Programmation orientée aspect pour Java/J2EE

Renaud Pawlak
Jean-Philippe Retaillé
Lionel Seinturier

© Groupe Eyrolles, 2004,
ISBN : 2-212-11408-7

EYROLLES

Chapitre 5

JBoss AOP

Ce chapitre aborde un troisième environnement de POA : JBoss AOP. La syntaxe et les concepts décrits dans ce chapitre correspondent à la version beta 3 de JBoss AOP.

L'annexe C fournit les instructions à suivre pour télécharger et installer JBoss AOP.

Comme JAC, JBoss AOP est un framework pour la programmation orientée aspect. Cela signifie que l'écriture d'un aspect se fait en Java pur, sans extension syntaxique. Alors que les coupes sont définies en Java avec JAC, elles le sont en XML avec JBoss AOP. Les codes advice sont par contre écrits en Java. Comme dans JAC, le tissage est dynamique et s'effectue à l'exécution.

JBoss AOP a été conçu et développé par Bill Burke avec la collaboration de contributeurs, dont Marc Fleury, le CEO du JBoss Group. JBoss AOP peut s'utiliser de façon autonome ou conjointement avec le serveur d'applications J2EE JBoss. Dans le premier cas, la version autonome est appelée standalone. Dans le deuxième cas, à partir de la version 4.0, le serveur d'applications JBoss inclut en standard le framework JBoss AOP.

JBoss AOP est un logiciel Open Source distribué librement et gratuitement selon les termes de la licence GNU LGPL (Lesser General Public License). Tout en étant fondé sur le modèle Open Source, cette licence autorise une utilisation de JBoss AOP dans des produits commerciaux. Le site Web de JBoss AOP est *http://www.jboss.org/developers/projects/jboss/aop.*

Première application avec JBoss AOP

Cette section fournit un exemple simple de mise en œuvre de la POA avec JBoss AOP, qui va nous permettre de présenter les bases de la syntaxe d'écriture des aspects, coupes et codes advice.

Reprenons l'application Gestion de commande décrite au chapitre 3. Rappelons qu'elle permet de gérer des commandes client. Nous allons lui appliquer le même aspect de gestion de traces applicatives qu'au chapitre 3. Cet aspect a pour fonction d'inspecter le comportement de l'application afin de connaître les méthodes appelées et l'ordre de ces appels.

Premier aspect de trace

Le premier aspect que nous allons écrire avec JBoss AOP trace les exécutions des méthodes de la classe Order. Le code de cet aspect comporte deux fichiers, **jboss-aop.xml** et **TraceInterceptor.java.** Le premier est un fichier XML qui définit une coupe, et le second un fichier Java qui fournit le code advice associé à cette coupe. Nous fournissons le contenu de ces fichiers dans les deux sections suivantes.

Les concepteurs de JBoss AOP parlent d'intercepteur plutôt que de code advice. Nous emploierons donc ce terme tout au long de ce chapitre. Il n'y a néanmoins pas de différence fondamentale entre un code advice AspectJ et un intercepteur JBoss AOP. Ce sont tous deux des blocs de code qui définissent le comportement d'un aspect. Le code d'un intercepteur, comme un code advice, s'exécute avant ou après un point de jonction.

Les coupes

Si AspectJ définit les coupes à l'aide de mots-clés et JAC en fournissant une méthode pointcut, JBoss AOP utilise un fichier XML : il s'agit de **jboss-aop.xml.**

JBoss AOP fournit donc un ensemble de balises XML et d'attributs pour définir des coupes et leurs intercepteurs associés. Les deux balises principales utilisées par JBoss AOP pour définir des coupes sont <bind> et <interceptor>. Nous examinons en détail la signification de ces balises à la section « Les coupes ».

Pour l'instant, il nous suffit de dire que <bind> permet de désigner les points de jonction appartenant à la coupe, tandis que <interceptor> indique l'intercepteur associé à ces points de jonction.

À titre d'exemple, examinons le fichier **jboss-aop.xml** suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<aop>
  <bind←❶
   pointcut="execution(←❷
   public void aop.jboss.Order->addItem(java.lang.String,int))"←❸
  >
      <interceptor class="aop.jboss.TraceInterceptor" />←❹
  </bind>
</aop>
```

La première ligne est l'en-tête standard pour tout fichier XML. La seconde ligne contient la balise <aop>, qui est la balise principale de tous les fichiers **jboss-aop.xml.**

La balise <bind> (repère ❶) débute la définition d'une coupe. L'attribut pointcut fournit l'expression de coupe. Cette expression est constituée à l'aide de mot-clé et de valeur. Le mot-clé cité dans l'exemple (execution repère ❷) correspond aux points de jonction de type exécution de méthodes. Nous reviendrons sur l'ensemble des mots-clés autorisés dans une expression de coupe à la section « Les coupes ».

La valeur fournie entre parenthèses après le mot-clé execution est une signature de méthode pouvant contenir des wildcards. Cette valeur désigne donc les méthodes que l'on souhaite inclure dans la coupe. Dans l'exemple (repère ❸) la

signature désigne simplement la méthode addItem de la classe aop.jboss.Order acceptant deux paramètres de type String et int.

La balise <interceptor> (repère ❹) fournit, *via* l'attribut class, le nom de la classe implémentant l'intercepteur. Il s'agit ici de la classe aop.jboss.TraceInterceptor dont nous fournissons le code à la section suivante.

Les intercepteurs

Un intercepteur JBoss AOP exécute du code avant ou après les points de jonction appartenant à une coupe. Le code d'un intercepteur est fourni dans une classe qui doit implémenter l'interface org.jboss.aop.advice.Interceptor. Cette interface définit deux méthodes, getName, qui doit retourner le nom de l'intercepteur, et invoke, qui doit fournir le code avant/après.

La classe TraceInterceptor suivante fournit le code de l'intercepteur associé à la coupe précédente :

```
package aop.jboss;

import org.jboss.aop.advice.Interceptor;
import org.jboss.aop.joinpoint.Invocation;
import org.jboss.aop.joinpoint.MethodInvocation;

public class TraceInterceptor implements Interceptor {
   public String getName() { return "TraceInterceptor"; }←❶

   public Object invoke(Invocation invocation)←❷
     throws Throwable {

      MethodInvocation mi = (MethodInvocation) invocation;←❸
      String methodName = mi.method.getName();

      System.out.println("Avant "+methodName);
      Object rsp = invocation.invokeNext();←❹
      System.out.println("Après "+methodName);
      return rsp;
}  }
```

Dans cet exemple, la méthode getName (repère ❶) retourne la chaîne de caractères TraceInterceptor. C'est le nom que l'on souhaite attribuer à l'intercepteur. Par convention, il s'agit souvent du nom de la classe, mais n'importe quelle autre valeur peut être choisie.

La méthode invoke (repère ❷) est appelée par le framework JBoss AOP juste avant un des points de jonction de la coupe associée à l'intercepteur. La signature de la méthode invoke est imposée : un seul paramètre de type Invocation est autorisé et le type de retour est Object. Le type d'exception Throwable doit être présent. Le paramètre de type Invocation permet d'introspecter le point de jonction. Le type Object constitue la valeur retournée par l'intercepteur à l'appelant. Throwable est le type racine de toutes les exceptions et erreurs en Java. Sa mention indique que la méthode invoke est susceptible de lever n'importe quelle exception ou erreur.

Dans la classe TraceInterceptor, la méthode invoke commence par récupérer, *via* le paramètre mi (repère ❸), le nom de la méthode dont nous interceptons l'exécution. Rappelons que la coupe définie dans le fichier **jboss-aop.xml** et

associée à cet intercepteur concerne uniquement les exécutions de méthodes. Le nom de méthode intercepté est affiché, précédé du message « Avant ». L'appel de la méthode invokeNext (repère ❹) exécute le code correspondant au point de jonction. invokeNext joue le même rôle que proceed dans AspectJ ou dans JAC. La valeur retournée par invokeNext doit être propagée à l'appelant : c'est ce que fait l'instruction return. Avant cela, un message « Après » suivi du nom de la méthode interceptée est affiché.

Compilation

Une application orientée aspect avec JBoss AOP est composée de fichiers Java pour l'application et les intercepteurs et d'un fichier XML pour la définition des coupes. Bien que n'importe quel nom puisse être employé, le fichier XML s'appelle de façon usuelle **jboss-aop.xml**.

La façon la plus simple de compiler un programme JBoss AOP consiste à utiliser un fichier de compilation Ant. Ant (*http://ant.apache.org*) est un outil standard utilisé très couramment par les développeurs Java pour automatiser les processus de compilation des programmes et de façon plus générale, n'importe quelle tâche (nettoyage de répertoire, lancement d'une exécution, etc.) en relation avec le développement des programmes. À la manière de l'outil make du monde Unix, Ant permet de découper une tâche en plusieurs sous-tâches, d'exprimer les dépendances entre ces sous-tâches (par exemple, d'exprimer qu'une sous-tâche doit être effectuée avant telle autre) et d'enchaîner leur exécution en respectant ces dépendances. Ces différentes informations sont définies dans un fichier XML **build.xml**. Ant permet également d'exécuter de façon incrémentale les tâches de compilation, c'est-à-dire d'éviter de re-exécuter inutilement une tâche de compilation déjà effectuée.

Le fichier **build.xml** suivant va nous permettre de compiler l'application Gestion de commandes. Par la même occasion, nous exploiterons ce fichier dans la section suivante pour exécuter l'application.

