La Programmation Orientée Aspect et AspectJ: Présentation et Application dans un Système Distribué.

Jean Baltus

Facultés Universitaires Notre-dame de la Paix,
Namur, Belgique
jbaltus@info.fundp.ac.be

Abstract. Les systèmes informatiques répondent à diverses exigences. L'implémentation de certaines exigences dans les langages actuels n'est pas toujours bien circonscrite. On se retrouve donc avec des problématiques dont l'implémentation se retrouve un peu partout dans les modules du système. C'est ce qu'on appelle des préoccupations qui se recoupent ("crosscutting concerns" dans la littérature américaine). La programmation orientée aspect apporte une solution élégante et simple à ce problème. Cette technique de programmation permet d'implémenter chaque problématique indépendamment des autres puis de les assembler selon des règles bien définies. AspectJ, une extension à Java, permet d'ajouter les fonctionnalités nécessaires à ce dernier pour obtenir un langage orienté aspect. Nous pouvons donc utiliser Java pour implémenter toutes les préoccupations et ensuite les nouvelles fonctionnalités d'AspectJ pour définir les règles d'assemblement. Un exemple concret d'application dans un système distribué sera présenté pour illustrer tous ces concepts.

1 Introduction

La plupart des logiciels actuels répondent à diverses exigences. Nous pouvons distinguer les exigences fonctionnelles des exigences non-fonctionnelles. Les exigences fonctionnelles décrivent le comportement du système et définissent les fonctions ou services que le système doit remplir. Les exigences non-fonctionnelles expriment les qualités ou contraintes imposées sur la manière de satisfaire les exigences fonctionnelles. Il s'agit donc principalement des exigences de performance, sécurité, sûreté, convivialité, concurrence, etc. Ces différents aspects du logiciel ne constituent pas une fin en soi et assurent donc un rôle de support aux fonctionnalités premières d'un système.

L'implémentation des exigences non-fonctionnelles pose souvent problème avec les méthodes de programmation actuelles car il est difficile d'en limiter la portée dans un module bien circonscrit. On se retrouve donc avec des problématiques dont l'implémentation se retrouve un peu partout dans les différents modules fonctionnels du système. C'est ce qu'on appelle des préoccupations qui se recoupent ("crosscutting concerns" dans la littérature américaine).

La programmation orientée aspect apporte une solution élégante et simple à ce problème. Cette nouvelle méthode de programmer permet d'implémenter chaque problématique indépendamment des autres, puis, de les assembler selon des règles bien définies. La programmation orientée aspect promet donc une meilleure productivité, une meilleure réutilisation du code et une meilleure adaptation du code aux changements.

Cet article se veut être une présentation intuitive de la programmation orientée aspect. Le but n'est donc pas d'être exhaustif ni de rentrer en profondeur dans les détails d'un langage mais plutôt d'expliquer cette nouvelle méthode sur base d'exemples pour en cerner les avantages potentiels. Un style rédactionnel compréhensible et facilement lisible a été délibérément choisi. Je présenterai donc, dans la suite de cet article, les problèmes de recoupement du code ainsi que les solutions actuelles en la matière. Nous verrons que ces solutions sont loin d'être parfaites et qu'elles ne font que déplacer le problème. J'introduirai ensuite les concepts clés de la programmation orientée aspect et montrerai les avantages qu'elle offre sur base d'un exemple. Je présenterai ensuite le langage orienté aspect le plus abouti actuellement, AspectJ, et ses fonctionnalités. Finalement, un exemple réel d'application dans un système distribué sera présenté pour concrétiser les idées.

2. Les Préoccupations qui se Recoupent

Tout système informatique peut être vu comme un ensemble de préoccupations. Une préoccupation est en fait un but particulier, une problématique d'un logiciel qui correspond à une de ses exigences. Dans la plupart des implémentations actuelles, on procède à ce qu'on appelle la découpe fonctionnelle du système, càd que l'on divise le système en modules, représentant chacun une fonctionnalité particulière. Les exigences non-fonctionnelles sont difficilement prises en compte dans une telle découpe et on se contente donc de les intégrer dans les différents modules fonctionnels. La figure 1 montre un exemple d'une application composée de 3 modules fonctionnels dans lesquels on a du y insérer l'implémentation d'exigences de performance, de journalisation (enregistrement d'informations dans un fichier journal) et de synchronisation.

2.1 Problèmes Actuels

Les exigences non-fonctionnelles qui traversent la modularisation fonctionnelle du système posent deux problèmes qui apparaissent clairement sur la figure 1 :

– Premièrement, la *dispersion du code* traitant un aspect du système à travers différents modules. Considérons un langage orienté objet, l'unité fonctionnelle est le package et si nous raffinons encore la découpe, cette unité est la classe. Il n'est pas rare de voir des méthodes traitant d'une exigence non-fonctionnelle se disperser dans l'implémentation des différentes classes composant le cœur fonctionnel du système. Par exemple, dans un système de gestion de base de données il se peut que la performance, la journalisation et la synchronisation concernent toutes les

classes accédant à la base de données. On voit donc que ces aspects seront implémentés dans plusieurs modules sans être bien circonscrits. Il s'agit très souvent de portions de code très similaires à ajouter un peu partout dans les modules concernés.

