

Introduction à AOP (Aspect-Oriented Programming) avec le framework Spring

par [Eric Wawszczyk \(Accueil\)](#)

Date de publication : 17/12/2007

Dernière mise à jour :

Dans beaucoup d'applications informatiques, un module ou composant métier est régulièrement pollué par de multiples appels à des composants utilitaires externes. La programmation par aspect (AOP) va permettre d'extraire les dépendances entre modules concernant des aspects techniques entrecroisés et de les gérer depuis l'extérieur de ces modules en les spécifiant dans des composants du système à développer nommés aspects ; ils sont développés à un autre niveau d'abstraction.

- I - Introduction
- II - La programmation orientée aspect
 - II-A - Principe
 - II-B - Lexique
 - II-C - Stratégies d'implémentation
- III - AOP : Exemple d'utilisation
 - III-A - Présentation
 - III-B - La classe à tracer
 - III-C - Le fichier de configuration Spring
 - III-D - La classe MonLogger
 - III-E - Résultat
- IV - AOP : Fonctions avancées
 - IV-A - Les pointcut
 - IV-B - Les différents Advice
- V - Conclusion
 - V-A - Avantages
 - V-B - Inconvénients
- VI - Remerciements
- VII - Liens

Cet article a pour but de vous présenter rapidement la programmation par aspect (AOP) et sa mise en application dans une application basée sur le framework Spring.

Dans la pratique, les considérations techniques que sont censées implémenter les modules d'une application non seulement s'entrecroisent (par exemple la gestion des utilisateurs fait aussi appel à la génération de trace) mais aussi sont répartis dans la couche métier. C'est l'intrication ou entrecroisement des aspects techniques. Ainsi, une couche logicielle initialement dédiée à gérer la logique métier applicative, va se retrouver dépendante de modules gérant les aspects transactionnels, journalisation, etc.

La programmation par aspect (AOP) va permettre d'extraire les dépendances entre ces modules.

I - Introduction

II - La programmation orientée aspect

II-A - Principe

Au lieu d'avoir un appel direct à un module technique depuis un module métier, ou entre deux modules techniques différents, en programmation par aspect, le code du module en cours de développement est concentré sur le but poursuivi (la logique métier), tandis qu'un aspect est spécifié de façon autonome, implémentant un aspect technique particulier, par exemple la persistance ou encore la génération de trace. Un ensemble de points d'insertions ou joinpoint en anglais sont ensuite définis pour établir la liaison entre l'aspect et le code métier ou un autre aspect. Ces définitions de joinpoint sont définies dans le cadre de la PAO. Selon les frameworks ou les langages d'aspects, la fusion du code technique avec le code métier est alors soit réalisée à la compilation, soit à l'exécution.

Bien sûr, si chaque aspect créé devait lui-même définir explicitement à quel point d'exécution il doit s'insérer dans le code métier ou dans un autre aspect, c'est-à-dire par exemple avec une dépendance directe vers le module métier où devra s'insérer le code technique, on n'aurait alors fait que décaler le problème. Aussi, l'astuce particulière de la programmation par aspect consiste à utiliser un système d'expressions rationnelles pour préciser à quels points d'exécution (en anglais, joinpoint) du système l'aspect spécifié devra être activé.

Exemple : Ajout de logs dans une application existante

Une approche fréquente consisterait en ce cas à « patcher » le code un peu partout pour rajouter des `log.debug()` au début et à la fin de chaque méthode. Avec les outils d'AOP on peut facilement spécifier les changements requis SANS toucher au code source original, dont la logique reste intacte.

Les outils de programmation par aspect sont en fait similaires aux modificateurs (`before`, `after` et `around`) que l'on trouve dans des langages comme LISP, auxquels on a ajouté la possibilité d'une description d'insertions déclaratives.

Un aspect permet donc de spécifier :

- les points d'action (**pointcut**), qui définissent les points de jonction satisfaisants aux conditions d'activation de l'aspect, donc le ou les moments où l'interaction va avoir lieu,
- les greffons c'est-à-dire les programmes (**advice**) qui seront activés avant, autour de ou après les points d'action définis.

II-B - Lexique

La programmation orientée aspect, parce qu'elle propose un paradigme de programmation et de nouveaux concepts, a développé un jargon bien spécifique qui ne facilite pas la compréhension de ses concepts qui sont, en définitive, simples mais puissants.

- **aspect** : un module définissant des greffons et leurs points d'activation,
- **greffon** (en anglais, **advice**) : un programme qui sera activé à un certain point d'exécution du système, précisé par un point de jonction,
- **point d'action**, de coupure, de greffe (en anglais, **pointcut**) : endroit du logiciel où est inséré un greffon par le tisseur d'aspect,
- **point de jonction**, d'exécution (en anglais, **join point**) : endroit spécifique dans le flot d'exécution du système, où il est valide d'insérer un greffon. Pour clarifier le propos, il n'est pas possible, par exemple, d'insérer un greffon au milieu du code d'une fonction. Par contre on pourra le faire avant, autour de, à la place ou après l'appel de la fonction.