```
<?xml version="1.0" encoding="UTF-8"?>
<project default="run" name="Gestion de commandes">

  <property name="jboss-aop.root" value="c:\jboss-aop-b1"/>🡐❶

  <target name="prepare">
    <path id="classpath">
      <pathelement path="." />
      <fileset dir="${jboss-aop.root}">
        <include name="*.jar" />
      </fileset>
    </path>
    <taskdef name="aopc" classname="org.jboss.aop.ant.AopC"
             classpathref="classpath"/>
  </target>

  <target name="compile" depends="prepare">🡐❷
    <javac srcdir="src" destdir="."
           debug="on" deprecation="on"
           optimize="off"
           includes="**">
      <classpath refid="classpath"/>
    </javac>
    <aopc compilerclasspathref="classpath" classpathref="classpath"
          verbose="true">
```

```
      <classpath path="."/>
      <src path="."/>
      <aoppath path="jboss-aop.xml"/>
    </aopc>
  </target>

  <target name="run" depends="compile">←❸
    <java fork="yes" failOnError="true"
          className="aop.jboss.Customer">←❹
      <sysproperty key="jboss.aop.path" value="jboss-aop.xml"/>
      <classpath refid="classpath"/>
    </java>
  </target>

</project>
```

Le repère ❶ désigne le répertoire dans lequel JBoss AOP a été installé (ici il s'agit du répertoire **c:\jboss-aop-b1**). En fonction du répertoire choisi lors de l'installation, la valeur désignée par le repère ❶ dans le fichier **jboss-aop.xml** doit éventuellement être modifiée.

Le fichier jboss-aop.xml précédent définit deux tâches principales : compile (repère ❷) et run (repère ❸). Comme leur nom le suggère, ces deux tâches permettent respectivement de compiler et d'exécuter l'application.

La compilation de l'application Gestion de commandes s'effectue en lançant la commande suivante dans un shell Windows ou Unix :

```
ant compile
```

Cela commande produit le résultat suivant :

```
Buildfile: C:\examples\build.xml

prepare:

compile:

      [javac] Compiling 4 source files to C:\examples

       [aopc] [deploying] file:/C:/examples/jboss-aop.xml

       [aopc] [trying to transform] aop.jboss.Catalog

       [aopc] [no comp needed] C:\examples\aop\jboss\Catalog.class

       [aopc] [trying to transform] aop.jboss.Customer

       [aopc] [no comp needed] C:\examples\aop\jboss\Customer.class

       [aopc] [trying to transform] aop.jboss.Order

       [aopc] [compiled] C:\examples\aop\jboss\Order.class

       [aopc] [trying to transform] aop.jboss.TraceInterceptor

       [aopc] [no comp
needed]C:\examples\aop\jboss\TraceInterceptor.class

BUILD SUCCESSFUL

Total time: 3 seconds
```

Exécution

Une fois compilée, l'application peut être lancée à l'aide de la commande :

```
ant run
```

Le résultat suivant est alors généré :

```
Buildfile: C:\examples\build.xml

prepare:

compile:

       [aopc] [deploying] file:/C:/examples/jboss-aop.xml

       [aopc] [trying to transform] aop.jboss.Catalog

       [aopc] [no comp needed] C:\examples\aop\jboss\Catalog.class

       [aopc] [trying to transform] aop.jboss.Customer

       [aopc] [no comp needed] C:\examples\aop\jboss\Customer.class

       [aopc] [trying to transform] aop.jboss.Order

       [aopc] [compiled] C:\examples\aop\jboss\Order.class

       [aopc] [trying to transform] aop.jboss.TraceInterceptor

       [aopc] [no comp
needed]C:\examples\aop\jboss\TraceInterceptor.class

run:🡐❶

       [java] Avant addItem

       [java] 1 article(s) DVD ajouté(s) à la commande

       [java] Après addItem

       [java] Avant addItem

       [java] 2 article(s) CD ajouté(s) à la commande

       [java] Après addItem

       [java] Montant de la commande : 50.0 euros

BUILD SUCCESSFUL

Total time: 1 second
```

Le résultat de l'exécution débute au repère ❶. La chaîne de caractère `[java]` qui précède chaque ligne est un préfixe ajouté automatiquement par Ant.

Toutes les lignes commençant par « Avant » correspondent à l'interception d'un point de jonction par la méthode invoke de la classe TraceInterceptor. Nous constatons que les deux exécutions de la méthode addItem et l'exécution de la méthode computeAmount ont été interceptées.

Les coupes

Cette section revient sur le mécanisme de définition de coupe et l'examine en détail.

Avec JBoss AOP, les coupes sont définies dans des fichiers XML. Chaque application est associée à un fichier XML de définition de coupe appelé habituellement **jboss-aop.xml.** Plusieurs coupes peuvent être définies dans un même fichier **jboss-aop.xml.**

La balise principale d'un fichier **jboss-aop.xml** de coupe est <aop>. Tout fichier de définition de coupe a la structure suivante :

```
<?xml version="1.0" encoding="UTF-8"?>
<aop>
  <bind pointcut=" ... expression de coupe ... " >
    <interceptor class=" ... classe d'interception ... " />
  </bind>
  ....
</aop>
```

La première ligne est un en-tête standard en XML. Les définitions de coupe sont comprises entre les balises <aop> et </aop>. La balise <bind> définit une coupe à l'aide d'une expression fournie par l'attribut pointcut. Un fichier **jboss-aop.xml** contient autant de balises <bind> qu'il y a de coupes dans l'application. À l'intérieur d'une balise <bind> la balise <interceptor> définit l'intercepteur associé à la coupe.

Les types de coupes

JBoss AOP permet de définir cinq types de coupes : exécution de méthode, constructeur, attribut, classe et appel de méthode. Chacun de ces types est associé à un mot-clé différent dans l'expression de coupe (attribut pointcut de la balise <bind>).

Dans la suite de cette section, nous commençons par présenter les principes généraux communs à toutes les expressions de coupes, puis nous présentons les différents types de coupes. Après cela, nous analysons la façon dont un ou plusieurs intercepteurs peuvent être associés à une coupe. Quel que soit le type de coupe, ce mécanisme est identique.

Les expressions de coupe

Les expressions de coupe sont construites à partir de mots-clés. Chaque mot-clé représente un type de coupe et est associé entre parenthèses, à une signature de méthode ou d'attribut. La structure type d'une expression de coupe est donc la suivante :

```
<bind pointcut="mot-clé(signature)" >
```

Comme nous le verrons par la suite, JBoss AOP fournit par exemple un mot-clé (il s'agit du mot-clé execution) pour les coupes de types exécutions de méthode. La signature représente alors la ou les méthodes que l'on veut inclure dans la coupe. La signature comprend les attributs de visibilité de la méthode (public, private, etc.), le type de retour de la méthode, le nom de la classe dans laquelle est définie la méthode, le nom de la méthode et les paramètres de la méthode. Par exemple, la coupe suivante :