– Deuxièmement, l'*enchevêtrement du code* de différentes problématiques. Ce problème découle directement du constat précédent: certaines préoccupations non-fonctionnelles viennent recouper l'implémentation des préoccupations fonctionnelles. Il en résulte la présence d'éléments de plusieurs problématiques dans l'implémentation d'un seul et même module. Reprenant l'exemple précédent, le code qui a trait à l'aspect performance, journalisation et synchronisation viendra s'enchevêtrer dans les classes d'accès à la base de données.

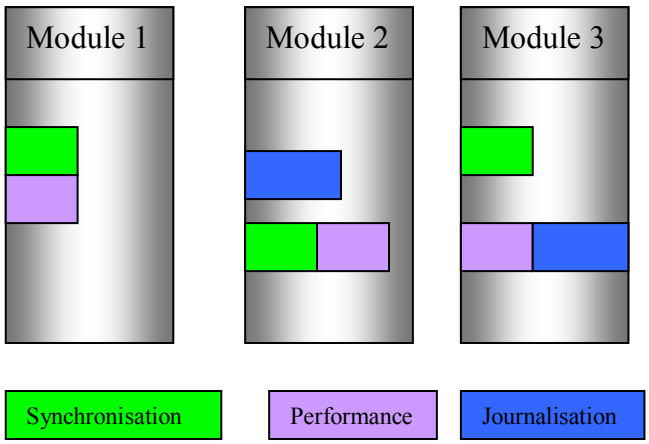

Fig. 1. Les exigences non-fonctionnelles traversent la modularisation fonctionnelle du système

Ces 2 problèmes entraînent des conséquences négatives sur le développement d'un logiciel [5]:

– traçage difficile: les différentes préoccupations d'un logiciel deviennent difficilement identifiables dans l'implémentation. Il en résulte une correspondance assez obscure entre les exigences et leurs implémentations.
– diminution de la productivité: la prise en considération de plusieurs exigences au sein d'un même module empêche le programmeur de se focaliser uniquement sur son but premier. Le danger d'accorder trop ou pas assez d'importance aux aspects accessoires d'un module en découle directement.
– diminution de la réutilisation du code: dans les conditions actuelles, un module implémente de multiples exigences. D'autres systèmes nécessitant des fonctionnalités similaires pourraient ne pas pouvoir réutiliser le module tel quel, entraînant de nouveau une diminution de la productivité à moyen terme.

- diminution de la qualité du code: les programmeurs ne peuvent pas se concentrer sur toutes les contraintes à la fois. L'implémentation disparate de certaines préoccupations peut entraîner des effets de bords non-désirés, i.e. des bugs.
- maintenance et évolutivité du code difficile: lorsque l'on veut faire évoluer le système, on doit modifier de nombreux modules. Modifier chaque sous-système pour répercuter les modifications souhaitées peut conduire à des incohérences.

2.2 Solutions Actuelles et leurs Désavantages

Diverses solutions ont été développées pour palier à ces problèmes. Le développement de "design patterns" tels que le "template", le "decorator" et le "visitor" permettent de séparer ou de déférer l'implémentation de certains aspects.

D'une manière générale, le "decorator pattern" permet d'ajouter un service, une responsabilité supplémentaire à une classe existante, i.e. de la décorer. Ce pattern pourrait par exemple être utilisé pour ajouter un service de journalisation (journal d'activités) ou de cache (exigence de performance) à une fonctionnalité existante du système.

Le "visitor pattern" est encore plus souple et permet, lui, d'ajouter des comportements à une classe existante sans y toucher. On crée ainsi une classe Visitor qui va aller visiter une classe et effectuer des opérations sur les données de cette dernière. Bien que cette approche semble aller à l'encontre de la philosophie orientée objet, il est parfois intéressant d'y recourir, notamment lorsque nous n'avons pas le contrôle des classes auxquelles nous voudrions ajouter des fonctionnalités.

Le "template pattern" permet de déférer l'implémentation de certains comportements d'une classe. Nous pourrions, par exemple, avoir une classe abstraite principale effectuant ses opérations dans un ordre prédéfini mais laissant l'implémentation d'un journal d'activités à une sous-classe concrète. On peut alors créer différentes sous-classes ayant chacune une implémentation différente du journal. La classe abstraite implémente donc sa fonctionnalité première sans se soucier des exigences secondaires et les sous-classes concrètes se focalisent sur l'implémentation de ces dernières.

Le problème avec ces solutions est que le nombre de lignes de code augmente facilement et qu'elles sont toujours soumises à certaines contraintes. Par exemple, pour le "decorator pattern", les classes implémentant les décorations doivent avoir une interface commune avec la classe qu'ils décorent. Ainsi, le contrôle des opérations reste dans les classes principales. Il y a donc toujours une trace d'exigences qui se recoupent. Si un programmeur veut implémenter ces patterns, il doit le décider à l'avance dans l'architecture de son système.

D'autres solutions "domain-specific" comme les cadres d'application (frameworks) ou les serveurs d'application permettent aux développeurs d'encapsuler des préoccupations qui se recoupent. Ainsi les "Enterprise Java Beans" (EJB) traitent certains aspects tels que la sécurité, la performance et la persistance dans des modules bien circonscrits. Ces solutions sont, par définition, spécialisées dans la résolution de problèmes spécifiques et ne couvrent pas toutes les exigences que l'on pourrait rencontrer dans une application. Les aspects non-pris en compte par ces architectures posent donc toujours problèmes et nécessitent une réponse adéquate.