Deux grandes stratégies de tissage d'aspects existent :

- le tissage **statique** par instrumentation du code source ou du pseudo-code machine intermédiaire
- le tissage **dynamique** lors de l'exécution du logiciel

C'est ce dernier cas que nous mettrons en pratique dans l'exemple qui va suivre.

II-C - Stratégies d'implémentation

III - AOP : Exemple d'utilisation

L'objectif de ce cas pratique est de tracer les appels aux méthodes objets dans une application basée sur Spring.

Ce besoin peut aussi bien se présenter en phase de développement, pendant laquelle, le développeur n'a pas forcément envie de mettre dans son code des `log.debug()` partout. Mais aussi en phase de maintenance, pour mieux comprendre le cheminement de l'application ou tracer un bug.

Avec Spring, il est possible de tracer les appels aux méthodes objets pendant l'exécution ET SANS modifier le code de l'application.

III-A - Présentation

III-B - La classe à tracer

Prenons l'exemple d'une simple classe métier "MonService" qui contient la méthode "hello()" et renvoie une String.

Nous allons donc voir comment modifier la configuration de Spring afin de mettre une trace à chaque appel et à chaque sortie de la méthode hello().

Le principe est d'intercepter les entrées/sorties de la méthode "hello()" et de les logger à l'aide d'une classe externe "MonLogger".

MonService.java

```
package ew.service;

public class MonService {

    public String hello(String msg){
        String s = "Hello "+msg;
        System.out.println(s);
        return s;
    }
}
```

Et maintenant le programme qui l'appelle :

MonServiceTest.java

```
package ew.test;

import junit.framework.TestCase;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import ew.service.MonService;

public class MonServiceTest extends TestCase {

    public void testMonService() {

        ApplicationContext context = new ClassPathXmlApplicationContext( new String[]
        {"springContext.xml"} );
```

MonServiceTest.java

```
MonService monService = (MonService) context.getBean("monService");
monService.hello("from Spring !");

}
```

Et exécutant le programme, nous obtenons :

```
Hello from Spring !
```

III-C - Le fichier de configuration Spring

Le fichier de configuration initial de Spring :

springContext.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/beans
         http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <bean name="monService" class="ew.service.MonService" />

</beans>
```

Le fichier Spring complété par la configuration AOP :

springContext.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: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">

  <!-- Debut de la configuration AOP -->
  <aop:config>
    <aop:pointcut id="servicePointcut" expression="execution(* ew.service.*(..))"/>
    <aop:aspect id="loggingAspect" ref="monLogger">
      <aop:before method="logMethodEntry" pointcut-ref="servicePointcut"/>
      <aop:after-returning method="logMethodExit" returning="result"
        pointcut-ref="servicePointcut"/>
    </aop:aspect>
  </aop:config>

  <bean id="monLogger" class="ew.aop.MonLogger"/>
  <!-- Fin de la configuration AOP -->

  <bean name="monService" class="ew.service.MonService" />

</beans>
```

```
springContext.xml
```

Explication de la configuration AOP :

```
<aop:config> ... </aop:config>
```

Définit le bloc de configuration AOP

```
<aop:pointcut id="servicePointcut" expression="execution(* ew.service.*(..))"/>
```

Permet de définir des points d'interception sur les objets.

Ici l'expression **ew.service.*.*** signifie que toutes les méthodes des objets qui sont dans le package "ew.service" seront interceptées.

```
<aop:aspect id="loggingAspect" ref="monLogger">
```

Les appels aux méthodes seront renvoyés vers le bean Spring "monLogger" (classe ew.aop.MonLogger)

```
<aop:before method="logMethodEntry" ...>
```

Avant l'appel (before) de la méthode "hello()", la méthode "MonLogger.logMethodEntry()" est appelée

```
<aop:after-returning method="logMethodExit" returning="result"...>
```

Après l'appel (after) de la méthode "hello()", la méthode "MonLogger.logMethodExit()" est appelée et le résultat de la méthode "hello()" lui sera passé en argument.

