come *JPA*, *JDO*, *Hibernate*, *iBatis*, ...
- ❑ Le varie soluzioni O/RM suddette possono essere usate in combinazione con le altre funzionalità di Spring, come la gestione dichiarativa delle transazioni => ***Spring come tecnologia di integrazione***

MVC Package

- ❑ Implementazione di ***Model-View-Controller (MVC)*** per applicazioni Web; buona separazione fra codice del modello di dominio e form Web



Architettura di Spring

AOP Package

- ❑ **Implementazione di aspect-oriented programming conforme allo standard AOP Alliance.** Permette di definire, ad esempio, **intercettori di metodo e pointcut** per disaccoppiamento pulito
- ❑ Possibilità di utilizzare **metadati a livello sorgente** per incorporare informazioni aggiuntive di comportamento all'interno del codice

Necessità di aprire una parentesi su AOP ☺?



Aspect Oriented Programming (AOP)

- ❑ Aspect Oriented programming (AOP) come approccio di design e tecnica per ***semplificare l'applicazione di cross-cutting concern (problematiche trasversali alla logica applicativa)***
- ❑ Esempi di cross-cutting concern
 - Logging
 - Locking
 - Gestione degli eventi
 - Gestione delle transazioni
 - Sicurezza e auditing
- ❑ Concetti rilevanti per AOP:
 - Joinpoint
 - Advice
 - Pointcut e Aspect
 - Weaving e Target
 - Introduction



AOP: Joinpoint & Advice

Joinpoint

- ❑ Punto ben definito del codice applicativo, anche determinato a runtime, ***dove può essere inserita logica aggiuntiva***
- ❑ Esempi di joinpoint
 - Invocazione di metodi
 - Inizializzazione di classi
 - Inizializzazione di oggetti (creazione di istanze)

Advice

- ❑ ***Codice con logica aggiuntiva*** che deve essere eseguito ad un determinato joinpoint
- ❑ Tipi di Advice
 - *before advice* eseguono ***prima*** del joinpoint
 - *after advice* eseguono ***dopo*** il joinpoint
 - *around advice* eseguono ***attorno*** (*around*) al joinpoint



AOP: Pointcut & Aspect

Pointcut

- ❑ ***Insieme di joinpoint*** usati per definire ***quando eseguire un advice***
- ❑ Controllo fine e flessibile su come applicare advice al codice applicativo
- ❑ Ad esempio:
 - Invocazione di metodo è un tipico joinpoint
 - Un tipico pointcut è l'insieme di tutte le invocazioni di metodo in una classe determinata
- ❑ ***Pointcut possono essere composti in relazioni anche complesse*** per vincolare il momento di esecuzione dell'advice corrispondente

Aspect

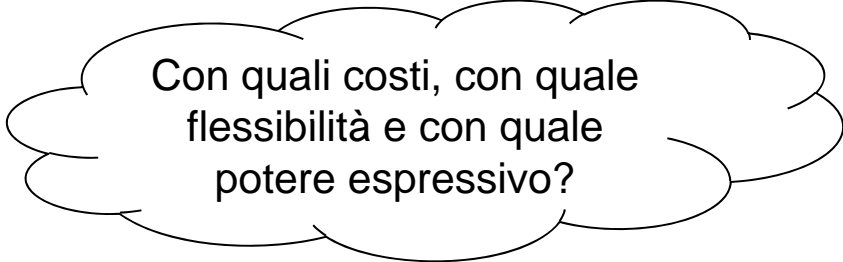
- ❑ Aspect come ***combinazione di advice e pointcut***



AOP: Weaving & Target

Weaving

- ❑ Processo **dell'effettivo inserimento di aspect dentro il codice applicativo** nel punto appropriato
- ❑ Tipi di weaving
 - A tempo di compilazione
 - Runtime



Con quali costi, con quale flessibilità e con quale potere espressivo?

Target

- ❑ Un **oggetto** il cui **flusso di esecuzione viene modificato** da qualche processo AOP
- ❑ Viene anche indicato qualche volta come *oggetto con advice* (*advised object*)



AOP Statico o Dinamico

❑ **AOP Statico**

- Il processo di weaving viene realizzato come ***passo ulteriore del processo di sviluppo***, durante il build dell'applicazione
- Incide sul ***codice dell'applicazione che viene eseguito***
- Ad esempio, in un programma Java, si può avere weaving attraverso la ***modifica del bytecode*** di una applicazione

❑ **AOP Dinamico**

- Processo di weaving realizzato dinamicamente a runtime
- Possibilità di ***cambiare weaving senza bisogno di ricompilazione***



AOP in Spring

A questo punto, se voi doveste implementare AOP in Spring (in ambiente Java in generale), che tipo di approccio usereste?