3. La Programmation Orientée Aspect

La programmation orientée aspect (OA) apporte une solution élégante et facile à implémenter aux problèmes de recoupement et aux conséquences négatives qui en résultent.

3.1 Description

La programmation OA est une méthode de programmation qui permet de séparer l'implémentation de toutes les exigences, fonctionnelles ou non, d'un logiciel. Le principe est donc de coder chaque problématique séparément et de définir leurs règles d'intégration pour les combiner en vue de former le système final.

Par rapport à l'orienté objet, cette nouvelle technique permet aux programmeurs d'encapsuler des comportements qui affectaient de multiples classes dans des modules réutilisables. La programmation OA permet donc d'encapsuler dans un module les préoccupations qui se recoupent avec d'autres.

Pour mieux comprendre ce principe, nous pouvons reprendre la description originale faite par Gregor Kiczales et ses collègues [3]. Une préoccupation qui doit être implémentée est soit:

- un composant si elle peut clairement être encapsulée dans un objet, ou un module. Les composants sont donc, par définition, des unités fonctionnelles d'un système. Ex : les services imprimantes, accès aux bases de données d'une librairie digitale.
- un aspect si elle ne peut pas être clairement encapsulée dans un composant. Les aspects correspondent donc souvent aux exigences non-fonctionnelles d'un système. Ex : la synchronisation, la performance et la fiabilité d'une librairie digitale.

En utilisant ces termes, on entrevoit le but de la programmation OA : aider le programmeur à séparer clairement les aspects et les composants les uns des autres en offrant des mécanismes qui permettent de les abstraire et de les composer pour obtenir le système général. On sent donc tout de suite la différence avec les langages orientés objet qui ne permettent, eux, que de séparer les composants, rendant impossible l'abstraction claire des aspects.

Avec la programmation OA, chaque problématique est implémentée de façon complètement indépendante, sans se soucier de l'existence des autres problématiques. Dans l'implémentation d'une librairie digitale en OA, les composants d'accès à la base de données ne sont absolument pas au courant qu'ils doivent être synchronisés. Le programmeur implémente l'aspect de synchronisation avec ses règles d'intégration séparément.

3.2 Etapes de Développement d'une Application Orientée Aspect

Maintenant que nous avons défini la programmation orientée aspect et ses buts, voyons ce que cela implique dans le processus de développement d'un logiciel.

L'implémentation d'une application orientée aspect peut se dérouler en 3 étapes comme illustré à la figure 2 :

1. La décomposition des éléments du système. Il s'agit donc d'identifier tous les composants et aspects. On sépare toutes les préoccupations, qu'elles soient fonctionnelles ou non.
2. L'implémentation de chaque préoccupation. Chaque problématique sera codée séparément dans un composant ou un aspect. Dans un aspect, le programmeur définit aussi les règles d'intégration de l'aspect avec les composants concernés.
3. L'intégration du système. Tout langage OA offre un mécanisme d'intégration appelé le "weaver". Le "weaver", à l'image d'un métier à tisser, va donc composer le système final sur base des règles et des modules qui lui ont été donnés.

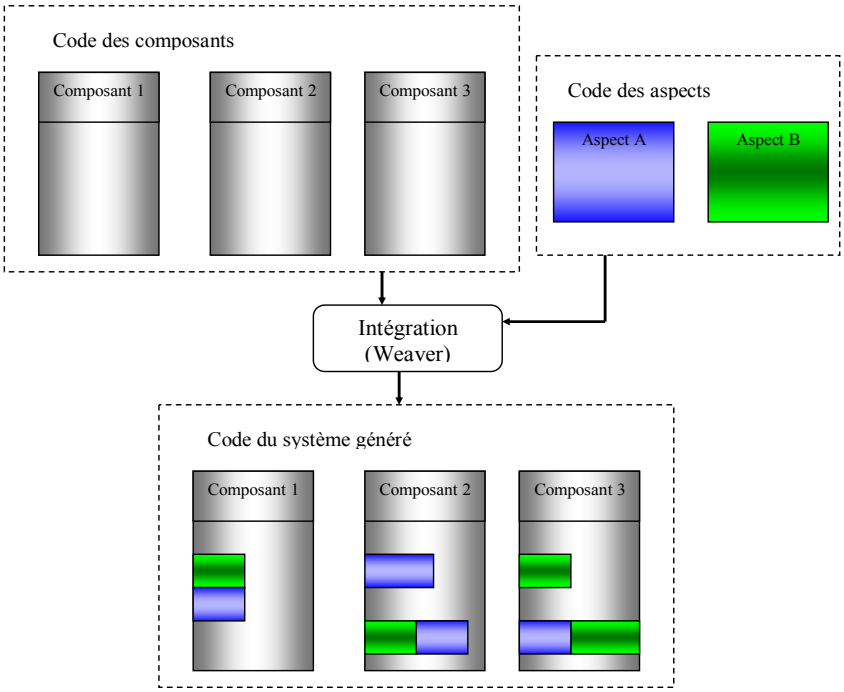

Fig. 2. Intégration du code des composants et des aspects pour former le système final

3.3 Exemple d'Utilisation

Pour bien illustrer ces étapes, je propose de comparer l'implémentation d'un système très simple de transaction bancaire dans un langage orienté objet avec son implémentation orientée aspect.