Cette classe contient les 2 méthodes "logMethodEntry()" et "logMethodExit()" qui vont être appelées par Spring-AOP pour tracer les appels des méthodes interceptées.

```
MonLogger.java
```

```
package ew.aop;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.JoinPoint.StaticPart;

public class MonLogger {

    // Cette méthode est appelée à chaque fois (et avant) qu'une méthode du package ew.service est
    // interceptée
    public void logMethodEntry(JoinPoint joinPoint) {

        Object[] args = joinPoint.getArgs();

        // Nom de la méthode interceptée
        String name = joinPoint.getSignature().toLongString();
    }
}
```

MonLogger.java

```
StringBuffer sb = new StringBuffer(name + " called with: [");

// Liste des valeurs des arguments reçus par la méthode
for(int i = 0; i < args.length; i++) {
    Object o = args[i];
    sb.append("'" + o + "'");
    sb.append((i == args.length - 1) ? " " : ", ");
}
sb.append("]");

System.out.println(sb);
}

// Cette méthode est appelée à chaque fois (et après) qu'une méthode du package ew.service est
interceptée
// Elle reçoit en argument 'result' qui est le retour de la méthode interceptée
public void logMethodExit(StaticPart staticPart, Object result) {

    // Nom de la méthode interceptée
    String name = staticPart.getSignature().toLongString();

    System.out.println(name + " returning: [" + result + "]");
}
}
```

III-D - La classe MonLogger**III-E - Résultat**

Et exécutant le programme, nous obtenons maintenant :

```
public java.lang.String ew.service.MonService.hello(java.lang.String) called with: ['from Spring
!']
Hello from Spring !
public java.lang.String ew.service.MonService.hello(java.lang.String) returning: [Hello from Spring
!]
```

IV - AOP : Fonctions avancées

IV-A - Les pointcut

Les méthodes à interceptées sont définies dans la balise `<aop:pointcut>` par l'attribut **expression** : `<aop:pointcut expression="..."/>`

Voici quelques exemples d'expression courantes pour un pointcut :

- **execution(public * *(..))** : Toutes les méthodes public
- **execution(* set*(..))** : Toutes les méthodes commençant par 'set'
- **execution(* com.xyz.service.IAccountService.*(..))** : Toutes les méthodes de l'interface 'IAccountService'
- **execution(* com.xyz.service.*.*(..))** : Toutes les méthodes du package 'com.xyz.service'
- **execution(* com.xyz.service..*.*(..))** : Toutes les méthodes du package 'com.xyz.service' et de ses sous-packages

Les différents types d'advice sont :

- **Before advice**: Exécuté avant le join point.
- **After returning advice**: Exécuté après le join point (Si la méthode interceptée s'exécute normalement - sans retourner d'exception).
- **After throwing advice**: Exécuté si la méthode interceptée retourne une exception.
- **After (finally) advice**: Exécuté en sortie du join point (exécution normale de la méthode ou sortie en exception).
- **Around advice**: Il englobe l'exécution de la méthode interceptée. Il s'agit du plus puissant des advices. Il permet de paramétrer le comportement avant et après exécution de la méthode.

Il permet ainsi d'exécuter ou non la méthode, de définir le retour souhaité, voir de lancer une exception.

IV-B - Les différents Advice

V - Conclusion

Grâce à l'AOP, le couplage entre les modules gérant des aspects techniques peut être réduit de façon très importante, en utilisant ce principe, ce qui présente de nombreux avantages :

- **Maintenance accrue** : les modules techniques, sous forme d'aspect, peuvent être maintenu plus facilement du fait de son détachement de son utilisation,
- **Meilleure réutilisation** : tout module peut être réutilisé sans se préoccuper de son environnement et indépendamment du métier ou du domaine d'application. Chaque module implémentant une fonctionnalité technique précise, on n'a pas besoin de se préoccuper des évolutions futures : de nouvelles fonctionnalités pourront être implémentées dans de nouveaux modules qui interagiront avec le système au travers des aspects.
- **Gain de productivité** : le programmeur ne se préoccupe que de l'aspect de l'application qui le concerne, ce qui simplifie son travail, et permet d'augmenter la parallélisation du développement.
- **Amélioration de la qualité du code** : La simplification du code qu'entraîne la programmation par aspect permet de le rendre plus lisible et donc de meilleure qualité.

V-A - Avantages

Le tissage d'aspect qui n'est finalement que de la génération automatique de code inséré à certains points d'exécution du système développé, produit un code qui peut être difficile à analyser (parce que généré automatiquement) lors des phases de mise au point des logiciels (débugage, test). Mais en fait cette difficulté est du même ordre que celle apportée par toute décomposition non linéaire (fonctionnelle ou objet par exemple).

V-B - Inconvénients

VI - Remerciements

Merci à ma société **Webnet** pour m'avoir laissé du temps pour rédiger cet article.

Merci à **RideKick** pour sa relecture.

VII - Liens

Source [Les sources du tutoriel](#)

 [Définition de la programmation orientée aspect](#)

 [Introduction au framework Spring](#)

 [Spring : théorie & pratique - Programmation orientée aspect](#)

 [Aspect Oriented Programming with Spring](#)