Spring realizza AOP sulla base **dell'utilizzo di proxy**

- ❑ Se si desidera creare una classe advised, occorre utilizzare la classe **ProxyFactory** per **creare un proxy per un'istanza di quella classe**, fornendo a **ProxyFactory** tutti gli aspect con cui si desidera informare il proxy



Dependency Injection in Spring

- ❑ ***Applicazione più nota e di maggiore successo del principio di Inversion of Control***
- ❑ “Hollywood Principle”
 - Don't call me, I'll call you
- ❑ Container (in realtà il container leggero di Spring) si occupa di ***risolvere (injection)*** le dipendenze dei componenti attraverso l'opportuna ***configurazione dell'implementazione dell'oggetto (push)***
- ❑ Opposta ai pattern più classici di ***istanziazione di componenti*** o ***Service Locator***, dove è il componente che deve determinare l'implementazione della risorsa desiderata (*pull*)

Martin Fowler chiamò per primo ***Dependency Injection*** questo tipo di IoC



Potenziati Benefici di Dependency Injection

Dopo avere visto EJB3.0, oramai ne siete esperti mondiali ☺...

❑ **Flessibilità**

- Eliminazione di codice di lookup nella logica di business

❑ **Possibilità e facilità di testing**

- Nessun bisogno di ***dipendere da risorse esterne o da container in fase di testing***
- Possibilità di abilitare ***testing automatico***

❑ **Manutenibilità**

- Permette riutilizzo in diversi ambienti applicativi cambiando semplicemente i file di configurazione (o in generale le specifiche di dependency injection) e ***non il codice***



Due Varianti per Dependency Injection in Spring

❑ Dependency injection **a livello di costruttore**

Dipendenze fornite attraverso i costruttori dei componenti

```
public class ConstructorInjection {  
    private Dependency dep;  
    public ConstructorInjection(Dependency dep) {  
        this.dep = dep;    } }  

```

❑ Dependency injection **a livello di metodi “setter”**

- Dipendenze fornite attraverso i **metodi di configurazione** (*metodi setter in stile JavaBean*) dei componenti
- Più frequentemente utilizzata nella comunità degli sviluppatori

```
public class SetterInjection {  
    private Dependency dep;  
    public void setMyDependency(Dependency dep) {  
        this.dep = dep;    } }  

```



BeanFactory

- ❑ L'oggetto **BeanFactory** è responsabile della **gestione dei bean che usano Spring e delle loro dipendenze**
- ❑ Ogni applicazione interagisce con la dependency injection di Spring (*IoC container*) tramite l'interfaccia **BeanFactory**
 - Oggetto **BeanFactory** viene creato dall'applicazione, tipicamente nella forma di **XmlBeanFactory**
 - Una volta creato, l'oggetto **BeanFactory** legge un **file di configurazione e si occupa di fare l'injection** (*wiring*)
- ❑ **XmlBeanFactory** è estensione di **DefaultListableBeanFactory** per leggere definizioni di bean da un documento XML

```
public class XmlConfigWithBeanFactory {  
  
    public static void main(String[] args) {  
        XmlBeanFactory factory = new XmlBeanFactory(new  
            FileSystemResource("beans.xml"));  
        SomeBeanInterface b = (SomeBeanInterface) factory.  
            getBean("nameOftheBean"); } }
```



File di Configurazione

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="renderer" class="StandardOutMessageRenderer">
    <property name="messageProvider">
      <ref local="provider"/>
    </property>
  </bean>
  <bean id="provider" class="HelloWorldMessageProvider"/> </beans>
```

Oppure a livello di costruttore

...

```
<beans>
<bean id="provider" class="ConfigurableMessageProvider">
  <constructor-arg>
    <value> Questo è il messaggio configurabile</value>
  </constructor-arg>
</bean> </beans>
```

Vedete anche l'esempio successivo, in cui in grande dettaglio sarà mostrato come il container IoC faccia dependency injection



Uso della Dependency Injection

```
public class ConfigurableMessageProvider implements
    MessageProvider {

    private String message;
    // usa dependency injection per config. del messaggio
    public ConfigurableMessageProvider(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
}
```



Tipi di Parametri di Injection

- ❑ Spring supporta ***diversi tipi di parametri*** con cui fare injection
 1. ***Valori semplici***
 2. ***Bean all'interno della stessa factory***
 3. ***Bean anche in diverse factory***
 4. ***Collezioni (collection)***
 5. ***Proprietà definite esternamente***
- ❑ Tutti questi tipi possono essere usati sia per injection sui costruttori che sui metodi *setter*

Ad esempio, injection di valori semplici

`<beans>`

```
<bean id="injectSimple" class="InjectSimple">
  <property name="name"> <value>John Smith</value>
</property>
  <property name="age"> <value>35</value>
</property>
  <property name="height"> <value>1.78</value>
</property>
</bean> </beans>
```



Esempio: Injection di Bean della stessa Factory

- ❑ Usata quando è necessario fare injection di un bean all'interno di un altro (*target bean*)
- ❑ Uso del tag `<ref>` in `<property>` o `<constructor-arg>` del target bean
- ❑ Controllo lasco sul tipo del bean "iniettato" rispetto a quanto definito nel target
 - Se il tipo definito nel target è ***un'interfaccia***, il bean injected deve essere ***un'implementazione di tale interfaccia***
 - Se il tipo definito nel target è una ***classe***, il bean injected deve essere della ***stessa classe o di una sottoclasse***