Pour simplifier l'exemple à l'extrême, nous ne considérerons que deux préoccupations : le processus de transaction et l'enregistrement des transactions dans un fichier journal.

Si nous prenons en considération le fait que toutes les transactions doivent être enregistrées dans un fichier journal, dans une implémentation Java, le code résultant ressemblerait à ceci (nous supposons que la classe Log existe déjà) :

```
public class ProcessusTransaction {
 //champs privés
 //champs concernant le fichier journal

  public void effectuerTransaction(Information info){
    Log.enregistrer("tentative transaction :", info);
    try {
      //effectuer les opérations fonctionnelles du système
      Log.enregistrer("transaction effectuée:", info);
    }
    catch (Exception ex){
      Log.enregistrer("transaction interrompue:", info, ex);
    }
  }

  //autres méthodes similaires
}
```

On réalise assez vite que le code ayant trait au journal de bord gène l'implémentation, tant dans sa rédaction que dans sa lecture ou son évolution. Les méthodes de la classe `ProcessusTransaction` s'occupent de choses qui dépassent le but principal auxquelles elles sont vouées. Avec un langage orienté aspect, on pourrait séparer l'implémentation des deux préoccupations. Nous pourrions alors réécrire le module principal comme suit :

```
public class ProcessusTransaction {
 //champs privés

 public void effectuerTransaction(Information info)
 throws Exception{
    //effectuer les opérations fonctionnelles du système
 }
 //autres méthodes similaires
}
```

On voit que la lisibilité du code en est améliorée; la classe ne s'occupe plus que des transactions, sans se soucier d'un quelconque enregistrement dans un journal. Pour que le système soit complet, il nous reste à définir l'aspect de journalisation avec ses règles d'intégration. Par exemple, l'aspect suivant assure que chaque fois que la méthode `effectuerTransaction` de la classe `ProcessusTransaction` est appelée, on enregistre les informations passées en argument.

```
public aspect TransactionLogging {
  Log log = new Log("fichier");
```

```
    pointcut appelTransaction(Information info):
      call(void
      ProcessusTransaction.effectuerTransaction(Information))
      &&  args(info);

    before(Information info): appelTransaction (info) {
      log.enregistrer("Tentative transaction:" + info);
    }
  }
```

D'une manière similaire, nous pouvons enregistrer dans le fichier journal les fins de transactions ainsi que les informations relatives aux exceptions renvoyées par la méthode `effectuerTransaction`.

La section 4 de cet article apportera des éclaircissements sur la syntaxe et la sémantique de ce bout de code implémenté en AspectJ, un langage orienté aspect. On remarque simplement que l'on a découplé le composant de transaction bancaire de son aspect journalisation.

Cette implémentation est donc beaucoup plus flexible. Si l'on veut maintenant généraliser les règles pour qu'elles s'appliquent à toutes les méthodes de la classe, nous n'avons plus qu'à aller changer le code à un seul endroit.

Pour obtenir le code final tel que présenté précédemment, il rester à intégrer le composant `ProcessusTransaction` avec l'aspect `TransactionLogging`. Ceci s'effectue simplement en lançant le processus d'intégration (le "weaver"). Par exemple, dans le langage AspectJ présenté par après, il suffit de lancer une commande "`ajc`" qui prend en argument tous les fichiers concernés et crée le code objet directement interprétable par une machine virtuelle Java.

3.4 Avantages Promis par la Programmation Orientée Aspect

Comme nous l'avons vu, l'implémentation séparée des préoccupations permet d'éviter la duplication du code. Chaque module adressant toujours une seule problématique, il est beaucoup plus facile de comprendre le code et de procéder à des changements.

Les systèmes orientés aspects sont beaucoup plus évolutifs pour deux autres raisons également. Premièrement, vu que les modules ne sont pas au courant des problématiques qui se recoupent, il est plus facile d'ajouter des nouvelles fonctionnalités au système. Deuxièmement, si on veut plus tard rajouter une exigence non-fonctionnelle comme la synchronisation, il suffit de créer un nouvel aspect qui s'en occupe, sans toucher au code des composants existants. Certains langages OA permettent même d'ajouter des aspects pendant que le programme s'exécute.

La programmation OA découplant les modules, leur réutilisation en est facilitée. Un aspect de journalisation générique pourrait être réutilisé dans plusieurs applications sans devoir y apporter une quelconque modification.

Le design du système à implémenter est aussi facilité par l'OA. Avec les langages traditionnels, le concepteur doit prévoir à l'avance les éléments susceptibles de changer dans l'architecture de son système. En résulte ce que certains appelle la "paralysie par l'analyse", i.e. le fait qu'un développeur ne sache jamais quand s'arrêter dans la spécification du design de son système. La programmation OA diminue

fortement ce risque puisqu'il est toujours possible de rajouter des aspects qui n'avaient pas nécessairement été prévu à l'avance.

Dernier avantage, une étude récente [8] a mis en évidence le fait que les programmeurs trouvent plus facilement la cause du problème en cas de bug dans le code source d'un langage orienté aspect. Les développeurs passent moins de temps à essayer de comprendre la sémantique des instructions et gagnent donc un temps considérable.

