

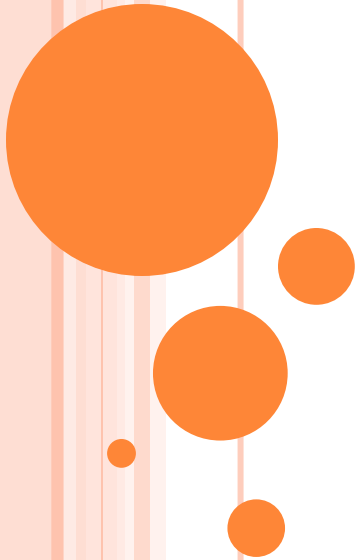


Spring

Enterprise Java Framework



spring



spring
source

SPRING : Historique

- Rod Johnson (2004)
- Faire avec un simple bean (POJO)
ce qui était seulement possible avec EJBs
- Environnement J2SE et J2EE
- Simplicité
- Testabilité
- Couplage faible. Programmation par interfaces



Documentation de référence:

<http://docs.spring.io/spring/docs/2.5.x/reference/index.html>



Problématiques des développements Java EE

- ✓ Mauvaise séparation des préoccupations. problématiques d'ordres différents (technique et métier) sont mal isolées
- ✓ Complexité: un serveur d'applications monolithique, implémentant la totalité de la spécification Java EE, dont à peine 20 % sont nécessaires pour développer la plupart des applications.
- ✓ Mauvaise interopérabilité: ex: une application Web pourra rarement passer d'un serveur d'applications à un autre sans réglage supplémentaire
- ✓ Mauvaise testabilité : les applications dépendent fortement de l'infrastructure d'exécution

- Spring a su comprendre les problèmes liés à J2EE .
- Spring propose un modèle de programmation plus adapté et plus productif
- Spring a comblé certaines lacunes de Java EE et a contribué à son amélioration



Les réponses de SPRING

➤ La notion de conteneur léger

- ✓ Opposé à un conteneur EJB jugé lourd technologiquement .
- ✓ Gère une application *via un* ensemble de composants [des objets présentant une interface].
- ✓ Gère le cycle de vie des composants (création, destruction) et leurs interdépendances .
- ✓ Permet la portabilité des applications [indépendantes du serveur d'applications]
- ✓ Approche par composants.
- ✓ Programmation par interface à faible couplage.
- ✓ Meilleure évolutivité et testabilité des applications.

Le cœur de Spring est un conteneur léger : le plus complet sur le marché.

Les réponses de SPRING

➤ La programmation orientée aspect [PAO]

- ✓ Paradigme de programmation permettant de modulariser des éléments logiciels en complément de la POO.
- ✓ Se concentre sur les éléments transversaux [éléments dupliqués dans de nombreuses parties d'une application, sans pouvoir être centralisés]
- ✓ Éléments transversaux : la gestion des transactions, la journalisation ou la sécurité
- ✓ Améliore la séparation des préoccupations dans une application
- ✓ Ajouter du comportement (décoration) de façon complètement transparente.
- ✓ N'impose aucune contrainte et permet d'ajouter très facilement du comportement à n'importe quel type d'objet.



Les réponses de SPRING

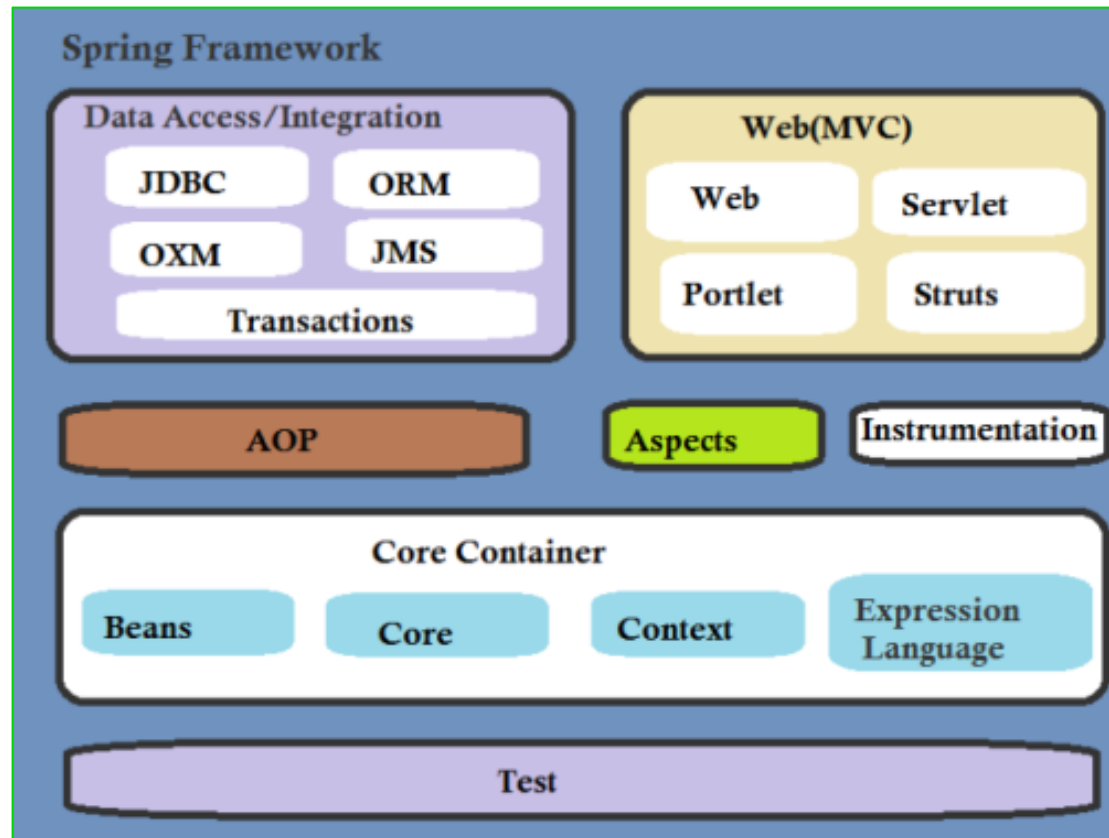
➤ L'intégration de Frameworks tiers

- ✓ Spring intègre un grand nombre de frameworks et de standards Java EE [JSF, Struts, Hibernate ...]
- ✓ Utilisation des l'API native des frameworks. Il est « non intrusif ».



L'écosystème SPRING

Modules
de Spring



- **Core** : le noyau. **classes utilisées** du framework et du conteneur léger.
- **AOP** : programmation orientée aspect. s'intègre avec AspectJ.
- **Data Access** : accès aux dépôts de données. Implémentation pour JDBC.
- **ORM** : mapping objet-relationnel [Hibernate - JPA - EclipseLink - Batis.]
- **Web** : support pour les applications Web [Spring Web MVC]
- **Test** : tests unitaires [JUnit - TestNG]

Les projets du portfolio

[projets gravitant autour de Spring]

Spring Web Flow : gestion des enchaînements de pages complexes dans une application.

Spring Web Services : support pour les services Web

Spring Security : authentification et autorisation dans une application Web.