```
<beans>
  <bean id="injectRef" class="InjectRef">
    <property name="oracle">
      <ref local="oracle"/>
    </property>
  </bean>
</beans>
```



Naming dei Componenti Spring

Come fa BeanFactory a trovare il bean richiesto (***pattern singleton come comportamento di default***)?

- ❑ Ogni bean deve avere un ***nome unico all'interno della BeanFactory contenente***

- ❑ Procedura di risoluzione dei nomi
 - Se un tag `<bean>` ha un attributo di nome **id**, il valore di questo attributo viene usato come nome
 - Se non c'è attributo **id**, Spring cerca un attributo di nome **name**
 - Se non è definito né **id** né **name**, Spring usa il nome della classe del bean come suo nome



Spring e Container Leggero

Quindi, perché si dice in letteratura che Spring rappresenta un buon esempio di ***tecnologia a container leggero*** (*lightweight container*)?

Non solo modularità...



Codice Base di HelloWorld

PARENTESI: proviamo a vedere con il ***più semplice degli esempi*** se a questo punto del corso capiamo fino in fondo che cosa si intende per ***dependency injection***, quali vantaggi produce e che tipo di supporto è necessario per realizzarla

```
public class HelloWorld {  
  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
    }  
}
```

Quali problemi?

Necessità di cambiare il codice (e di ricompilare) per imporre una modifica del messaggio

Codice ***non estensibile e modificabile***



Versione di HelloWorld con Argomenti a Linea di Comando

```
public class HelloWorldWithCommandLine {  
    public static void main(String[] args) {  
        if(args.length > 0) {  
            System.out.println(args[0]);  
        } else { System.out.println("Hello World!"); }  
    }  
}
```

In questo modo si **“esternalizza” il contenuto del messaggio**, che viene letto a runtime dagli argomenti a linea di comando

Problemi?

- ❑ Codice responsabile del rendering del messaggio (`println`) si **occupa anche di ottenere il messaggio**
 - Cambiare come il messaggio deve essere ottenuto obbliga a cambiare il codice del renderer
- ❑ Il renderer **non può essere modificato facilmente** (messaggio verso stderr? O in tag HTML invece che plain text?)



Disaccoppiamento

- 1) Disaccoppiamento dell'**implementazione della logica del message provider** rispetto al resto del codice tramite **creazione di una classe separata**

```
public class HelloWorldMessageProvider {  
  
    public String getMessage() {  
        return "Hello World!";  
    }  
}
```

- 2) Disaccoppiamento **dell'implementazione della logica di message rendering** dal resto del codice

La logica di message rendering è data all'oggetto [HelloWorldMessageProvider](#) da **qualcun altro** – questo è ciò che si intende con Dependency Injection

```
public class StandardOutMessageRenderer {  
    private HelloWorldMessageProvider messageProvider = null;  
    public void render() {  
        if (messageProvider == null) {  
            throw new RuntimeException("You must set the property  
messageProvider of class:" + StandardOutMessageRenderer.class.  
getName());  
        }  
    }  
}
```




Disaccoppiamento

```
// continua
System.out.println(messageProvider.getMessage()); }

// dependency injection tramite metodo setter
public void setMessageProvider>HelloWorldMessageProvider provider)
{   this.messageProvider = provider;
}

public HelloWorldMessageProvider getMessageProvider() {
    return this.messageProvider;
}
}
```



HelloWorld con Disaccoppiamento

```
public class HelloWorldDecoupled {  
  
    public static void main(String[] args) {  
        StandardOutMessageRenderer mr =  
            new StandardOutMessageRenderer();  
        HelloWorldMessageProvider mp =  
            new HelloWorldMessageProvider();  
        mr.setMessageProvider(mp);  
        mr.render();  
    }  
}
```

A questo punto, la logica del message provider e quella del message renderer sono separate dal resto del codice

Quali problemi ancora?

- ❑ Implementazioni specifiche di MessageRenderer e di MessageProvider sono hard-coded nella logica applicativa (in questo launcher in questo lucido)
- ❑ Aumentiamo il disaccoppiamento ***tramite interfacce***



HelloWorld con Disaccoppiamento: Interfacce

```
public interface MessageProvider {  
    public String getMessage(); }
```

```
public class HelloWorldMessageProvider  
    implements MessageProvider {  
    public String getMessage() {  
        return "Hello World!"; }  
}
```

```
public interface MessageRenderer {  
    public void render();  
    public void setMessageProvider(MessageProvider provider);  
    public MessageProvider getMessageProvider();  
}
```



HelloWorld con Disaccoppiamento: Interfacce

```
public class StandardOutMessageRenderer
    implements MessageRenderer {
    // MessageProvider è una interfaccia Java ora
    private MessageProvider messageProvider = null;
    public void render() {
        if (messageProvider == null) {
            throw new RuntimeException( "You must set the property
messageProvider of class:" + StandardOutMessageRenderer.class.
getName());    }
        System.out.println(messageProvider.getMessage()); }
    ...