4. AspectJ

Après, je l'espère, avoir suscité votre intérêt pour la programmation orientée aspect, je propose, à titre d'illustration, d'étudier un langage OA particulier : AspectJ.

Tout comme n'importe quel langage traditionnel, AspectJ est composé de deux parties [6]: une spécification qui décrit la syntaxe et la sémantique du langage et une implémentation qui vérifie la conformité des codes sources aux spécifications et les transforme en code exécutable ou interprétable par une machine.

AspectJ est en fait une extension à Java. Les composants sont toujours écrits en Java pur. Grâce aux nouvelles possibilités offertes par la syntaxe d' AspectJ, nous pouvons définir des aspects accompagnés de leurs règles d'intégration. Ces règles s'expriment sous forme de points de jointure, "pointcuts" et "advice", concepts expliqués dans les sections qui suivent. J'utiliserai ici la terminologie anglaise pour faire le lien avec la syntaxe d'AspectJ. Un aspect est donc un mélange de règles AspectJ et d'instructions Java qui modularise une préoccupation.

AspectJ dispose de son propre compilateur qui permet de combiner les composants et les aspects entre-eux et de convertir le code source obtenu en code interprétable par une machine virtuelle Java.

4.1 Les Points de Jointure

Les points de jointure désignent des points précis dans l'exécution d'un programme. A la différence des "break point" utilisés pour le débugage, il s'agit ici de points bien précis et exploitables comme l'appel d'une méthode, l'affectation d'une variable, etc.

AspectJ permet de définir les points de jointure suivants :

- un appel de méthode ou de constructeur (on se trouve dans le contexte de l'appelant)
- l'exécution d'une méthode ou d'un constructeur (on se trouve dans le contexte de l'exécution, donc de l'appelé)
- l'accès en lecture ou écriture d'un champ
- l'exécution d'un bloc "catch" qui traite une exception Java
- l'initialisation d'un objet ou d'une classe (càd des membres statiques d'une classe)

4.2 Les PointCuts.

Les pointcuts correspondent à la définition syntaxique d'un ensemble de points de jointure, plus, éventuellement, certaines valeurs dans le contexte d'exécution ce ces points de jointure. Voici une des formes les plus simples de définition de pointcut :

```
pointcut <nom du pointcut> :
 call ( <signature de méthode> )
```

Ce pointcut définit tous les points de jointures qui correspondent à l'appel (`call`) d'une méthode dont la signature correspond à `<signature de méthode>`. Dans la suite du programme, on pourra donc y référer en utilisant le nom `<nom du pointcut>`.

Reprenons l'exemple de la section 3.3 de cet article; le pointcut suivant avait été défini:

```
pointcut appelTransaction(Information info):
    call(void
    ProcessusTransaction.effectuerTransaction(Information))
    &&  args(info);
```

Le nom du pointcut est `appelTransaction`. Nous lui avons associé un contexte `info` qui pourra être utilisé par les advices le concernant. La définition se trouve après les deux points. Il s'agit ici d'un appel de méthode (`call`) dont la signature est `void ProcessusTransaction.effectuerTransaction(Information)`. `void` indique que la méthode capturée doit avoir `void` comme type de retour. `ProcessusTransaction.effectuerTransaction` spécifie le nom de la méthode qui doit être capturée dans la classe `ProcessusTransaction`. Ensuite, `(Information)` signifie que la méthode capturée doit prendre en argument un objet de type `Information`. Finalement, `args(info)` signifie que le pointcut expose l'argument `info` pour que les advices puissent effectuer des opérations dessus (cf. section suivante).

Quelques exemples de poincuts particuliers sont présentés dans le tableau suivant :

Table 1. Exemples de pointcuts et leurs significations

Pointcut	Description
`call(void MaClasse.maMethode(..))`	Appel de `maMethode` de `MaClasse` prenant un nombre quelconque d'arguments, avec comme type de retour `void` et n'importe quel droit d'accès (public, private, etc…)
`call(* MaClasse.maMethode(..))`	Appel de `maMethode()` de `MaClasse` prenant un nombre quelconque d'arguments, avec n'importe quel type de retour
`call( MaClasse.new(String))`	Appel du constructeur de `MaClasse` avec un argument de type `String`

`execution (void MaClasse.maMethode())`	Exécution de `maMethode()` de `MaClasse` retournant `void`.
`set(int MaClasse.x)`	Ecriture dans le champ `x` de type entier `(int)` de la classe `MaClasse`
`get(String Toto.nom)`	Lecture du champ `nom` de type `String` de la classe `Toto`
`Handler(IOException)`	Exécution d'un bloc "`catch`" (en java) qui traite une exception de type `IOException`

Nous pouvons compléter tous ces exemples avec des contraintes supplémentaires sur le contexte d'exécution de ces points de jointure. Par exemple, la définition

```
pointcut appelMethode(Foo in):
    call(SuperFoo.methode())
```

peut –être complétée par :

```
 &&  target(in);
```

Ceci signifie que l'appel `methode()` doit être effectué sur un objet du même type que `in`, càd de type `Foo`. Ce genre de contrainte impliquerait donc que `Foo` soit une sous-classe de `SuperFoo`.