```
<bind
  pointcut="execution(
    public void aop.jboss.Order->addItem(java.lang.String,int))" >
```

représente toutes les exécutions de la méthode publique addItem, définie dans la classe aop.jboss.Order, prenant deux paramètres de type String et int et retournant void. Notons que le nom de la classe et le nom de la méthode sont séparés par les deux caractères ->. Notons également que plusieurs attributs de visibilité peuvent être fournis (par exemple public static) ou aucun (dans ce cas, toutes les méthodes spécifiées sont prises en compte, quelle que soit leur visibilité).

Les noms de classes peuvent être remplacés par l'opérateur $instanceof suivi d'un nom de type (classe ou interface) entre accolades. Plutôt que de spécifier une classe cible, on spécifie donc un ensemble de classes cibles sous-types d'un type particulier. Des exemples d'utilisation de l'opérateur $instanceof sont fournis au tableau 5.1.

Des wildcards peuvent être utilisés dans les signatures de méthodes. Il est ainsi possible d'inclure plusieurs méthodes dans une même coupe. De façon similaire à ce que propose AspectJ, JBoss AOP utilise les wildcards étoile (*) et point point (..). Le wildcard .. est utilisé à la place des paramètres de méthodes pour indiquer que n'importe quel profil de paramètres est autorisé. Par exemple, la coupe suivante :

```
<bind
  pointcut="execution(public void aop.jboss.Order->addItem(..))" >
```

désigne les exécutions de toutes les méthodes addItem, quel que soit leur profil de paramètres.

Le wildcard * est utilisé pour remplacer un type de retour, le nom d'un package, d'une classe, d'une méthode ou le type d'un paramètre. Le tableau 5.1 fournit quelques exemples d'utilisation du wildcard *.

Tableau 5.1 Exemples d'utilisation des wildcards dans les expressions de coupe JBoss AOP

EXPRESSION DE COUPE	DESCRIPTION
`public void aop.jboss.Order->addItem(*,int,*)`	Toutes les méthodes publiques addItem de la classe aop.jboss.Order, qui retournent void, qui acceptent trois paramètres et dont le deuxième est de type int.
`public void aop.jboss.Order->add*(..)`	Toutes les méthodes publiques de la classe aop.jboss.Order, qui retournent void, dont le nom commence par add et quel que soit leur profil de paramètres.
`private * aop.jboss.Order->*(..)`	Toutes les méthodes privées de la classe aop.jboss.Order, quels que soient leur type de retour et leur profil de paramètres.
`* aop.*.O*->getX()`	Les méthodes getX qui ne prennent pas de paramètres et dont le type de retour est quelconque, définies dans les classes dont le nom commence par O dans tous les sous-packages du package aop.
`* $instanceof{Remote}->*(..)`	Toutes les méthodes définies dans une classe implémentant l'interface `Remote`.

Les expressions de coupes peuvent être combinées à l'aide des opérateurs logiques AND, OR et ! (NOT logique), et regroupées à l'intérieur de parenthèses. Par exemple, l'expression :

```
<bind
  pointcut="execution(* Foo->foo()) OR execution(* Bar->bar())" >
```

désigne toutes les exécutions des méthodes foo et bar définies respectivement dans les classes Foo et Bar.

Afin de faciliter la réutilisation d'expressions de coupe d'usage courant, il est possible de nommer ces expressions et d'utiliser ces noms dans la définition d'autres coupes. L'attribution d'un nom à une expression de coupe s'effectue à l'aide de la balise <pointcut> et des attributs name et expr. Comme leurs titres le suggèrent, ces attributs définissent respectivement le nom et la valeur de l'expression. Par exemple, les déclarations XML suivantes :

```
<pointcut name="foo" expr="execution(* Foo->foo())" />
<pointcut name="bar" expr="execution(* Bar->bar())" />
```

définissent deux expressions de coupes nommées respectivement foo et bar. Ces noms peuvent alors s'utiliser dans d'autres expressions de la façon suivante :

```
<bind pointcut="foo OR bar" >
```

Le type exécution de méthode

Comme nous l'avons mentionné précédemment, un des cinq types de coupes de JBoss AOP concerne les exécutions de méthodes. Il se définit avec le mot-clé execution. Le mot-clé est associé à une expression formée par un signature contenant éventuellement des wildcards : toutes les méthodes dont la signature vérifie l'expression sont inclues dans la coupe.

Le type constructeur

Le deuxième type de coupe fournit par JBoss AOP concerne les exécutions de constructeurs. Le mot-clé correspondant est, comme pour les exécutions de méthodes, execution. Une signature est associée à ce mot-clé. Elle peut contenir des wildcards, mais doit obligatoire contenir le nom de méthode new. Cette dernière caractéristique a pour but de différencier les exécutions de constructeurs des exécutions de méthodes. Par exemple, l'expression suivante :

```
<bind pointcut="execution(public aop.jboss.Order->new(..))" >
```

désigne les exécutions de tous les constructeurs de la classe aop.jboss.Order.

Le type attribut

Les coupes de type attribut concernent les lectures et les écritures d'attributs. Trois mots-clés sont fournis par JBoss AOP : get, set et field. Le premier désigne les opérations de lecture d'attributs, le deuxième les écritures et le troisième à la fois les lectures et les écritures. Les trois mots-clés sont associés à un profil pouvant contenir des wildcards. Contrairement aux types de coupe précédents, les profils ne concernent pas ici des signatures de méthodes, mais des définitions d'attributs. On y trouve donc des éléments de visibilité, un type, un nom de classe et un nom d'attribut. Par exemple, l'expression suivante :

```
<bind pointcut="set(private * aop.jboss.Order->articles)" >
```

désigne les opérations d'écriture de l'attribut privé articles défini dans la classe aop.jboss.Order.

Le type classe

Les coupes de type classe regroupe les trois types vus précédemment : exécution de méthodes, constructeur et attribut. Tous les points de jonction de ces trois types sont donc inclus dans les coupes de type classe. Celles-ci se déclarent à l'aide du mot-clé all associé à un nom de classe pouvant contenir un ou plusieurs wildcards *. Par exemple, l'expression suivante :

```
<bind pointcut="all(aop.jboss.O*)" >
```

désigne toutes les exécutions de méthodes, de constructeurs et toutes les opérations de lecture et d'écriture d'attributs situées dans toutes les classes du package aop.jboss dont le nom commence par O.

Le type appel de méthode

Le dernier type de coupe fourni par JBoss AOP concerne les appels de méthodes. Il s'agit donc d'inclure dans la coupe tous les points de jonction qui correspondent à l'appel d'une ou de plusieurs méthodes. Ce type de coupe se déclare à l'aide du mot-clé call associé à une signature de méthode pouvant contenir des wildcards. Par exemple, la coupe suivante :

```
<bind pointcut="call(* aop.jboss.Order->*(..))" >
```

désigne tous les appels à une des méthodes de la classe aop.jboss.Order. Il n'est pas rare qu'une méthode d'usage courant soit appelée en de nombreux emplacements d'un programme. Il peut alors être utile de filtrer ces appels afin de ne retenir que ceux qui proviennent d'une localisation donnée. Le mot-clé within associé à une signature de méthode contenant éventuellement des wildcards permet de faire cela. Par exemple, la coupe suivante :

```
<bind pointcut="call(* aop.jboss.Order->addItem(..) AND
                      within(* aop.jboss.Customer->run()))" >
```

désigne les appels à la méthode addItem issue de la méthode run de la classe Customer. Notons que le mot-clé within est situé à l'intérieur des parenthèses de call. Il est également possible de combiner within avec la négation logique ! afin d'exclure certaines localisations.

Les intercepteurs associés à une coupe

Nous avons vu comment désigner l'ensemble des points de jonction d'une coupe. Nous allons maintenant compléter la définition d'une coupe en fournissant le ou les intercepteurs qui lui sont associés.

Déclaration d'un intercepteur

Quel que soit le type de coupe, la déclaration du ou des intercepteurs associés suit les mêmes règles. La balise <interceptor> annonce un intercepteur. Cette balise est associée à un attribut class qui fournit le nom de la classe qui implémente l'intercepteur.

À titre d'exemple, la déclaration suivante :