Spring Dynamic Modules : intégration des applications d'entreprise sur la plateforme OSGi

Spring Batch : traitements de type batch, manipulant de grands volumes de données.



Spring Integration : « *Enterprise Integration Patterns* ». interagir avec des systèmes externes.

Spring LDAP : gestion des accès aux données de pour les annuaires LDAP.

Spring IDE : plug-ins Eclipse facilitant l'édition de configurations Spring.

Spring Modules : intégration dans de différents outils et frameworks.

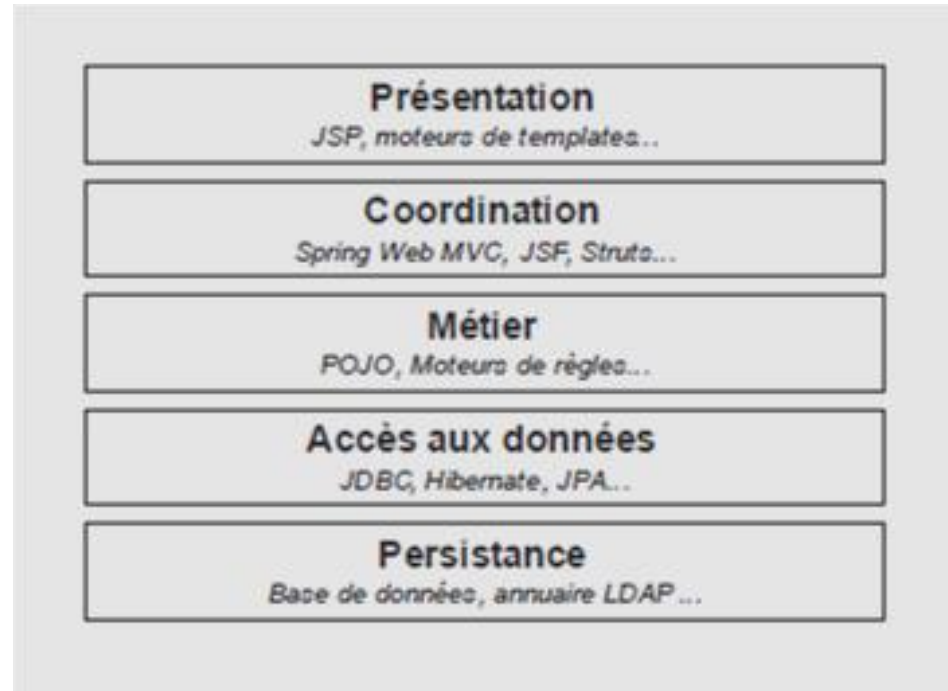
Spring JavaConfig : Implémentation d'une approche Java pour la configuration de contextes Spring



Notions d'architecture logicielle

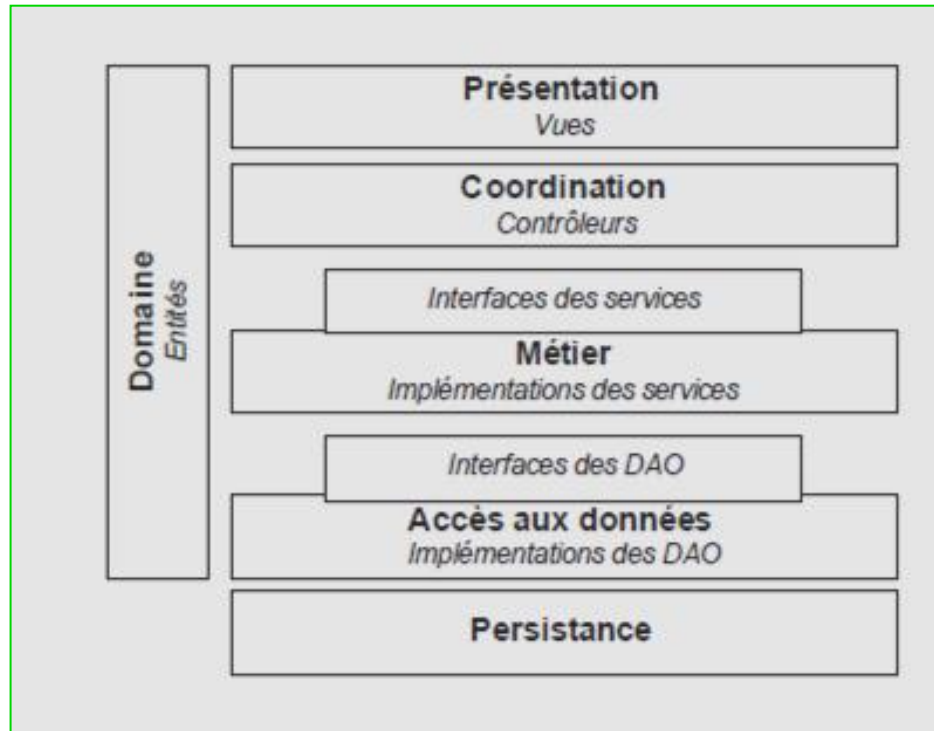
➤ Couches logicielles

Modèle à cinq couches et technologies associées



Nom	Fonction	Composant
Présentation	Gestion du code de présentation de l'interface utilisateur	Vue
Coordination	Analyse des requêtes utilisateur, orchestration des appels métier, gestion de la cinématique	Contrôleur
Métier	Implémentation des règles fonctionnelles d'une application	Service
Accès aux données	Interaction avec l'entrepôt de données (récupération, mise à jour, etc.)	Data Access Object (DAO)
Persistence	Stockage des données	Entrepôt de données

➤ La programmation par interface



✓ La communication entre couches :

- Une couche ne connaît que la couche inférieure et ne doit jamais faire référence à la couche supérieure (elle ignore complètement son existence)
- la communication entre couches s'effectue en établissant des contrats *via des interfaces (Java)*
- *La* notion de dépendance: instructions d'import de packages.



✓ La programmation par interface:

- ➔ Découpler un composant de l'implémentation d'un autre composant.
- ➔ L'interface définit le contrat de communication, l'implémentation du composant le remplit, l'autre composant pourra l'utiliser.
- ➔ but : réduire le couplage entre les composants.



➤ L'inversion de contrôle : IoC (Inversion of Control)

Origine du concept:

- ✓ fait référence au flot d'exécution d'une application
- ✓ programmation procédural ➔ contrôle total du flot d'exécution du programme (*instructions, conditions et boucles*)
- ✓ *Frameworks* ➔ pas de maîtrise de bout en bout par l'application du flot d'exécution.
 - ➔ prise en charge de l'essentiel du flot d'exécution par les frameworks. L'application s'insère dans le cadre du fonctionnement des frameworks.
 - ➔ inversion du contrôle du flot d'exécution

L'inversion de contrôle dans les conteneurs légers

version spécialisée: résoudre les problématiques d'instanciation et de dépendances entre les composants d'une application.