    public void setMessageProvider(MessageProvider provider) {
        this.messageProvider = provider;    }

    public MessageProvider getMessageProvider() {
        return this.messageProvider;    }
}
```



HelloWorld con Disaccoppiamento: Interfacce

- ❑ Rimane responsabilità del launcher di effettuare la “dependency injection”

```
public class HelloWorldDecoupled {  
  
    public static void main(String[] args) {  
        MessageRenderer mr = new StandardOutMessageRenderer();  
        MessageProvider mp = new HelloWorldMessageProvider();  
        mr.setMessageProvider(mp);  
        mr.render();  
    }  
}
```

- ❑ Ora è possibile modificare la logica di message rendering senza alcun impatto sulla logica di message provider
- ❑ Allo stesso modo, è possibile cambiare la logica di message provider senza bisogno di modificare la logica di message rendering



HelloWorld con Disaccoppiamento: Interfacce

Quali problemi ancora?

- ❑ L'uso di differenti implementazioni delle interfacce `MessageRenderer` o `MessageProvider` necessita comunque di una *modifica (limitata) del codice della logica di business (launcher)*

=> Creare una ***semplice classe factory*** che legga i nomi delle classi desiderate per le implementazioni delle interfacce da un file (*property file*) e le istanzi a runtime, facendo le veci dell'applicazione



HelloWorld con Classe Factory

```
public class MessageSupportFactory {
    private static MessageSupportFactory instance = null;
    private Properties props = null;
    private MessageRenderer renderer = null;
    private MessageProvider provider = null;

    private MessageSupportFactory() {
        props = new Properties();
        try {
            props.load(new FileInputStream("msf.properties"));
            // ottiene i nomi delle classi per le interfacce
            String rendererClass = props.getProperty("renderer.class");
            String providerClass = props.getProperty("provider.class");
            renderer = (MessageRenderer) Class.forName(rendererClass).
                newInstance();
            provider = (MessageProvider) Class.forName(providerClass).
                newInstance();
        } catch (Exception ex) { ex.printStackTrace(); }
    }
}
```



HelloWorld con Classe Factory

```
static { instance = new MessageSupportFactory(); }

public static MessageSupportFactory getInstance() {
    return instance; }

public MessageRenderer getMessageRenderer() {
    return renderer; }

public MessageProvider getMessageProvider() {
    return provider; }
}

public class HelloWorldDecoupledWithFactory {

    public static void main(String[] args) {
        MessageRenderer mr = MessageSupportFactory.getInstance().
            getMessageRenderer();
        MessageProvider mp = MessageSupportFactory.getInstance().
            getMessageProvider();
        mr.setMessageProvider(mp);
        mr.render(); }
}
```




HelloWorld con Classe Factory

File di proprietà

```
# msf.properties  
renderer.class=StandardOutMessageRenderer  
provider.class=HelloWorldMessageProvider
```

- ❑ Ora le implementazioni di message provider e message renderer ***possono essere modificate tramite semplice modifica del file di proprietà***

Quali problemi ancora?

- ❑ Necessità di ***scrivere molto “glue code”*** per mettere insieme l'applicazione
- ❑ Necessità di scrivere una ***classe MessageSupportFactory***
- ❑ L'istanza di ***MessageProvider*** deve essere ancora ***iniettata manualmente*** nell'implementazione di MessageRenderer



HelloWorld usando Spring

```
public class HelloWorldSpring {  
  
    public static void main(String[] args) throws Exception {  
        // ottiene il riferimento a bean factory  
        BeanFactory factory = getBeanFactory();  
        MessageRenderer mr = (MessageRenderer) factory.  
            getBean("renderer");  
        MessageProvider mp = (MessageProvider) factory.  
            getBean("provider");  
        mr.setMessageProvider(mp);  
        mr.render();  
    }  
  
    // continua...
```



HelloWorld usando Spring

```
// Possibilità di scrivere il proprio metodo getBeanFactory()  
// a partire da Spring DefaultListableBeanFactoryclass  
  
private static BeanFactory getBeanFactory() throws Exception {  
    DefaultListableBeanFactory factory = new  
        DefaultListableBeanFactory();  
    // creare un proprio lettore delle definizioni  
    PropertiesBeanDefinitionReader rdr = new  
        PropertiesBeanDefinitionReader(factory);  
    // caricare le opzioni di configurazione  
    Properties props = new Properties();  
    props.load(new FileInputStream("beans.properties"));  
    rdr.registerBeanDefinitions(props);  
    return factory;  
}  
}
```



HelloWorld con Spring: Quali Problemi?

Quali vantaggi già raggiunti in questo modo?

- ❑ Eliminata la necessità di produrre glue code (*MessageSupportFactory*)
- ❑ **Migliore gestione degli errori** e meccanismo di **configurazione completamente disaccoppiato**

Quali problemi ancora?

- ❑ **Il codice di startup deve avere conoscenza delle dipendenze di *MessageRenderer*, deve ottenerle e deve passarle a *MessageRenderer***
 - In questo caso **Spring agirebbe come non più di una classe factory sofisticata**
 - Rimarrebbe al programmatore il compito di fornire il proprio metodo `getBeanFactory()` usando le API di basso livello del framework Spring



HelloWorld con Spring DI

- ❑ Finalmente 😊, utilizzo della Dependency Injection (DI) del framework Spring

File di configurazione