```
<bind pointcut="execution(* aop.jboss.O*->*(..))">
   <interceptor class="aop.jboss.MyInterceptor" />
</bind>
```

associe l'intercepteur implémenté par la classe aop.jboss.MyInterceptor à la coupe comprenant les exécutions de toutes les méthodes des classes dont le nom commence par O dans le package aop.jboss.

Les intercepteurs nommés

Lorsqu'un même intercepteur est utilisé dans plusieurs coupes, il peut être fastidieux de répéter la classe qui l'implémente à chaque utilisation. De plus, si cette classe change, il est nécessaire de répercuter ce changement plusieurs fois pour toutes les coupes qui utilisent l'intercepteur. Pour résoudre ce problème, JBoss AOP permet de déclarer globalement les intercepteurs dans le fichier **jboss-aop.xml.** Les intercepteurs sont alors nommés, et les coupes font référence à ce nom pour indiquer qu'elles utilisent l'intercepteur.

La balise <interceptor> permet de déclarer globalement un intercepteur dans le fichier **jboss-aop.xml.** Cette balise est associée à un attribut name qui fournit le nom de l'intercepteur et à un attribut class qui indique la classe implémentant l'intercepteur.

Par exemple, la ligne suivante :

```
<interceptor name="myInter" class="aop.jboss.MyInterceptor" />
```

définit l'intercepteur myInter implémenté par la classe aop.jboss.MyInterceptor.

Les coupes font référence à un intercepteur nommé à l'aide de la balise <interceptor-ref> associée à l'attribut name.

La déclaration suivante :

```
<bind pointcut="execution(* aop.jboss.Order->addItem(..))">
  <interceptor-ref name="myInter" />
</bind>
```

associe l'intercepteur myInter aux exécutions de la méthode addItem.

Les piles d'intercepteurs

Plusieurs intercepteurs peuvent être associés à une coupe. JBoss AOP parle en ce cas de pile d'intercepteurs.

Les intercepteurs sont exécutés dans leur ordre de définition dans la pile. Un même intercepteur peut apparaître plusieurs fois dans une pile. Il est alors exécuté plusieurs fois. L'appel de la méthode invokeNext dans le code d'un intercepteur permet de passer à l'exécution de l'intercepteur suivant ou à l'exécution du code correspondant au point de jonction s'il n'y a plus d'intercepteur.

Une pile d'intercepteurs se définit soit en utilisant plusieurs balises <interceptor> entre les balises <interceptors> et </interceptors> dans la définition d'une coupe, soit en déclarant une pile de façon globale dans le fichier **jboss-aop.xml** à l'aide de la balise <stack>.

La balise <stack> pour définir une pile d'intercepteurs est suivie de la liste des intercepteurs faisant partie de la pile. Ceux-ci peuvent être définis explicitement

à l'aide de la balise <interceptor> ou correspondre à un intercepteur nommé. Dans ce dernier cas, la balise <interceptor-ref> est utilisée.

La déclaration suivante :

```
<stack name="myStack">
  <interceptor-ref name="myInter" />
  <interceptor class="aop.jboss.Interceptor2" />
</stack>
```

définit la pile myStack, qui comprend deux intercepteurs. Le premier est l'intercepteur myInter défini précédemment. Le second est implémenté par la classe aop.jboss.Interceptor2.

Les coupes font référence à une pile d'intercepteurs à l'aide de la balise <stack-ref>.

La déclaration suivante :

```
<bind pointcut="execution(* aop.jboss.Order->addItem(..))">
   <stack-ref name="myStack" />
 </bind>
```

associe la pile d'intercepteurs myStack aux exécutions de la méthode addItem.

Les piles peuvent inclure d'autres piles. Ainsi, la pile myStack2 suivante :

```
<stack name="myStack2">
  <interceptor class="aop.jboss.Interceptor3" />
  <stack-ref name="myStack" />
</stack>
```

contient l'intercepteur implémenté par la classe aop.jboss.Interceptor3 et les intercepteurs de la pile myStack.

Les intercepteurs

Nous avons vu à la section précédente comment définir en XML des coupes avec JBoss AOP. Cette section s'intéresse à l'écriture des intercepteurs.

Rappelons que les intercepteurs de JBoss AOP fournissent du code qui s'exécute avant ou après les points de jonction. Les intercepteurs de JBoss AOP sont équivalents aux codes advice d'AspectJ et aux wrappers de JAC.

Implémentation d'un intercepteur

Les intercepteurs JBoss AOP s'écrivent en Java dans des classes qui doivent implémenter l'interface org.jboss.aop.advice.Interceptor.

Les deux méthodes suivantes sont définies dans cette interface :

- getName : retourne le nom de l'intercepteur. Le nom est choisi librement par le développeur.
- invoke : définit le code à exécuter avant et après un point de jonction.

De ces deux méthodes, invoke est la plus importante. C'est la méthode invoquée par le framework JBoss AOP juste avant un point de jonction associé à l'intercepteur.

La signature de la méthode invoke est imposée. Elle accepte un seul paramètre de type org.jboss.aop.joinpoint.Invocation. Comme nous le verrons à la section suivante, ce paramètre permet de faire de l'introspection de point de jonction. N'importe quelle exception est susceptible d'être levée par la méthode invoke. Sa signature doit donc comporter l'exception Throwable, classe ancêtre de toutes les exceptions Java. Finalement, le type de retour de la méthode invoke doit être Object.

Dans un intercepteur, l'appel de la méthode invokeNext permet de délimiter les parties de code qui s'exécutent avant et après le point de jonction. La méthode invokeNext doit être appelée sur l'objet de type Invocation passé en paramètre de la méthode invoke. La méthode invokeNext joue pour JBoss AOP le même rôle que proceed pour AspectJ et JAC.

Lorsqu'un point de jonction ne comporte qu'un seul intercepteur, la méthode invokeNext exécute le code correspondant au point de jonction. Lorsque plusieurs intercepteurs sont greffés autour du même point de jonction, invokeNext invoque l'intercepteur suivant ou le code correspondant au point de jonction si le dernier intercepteur est atteint.

L'appel de la méthode invokeNext est facultatif. Si elle n'est pas appelée, le code correspondant au point de jonction n'est pas exécuté. Ce comportement correspond à des aspects, par exemple un aspect de sécurité, qui ne souhaitent pas exécuter ce code ou qui souhaitent le remplacer par un autre comportement.

La méthode invokeNext peut être appelée plusieurs fois. Dans ce cas, le code correspondant au point de jonction est exécuté à plusieurs reprises. Ce comportement correspond à des aspects qui tentent d'exécuter à plusieurs reprises l'application, par exemple en cas de défaillance.

La méthode invoke retourne une valeur de type Object. Ce type correspond à la valeur retournée par le code du point de jonction. Il est nécessaire de propager cette valeur à l'appelant.

Le format général de la classe Java implémentant un intercepteur est donc :

```
package aop.jboss;

import org.jboss.aop.advice.Interceptor;
import org.jboss.aop.joinpoint.Invocation;

public class MyInterceptor implements Interceptor {
   public String getName() { return "unNom"; }