➤ *L'injection de dépendances*

- ✓ S'appuie sur le conteneur léger qui gère le cycle de vie des composants d'une application, ainsi que leurs dépendances, en les injectant de manière appropriée.
- ✓ 2 formes d'injections en SPRING:
 - Injection par constructeur (*un objet se voit injecter ses dépendances au moment où il est créé, c'est-à dire via les arguments de son constructeur*)
 - injection par modificateurs (*un objet est créé, puis ses dépendances lui sont injectées par les modificateurs correspondants*).
- ✓ Référentiel de description des dépendances en XML.



Les fondations de Spring

deux piliers

```
graph TD; A[Les fondations de Spring] --> B[deux piliers]; B --> C[Le conteneur léger]; B --> D[Le framework de POA];
```

The diagram illustrates the foundational components of Spring. It starts with the title 'Les fondations de Spring' at the top. An orange arrow points down to the text 'deux piliers'. From this central text, two red arrows branch out to two separate boxes: 'Le conteneur léger' on the left and 'Le framework de POA' on the right. Both boxes have a green border.

Le conteneur léger

Le framework de POA

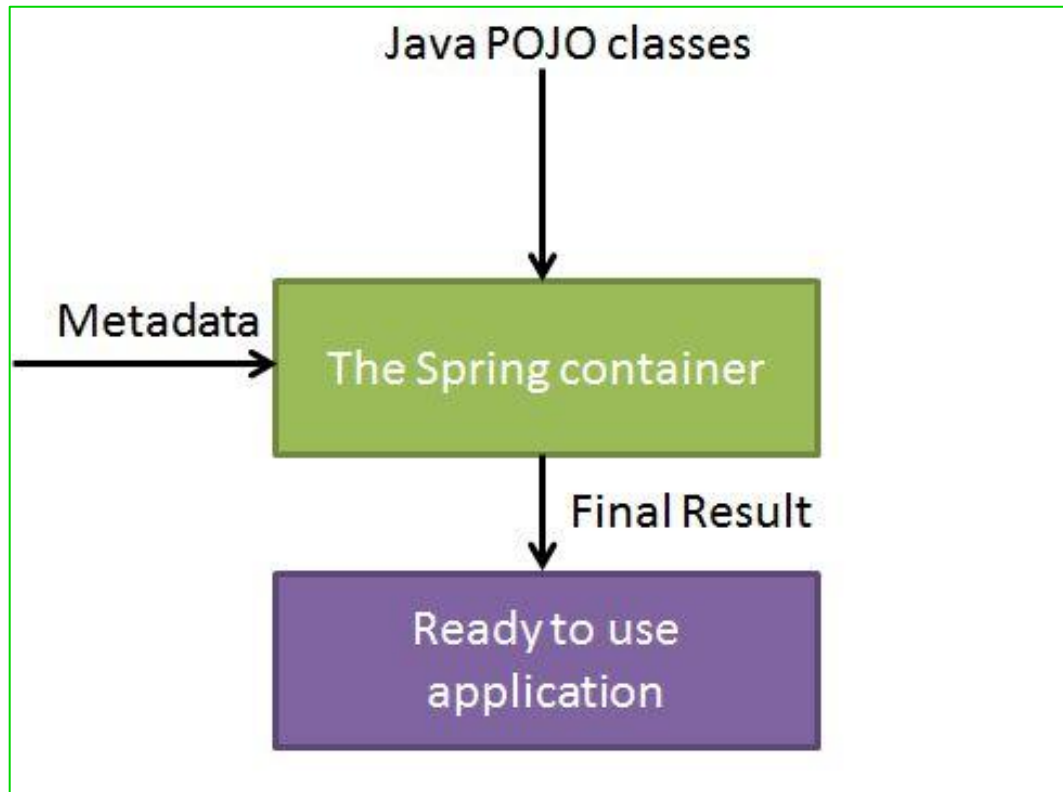


Le conteneur léger de Spring



THE FIRST STEP





➤ Le conteneur léger de Spring:

- ✓ simple fabrique d'objets Java.
- ✓ *fonctionnement de base* : définir des objets dans un fichier XML. Spring charge ce fichier pour gérer l'instanciation des objets.

➤ « Context » :

- ✓ L'objet Spring contenant les objets décrits dans le fichier XML.
- ✓ Le contexte propose un ensemble de méthodes pour récupérer les objets qu'il contient.



Instanciation du conteneur léger de Spring

Définition d'un Bean dans le conteneur léger de Spring.

Voici le fichier de configuration XML correspondant :

```
<?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/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">← ❶

  <bean id="user" class="tudu.domain.model.User">← ❷
    <property name="firstName" value="Frédéric" />← ❸
    <property name="lastName" value="Chopin" />← ❹
  </bean>

</beans>
```

- (1) déclaration du schéma utilisé [les balises utilisables].
utilisation du schéma **beans**, qui permet de définir des objets Java ainsi que leurs propriétés, avec des définitions explicites
- (2) Définition du Bean (balise bean) avec un identifiant (attribut id) au sein du contexte Spring et sa classe (attribut class).
- (3) (4) assignation de deux propriétés

Le fichier de configuration XML est ensuite être chargé en faisant appel aux différentes classes disponibles dans Spring :

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.
    FileSystemXmlApplicationContext;

import tudu.domain.model.User;

public class StartSpring {

    public static void main(String[] args) {

        ApplicationContext context = new
            FileSystemXmlApplicationContext(
                "applicationContext.xml"
            ); ← ❶

        User user = (User) context.getBean("user"); ← ❷

        System.out.println(
            "Utilisateur : "+user.getFirstName()+" "+user.getLastName() ← ❸
        );