```
#Message renderer
```

```
renderer.class=StandardOutMessageRenderer
```

```
# Chiede a Spring di assegnare l'effettivo provider alla
```

```
# proprietà MessageProvider del bean Message renderer
```

```
renderer.messageProvider(ref)=provider
```

```
#Message provider
```

```
provider.class=HelloWorldMessageProvider
```



HelloWorld con Spring DI

```
public class HelloWorldSpringWithDI {  
    public static void main(String[] args) throws Exception {  
        BeanFactory factory = getBeanFactory();  
        MessageRenderer mr = (MessageRenderer) factory.  
            getBean("renderer");  
        // Nota che non è più necessaria nessuna injection manuale  
        // del message provider al message renderer  
        mr.render();    }  
    private static BeanFactory getBeanFactory() throws Exception {  
        DefaultListableBeanFactory factory = new  
            DefaultListableBeanFactory();  
        PropertiesBeanDefinitionReader rdr = new  
            PropertiesBeanDefinitionReader(factory);  
        Properties props = new Properties();  
        props.load(new FileInputStream("beans.properties"));  
        rdr.registerBeanDefinitions(props);  
        return factory; }  
}
```



HelloWorld con Spring DI: Ultime Osservazioni

- ❑ Il metodo `main()` deve semplicemente ottenere il bean `MessageRenderer` e richiamare `render()`
 - Non deve ottenere prima il `MessageProvider` e configurare la proprietà `MessageProvider` del bean `MessageRenderer`
 - **“wiring” realizzato automaticamente dalla *Dependency Injection di Spring***
- ❑ Nota che non serve ***nessuna modifica*** alle classi da collegare insieme tramite DI
- ❑ Queste classi ***NON fanno alcun riferimento a Spring***
 - ***Nessun bisogno di implementare interfacce*** del framework Spring
 - ***Nessun bisogno di estendere classi*** del framework Spring
- ❑ ***Classi come POJO puri*** che possono essere sottoposte a ***testing senza alcuna dipendenza da Spring***



Spring DI con file XML

- ❑ Più usualmente, dipendenze dei bean sono specificate tramite un file XML

```
<beans>
  <bean id="renderer"
        class="StandardOutMessageRenderer">
    <property name="messageProvider">
      <ref local="provider"/>
    </property>
  </bean>
  <bean id="provider"
        class="HelloWorldMessageProvider"/>
</beans>
```




Spring DI con file XML

```
public class HelloWorldSpringWithDIXMLFile {  
  
    public static void main(String[] args) throws Exception {  
        BeanFactory factory = getBeanFactory();  
        MessageRenderer mr = (MessageRenderer) factory.  
            getBean("renderer");  
        mr.render();  
    }  
  
    private static BeanFactory getBeanFactory() throws Exception {  
        BeanFactory factory = new XmlBeanFactory(new  
            FileSystemResource("beans.xml"));  
        return factory;  
    }  
}
```



Spring DI con file XML: Uso di Costruttore per MessageProvider

```
<beans>
  <bean id="renderer" class="StandardOutMessageRenderer">
    <property name="messageProvider">
      <ref local="provider"/>
    </property>
  </bean>
  <bean id="provider" class="ConfigurableMessageProvider">
    <constructor-arg>
      <value>Questo è il messaggio configurabile</value>
    </constructor-arg>
  </bean>
</beans>
```



Spring DI con file XML: Uso di Costruttore

```
public class ConfigurableMessageProvider
    implements MessageProvider {

    private String message;
    public ConfigurableMessageProvider(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }

}
```



HelloWorld usando Spring AOP

Infine, se volessimo scrivere a video “Hello World!” sfruttando AOP

```
public class MessageWriter implements IMessageWriter{  
  
    public void writeMessage() {  
        System.out.print("World");  
    }  
  
}
```

- ❑ *joinpoint* è l'invocazione del metodo `writeMessage()`
- ❑ ***Necessità di un “around advice”***



HelloWorld usando Spring AOP

- ❑ Gli **advice Spring** sono **scritti in Java** (nessun linguaggio AOP-specific)
- ❑ Pointcut tipicamente specificati in file XML di configurazione
- ❑ Spring supporta **solo joinpoint a livello di metodo** (ad esempio, impossibile associare advice alla modifica di un campo di un oggetto)

```
public class MessageDecorator implements MethodInterceptor {  
  
    public Object invoke(MethodInvocation invocation)  
                        throws Throwable {  
        System.out.print("Hello ");  
        Object retVal = invocation.proceed();  
        System.out.println("!");  
        return retVal;  
    }  
}
```



HelloWorld usando Spring AOP

- ❑ Uso della classe *ProxyFactory* per creare il proxy dell'oggetto target
- ❑ Anche modalità più di base, ***tramite uso di possibilità predeterminate*** e file XML, ***senza istanziare uno specifico proxy per AOP***