La même contrainte peut être appliquée sur les pointcuts définissant l'exécution d'une méthode, dans ce cas, nous utiliserons le mot réservé `this` à la place de `target`. Exemple:

```
pointcut appelMethode(Foo in):
    execution (SuperFoo.methode())  &&  this(in);
```

4.3 Les Advices.

Un advice est un mécanisme (similaire à une méthode) utilisé pour déclarer l'exécution de code à tous les points de jointure d'un "pointcut". Il y a trois types d'advice : les "before advices", les "around advices" et les "after advices". Comme leurs noms l'indiquent, les "before advices" s'exécutent avant que les points de jointures ne soient exécutés; les "around advices" permettent d'exécuter du code avant et après les points de jointure; et les "after advices" s'exécutent uniquement après l'exécution des points de jointure. L'avantage des advices vient du fait qu'ils peuvent avoir accès à certaines valeurs du contexte d'exécution d'un pointcut.

Si nous reprenons l'exemple de la section 3.3, l'advice suivant avait été défini:

```
before(Information in): appelTransaction (in) {
    log.enregistrer("Tentative transaction:" + in);
}
```

Ce "before advice" concerne tous les points de jointure du pointcut `appelTransaction`. Avant d'appeler les méthodes désignées par `appelTransaction`, on fait appel à `log.enregistrer(...)` qui va enregistrer l'argument `in` de ces méthodes dans un fichier journal.

De la même manière, on aurait pu définir un "after advice" qui allait enregistrer les informations après les appels de méthode. Il suffit pour cela de remplacer le mot réservé `before` par `after` dans la définition ci-dessus.

Autre exemple, l'advice suivant capture tous les appels de `Connection.close()` et affiche le temps de l'horloge système avant et après ceux-ci :

```
void around(Connection conn) :
call(Connection.close()){
    System.out.println(System.currentTimeMillis());
    proceed();
    System.out.println(System.currentTimeMillis());
}
```

On remarque que si on n'avait pas fait appel à `proceed()` dans l'advice, l'appel de méthode `Connection.close` n'aurait pas été effectué. Ce mécanisme permet donc aussi de supprimer l'exécution de certains blocs de code.

4.4 Les Aspects.

Un aspect rassemble des pointcuts et des advices pour former une unité de recoupement. Les pointcuts et les advices définissent à eux deux les règles d'intégration.

Un aspect est similaire à une classe Java, dans ce sens où, il contient des champs et des méthodes et peut même étendre d'autres classes ou d'autres aspects. Je n'entrerai pas plus loin dans le sujet et signale simplement qu'un aspect se déclare comme une classe Java en remplaçant le mot `class` par `aspect`. L'exemple présenté à la section 3.3 illustre bien cette différence.

4.5 Intégration d'AspectJ dans un Processus de Développement

Nous avons vu plus haut la distinction entre composants et aspects. Nous pouvons encore raffiner cette classification en distinguant deux types d'aspects : les aspects de développement et les aspects de production. Jusque maintenant, nous avons principalement parlé de la dernière catégorie. Les aspects de production sont ceux qui doivent rester dans le système final pour qu'il fonctionne correctement. Il s'agit donc, par exemple, des aspects de sécurité, de synchronisation, etc. Les aspects de développement sont ceux que le programmeur utilise à des fins personnelles de débugage, traçage, vérification, etc. Il s'agit donc d'aspects qui sont utiles, non pas pour le système en lui-même, mais pour son processus de développement.

Pour faciliter l'intégration de la programmation orientée aspect dans votre processus de travail, il est intéressant de commencer par quelques aspects de développement :

- Vous pouvez utiliser AspectJ pour tracer ou inscrire des informations intéressantes dans un journal de bord. Cette technique s'avère bénéfique notamment pour débuguer un programme : les lignes de code qui impriment les variables ou autres informations souhaitées ne viennent pas s'immiscer dans le code source des composants, il n'est donc plus nécessaire de les enlever par après.
- Dans la même optique, vous pouvez utiliser AspectJ pour tester le système. Il devient très facile d'insérer le code qui vérifiera les invariants de vos modules ou les pré-conditions et les post-conditions de vos méthodes.

Ces deux exemples n'entraînent aucune conséquence sur l'architecture des systèmes et permettent de se familiariser avec le langage AspectJ avant de passer à la conception d'aspect de production.

5. Exemple Réel de Programmation OA dans un Système Distribué

Pour bien concrétiser tous les concepts vus jusque maintenant et sentir les avantages que peut apporter la programmation orientée aspect, je propose ici un exemple d'application client-serveur qui utilise le mécanisme des aspects pour implémenter un pool de threads qui améliorera la performance du serveur. Cet exemple est une adaptation de celui présenté par Ramnivas Laddad dans la troisième partie de sa série d'article sur la programmation orientée aspect [7].

Le but n'étant pas de montrer l'implémentation complète d'une application mais de présenter les avantages de la programmation orientée aspect, j'ai simplifié l'exemple pour ne présenter que ce qui était nécessaire. Ainsi, on se concentrera uniquement sur l'aspect du pool de threads. L'aspect de journalisation, par exemple, est encore traité dans les classes serveurs principales.