    }

}
```

applicationContext.xml ➔ nom du fichier XML

FileSystemXmlApplicationContext ➔ localise le fichier sur le système de fichiers

ClassPathXmlApplicationContext ➔ localise le fichier dans le classpath



Le contexte d'application de Spring

contexte d'application de Spring → l'interface [**ApplicationContext**]

(le type de conteneur léger de SPRING le plus puissant)

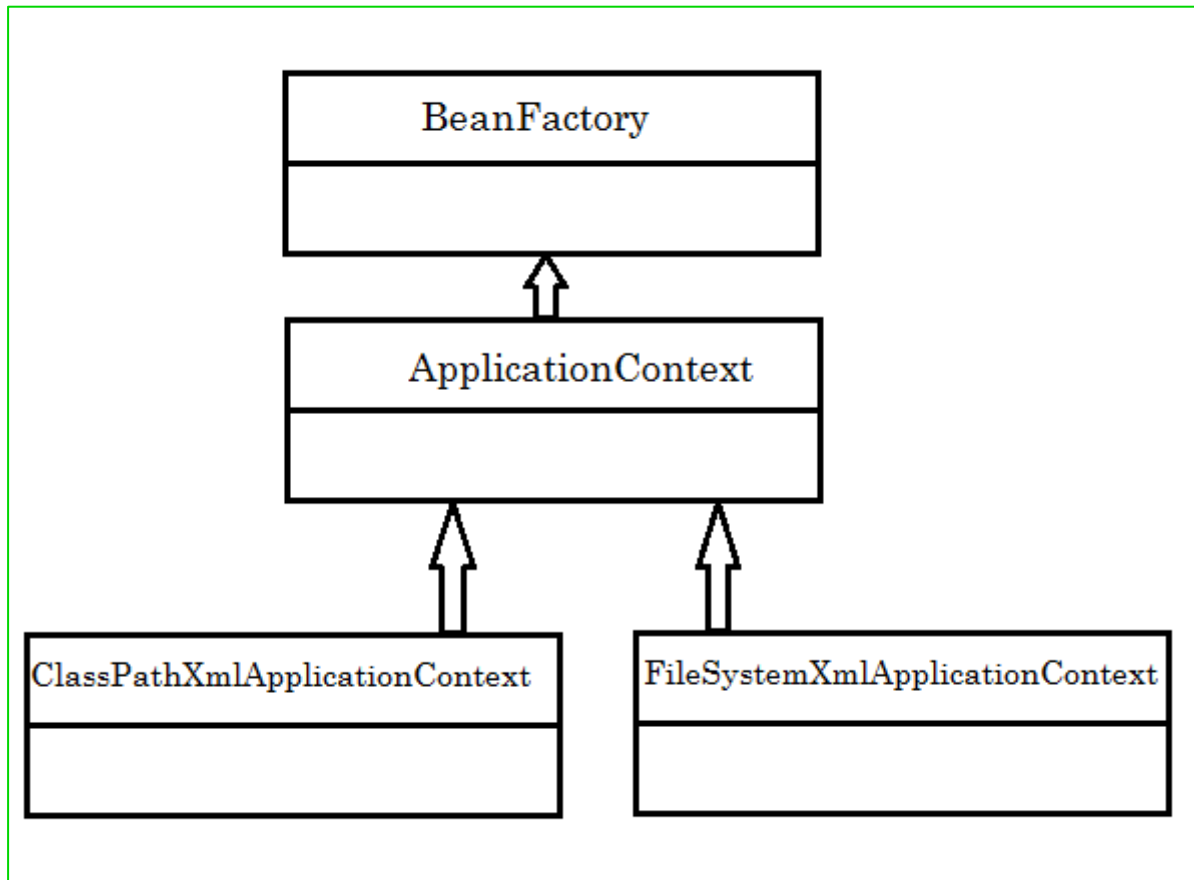
quelques méthodes de l'interface :

```
Object getBean(String name) throws BeansException;  
Object getBean(String name, Class requiredType)  
throws BeansException;  
boolean containsBean(String name);  
boolean isSingleton(String name)  
throws NoSuchBeanDefinitionException;  
Class getType(String name) throws NoSuchBeanDefinitionException;  
String[] getAliases(String name)  
throws NoSuchBeanDefinitionException;
```

EX : [**getBean()**] récupère l'instance d'un objet à partir de son nom

[**ApplicationContext**] hérite de l'interface [**BeanFactory**]





la **BeanFactory** : correspond véritablement à la notion de fabrique de Beans.

les applications d'entreprise utilisent [**ApplicationContext**] → propose des fonctionnalités plus puissantes pour la gestion des Beans.

Définition d'un Bean

- configuration XML
- utilisation des schémas XML (dictent les balises utilisables dans un fichier de config)

les schémas XML disponibles dans Spring 2.5

Nom	Description
beans	Définition des Beans et de leurs dépendances
aop	Gestion de la programmation orientée aspect
context	Activation des annotations et positionnement de post-processeurs
util	Déclaration de constantes et de structures de données
jee	Fonctions pour s'interfacer avec JNDI et les EJB
jms	Configuration de Beans JMS
lang	Déclaration de Beans définis avec des langages de script, tels que JRuby ou Groovy
p	Définition des propriétés de Beans
tx	Déclarations des transactions sur des Beans

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:jee="http://www.springframework.org/schema/jee"
  xmlns:jms="http://www.springframework.org/schema/jms"
```



Nommage des Beans

```
<bean id="user" class="tudu.domain.model.User">
```

l'attribut [**name**] permet de définir des alias :

```
<bean id="user" class="tudu.domain.model.User"  
name="tuduListsUser,pianist">
```



Les méthodes d'injection

la puissance de Spring → l'injection de dépendances.

[injection de dépendances == initialisation des propriétés d'un Bean]

- ✓ Propriétés simples : [entier, réel, chaîne de caractères]
- ✓ Référence à d'autres Beans : [*collaborateurs*]

L'injection par modificateur

```
<bean id="user" class="tudu.domain.model.User">  
  <property name="firstName" value="Frédéric" />  
  <property name="lastName" value="Chopin" />  
</bean>
```

```
<bean id="user" class="tudu.domain.model.User">  
  <property name="firstName"><value>Frédéric</value></property>  
  <property name="lastName"><value>Chopin</value></property>  
</bean>
```



L'injection par constructeur

```
public class UnBean {  
  
    private String chaine;  
    private Integer entier;  
  
    public UnBean(String chaine, Integer entier) {  
        super();  
        this.chaine = chaine;  
        this.entier = entier;  
    }  
}
```

```
<bean id="monBean" class="UnBean">  
    <constructor-arg value="chaine" />  
    <constructor-arg value="10" />  
</bean>
```

indexation (à partir de 0)

```
<bean id="monBean" class="UnBean">  
    <constructor-arg value="10" index="1" />  
    <constructor-arg value="chaine" index="0" />  
</bean>
```

lever l'ambiguïté avec [type]

```
<bean id="monBean" class="UnBean">  
    <constructor-arg value="10" type="java.lang.Integer" />  
</bean>
```

```
<bean id="monBean" class="UnBean">  
    <constructor-arg value="chaine" type="java.lang.String" />  
</bean>
```

Injection des propriétés

Injection de valeurs simples

types de propriétés supportés :

- booléens
- type char et java.lang.Character ;
- type java.util.Properties ;
- type java.util.Locale ;
- type java.net.URL ;
- type java.io.File ;
- type java.lang.Class ;
- tableaux de bytes (chaîne de caractères transformée *via la méthode
getBytes de String*) ;
- tableaux de chaînes de caractères (chaînes séparées par une virgule,
selon le format CSV).



```

public class UnBean {
    private String chaine;
    private int entier;
    private float reel;
    private boolean booleen;
    private char caractere;
    private java.util.Properties proprietes;
    private java.util.Locale localisation;
    private java.net.URL url;
    private java.io.File fichier;
    private java.lang.Class classe;
    private byte[] tab2bytes;
    private String[] tab2chaines;

    // définition des accesseurs et modificateurs de chaque attribut
    (...)
}

```

```

<bean id="monBean" class="UnBean">
  <property name="chaine" value="valeur" />
  <property name="entier" value="10" />
  <property name="reel" value="10.5" />
  <property name="booleen" value="true" />
  <property name="caractere" value="a" />
  <property name="proprietes">
    <value>
      log4j.rootLogger=DEBUG,CONSOLE
      log4j.logger.tudu=WARN
    </value>

```

```

  </property>
  <property name="localisation" value="fr_FR" />
  <property name="url" value="http://tudu.sf.net" />
  <property name="fichier" value="file:c:\\temp\\test.txt" />
  <property name="classe" value="java.lang.String" />
  <property name="tab2bytes" value="valeur" />
  <property name="tab2chaines" value="valeur1,valeur2" />
</bean>