```
public static void main(String[] args) {  
    MessageWriter target = new MessageWriter();  
    ProxyFactory pf = new ProxyFactory();  
    // aggiunge advice alla coda della catena dell'advice  
    pf.addAdvice(new MessageDecorator());  
    // configura l'oggetto dato come target  
    pf.setTarget(target);  
    // crea un nuovo proxy in accordo con le configurazioni  
    // della factory  
    MessageWriter proxy = (MessageWriter) pf.getProxy();  
    proxy.writeMessage();  
    // Come farei invece a supportare lo stesso comportamento  
    // con chiamata diretta al metodo dell'oggetto target?  
    ... } }
```



Intercettori Spring

Guarda caso ☺, anche in **Spring possiamo definire intercettori**, ma questa volta in modo molto diverso e meno trasparente che non in modello container pesante, **sfruttando i concetti di AOP e gli oggetti con proxy**

- ❑ Intercettore Spring può eseguire immediatamente prima o dopo l'invocazione della richiesta corrispondente
- ❑ Implementa l'interfaccia [HandlerInterceptor](#) o estende [HandlerInterceptorAdaptor](#)

```
public class MyService {
    public void doSomething() {
        for (int i = 1; i < 10000; i++) {
            System.out.println("i=" + i); } }
}

public class ServiceMethodInterceptor implements MethodInterceptor {
    public Object invoke(MethodInvocation methodInvocation) throws Throwable {
        long startTime = System.currentTimeMillis();
        Object result = methodInvocation.proceed();
        long duration = System.currentTimeMillis() - startTime;
```



Intercettori Spring

```
Method method = methodInvocation.getMethod();
String methodName = method.getDeclaringClass().getName() + "." +
    method.getName();
System.out.println("Method '" + methodName + "' took " + duration
    + " milliseconds to run");
return null; }
}
```

```
<beans>
  <bean id="myService" class="com.test.MyService"> </bean>
  <bean id="interceptor" class="com.test.ServiceMethodInterceptor">
    </bean>
  <bean id="interceptedService"
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target">
      <ref bean="myService"/> </property>
    <property name="interceptorNames">
      <list> <value>interceptor</value> </list> </property>
    </bean> </beans>
```




Tipi di Transazionalità verso DB in Spring (e non solo...)

Discorso generale su transazioni e DB. Con scarsa sorpresa in Spring:

- ❑ **Transazione locale**

- Specifica per una singola risorsa transazionale, ad esempio unica risorsa JDBC

- ❑ **Transazione globale**

- **Gestita dal container**
- Può includere **risorse transazionali multiple e distribuite**

- ❑ Il programmatore può **specificare in modo dichiarativo** che un **metodo di bean deve avere proprietà transazionali**

- ❑ Anche ***l'implementazione delle transazioni è basata su AOP***

- Intercettazione di chiamate a metodi per gestione transazioni

- ❑ Nessuna necessità di modificare la logica di business, né al cambio della transazionalità desiderata né al cambio del provider di transazionalità



Livelli di Isolamento delle Transazioni

- ❑ **ISOLATION_DEFAULT**
- ❑ **ISOLATION_READ_UNCOMMITTED**
 - Possono accadere *letture “sporche” (dirty read), non ripetibili e fantasma (phantom read)*
- ❑ **ISOLATION_READ_COMMITTED**
 - Letture sporche rese impossibili; possibilità di accadimento di *letture non ripetibili e fantasma*
- ❑ **ISOLATION_REPEATABLE_READ**
 - Possibilità delle sole *letture fantasma*; dirty e non-repeatable rese non possibili
- ❑ **ISOLATION_SERIALIZABLE**
 - Tutte le possibilità “spiacevoli” sopra per la lettura sono rese impossibili



Parentesi su Letture Phantom/Non-Repeatable/Dirty

Phantom read

Quando, nel corso di una transazione, vengono eseguite **due query identiche e i risultati restituiti dalla seconda query sono differenti da quelli per la prima**. Causa: “**lock di range**” non **acquisiti** (solo acquisizione di read lock) in fase di SELECT

Transaction 1

```
/* Query 1 */
```

```
SELECT * FROM users
```

```
WHERE age BETWEEN 10 AND 30;
```

```
/* più tardi, dopo l'esecuzione di Query 2 */
```

```
SELECT * FROM users
```

```
WHERE age BETWEEN 10 AND 30;
```

Transaction 2

```
/* Query 2 */
```

```
INSERT INTO users VALUES ( 3, 'Bob', 27 );
```

```
COMMIT;
```



Parentesi su Letture Phantom/Non-Repeatable/Dirty

Non-repeatable read

In soluzioni di controllo della concorrenza basate su lock, ***letture non-repeatable possono avvenire quando i lock in lettura non sono acquisiti*** durante una SELECT. In soluzioni di controllo della concorrenza multiversion, letture non-repeatable possono avvenire quando si ***rilassa il vincolo che una transazione con conflitto di commit debba effettuare rollback***

Transaction 1