   public Object invoke(Invocation invocation) throws Throwable {

      System.out.println("Code avant");
      Object rsp = invocation.invokeNext();
      System.out.println("Code après");
      return rsp;
}  }
```

Le mécanisme d'introspection de point de

jonction

Le terme introspection peut être interprété comme l'action d'« inspecter à l'intérieur ». Il s'agit d'obtenir des informations sur le point de jonction courant. L'introspection est réalisée à l'aide du paramètre de type org.jboss.aop.joinpoint.Invocation de la méthode invoke d'un intercepteur.

Le type Invocation correspond à une classe qui possède plusieurs sous-classes. Chaque sous-classe correspond à un type de point de jonction particulier. La figure 5.1 fournit une vue de cette hiérarchie de classes. Seuls les attributs et les méthodes principales sont représentés. La documentation javadoc fournie avec JBoss AOP contient une description exhaustive de ces classes.

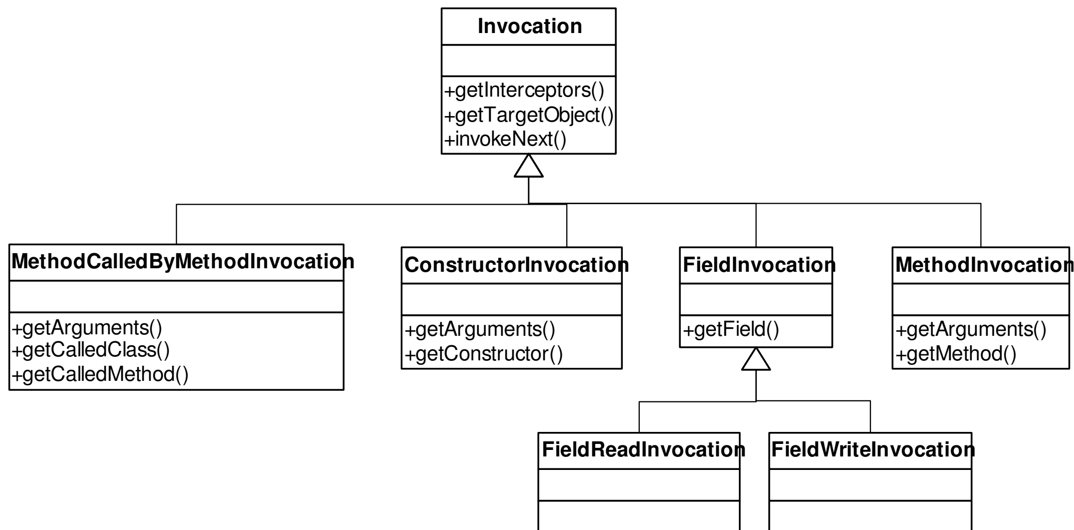

Figure 5.1 Extrait de la hiérarchie de classes org.jboss.aop.joinpoint.Invocation

La classe Invocation est la classe commune à tous les points de jonction. Elle possède une méthode getTargetObject qui retourne l'objet du point de jonction courant. La méthode getInterceptors retourne une liste de tous les intercepteurs greffés autour du point de jonction. La méthode invokeNext permet d'exécuter le code correspondant au point de jonction.

Chacune des sous-classes de Invocation correspond à un type de point de jonction différent :

- La classe MethodCalledByMethodInvocation représente les points de jonction de type appel de méthode. Ses méthodes getArguments, getCalledClass et getCalledMethod retournent respectivement les paramètres de l'appel, le nom de la classe appelée et la méthode appelée.

- La classe ConstructorInvocation correspond aux exécutions de constructeur. Sa méthode getArguments retourne les paramètres d'appel du constructeur. La méthode getConstructor dont le type de retour est

java.lang.reflect.Constructor fournit une représentation du constructeur *via* l'API de réflexion de Java.

- La classe FieldInvocation est associée aux points de jonction concernant les attributs. Ses deux sous-classes, FieldReadInvocation et FieldWriteInvocation, concernent respectivement les lectures et les écritures d'attributs. La méthode getField de la classe FieldInvocation retourne une représentation de l'attribut intercepté.

- La classe MethodInvocation correspond aux exécutions d'une méthode. Les paramètres d'appel de la méthode, ainsi qu'une représentation de celle-ci, peuvent être récupérés respectivement à l'aide des méthodes getArguments et getMethod.

Groupement de méthodes d'interception

Comme nous l'avons vu précédemment, les intercepteurs JBoss AOP doivent implémenter l'interface Interceptor et pour cela, fournir du code pour la méthode invoke. Comme il ne peut pas y avoir plusieurs méthodes avec la même signature dans une classe Java, cela implique qu'il ne peut y avoir qu'une seule méthode d'interception par intercepteur. Cette limitation peut être gênante lorsque de nombreuses méthodes d'interception sont nécessaires.

À partir de la version 1.0 beta, JBoss AOP offre un mécanisme de groupement de méthodes d'interception. Ce mécanisme permet de définir plusieurs méthodes d'interception dans une seule classe. Cette classe porte alors, dans la terminologie JBoss AOP, le nom d'aspect.

Notons cependant que ces aspects JBoss AOP sont différents de ceux d'AspectJ. Un aspect AspectJ comporte des coupes et des blocs de code advice. Un aspect JBoss AOP comporte seulement des blocs de code advice. Les coupes sont définies, comme c'est le cas pour les intercepteurs, dans un fichier **jboss-aop.xml**.

Un aspect JBoss AOP est une classe Java sans contrainte particulière (i.e. il n'y a ni classe de base à étendre, ni interface à implémenter). C'est simplement une classe comportant des méthodes d'interception. Le nom de ces dernières est libre, mais le reste de leur signature doit respecter les contraintes suivantes : un seul paramètre de type Invocation (ou d'un des sous-types de Invocation présentés à la section précédente) est autorisé, le type de retour doit être Object et l'exception Throwable doit être déclarée.

À titre d'exemple, le format général d'un aspect JBoss AOP définissant une méthode d'interception est :

```
package aop.jboss;

import org.jboss.aop.joinpoint.Invocation;

public class MyJBossAOPAspect {

   public Object methInterceptor(Invocation invocation)
      throws Throwable {

      System.out.println("Code avant");
```

```
        Object rsp = invocation.invokeNext();
        System.out.println("Code après");
        return rsp;
}  }
```

Un aspect JBoss AOP peut contenir autant de méthodes d'interceptions que l'on souhaite. Comme pour les intercepteurs, l'association entre une coupe et un aspect se fait via un fichier **jboss-aop.xml**. À titre d'exemple, le fichier suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<aop>
   <aspect class="aop.jboss.MyJBossAOPAspect"/>❶
   <bind pointcut="execution(void *->addItem(..))">❷
       <advice name="methInterceptor"❸
               aspect="aop.jboss.MyJBossAOPAspect"/>❹
   </bind>
</aop>
```

déclare que la classe aop.jboss.MyJBossAOPAspect est un aspect JBoss AOP (repère ❶), et associe cet aspect (repère ❹) à la coupe qui correspond à toutes les exécutions de la méthode addItem (repère ❷). La méthode methInterceptor (repère ❸) de l'aspect MyJBossAOPAspect prendra en charge toutes les interceptions de points de jonction de cette coupe.

Le mécanisme mix-in

Le mécanisme mix-in de JBoss AOP permet d'étendre le comportement d'une application. Ce mécanisme est identique au mécanisme d'introduction d'AspectJ.

Concrètement le mécanisme mix-in consiste à ajouter aux classes existantes de l'application des interfaces ou des attributs et des méthodes provenant d'une classe.

Le mécanisme mix-in atteint donc, par d'autres biais, le même objectif que l'héritage : il permet d'étendre une classe. Contrairement à l'héritage, cependant, il ne permet pas de redéfinir ou de surcharger des méthodes existantes.

Définition

Le mécanisme mix-in se définit dans le fichier **jboss-aop.xml** au moyen de la balise <introduction> associée à l'attribut class. Cet attribut est une expression régulière Java qui indique la ou les classes étendues par le mécanisme mix-in. La balise est suivie d'une balise <mixin>.

Le fragment de fichier XML suivant illustre la définition d'un mécanisme mix-in concernant la classe aop.jboss.Order :

```
<introduction class="aop.jboss.Order">
<mixin>
  ...
</mixin>
</introduction>
```

Nous complétons ce fragment de fichier à la section suivante (« Exemple »).

Les balises <introduction> cohabitent au sein des fichiers **jboss-aop.xml** avec les balises de définition de coupe que nous avons vues jusqu'à présent. Elles sont écrites, comme elles, entre les balises <aop> et </aop>, qui délimitent le début et la fin d'un fichier **jboss-aop.xml.**

Les points de suspension de l'exemple précédent accueillent la définition des éléments qui vont être ajoutés, ici dans la classe aop.jboss.Order. Deux types d'éléments peuvent être ajoutés : des interfaces et des classes. Chacun de ces éléments est associé à une balise XML.

La balise <interfaces> fournit les noms de la ou des interfaces ajoutées. Lorsqu'il y a plusieurs interfaces, leurs noms sont séparés par une virgule.

La balise <class> donne le nom de la classe à ajouter. Tous les attributs et toutes les méthodes de cette classe sont ajoutés à la ou aux classes initiales, ici à la classe aop.jboss.Order.

Conceptuellement, le mécanisme mix-in étend une classe existante. Cependant, en pratique, deux instances subsistent lorsque le programme s'exécute : celle de la classe initiale, non étendue, et celle de la classe correspondant à l'extension. Une troisième balise <construction> est disponible dans la définition d'un mécanisme mix-in pour indiquer comment cette seconde instance doit être créée. Concrètement, cette balise fournit, à l'aide de l'instruction new, la ligne de code qui permet de créer l'instance.

Par exemple, le fragment XML suivant :

```
<construction> new aop.jboss.Calendar(this) </construction>
```

indique que l'instance étendue doit être créée en appelant le constructeur de la classe aop.jboss.Calendar et en lui passant en paramètre la référence this.