Supposons que nous ayons implémenté un serveur qui permet de convertir des strings en majuscules. Une implémentation possible en Java est illustrée ci-dessous. Ce module est divisé en deux classes: `UppercaseWorker` et `UppercaseServer`. `UppercaseServer` est composé d'une méthode principale qui alloue un nouveau thread `UppercaseWorker` chaque fois qu'un client se connecte. `UppercaseWorker` traite donc les requêtes des clients.

```
// UppercaseServer.java
import java.io.*;
import java.net.*;

class UppercaseWorker implements Runnable {
  private Socket requestSocket;
  private BufferedReader requestReader;
  private PrintWriter responseWriter;

  public UppercaseWorker(Socket rs) throws IOException {
    System.out.println("Creating new worker");
```

```
    this.requestSocket = rs;
    this.requestReader = new BufferedReader(
           new inputStreamReader(
           requestSocket.getInputStream(),"ASCII"),560);
    this.responseWriter = new PrintWriter(
            requestSocket.getOutputStream(),true);
  }

  public void run() {

    try {
      while(true) {
        String requestString = requestReader.readLine();
        if (requestString == null) {
          break;
        }
        else{
          System.out.println("Got request: " + requestString);
          responseWriter.println(requestString.toUpperCase());
        }
      }
    }
    catch(IOException ex) {
     System.err.println(
     "\t@"+requestSocket.getInetAddress().getHostName()+ " error");
    }
    finally {
      try {
        responseWriter.close();
        requestReader.close();
        requestSocket.close();
      }
      catch (IOException skip) {
      }
    }
    System.out.println("Ending the session");
  }
}

public class UppercaseServer {

  public static void main(String[] args) throws Exception {
    if (args.length != 1) {
      System.out.println("Usage: java UppercaseServer <portNum>");
      System.exit(1);
    }

    int portNum = Integer.parseInt(args[0]);
    ServerSocket serverSocket = new ServerSocket(portNum);

    while(true) {
      Socket requestSocket = serverSocket.accept();
      Thread serverThread
        = new Thread(new UppercaseWorker(requestSocket));
```

```
      serverThread.start();
    }
  }
}
```

Supposons maintenant que, pour des raisons de performance, on veuille améliorer ce serveur en utilisant un pool de threads. Au lieu de créer un thread pour chaque nouveau client, on propose donc de réutiliser des threads créés au préalable. Pour se faire, nous avons tout d'abord besoin de créer la structure de ce pool de thread. Un exemple d'implémentation est proposé ci-dessous :

```
// ThreadPool.java
import java.util.*;

public class ThreadPool {
  private List waitingThreadList =
    Collections.synchronizedList(new LinkedList());

  public void put(DelegatingThread thread) {
    System.out.println("Putting back: " + thread);
    waitingThreadList.add(thread);
  }

  public DelegatingThread get() {
    if (!waitingThreadList.isEmpty()) {
      DelegatingThread availableThread
        = (DelegatingThread)waitingThreadList.remove(0);
      System.out.println("Providing for work: " +
                                  availableThread);
      return availableThread;
    }
    return null;
  }

  static class DelegatingThread extends Thread {
    private Runnable delegatee;

    public void setDelegatee(Runnable d) {
      delegatee = d;
    }

    public void run() {
      delegatee.run();
    }
  }
}
```

Ce `ThreadPool` a une structure simple s'apparentant à une pile dans laquelle nous pouvons ajouter des threads (via la méthode `put`), et en retirer (via la méthode `get`). Notons que si le pool est vide, la méthode `get` retourne `null` et ne crée pas de nouveau thread. La classe `DelagatingThread` est un simple thread auquel nous

pouvons déléguer le travail d'une classe implémentant `Runnable` via la méthode `setDelegate`.

Maintenant que l'on dispose de la structure du pool, nous pouvons écrire les règles d'intégration qui vont pouvoir changer l'exécution du serveur sans toucher à son code source. Nous allons créer 3 advices relatifs à trois poincuts différents :

```
// ThreadPooling.java
public aspect ThreadPooling {
  ThreadPool pool = new ThreadPool();

//================================================================
// creation de Thread -> renvoie un thread du pool
//================================================================
  pointcut threadCreation(Runnable runnable)
    : call(Thread.new(Runnable)) && args(runnable);

  Thread around(Runnable runnable) : threadCreation(runnable) {
    ThreadPool.DelegatingThread availableThread = pool.get();
    if (availableThread == null) {
      availableThread = new ThreadPool.DelegatingThread();
    }
    availableThread.setDelegatee(runnable);
    return availableThread;
  }


//================================================================
// Session - evite qu'un Thread ne meure a la fin de son execution
//================================================================
  pointcut session(ThreadPool.DelegatingThread thread)
    : execution(void ThreadPool.DelegatingThread.run())
      && this(thread);

  void around(ThreadPool.DelegatingThread thread) :
  session(thread){
    while(true) {
      proceed(thread);
      pool.put(thread);
      synchronized(thread) {
        try {
          thread.wait();
        }
        catch(InterruptedException skip) {
        }
      }
    }
  }


//================================================================
//Thread start - verifie si le Tread a deja demarre et le reveille
//================================================================
  pointcut threadStart(ThreadPool.DelegatingThread thread)
```

```
    : call(void Thread.start()) && target(thread);

  void around(Thread thread) : threadStart(thread) {
    if (thread.isAlive()) {
      synchronized(thread) {
        thread.notify(); // wake it up
      }
    } else {
      proceed(thread);
    }
  }
}
```

L'idée principale de cet aspect est de capturer les points de jointure qui créent les nouvelles connexions pour les clients et de les remplacer par une connexion déjà existante dans le pool. Pour que cela fonctionne, nous devons aussi capturer les points de jointures qui ferment les connexions pour remettre ces dernières dans le pool à la place.