```
/* Query 1 */  
SELECT * FROM users WHERE id = 1;  
/* Dopo Query 2 e suo commit parziale */  
SELECT * FROM users WHERE id = 1;
```

Transaction 2

```
/* Query 2 */  
UPDATE users SET age = 21 WHERE id = 1;  
COMMIT; /* ad es. per read-committed isolation: read lock rilasciati prima  
        della fine della transazione totale */  
COMMIT; /* fine della transazione totale, anche write lock rilasciati */
```



Parentesi su Letture Phantom/Non-Repeatable/Dirty

Dirty read

Quando una transazione legge ***dati che sono stati modificati da un'altra transazione non ancora committed***. Letture dirty sono simili a letture non-repeatable, ma la seconda transazione non necessita di commitment per la prima query per restituire un risultato diverso

Transaction 1

`/* Query 1 */`

`SELECT * FROM users WHERE id = 1;`

`/* Dopo Query 2 */`

`SELECT * FROM users WHERE id = 1;`

Transaction 2

`/* Query 2 */`

`UPDATE users SET age = 21 WHERE id = 1;`

`/* Nessun commit immediato, solo alla fine della transazione */`

`COMMIT;`



Invece Vecchia Conoscenza: Propagazione delle Transazioni in Spring

- ❑ **PROPAGATION_REQUIRED**
 - Supporto alla propagazione della transazione di partenza; crea una nuova transazione se non era transazionale il contesto di partenza
- ❑ **PROPAGATION_SUPPORTS**
 - Supporto alla propagazione della transazione di partenza; esegue non transazionalmente se la partenza non era transazionale
- ❑ **PROPAGATION_MANDATORY**
 - Supporto alla propagazione della transazione di partenza; lancia un'eccezione se la partenza non era transazionale
- ❑ **PROPAGATION_REQUIRES_NEW**
 - Crea una nuova transazione, sospendendo quella di partenza, se esistente
- ❑ **PROPAGATION_NOT_SUPPORTED**
- ❑ **PROPAGATION_NEVER**
- ❑ **PROPAGATION_NESTED**



Spring: qualche Considerazione Avanzata (1)

Abbiamo visto che alla base dell'architettura Spring c'è l'idea di ***Inversion of Control*** (prima che in EJB3.0!) e di ***factory leggera*** per l'istanziamento, il ritrovamento e la gestione delle relazioni fra oggetti

Factory supportano due modalità di oggetto:

- ❑ ***Singleton*** (default) – ***unica istanza condivisa*** dell'oggetto con nome specificato, ***ideale per oggetti stateless***. Riduce la proliferazione di singleton nel codice applicativo
- ❑ ***Prototype*** – ogni operazione di ritrovamento di un oggetto produrrà la ***creazione di una nuova istanza***. Utile per far avere ad ogni invocante una istanza distinta

Un bean Spring può essere anche un **FactoryBean** (implementazione dell'interfaccia corrispondente)

- ***Aggiunge un livello di indirettezza***
- Usato di solito per ***creare oggetti con proxy*** utilizzando ad es. AOP (concettualmente simile a interception in EJB, ma di più semplice utilizzo)



Spring: qualche Considerazione Avanzata (2)

- ❑ La possibilità di semplice Dependency Injection tramite costruttori o metodi ***semplifica il testing delle applicazioni Spring***
 - Per esempio è semplice scrivere un test JUnit che crea l'oggetto Spring e configura le sue proprietà a fini di testing
- ❑ Il ***container IoC non è invasivo***: molti oggetti di business non dipendono dalle API di invocazione del container => possono ***essere portabili verso altre implementazioni di container*** (*PicoContainer, HiveMind, ...*) ed è facile “introdurre” vecchi POJO in ambiente Spring
- ❑ ***Factory Spring sono leggere***: anche implementazioni all'interno di singole applet o come applicazioni Swing standalone
- ❑ Secondario, ma anche successo per ***unchecked runtime exception***



Inoltre, Autowiring

- ❑ Spring può occuparsi **automaticamente di risolvere dipendenze tramite introspezione delle classi** bean. In questo modo il programmatore non si deve preoccupare di specificare esplicitamente le proprietà del bean o gli argomenti del costruttore
 - Ovvero, **non necessario l'utilizzo di <ref>**
- ❑ Le proprietà del bean sono “autowired” attraverso matching basato su nome o su tipo
 - *autowire=“name” (configurazione di default)*
 - Autowiring fatto sui **nomi delle proprietà** (metodi di nome `set<Property-name>()` del bean)
 - *autowire=“type”*
 - Autowiring fatto sui **tipi di proprietà** del bean (`set<Property-name>(ArgumentType arg)`)
 - *autowire=“constructor”*
 - Match fatto sui **tipi degli argomenti del costruttore**
 - ...

Vi ricorda qualcosa?



ApplicationContext

In realtà, per accedere ad ***alcune funzionalità avanzate di Spring***, non è sufficiente l'uso della semplice interfaccia **BeanFactory** => **ApplicationContext** è ***l'estensione*** dell'interfaccia **BeanFactory**

- Fornisce tutte le funzionalità base + gestione delle transazioni e di AOP, ad esempio (non supportate dalla BeanFactory di base)
- ❑ **ApplicationContext** si utilizza in ***modalità più “tradizionale” e framework-oriented***
- ❑ Funzionalità aggiuntive di **ApplicationContext**
 - Interfacce per la ***gestione del ciclo di vita***
 - ***Propagazione di eventi*** a bean che implementano l'interfaccia **ApplicationListener**
 - Accesso a risorse come URL e file
 - ...



Gestione del Ciclo di Vita

La gestione del ciclo di vita si basa su di un accordo sull'implementazione di **interfacce standardizzate**.
Ad esempio:

- ❑ Interfaccia [ApplicationContextAware](#)
 - Un bean che implementa questa interfaccia avrà il suo metodo di interfaccia [setApplicationContext\(\)](#) automaticamente invocato alla creazione del bean stesso. Riceverà così **un riferimento al contesto** su cui poter effettuare invocazioni nel seguito