La création de l'instance étendue s'effectue dans le contexte de l'instance initiale. La référence this correspond à la référence de l'instance initiale. Nous sommes de la sorte à même de transmettre à l'instance étendue la référence de l'instance initiale. L'instance étendue connaît la référence de l'instance qu'elle étend.

Exemple

Afin d'illustrer le mécanisme mix-in, nous allons étendre la classe Order de l'application Gestion de commande. Il s'agit de faire en sorte que toutes les commandes soient datées. Pour cela, nous allons définir une interface et une classe.

L'interface CalendarItf fournit la signature des méthodes que nous souhaitons ajouter à la classe Order. Il s'agit de deux méthodes permettant respectivement de positionner et de récupérer la date :

```
package aop.jboss;
import java.util.Date;
public interface CalendarItf {
  public void setDate(Date date);
  public Date getDate();
}
```

La classe Calendar fournit l'implémentation de l'interface CalendarItf :

```
package aop.jboss;
import java.util.Date;
public class Calendar implements CalendarItf {
  private Object initial;
  private Date date;
  public Calendar( Object initial ) {←❶
    this.initial = initial;
    date = new Date();
  }
  public void setDate(Date date) {
    this.date = date;
  }
  public Date getDate() {
    return date;
  }
}
```

En plus de fournir l'implémentation des méthodes setDate et getDate, la classe Calendar définit un attribut date et un constructeur (repère ❶) prenant en paramètre un objet. Ce constructeur est appelé par le framework JBoss AOP. Le paramètre contient alors la référence de l'objet initial étendu.

Le fichier **jboss-aop.xml** suivant permet de déclarer le mécanisme mix-in :

```
<?xml version="1.0" encoding="UTF-8"?>
<aop>
  <introduction class="aop.jboss.Order">
  <mixin>
    <interfaces> aop.jboss.CalendarItf </interfaces>
    <class> aop.jboss.Calendar </class>
    <construction> new aop.jboss.Calendar(this) </construction>
  </mixin>
  </introduction>
</aop>
```

L'interface Calendar et la classe CalendarItf sont ajoutées à la classe Order.

Les éléments introduits par le mécanisme mix-in sont principalement utilisés par les aspects. Ainsi, l'interface CalendarItf et les méthodes setDate et getDate de la classe Calendar sont utilisées dans des intercepteurs associés à une coupe concernant la classe Order.

De façon complémentaire, tout objet étendu par le mécanisme mix-in peut être converti vers le type correspondant à l'interface introduite. Dans l'exemple, les objets de la classe Order peuvent être convertis vers CalendarItf. Il est de la sorte possible à tout moment de tirer parti des méthodes introduites, ici setDate et getDate. Cette caractéristique est particulièrement intéressante lorsqu'une application, tout en étant indépendante de l'implémentation d'un aspect, sait qu'une interaction avec cet aspect doit être mise en œuvre. Il est alors possible de convertir tout objet applicatif vers l'interface introduite et d'utiliser les méthodes introduites par l'aspect.

Les métadonnées

Les concepts abordés jusqu'à présent nous ont permis d'écrire des programmes orientés aspect avec JBoss AOP. Nous avons pu écrire des coupes et associer des intercepteurs aux points de jonction désignés par ces coupes. Nous avons vu

également que le mécanisme mix-in permettait d'étendre le comportement d'une application.

Cette section s'intéresse aux métadonnées. Les métadonnées représentent des informations, donc des données, qui sont associées à une application. On leur associe le préfixe *méta* car ce sont des données qui ne sont pas manipulées directement par l'application mais qui apportent des informations décrivant l'application elle-même.

Les métadonnées ne sont pas propres à la POA. On les trouve dans d'autres domaines de l'informatique, comme les bases de données ou la programmation réflexive. Elles sont néanmoins très prisées en POA car elles permettent d'associer des informations concernant un aspect aux éléments d'une application (classe, méthode, attribut, etc.). Grâce à elles, nous pouvons par exemple désigner les méthodes dont les exécutions doivent être tracées par un aspect de trace ou les attributs dont les valeurs doivent être sauvegardées par un aspect de persistance.

Les métadonnées de JBoss AOP améliorent la réutilisabilité des aspects en adaptant chaque aspect au contexte particulier d'une application. En cela, elles jouent un rôle similaire à celui des fichiers de configuration d'aspect de JAC ou à celui des aspects abstraits d'AspectJ.

Définition des métadonnées

Les métadonnées sont définies en JBoss AOP à l'aide d'annotations écrites sous forme de commentaires dans le code source des programmes. Ces annotations sont similaires aux balises javadoc : elles sont attachées à des classes, des attributs ou des méthodes. Les annotations JBoss AOP sont précédées des symboles @@, elles ont un nom et éventuellement des attributs et des valeurs. Par exemple, la méthode message suivante :

```
/**
 * @@intl country=france code=fr
 * @@important
 */
public void message() { ... }
```

est attachée aux métadonnées intl et important. La seconde n'a pas d'attribut, tandis que la première en a deux, country et code, dont les valeurs respectives sont france et fr. Lorsque les valeurs contiennent des espaces, il est nécessaire de les écrire entre guillemets.

Une fois définies, les annotations vont pouvoir être exploitées dans la définition de coupes et dans les intercepteurs. Avant cela, il va être nécessaire de modifier le fichier Ant **build.xml** de compilation de l'application que nous avons fourni en début de chapitre. Le fichier à utiliser est le suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<project default="run" name="Gestion de commandes">

  <property name="jboss-aop.root" value="c:\jboss-aop-b1"/>

  <target name="prepare">
    <path id="classpath">
      <pathelement path="." />
      <fileset dir="${jboss-aop.root}">
```

```
        <include name="*.jar" />
      </fileset>
    </path>
    <taskdef name="aopc" classname="org.jboss.aop.ant.AopC"
             classpathref="classpath"/>
    <taskdef name="annotationc"➔❶
             classname="org.jboss.aop.ant.AnnotationC"
             classpathref="classpath"/>
  </target>

  <target name="compile" depends="prepare">
    <annotationc compilerclasspathref="classpath"➔❷
       classpathref="classpath" output="metadata-aop.xml"
       xml="true">
         <src path="."/>
    </annotationc>
    <javac srcdir="src" destdir="."
           debug="on" deprecation="on"
           optimize="off"
           includes="**">
      <classpath refid="classpath"/>
    </javac>
    <aopc compilerclasspathref="classpath" classpathref="classpath"
          verbose="true">
      <classpath path="."/>
      <src path="."/>
      <aoppath>
        <pathelement path="jboss-aop.xml"/>
        <pathelement path="metadata-aop.xml"/>➔❸
      </aoppath>
    </aopc>
  </target>

  <target name="run" depends="compile">
    <java fork="yes" failOnError="true"
          className="aop.jboss.Customer">
      <sysproperty key="jboss.aop.path"
                   value="metadata-aop.xml;jboss-aop.xml"/>➔❹
      <classpath refid="classpath"/>
    </java>
  </target>

</project>
```

Les modifications concernent la définition d'une nouvelle tâche annotationc (repère ❶), l'appel de cette tâche (repère ❷) afin de rassembler les annotations dans un fichier **metadata-aop.xml**, l'ajout de ce fichier lors des phases de tissage (tâche aopc, repère ❸) et lors de l'exécution du programme (repère ❹).

Utilisation des métadonnées

Une fois définies dans un programme, les métadonnées peuvent être utilisées dans la définition des coupes et dans les intercepteurs. Afin d'illustrer ces utilisations, nous allons annoter la classe Order de l'application Gestion de commande.