Le premier pointcut "`threadCreation`" défini les points de jointures qui correspondent aux créations de thread (`new Thread`) qui prennent en argument un object de type `Runnable`. L'advice concernant ce premier pointcut vérifie si un `DelegatingThread` est disponible dans le pool. Si aucun thread n'est disponible, on en crée alors un nouveau. Dans les 2 cas, on délègue ensuite le travail de l'argument `runnable` au `DelegatingThread` et on renvoie ce dernier à la place. Nous pouvons remarquer que cet advice ne fait pas appel à la méthode `proceed`, et que donc le code capturé par ce point de jointure (`new Thread(runnable)`) ne sera pas exécuté.

Le deuxième poincut "`session`" capture les points de jointures correspondant aux exécutions de la méthode `run()` d'un objet de la classe `ThreadPool.DelegatingThread`. Le but de l'advice consiste à empêcher le thread de mourir lorsqu'il a fini son travail. Pour ce faire, on englobe les opérations de la méthode `run` dans une boucle infinie. A chaque fois que la méthode a terminé, on endort le thread lui correspondant et on le remet dans le pool.

Le dernier pointcut "`threadStart`" capture tous les appels de méthode `Thread.start(t)` s'appliquant sur un objet de type `ThreadPool.DelegatingThread`. L'advice correspondant vérifie si le thread en argument a déjà été démarré (càd qu'il provient du pool et qu'il est juste endormi). Dans l'affirmative on se contente donc de le réveiller et dans le cas contraire on exécute le code original grâce à l'appel de `proceed()`.

Cet exemple est assez intéressant dans ce sens où on aperçoit clairement la méthode de développement orientée aspect. Nous avons commencé par écrire le composant principal càd le serveur, puis nous avons voulu prendre en compte l'aspect performance et avons décider de créer un pool de thread. Finalement, nous avons intégré l'aspect de performance dans le serveur original. Nous remarquons aussi qu'aucun changement n'a été apporté au serveur initial. Cette technique peut aussi servir pour améliorer l'accès aux bases de données d'une librairie digitale dont je parlais au début de cet article.

Conclusions

J'ai présenté ici les problèmes de recoupement qui existent dans la plupart des implémentations actuelles des systèmes informatiques. La programmation orientée aspect apporte une solution nouvelle et élégante à ces problèmes. Elle permet de séparer l'implémentation des composants et des aspects d'un système. Rappelons-nous que les composants sont les problématiques d'un système qui peuvent être clairement encapsulées dans un module bien circonscrit. Les aspects sont les problématiques d'un système qui ne peuvent pas être clairement encapsulées dans un module bien circonscrit. L'avantage premier des langages orientés aspect est qu'ils permettent d'encapsuler l'implémentation des aspects sous une certaine forme d'abstraction.

Les langages orientés aspect, et plus particulièrement AspectJ, commencent à être utilisés de plus en plus. Certains programmeurs intègrent cette nouvelle méthode dans leur processus de développement à des fins de tests internes, débugage, traçage d'information, et d'autres l'utilisent déjà pour développer des applications complètes. Les systèmes distribués se prêtent particulièrement bien aux techniques de l'orienté aspect: ils contiennent souvent des problématiques de synchronisation, d'authentification, et autres aspects dont la programmation OA favorise l'encapsulation.

Même si la programmation OA apporte un avantage non-réfutable pour certains types d'application, il est encore trop tôt pour la présenter comme le prochain grand paradigme de programmation. Son utilisation et son expérimentation sous divers points de vue devraient nous en dire plus dans les quelques années à venir.

Bibliographie

1. Gal A., Schröder-Preikschat W. and Spinczyk O., *Aspect C++ : Language Proposal and Prototype Implementation*, submitted to the AOP workshop at the ACM conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), 2001.

2. Gal A., Schröder-Preikschat W. and Spinczyk O., *On Aspect-Orientation in Distributed Real-time Dependable System,* University of Magdeburg, Germany, 2001.

3. Kiczales G., et al., *Aspect-Oriented Programming*, in Proceeding of the European Conference on Object-Oriented Programming (ECOOP), Finland, 1997.

4. Kiczales G., et al., An *Overview of AspectJ*, in European Conference on Object-Oriented Programming (ECOOP), 2001.

5. Laddad R., *Use AspectJ to modulariwe crosscutting concerns in real-world problems,* in Javaworld magazine, Web pages, http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect_p.html, 2002.

6. Laddad R., *Learn AspectJ to better understand aspect-oriented programming,* in Javaworld magazine, Web pages, http://www.javaworld.com/javaworld/jw-03-2002/jw-0301-aspect2_p.html, 2002.

7. Laddad R., *Separate software concerns with aspect-oriented programming,* in Javaworld magazine, Web pages, http://www.javaworld.com/javaworld/jw-04-2002/jw-0412-aspect3_p.html, 2002.

8. Walker R., Baniassad E. and Murphy G., *An Initial assessment of Aspect-oriented Programming,* in Proceedings of the 21st International Conference on Software Engineering, Los Angeles, CA, 1999.