```
public class Publisher implements ApplicationContextAware {  
    private ApplicationContext ctx;  
  
    // Questo metodo sarà automaticamente invocato da IoC container  
    public void setApplicationContext(  
        ApplicationContext applicationContext)  
        throws BeansException {  
        this.ctx = applicationContext;  
    }  
}
```



Possibilità di Effettuare Dependency Checking

- ❑ Utilizzabile per **controllare l'esistenza di dipendenze non risolte** quando abbiamo già fatto il deployment di un bean all'interno di un container Spring
 - Proprietà che **non hanno valori configurati all'interno della definizione del bean**, per i quali anche autowiring non ha prodotto alcun setting
- ❑ Caratteristica utile quando ci si vuole assicurare che **tutte le proprietà** (o tutte le proprietà di un determinato tipo) **siano state configurate** su un bean

Modalità possibili:

- **none** – nessun check
- **simple** – dependency checking effettuato solo per **tipi primitivi e collection**
- **object** – dependency checking effettuato solo per **altri bean associati** all'interno della stessa factory (*collaborator*)
- **all** – dependency checking effettuato per collaborator, tipi primitivi e collection



Propagazione di Eventi

La **gestione degli eventi** in `ApplicationContext` è realizzata tramite la classe `ApplicationEvent` e l'interfaccia `ApplicationListener`

- ❑ Se un bean implementa l'interfaccia `ApplicationListener` e ne è fatto il deployment in un `ApplicationContext ac1`, quel bean **viene notificato** ogni volta che un `ApplicationEvent` viene pubblicato in `ac1`
- ❑ Essenzialmente, il ***solito design pattern Observer***

Tre tipologie di eventi built-in:

- ❑ `ContextRefreshEvent`
 - Inizializzazione o refresh di `ApplicationContext`
- ❑ `ContextClosedEvent`
 - Chiusura di `ApplicationContext`
- ❑ `RequestHandleEvent`
 - Evento specifico per il Web – una richiesta HTTP è stata appena servita



Esempio di Gestione Eventi

Ad esempio, configurazione in [ApplicationContext.xml](#) del comportamento “ad ogni ricezione di un email da un indirizzo in black list, invia un email di notifica a spam@list.org”

```
<bean id="emailer" class="example.EmailBean">  
  <property name="blackList">  
    <list>  
      <value>black@list.org</value>  
      <value>white@list.org</value>  
      <value>john@doe.org</value>  
    </list>  
  </property> </bean>
```

```
<bean id="blackListListener"  
  class="example.BlackListNotifier">  
  <property name="notificationAddress"  
    value="spam@list.org"/>  
</bean>
```



Esempio di Gestione Eventi

Classe bean che pubblica eventi tramite l'oggetto [ApplicationContext](#)

```
public class EmailBean implements ApplicationContextAware {  
    private List blackList;  
    public void setBlackList(List blackList) {  
        this.blackList = blackList;  
    }  
    public void setApplicationContext(ApplicationContext ctx) {  
        this.ctx = ctx;  
    }  
    public void sendEmail(String address, String text) {  
        if (blackList.contains(address)) {  
            BlackListEvent evt = new BlackListEvent(address, text);  
            ctx.publishEvent(evt);  
            return;  
        }  
    }  
}
```



Esempio di Gestione Eventi

Classe Notifier, che riceve le notifiche degli eventi generati

```
public class BlackListNotifier implement ApplicationListener {  
  
    private String notificationAddress;  
  
    public void setNotificationAddress(String notificationAddress)  
    {  
        this.notificationAddress = notificationAddress;  
    }  
  
    public void onApplicationEvent(ApplicationEvent evt) {  
        if (evt instanceof BlackListEvent) {  
            // invio dell'email di notifica all'indirizzo appropriato  
            }  
        }  
    }  
}
```