```
package aop.aspectj;

import java.util.*;

/**
 * @@annotatedClass (level=1)➔❶
```

```
 */
public class Order {

    private Map items = new HashMap();

    /**
     * @@toBeTraced🡐❷
     * @@intl (country=france,code=fr)🡐❸
     */
    public void addItem(String reference,int quantity) {
        items.put(reference,new Integer(quantity));
        System.out.println(
            quantity+" item(s) "+reference+
            " ajouté(s) à la commande" );
    }

    public double computeAmount() {
        double amount = 0.0;
        Iterator iter = items.entrySet().iterator()
        while ( iter.hasNext() ) {
            Map.Entry entry = (Map.Entry) iter.next();
            String item = (String) entry.getKey();
            Integer quantity = (Integer) entry.getValue();
            double price = Catalog.getPrice(item);
            amount += price*quantity.intValue();
        }
        return amount;
    }
}
```

Comme nous pouvons le constater, trois annotations ont été définies : annotatedClass (repère ❶) pour la classe Order, toBeTraced (repère ❷) et intl (repère ❸) pour la méthode addItem. Dans des cas plus complexes, une même annotation peut être associées à plusieurs classes, méthodes ou attributs.

Nous allons maintenant utiliser ces annotations dans la définition d'une coupe. De façon intuitive, les métadonnées associées aux méthodes peuvent être utilisées dans les coupes en lieu et place des noms de méthodes, tandis que les métadonnées associées aux classes peuvent naturellement remplacer les noms de classes. Par exemple, l'expression suivante :

```
<bind pointcut="execution(* aop.jboss.Order->@toBeTraced(..))" >
```

capture toutes les exécutions des méthodes de la classe Order qui sont annotées avec la métadonnée de toBeTraced. De même, l'expression :

```
<bind pointcut="execution(* @annotatedClass->@toBeTraced(..))" >
```

concerne toutes les exécutions de méthodes annotées avec la métadonnée toBeTraced et définies dans des classes annotées avec la métadonnée annotatedClass.

De façon complémentaire, les métadonnées peuvent également être exploitées dans les intercepteurs. La classe MyInterceptor2 suivante en fournit un exemple.

```
package aop.jboss;

import org.jboss.aop.Adviced;
import org.jboss.aop.ClassAdvisor;
import org.jboss.aop.advice.Interceptor;
import org.jboss.aop.joinpoint.Invocation;

public class MyInterceptor2 implements Interceptor {
  public String getName() { return "MyInterceptor2"; }
  public Object invoke(Invocation invocation) throws Throwable {
```

```
    String methodMetaData = (String)
       invocation.getMetaData("intl","country");←❶

    Advised obj = (Advised) invocation.targetObject;
    ClassAdvisor cadvisor = (ClassAdvisor) obj.getAdvisor();
    String classMetaData = (String)
        cadvisor.getClassMetaData().getMetaData(
            "annotatedClass","level");←❷

    System.out.println(methodMetaData);
    System.out.println(classMetaData);

    return invocation.invokeNext();
  }
}
```

Le paramètre de type Invocation fournit une méthode getMetaData. Comme son nom le suggère, cette méthode permet de récupérer les métadonnées associées à une invocation. Ici, il s'agit en l'occurrence d'une invocation de méthode. Le principe est identique pour les opérations sur les attributs. La méthode getMetaData accepte deux paramètres qui correspondent respectivement au nom et à l'attribut de la métadonnée que l'on souhaite interroger. Elle retourne la valeur de l'attribut ou la référence null si la métadonnée et/ou l'attribut n'existent pas. Ainsi, dans le programme précédent, nous interrogeons l'attribut country de la métadonnée intl (repère ❶). Dans le cas d'une interception concernant la méthode addItem, la valeur retournée sera france, tandis que la valeur sera null dans tous les autres cas.

La mécanique à mettre en œuvre pour récupérer les métadonnées associées aux classes nécessitent un peu plus d'étapes (voir le paragraphe de code qui se termine au repère ❷). De façon synthétique, tous les objets métiers d'un programme JBoss AOP sont associés à une instance de la classe ClassAdvisor qui permet, entre autres, d'accéder aux métadonnées de la classe. C'est ce que nous faisons dans l'exemple en interrogeant la valeur de l'attribut level de la métadonnée annotedClass.

Modification des métadonnées

Nous venons de voir comment définir et utiliser des métadonnées. Ce sont les deux opérations les plus couramment effectuées. En complément à cette présentation des métadonnées de JBoss AOP, la présente section montre comment des métadonnées peuvent être modifiées pendant l'exécution du programme.

JBoss AOP fournit une API qui permet de modifier les métadonnées d'un programme en cours d'exécution. Avant d'indiquer la procédure à suivre pour aboutir à cela, il est nécessaire de préciser que plusieurs niveaux de métadonnées existent dans JBoss AOP.

Les niveaux de métadonnées

Les métadonnées de JBoss AOP peuvent appartenir à l'un des niveaux suivants : invocation, thread, instance ou classe. Il y a une relation d'inclusion entre ces

niveaux puisqu'une invocation est contenue dans un thread qui lui-même concerne une instance qui appartient à une classe.

Le niveau invocation

Le niveau invocation est le plus fin. Les modifications faites sur les métadonnées à ce niveau ne sont prises en compte que pendant la durée de vie de l'invocation, c'est-à-dire tant que nous restons dans le corps de la méthode invoke d'un intercepteur. Dès que cette méthode se termine, les modifications sont perdues. De plus, si plusieurs invocations se déroulent en parallèle, les modifications de métadonnées faites dans une invocation ne sont pas vues par les autres.

La méthode setMetaData de la classe Invocation permet de modifier une métadonnée au niveau invocation. Elle prend en paramètre une instance de la classe org.jboss.aop.metadata.SimpleMetaData. Cette dernière possède une méthode addMetaData avec trois paramètres de type String : un nom de groupe de métadonnées, un nom de métadonnées et une valeur.

Les trois lignes de code suivantes dans une méthode invoke :

```
SimpleMetaData smd = new SimpleMetaData();
smd.addMetaData( "toBeTraced", "tracedMethod", "false" );
invocation.setMetaData(smd);
```

positionnent la valeur false pour la métadonnée tracedMethod du groupe toBeTraced.

Insistons sur le fait que dès que la méthode invoke se termine, cette modification est perdue.

Le niveau thread

Dans ce niveau, les métadonnées sont attachées à un thread. Lorsque l'application est multithread, chaque thread possède ses propres métadonnées. Les métadonnées de niveau thread existent tant que le thread s'exécute.

La méthode statique instance de la classe org.jboss.aop.metadata.ThreadMetaData retourne l'instance de la classe SimpleMetaData, qui permet de modifier les métadonnées de niveau thread.

Le code suivant effectue la même modification de métadonnée que précédemment, mais cette fois au niveau thread :

```
ThreadMetaData.instance().
  addMetaData( "toBeTraced", "tracedMethod", "false");
```

Le niveau instance

Dans ce niveau, les métadonnées sont attachées à une instance, c'est-à-dire à un objet de l'application.

Avec JBoss AOP, chaque objet de l'application peut être converti vers le type org.jboss.aop.Advised. Celui-ci fournit une méthode _getInstanceAdvisor, qui

retourne l'instance de SimpleMetaData correspondant aux métadonnées attachées à cet objet.

À titre d'exemple, nous allons modifier une métadonnée de l'objet applicatif contenant le point de jonction courant. Rappelons pour cela que, dans une méthode invoke, cet objet s'obtient à l'aide de l'attribut targetObject du paramètre invocation.

La modification de métadonnée au niveau instance s'effectue de la façon suivante :

```
Advised obj = (Advised) invocation.targetObject;
obj._getInstanceAdvisor().getMetaData().
   addMetaData( "toBeTraced", "tracedMethod", "false" );
```

Le niveau classe

Le niveau classe est le plus macroscopique. Les métadonnées de niveau classe sont connues de façon globale par toutes les instances de cette classe et sont accessibles dans tous les threads, quelle que soit l'invocation.

La modification de métadonnées au niveau classe s'effectue de la manière suivante :

```
Advised obj = (Advised) invocation.targetObject;
ClassAdvisor advisor = (ClassAdvisor) obj.getAdvisor();
ClassMetaDataBinding cmd =
  new SimpleClassMetaDataBinding(
    new SimpleClassMetaDataLoader(),
    "toBeTraced","tracedMethod","false");
advisor.addClassMetaData(cmd);
```

Fonctionnalités avancées

Ce panorama de JBoss AOP ne serait pas complet sans la présentation de fonctionnalités avancées. Ces fonctionnalités concernent l'instanciation, la configuration, l'ordonnancement des intercepteurs et les piles de méthodes .

Instanciation des intercepteurs

Par défaut, JBoss AOP crée une instance d'intercepteur par classe incluse dans une coupe. Il y a deux façons de modifier ce comportement par défaut. La première consiste à indiquer que l'intercepteur est un singleton, et la seconde à fournir une classe dite factory pour contrôler finement la façon dont les intercepteurs sont instanciés.

Singleton

La première technique permettant de modifier la façon dont JBoss AOP instancie les intercepteurs consiste à indiquer que l'intercepteur est un singleton. Dans ce cas, chaque classe d'intercepteur est associée à une seule instance. Celle-ci est chargée de traiter les interceptions pour tous les points de jonction de

l'application. Cette technique est utile si nous souhaitons partager des données définies au niveau de l'intercepteur entre tous les points de jonction de l'application.

Un intercepteur singleton se définit dans le fichier **jboss-aop.xml** à l'aide de l'attribut singleton de la balise <interceptor>.

Le fragment XML suivant :