```



Injection de la valeur null

```
<bean id="monBean" class="UnBean">  
  <constructor-arg><null /></constructor-arg>  
  <constructor-arg value="10" />  
</bean>
```

Injection de structures de données

Le type java.Util.Map

assigner une Map à la propriété d'un Bean (balises map et entry) :

```
<property name="map">  
  <map>  
    <entry key="cle1" value="valeur1"/>  
    <entry key="cle2" value="valeur2"/>  
  </map>  
</property>
```



définir explicitement une Map en tant que Bean :

```
<util:map id="mapBean">  
  <entry key="cle1" value="valeur1"/>  
  <entry key="cle2" value="valeur2"/>  
</util:map>
```

Nous pouvons préciser la classe d'implémentation avec l'attribut map-class :

```
<util:map id="mapBean" map-class="java.util.HashMap">
```



Le type java.util.Set

```
<property name="set">
  <set>
    <value>valeur1</value>
    <value>valeur2</value>
  </set>
</property>
```

```
<util:set id="setBean">
  <value>valeur1</value>
  <value>valeur2</value>
</util:set>
```

```
<util:set id="setBean" set-class="java.util.HashSet">
  <value>valeur1</value>
  <value>valeur2</value>
</util:set>
```



Le type java.util.List

```
<property name="list">
  <list>
    <value>valeur1</value>
    <value>valeur2</value>
  </list>
</property>
```

```
<util:list id="listBean">
  <value>valeur1</value>
  <value>valeur2</value>
</util:list>
```

```
<util:list id="listBean" list-class="java.util.ArrayList">
```

Le type java.util.Properties

```
<property name="props">
  <props>
    <prop key="cle1">valeur1</prop>
    <prop key="cle2">valeur2</prop>
  </props>
</property>
```

```
<util:properties id="propsBean">
  <prop key="cle1">valeur1</prop>
  <prop key="cle2">valeur2</prop>
</util:properties>
```

Injection des collaborateurs

Injection explicite des collaborateurs

Ex : injecter une implémentation d'un DAO dans un service métier :

```
<bean id="userDAO" class="tudu.domain.dao.jpa.UserDAOJpa" />← ❶  
  
<bean id="userManager"  
  class="tudu.service.impl.UserManagerImpl"← ❷  
  <property name="userDAO" ref="userDAO" />← ❸  
</bean>
```

- (1) déclaration du DAO (),
- (2) déclaration du service ().
- (3) Le DAO est injecté *via la balise property*. l'attribut *ref* référence le DAO

injection unique

```
<bean id="userManager" class="tudu.service.impl.UserManagerImpl">  
  <property name="userDAO">  
    <bean class="tudu.domain.dao.jpa.UserDAOJpa" />  
  </property>  
</bean>
```



Injection automatique des collaborateurs

réduction de lignes de configuration ➔ injection automatique : [*autowiring*]

modes d'autowiring proposés par Spring :

Mode	Description
no	Aucun autowiring n'est effectué, et les dépendances sont assignées explicitement. Il s'agit du mode par défaut.
byName	Spring recherche un Bean ayant le même nom que la propriété pour réaliser l'injection.
byType	Spring recherche un Bean ayant le même type que la propriété pour réaliser l'injection. Si plusieurs Beans peuvent convenir, une exception est lancée. Si aucun Bean ne convient, la propriété est initialisée à null.
constructor	Similaire à byType, mais fondé sur les types des paramètres du ou des constructeurs
autodetect	Choisit d'abord l'injection par constructeur. Si un constructeur par défaut est trouvé (c'est-à-dire un constructeur sans argument), passe à l'injection automatique par type.

deux moyens d'activer l'**autowiring** dans Spring :

- ✓ La configuration XML
- ✓ Les annotations



Injection automatique en XML

- ✓ Activation Bean par Bean de l'autowiring avec l'attribut [autowire] de la balise [bean] → utilisation de ce mode d'injection de manière ciblée

Ex: injection automatique d'un DAO dans un service métier

```
<bean id="userManager" class="tudu.service.impl.UserManagerImpl"  
    autowire="byType" />  
  
<bean id="userDAO" class="tudu.domain.dao.jpa.UserDAOJpa" />
```

la propriété se nomme userDAO, l'autowiring fonctionnerait par nom

```
<bean id="userManager" class="tudu.service.impl.UserManagerImpl"  
    autowire="byName" />  
  
<bean id="userDAO" class="tudu.domain.dao.jpa.UserDAOJpa" />
```

- ✓ pour définir globalement le mode d'autowiring:

```
<beans (...) default-autowire="byType">
```



Injection automatique par annotation

- ✓ L'autowiring *via XML a des limites* (agit globalement sur les propriétés d'un objet)
➔ l'autowiring avec des annotations
- ✓ 2 annotations:
[apposées sur des propriétés, des constructeurs ou des modificateurs]
 - **@Autowired** (issue de Spring)
 - **@Resource** (injecter les ressources JNDI)
- ✓ 2 façons d'activer la détection des annotations:
 - déclarer dans le contexte le Bean spécifique **BeanPostProcessor**, qui effectuera l'injection automatiquement lors du chargement du contexte :

```
<bean class="org.springframework.beans.factory.annotation.➔  
    AutowiredAnnotationBeanPostProcessor" />
```



- utiliser la balise **annotation-config** du schéma **context**, qui active la détection d'un ensemble d'annotations dans Spring :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

  <context:annotation-config />

</beans>
```



Exemple : un DAO est injecté dans un service métier,
(la détection des annotations est activée)

- la déclaration des deux Beans :

```
<bean id="userManager" class="tudu.service.impl.UserManagerImpl" />  
  
<bean id="userDAO" class="tudu.domain.dao.jpa.UserDAOJpa" />
```

- annotation **@Autowired** sur la propriété **userDAO** du service :

```
(...)  
import org.springframework.beans.factory.annotation.Autowired;  
(...)  
public class UserManagerImpl implements UserManager {  
  
    @Autowired  
    private UserDAO userDAO;  
  
    (...)  
}
```

- ✓ rendre une dépendance optionnelle avec le paramètre *required* :

```
@Autowired(required=false)  
private UserDao userDao;
```

- ✓ Spring effectue un autowiring par type.
 - ➔ cas : plusieurs Beans correspondent au type d'une dépendance
 - ➔ lever les ambiguïtés avec *@Qualifier*

```
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;  
  
public class AlerteClientManager {  
  
    @Autowired  
    @Qualifier("goldMessageSender")  
    private MessageSender senderPourAbonnementGold;  
  
    @Autowired  
    @Qualifier("standardMessageSender")  
    private MessageSender senderPourAbonnementStandard;  
  
    (...)  
}
```

```
<bean class="SmsMessageSender">  
    <qualifier value="goldMessageSender" />  
</bean>  
  
<bean class="EmailMessageSender">  
    <qualifier value="standardMessageSender" />  
</bean>
```



Injection avec le schéma p

schéma XML [P] ➔ injection des propriétés (simples et collaborateurs)

l'en-tête du fichier 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"
xmlns:p="http://www.springframework.org/schema/p"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">
```

injection d'un
Collaborateur
avec [-ref]

```
<bean id="userManager" class="tudu.service.impl.UserManagerImpl"
p:userDAO-ref="userDAO" />
<bean id="userDAO" class="tudu.domain.dao.jpa.UserDAOJpa" />
```

injection d'une
Propriété

```
<bean id="user" class="tudu.domain.model.User"
p:firstName="Frédéric" p:lastName="Chopin" />
```



Sélection du mode d'instanciation, ou portée

- ✓ Récupération du Bean avec ***getBean*** → Spring décide à partir de cette définition comment l'objet doit être créé
- ✓ les Beans Spring sont par défaut des singletons

portée → *mode* d'instanciation

Portées proposées par Spring :

Portée	Description
singleton	Un seul Bean est créé pour la définition.
prototype	Une nouvelle instance est créée chaque fois que le Bean est référencé.
request	Une instance est créée pour une requête HTTP, c'est-à-dire que chaque requête HTTP dispose de sa propre instance de Bean, pour toute sa durée de vie. Portée valable seulement dans le contexte d'une application Web.
session	Une instance est créée pour une session HTTP, c'est-à-dire que la session HTTP dispose de sa propre instance de Bean, pour toute sa durée de vie. Portée valable seulement dans le contexte d'une application Web.
globalSession	Une instance est créée pour une session HTTP globale, c'est-à-dire que la session HTTP globale dispose de sa propre instance de Bean, pour toute sa durée de vie. Portée valable seulement dans un contexte de portlet.

attribut [***scope***] :

```
<bean id="user" class="tudu.domain.model.User" scope="prototype" />
```

Portées singleton et prototype

- ✓ les plus fréquemment utilisées
- ✓ cas d'une application d'entreprise :
 - Singleton ➔ objets accédée par plusieurs utilisateurs simultanément (services métiers et des DAO)
 - Prototype ➔ objets à usage unique. un objet pour chaque utilisation (Les actions (contrôleurs) de Struts 2)



- configuration 100 % XML
- configuration par des annotations

pas de règle absolue

styles de configuration,
fondés sur le bon sens,
la cohérence
et l'uniformité



Détection automatique de composants

- ✓ Approche 100 % annotation
- ✓ annotations sur les classes des Beans

Annotations pour la détection de composants:

Annotation	Description
@Component	Annotation générique des composants
@Repository	Annotation dénotant un Bean effectuant des accès de données (par exemple DAO)
@Service	Annotation dénotant un Bean effectuant des traitements métier
@Controller	Annotation dénotant un contrôleur de l'interface graphique (généralement un contrôleur Web)

@Component ➔ qualifier un composant (objets de type boîte noire)

@Repository - @Service - @Controller ➔ qualifier le type du composant (renseigner le conteneur sur l'utilité du Bean)



Paramétrages pour la détection automatique

- ✓ définir les classes des composants et les annoter
- ✓ préciser dans le fichier XML de configuration de Spring le package dans lequel les composants doivent être recherchés

singltons par défaut

```
@Repository
public class UserDAOJpa implements UserDAO {
    (...)
}
```

```
@Repository("userDAO")
@Scope("prototype")
public class UserDAOJpaExplicitName implements UserDAO {
    (...)
}
```

balise ***component-scan*** du schéma XML ***context***

Le package à analyser est précisé avec l'attribut ***base-package***

```
http://www.springframework.org/schema/context/spring-context.xsd">
  <context:component-scan base-package="tudu.domain.dao.jpa" />
</beans>
```



Filtrer les composants à détecter

Types de filtre pour la détection de composants

Type de filtre	Description
annotation	L'annotation précisée est apposée sur la classe du composant.
assignable	La classe du composant peut être transtypée en la classe/interface précisée.
aspectj	La classe du composant correspond à la coupe AspectJ précisée.
regex	La classe du composant correspond à l'expression régulière précisée.

✓ modifier du comportement en paramétrant des filtres dans la balise ***component-scan***.

```
<context:component-scan base-package="tudu">
  <context:include-filter type="aspectj"
    expression="tudu..*DAO*" />
  <context:exclude-filter type="aspectj"
    expression="tudu..*Service*" />
</context:component-scan>
```



détecter tous les DAO, mais pas les services
via une expression AspectJ



Les post-processeurs

- Bean spécifique capable d'influer sur le mode de création des Beans
- effectuer des vérifications sur la bonne configuration des Beans ou à effectuer des modifications de façon transverse
- 2 types:
 - *BeanPostProcessor* ➔ agit directement sur les Beans.
capable de modifier un Bean ou de faire des vérifications sur un Bean déjà initialisé.
 - *BeanFactoryPostProcessor* ➔ agit sur les définitions des Beans.
capable d'agir sur la définition des Beans avant leur création



Exemple : *BeanPostProcessor*

```
public class SimpleBeanPostProcessor
    implements BeanPostProcessor {

    public Object postProcessBeforeInitialization(
        Object bean, String beanName) throws BeansException {
        if(bean instanceof User) {←❶
            ((User) bean).setCreationDate(new Date());←❷
        }
        return bean;
    }

    public Object postProcessAfterInitialization(
        Object bean, String beanName) throws BeansException {
        return bean;←❸
    }

}
```

Activation: le déclarer dans le contexte pour qu'il prenne effet

```
<bean class="splp.postproc.SimpleBeanPostProcessor" />
<bean id="user" class="tudu.domain.model.User" />
```



Exemple : *BeanFactoryPostProcessor*

```
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.➡
    BeanFactoryPostProcessor;
import org.springframework.beans.factory.config.➡
    ConfigurableListableBeanFactory;

public class SimpleBeanFactoryPostProcessor
    implements BeanFactoryPostProcessor {

    public void postProcessBeanFactory(
        ConfigurableListableBeanFactory beanFactory)
        throws BeansException {
        System.out.println(
            beanFactory.getBeanDefinitionCount()+" bean(s) défini(s)"
        );
    }
}
```

Activation :

- le déclarer en tant que Bean dans un *ApplicationContext*.
- Pour une *BeanFactory*, il faut l'ajouter de façon programmatique.