```
<interceptor name="myInter" class="aop.jboss.MyInterceptor"
   singleton="true" />
```

indique que l'intercepteur myInter implémenté par la classe aop.jboss.MyInterceptor est un singleton.

Factory

La seconde technique permettant de modifier la façon dont JBoss AOP instancie les intercepteurs consiste à fournir une classe dite factory chargée de créer des instances d'intercepteurs de façon personnalisée.

La classe factory est invoquée par le framework JBoss AOP chaque fois qu'il a besoin d'une instance d'intercepteur. La classe factory peut alors choisir de lui en fournir une nouvelle ou de lui fournir un intercepteur existant.

La classe factory doit implémenter l'interface org.jboss.aop.InterceptorFactory et fournir un constructeur sans argument. La classe InterceptorFactory définit une seule méthode create. Celle-ci a un paramètre de type org.jboss.aop.Advisor qui permet d'obtenir des métadonnées associées à la classe pour laquelle JBoss AOP souhaite un intercepteur. Le type de retour de la méthode create est Interceptor.

La classe suivante fournit un exemple de classe factory :

```
package aop.jboss;

import org.jboss.aop.Advisor;
import org.jboss.aop.Interceptor;
import org.jboss.aop.InterceptorFactory;

public class TraceInterceptorFactory implements InterceptorFactory{
  private index;
  public TraceInterceptorFactory() {}
  public Interceptor create(Advisor advisor) {
    return new TraceInterceptor(index++);
  }
}
```

Cette classe TraceInterceptorFactory crée une nouvelle instance de l'intercepteur TraceInterceptor à chaque appel de la méthode create. Chaque intercepteur reçoit un numéro unique *via* l'attribut index qui est incrémenté pour chaque nouvel intercepteur.

Les classes factory se définissent dans le fichier **jboss-aop.xml** à l'aide de l'attribut factory associé à la balise <interceptor>.

Le fragment XML suivant :

```
<interceptor name="myInter" class="aop.jboss.MyInterceptor"
   factory="aop.jboss.TraceInterceptorFactory" />
```

indique d'utiliser la classe aop.jboss.TraceInterceptorFactory comme classe factory pour l'intercepteur myInter.

Configuration des intercepteurs

Nous avons vu que le fichier **jboss-aop.xml** permettait de définir des coupes, des intercepteurs, le mécanisme mix-in et des métadonnées. Nous allons voir dans cette section qu'il permet également de configurer les intercepteurs.

La balise <interceptor> permet de définir un intercepteur. Trois attributs peuvent lui être associés : name, class, factory et singleton. JBoss AOP permet d'inclure entre les balises <interceptor> et </interceptor> n'importe quel fragment XML syntaxiquement correct. Ce fragment n'est pas exploité directement par JBoss AOP mais fait partie de la configuration de l'intercepteur. L'idée est de pouvoir passer à l'intercepteur des paramètres de configuration définis en XML.

Par exemple, la déclaration suivante :

```
<interceptor name="myInter" class="aop.jboss.MyInterceptor"
   factory="aop.jboss.TraceInterceptorFactory2">
   <type>verbose</type>
   <level>10</level>
</interceptor>
```

définit une configuration de l'intercepteur myInter comprenant deux balises XML, <type> et <level>. Ces balises sont associées respectivement aux valeurs verbose et 10. Les noms des balises ne sont pas imposés par JBoss AOP. Chaque développeur est donc libre de choisir les noms qu'il souhaite dès lors que ces derniers n'entrent pas en conflit avec les balises existantes de JBoss AOP.

Pour pouvoir exploiter les balises XML de configuration, les classes factory doivent implémenter l'interface org.jboss.util.xml.XmlLoadable. Cette interface définit la méthode importXml suivante :

```
public void importXml( org.w3c.dom.Element element )
  throws Exception;
```

La méthode importXml est appelée par le framework JBoss AOP. Le paramètre element contient l'arbre XML DOM de la balise <interceptor>. Il est alors possible de récupérer, *via* l'API org.w3c.dom, l'ensemble des balises incluses dans la balise <interceptor>, notamment les balises <type> et <level>. L'API org.w3c.dom est fournie en standard dans J2SE à partir de la version 1.4.

La classe TraceInterceptorFactory2 suivante illustre l'utilisation des paramètres de configuration :

```
package aop.jboss;

import org.jboss.aop.Advisor;
import org.jboss.aop.Interceptor;
import org.jboss.aop.InterceptorFactory;
import org.jboss.util.xml.XmlLoadable;
import org.w3c.dom.Element;

public class TraceInterceptorFactory2
  implements InterceptorFactory, XmlLoadable {

  private index;

  public TraceInterceptorFactory() {}
```

```
  public Interceptor create(Advisor advisor) {
    return new TraceInterceptor(index++);
  }

  public void importXml( org.w3c.dom.Element element )
    throws Exception {

    Element type =
      (Element) element.getElementsByTagName("type").item(0);
    String typeValue = type.getFirstChild().getNodeValue();

    Element level =
      (Element) element.getElementsByTagName("level").item(0);
    String levelValue = type.getFirstChild().getNodeValue();
  }
}
```

La méthode getElementByTagName permet de récupérer un élément fils. Ici, nous récupérons successivement les fils des noms type et level. Plusieurs éléments fils pouvant correspondre, l'appel de la méthode item(0) permet de ne retenir que le premier. Les valeurs des éléments type et level sont à nouveau des fils de ces éléments. Nous les récupérons à l'aide de l'appel getFirstChild().getNodeValue().

Avec le fichier **jboss-aop.xml** précédent, les variables typeValue et levelValue contiennent respectivement, à la fin de la méthode importXml, verbose et 10.

Ordonnancement des intercepteurs

Nous avons vu précédemment que le mécanisme de pile d'intercepteurs permettait d'associer plusieurs intercepteurs à une coupe. Les intercepteurs sont alors exécutés dans leur ordre de déclaration dans la pile.

Il se peut qu'un même point de jonction soit désigné par plusieurs coupes. Dans ce cas, il est associé à plusieurs intercepteurs ou à plusieurs piles. L'ordre d'exécution des intercepteurs et des piles correspond à l'ordre de déclaration de la coupe dans le fichier **jboss-aop.xml.** Les intercepteurs ou les piles de la coupe apparaissant en premier sont d'abord appliqués, puis vient le tour des piles ou des intercepteurs des coupes suivantes, toujours dans leur ordre de déclaration.

Piles de méthodes

Les piles de méthodes permettent de spécifier des suites d'appels de méthodes. Ces suites vont pouvoir être utilisées dans la définition des coupes pour filtrer les points de jonction en fonction du contexte d'exécution. Cela permet de ne retenir des points de jonction que s'ils se trouvent dans le contexte d'exécution d'autres méthodes. Ce mécanisme est similaire à celui fourni par l'opérateur cflow de AspectJ.

Une pile de méthode se définit dans un fichier **jboss-aop.xml** à l'aide de la balise <cflow-stack>. Par exemple, la déclaration suivante :

```
<cflow-stack name="foo">
  <called expr="void A->a()" />
  <called expr="void B->b()" />
</cflow-stack>
```

définit une pile foo constituée par les méthodes a de la classe A et b de la classe B. Cette pile peut être utilisée dans une coupe à l'aide de l'attribut cflow. Par exemple :

```
<bind pointcut="..." cflow="foo" >
```

définit une coupe associée à la pile foo. Seuls les points de jonction correspondant à l'expression de coupe (ici non spécifiée et désignée par des points de suspension) et se trouvant dans un contexte d'exécution tel que la méthode a a été appelée, puis b (i.e. a et b se trouvent dans la pile d'exécution dans cet ordre) seront retenus.