Spring Expression Language: SpEL

- ✓ possibilité de l'utiliser pour la définition des Beans dans la configuration XML ou dans celle par annotation
- ✓ très proche dans sa syntaxe de langages tels que Java Unified EL ou OGNL

```
<bean id="user1" class="tudu.domain.model.User">← 1
  <property name="login" value="acogoluegnes" />
  <property name="firstName" value="Arnaud" />
  <property name="lastName" value="Cogoluegnes" />
</bean>

<bean id="user2" class="tudu.domain.model.User">← 2
  <property name="login" value="#{user1.login}" />
  <property name="firstName" value="#{user1.firstName}" />
  <property name="lastName" value="#{user1.lastName}" />
</bean>
```



variable ***systemProperties*** → accès aux propriétés système :

- définition des propriétés :

```
System.setProperty("databaseDriver", "org.hsqldb.jdbcDriver");  
System.setProperty("databaseUrl", "jdbc:hsqldb:mem:tudu-el");  
System.setProperty("databaseUser", "sa");  
System.setProperty("databasePassword", "");
```

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.➡  
    SingleConnectionDataSource">  
    <property name="driverClassName"  
        value="#{systemProperties.databaseDriver}" />  
    <property name="url" value="#{systemProperties.databaseUrl}" />  
    <property name="username"  
        value="#{systemProperties.databaseUser}" />  
    <property name="password"  
        value="#{systemProperties.databasePassword}" />  
</bean>
```

Spring AOP



Les concepts de la POA

- ✓ POA (programmation orientée aspect) ou **AOP** (Aspect-Oriented Programming)
- ✓ AOP est un paradigme → principes qui structurent la manière de modéliser les applications . La façon de les développer.
- ✓ Problématiques des approches classiques de modélisation (POO) :
 - Modélisation logicielle idéale → séparation totale entre les différentes préoccupations d'une application. pérenniser au maximum le code de l'application
 - 2 sortes de préoccupations → fonctionnel (classes métiers) et techniques (persistance des données DAO).
 - Problème de séparation des préoccupations dans le cas de l'intégration des fonctionnalités transversales .
[***transversales*** : fonctions et exigences qui concernent plusieurs modules d'une application(sécurité, la journalisation, la validation, gestion des transactions ...)]



- ces fonctionnalités sont implémentées dans chaque classe concernée
 - ➔ génération du phénomène de dispersion du code
 - ➔ une évolution implique la modification de plusieurs classes
 - ➔ dégradation de la qualité de la modélisation
- ✓ AOP capture les fonctionnalités transversales au sein d'entités spécifiques préservant la flexibilité de la conception de l'application
- ✓ AOP complète la POO (Programmation Orientée Objets) en offrant des mécanismes complémentaires pour mieux modulariser les préoccupations d'une application et améliorer leur séparation.
- ✓ Objectif de POA ➔ améliorer la modularité des logiciels et faciliter la réutilisation et la maintenance
- ✓ Frameworks POA du marché :
 - **AspectJ** : le plus complet <http://www.eclipse.org/aspectj/>
 - **JBoss AOP** : sous-projet de JBoss <http://labs.jboss.com/jbossaop/>
 - **Spring AOP**: framework Spring <http://www.springframework.org/>



Notions de base de la POA

La notion d'aspect

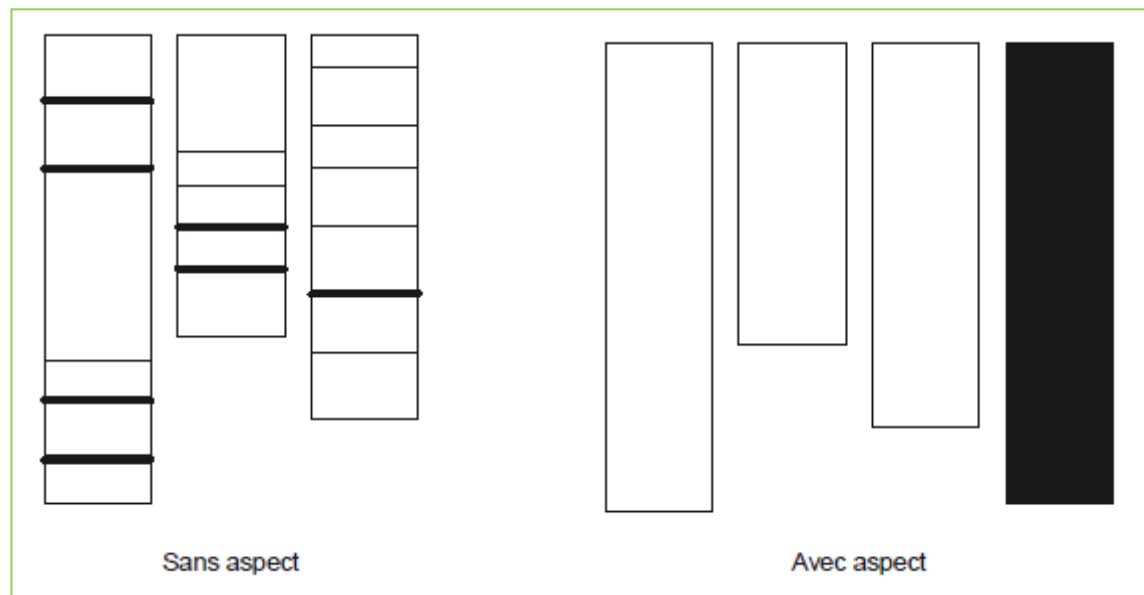
- ✓ entité logicielle qui rassemble le code d'une fonctionnalité transversale dispersé au sein de l'application

Aspect

Entité qui capture une fonctionnalité transversale à une application.

POA: aspect [la sécurité, la persistance] == POO: classe [clientèle, commandes]

Impact d'un aspect sur la localisation d'une fonctionnalité transversale



Les points de jonction (*join point*)

- ✓ désigne un endroit du programme où nous souhaitons ajouter un aspect. Il est fourni de manière « programmatique » par le développeur *via un numéro de ligne dans le code source*

join point

Point dans l'exécution d'un programme autour duquel un ou plusieurs aspects peuvent être ajoutés.

- ✓ l'expression [**UserManagerImpl.***] définit l'ensemble des méthodes de la classe *UserManagerImpl*. Chacune des méthodes est un point de jonction.

Les coupes (*pointcut*)

- ✓ Désigne un ensemble de points de jonction
- ✓ définie où un aspect devait être greffé dans une application
- ✓ définie à l'intérieur d'un aspect



Les greffons (*code advice*)

- ✓ définit ce que l'aspect greffe dans l'application. les instructions ajoutées par l'aspect
- ✓ représente l'implémentation de la fonctionnalité transversale
- ✓ Bloc de code définissant le comportement d'un aspect.
- ✓ joue le même rôle qu'une méthode et sont associé a une coupe et donc a des points de jonction

Chaque greffon est associé à une coupe. La coupe fournit l'ensemble des points de jonction autour desquels sera greffé le bloc de code du greffon. Une même coupe peut être utilisée par plusieurs greffons. Dans ce cas, différents traitements sont à greffer autour des mêmes points de jonction.



types de greffons

- ***before*** : le code est exécuté avant les *points de jonction*, c'est-à-dire avant l'exécution des méthodes.
- ***after returning*** : le code est exécuté après les points de jonction, c'est-à-dire après l'exécution des méthodes.
- ***after throwing*** : le code est exécuté après les points de jonction si une exception a été générée.
- ***after*** : le code est appelé après les points de jonction, même si une exception a été générée.
- ***around*** : le code est exécuté avant et après les points de jonction.

(le type ***around*** est l'union des types ***before*** et ***after***)



Le mécanisme d'introduction

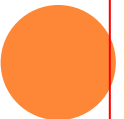
- ✓ permet d'étendre le comportement d'une classe en modifiant sa structure avec l'ajout d'attributs ou de méthodes.
- ✓ l'équivalent de l'héritage en POO. Ne permet pas de redéfinir une méthode. Mécanisme purement *additif*
- ✓ utilisation du terme [***introduction***] ➔ *les éléments sont ajoutés à la classe.*

Le tissage d'aspect (aspect weaver)

- ✓ tissage (en anglais *weaving*) : opération automatique pour obtenir une application opérationnelle intégrant les fonctionnalités des classes et celles des aspects
- ✓ le tissage s'effectue uniquement à l'exécution, au sein du conteneur léger

(aspect weaver)

Programme qui réalise une opération d'intégration entre un ensemble de classes et un ensemble d'aspects.




- ✓ Le tissage repose sur l'utilisation de **proxy dynamiques** créés avec l'API standard Java SE (classe: *java.lang.reflect.Proxy*).

proxy dynamiques :

- intercepte les appels aux méthodes de l'interface
- appelle les greffons implémentés par les aspects
- redirige, en fonction du type de greffon, les appels à la classe implémentant l'interface.

illustration fonctionnement:

- classe **UneImpl** implémente l'interface **UneInterface**.
 - un aspect : une coupe : appel à la méthode **uneMethode** de l'interface.
 - Injection des dépendances : le conteneur léger génère un proxy dynamique de l'interface qui intercepte les appels à l'ensemble des méthodes définies dans UneInterface.
 - Quand **uneMethode** est appelée: le proxy exécute le greffon :
 - Si le greffon est de type **before**, il est exécuté en premier, puis l'appel est redirigé vers UneImpl.
 - Si le greffon est de type **after**, l'appel est redirigé vers UneImpl, puis le greffon est exécuté.
 - Si le greffon est de type **around**, seul le greffon est appelé par le proxy qui redirige l'appel à UneImpl *via l'instruction **proceed***.
- 

Implémentations:

✓ définition d'un **aspect** :

1. implémentation du greffon sous forme d'une classe Java
2. définition de la coupe sous la forme d'un Bean
3. l'assemblage des deux dans un **advisor**
 ➔ classe: *DefaultPointcutAdvisor*
 (*advisor* se voit injecter le *greffon* et la *coupe*)

✓ définition d'une **coupes**:

interface *Pointcut* -

package : *org.springframework.aop*.

méthodes: *getClassFilter* - *getMethodMatcher*

(savoir quelles sont les classes et méthodes concernées par la coupe)



Implémentations:

✓ définition d'une **greffons** : (sous forme de bean):

- *arround*:

- interface : *MethodInterceptor* .
 - package : *org.aopalliance.intercept*.
 - méthode : *invoke()*

- *before*:

- interface : *MethodBeforeAdvice*.
 - package : *org.springframework.aop*.
 - méthode : *before()*

- *after returning*:

- interface : *AfterReturningAdvice*.
 - package: *org.springframework.aop*.
 - méthode: *afterReturning*

- *after throwing*:

- interface : *ThrowsAdvice*
 - package : *org.springframework.aop*.



Déclaration avec des configurations XML

- ✓ Fichier de configuration des beans
- ✓ déclaration dans l'élément **<aop:config>**
- ✓ *aspects* : **<aop: aspect>**
- ✓ *points d'action* : **<aop: pointcut>**
- ✓ *greffons* : *élément XML pour chaque type*
 - attribut : **pointcut-ref** → référence à un point d'action
 - attribut : **method** → nom de la méthode de greffon dans la classe d'aspect
- ✓ *introductions* : **<aop:declareparents>**



Déclarer des aspects

```
<beans ...>
  <aop:config>
    <aop:aspect id="loggingAspect" ref="calculatorLoggingAspect">
    </aop:aspect>

    <aop:aspect id="validationAspect" ref="calculatorValidationAspect">
    </aop:aspect>

    <aop:aspect id="introduction" ref="calculatorIntroduction">
    </aop:aspect>
  </aop:config>

  <bean id="calculatorLoggingAspect"
    class="com.apress.springrecipes.calculator.CalculatorLoggingAspect" />

  <bean id="calculatorValidationAspect"
    class="com.apress.springrecipes.calculator.
      CalculatorValidationAspect" />

  <bean id="calculatorIntroduction"
    class="com.apress.springrecipes.calculator.CalculatorIntroduction" />
  ...
</beans>
```

Déclarer des points d'action

```
<aop:config>
  <aop:pointcut id="loggingOperation" expression=
    "within(com.apress.springrecipes.calculator.ArithmeticCalculator+) ||
    within(com.apress.springrecipes.calculator.UnitCalculator+)" />

  <aop:pointcut id="validationOperation" expression=
    "within(com.apress.springrecipes.calculator.ArithmeticCalculator+) ||
    within(com.apress.springrecipes.calculator.UnitCalculator+)" />
```



Déclarer des greffons

```
<aop:config>
  ...
  <aop:aspect id="loggingAspect" ref="calculatorLoggingAspect">
    <aop:before pointcut-ref="loggingOperation"
      method="logBefore" />

    <aop:after-returning pointcut-ref="loggingOperation"
      returning="result" method="logAfterReturning" />

    <aop:after-throwing pointcut-ref="loggingOperation"
      throwing="e" method="logAfterThrowing" />

    <aop:around pointcut-ref="loggingOperation"
      method="logAround" />
  </aop:aspect>

  <aop:aspect id="validationAspect" ref="calculatorValidationAspect">
    <aop:before pointcut-ref="validationOperation"
      method="validateBefore" />
  </aop:aspect>
</aop:config>
```

Déclarer des introductions

```
<aop:config>
  ...
  <aop:aspect id="introduction" ref="calculatorIntroduction">
    <aop:declare-parents
      types-matching=
        "com.apress.springrecipes.calculator.ArithmeticCalculatorImpl"
      implement-interface=
        "com.apress.springrecipes.calculator.MaxCalculator"
      default-impl=
        "com.apress.springrecipes.calculator.MaxCalculatorImpl" />
  </aop:aspect>
</aop:config>
```



Déclaration avec des annotations *AspectJ*

- ✓ activation de la prise en charge des annotations *AspectJ* avec l'élément `<aop:aspectj-autoproxy>`

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

  <aop:aspectj-autoproxy />

  <bean id="arithmeticCalculator"
    class="com.apress.springrecipes.calculator.ArithmeticCalculatorImpl" />
```

- ✓ les annotations AspectJ:
 - un aspect : `@Aspect`.
 - un greffon : `@Before` - `@After` - `@AfterReturning`
`@AfterThrowing` - `@Around`
 - un point d'action : `@pointcut`



Exemples :

```
@Aspect
public class CalculatorValidationAspect {

    @Before("execution(* *.*(double, double))")
    public void validateBefore(JoinPoint joinPoint) {
        for (Object arg : joinPoint.getArgs()) {
            validate((Double) arg);
        }
    }

    private void validate(double a) {
        if (a < 0) {
            throw new IllegalArgumentException("Nombre")
        }
    }
}
```

```
@Aspect
public class CalculatorLoggingAspect {
    ...
    @Pointcut("execution(* *.*(..))")
    private void loggingOperation() {}

    @Before("loggingOperation()")
    public void logBefore(JoinPoint joinPoint) {
        ...
    }

    @AfterReturning(
        pointcut = "loggingOperation()",
        returning = "result")
    public void logAfterReturning(JoinPoint joinPoint, Object result) {
        ...
    }
}
```















