

Le point sur la programmation par aspects

Noury M. N. Bouraqadi-Saâdani — Thomas Ledoux

École des Mines de Nantes
Département Informatique
B.P. 20722
44307 Nantes Cedex 03
{Noury.Bouraqadi, Thomas.Ledoux}@emn.fr
http://www.emn.fr/{bouraqadi, ledoux}

Ce travail a été partiellement financé par l'union européenne dans le cadre du projet EasyComp n° 1999-14191 (www.easycomp.org).

RÉSUMÉ. La réutilisation est l'une des principales promesses de la programmation par objets. Malheureusement, elle n'est pas toujours possible du fait du mélange, lors de l'implémentation des applications, des définitions des services réalisés ("fonctionnalités") avec des propriétés non-fonctionnelles représentant des mécanismes d'exécution spécifiques (distribution, tolérance aux fautes, persistance, ...). Afin de pallier ce défaut, le paradigme de la programmation par aspects considère que les services d'une application et ses propriétés non-fonctionnelles correspondent à des aspects qui doivent être découplés les uns des autres. L'application est obtenue par la composition ("assemblage") de ces différents aspects. Nous avons identifié trois approches permettant la mise en œuvre de la programmation par aspects. Dans le présent article, nous faisons le point sur ces approches ainsi que sur les concepts sous-jacents à la programmation par aspects.

ABSTRACT. Reuse is one of the major promises of object-oriented programming. Unfortunately, this promise is only partially achieved. One of the sources of this failure is the mix of the applications basic functionalities code with non-functional properties code (e.g. distribution, fault tolerance, persistence). In order to avoid this weakness, the aspect-oriented programming (AOP) paradigm proposes to split applications implementation into aspects representing either functionalities or non-functional properties. Then, an application is built by composing ("assembling") various aspects. We have identified three different approaches allowing AOP. In this paper, we present and discuss these approaches and the concepts behind AOP.

MOTS-CLÉS : Séparation des aspects, composition d'aspects, réutilisation.

KEYWORDS: Separation of concerns, aspect composition, reuse.

1. Introduction

Du fait de ses nombreux atouts (encapsulation, polymorphisme, modularité, ...), la programmation par objets constitue indéniablement une technologie importante pour aider à la construction d'applications complexes. En effet, elle favorise la *réutilisation* et permet ainsi la réduction des délais de développement et de maintenance. Cependant, cette réutilisation n'est pas toujours aisée [MAT 93] [GAM 95] [KIC 96] [STE 96]. C'est notamment le cas pour les applications dont la construction ne se limite pas à la simple définition d'un ensemble de services donnés, mais nécessite également la prise en compte de différentes *propriétés non-fonctionnelles* telles que la distribution, la persistance ou encore le temps réel. Dans de telles applications, l'implémentation des services réalisés et celle des différentes propriétés non-fonctionnelles se trouvent intimement enchevêtrées. De ce fait, la réutilisation se trouve compromise. En effet, il n'est pas toujours possible de réutiliser les différents *aspects* (i.e. services ou propriétés non-fonctionnelles) d'une application, indépendamment les uns des autres.

Dans le but de permettre une meilleure réutilisation, le paradigme de la *programmation par aspects* [DIJ 76] [HUR 95] [KIC 97] propose de structurer les applications sur la base du concept d'*aspect*. Un *aspect* est une "brique" de l'application représentant totalement et exclusivement une propriété donnée de l'application (les services réalisés, la persistance, la distribution, ...). La programmation par aspects consiste à réaliser une application en deux temps. Tout d'abord, les différents aspects sont définis *séparément* les uns des autres. Les définitions des aspects doivent être découplées, de sorte qu'elles ne se référencent pas les unes les autres. Ensuite, l'application est produite par la *composition* ("assemblage") des aspects ainsi définis. En proposant de retarder la composition, le paradigme de la programmation par aspects relâche le couplage entre les aspects, offrant ainsi de nouvelles perspectives de réutilisation. Il devient, en effet, possible d'envisager la réutilisation des aspects indépendamment les uns des autres.

La représentation des aspects et la manière de les composer sont des questions ouvertes, auxquelles chaque mise en œuvre de la programmation par aspects doit répondre. L'objectif du présent article est de présenter les principales techniques de mise en œuvre de la programmation par aspects et de faire le point sur l'utilisabilité de ce paradigme.

Il est à noter que la programmation par aspects est indépendante de la programmation par objets. En effet, il est possible d'utiliser la programmation par aspects conjointement à d'autres paradigmes de programmation (fonctionnelle, procédurale...). Néanmoins, nous plaçons cette étude dans le contexte de la programmation par objets, du fait des nombreux atouts de ce paradigme.

Dans ce qui suit, nous motivons et présentons d'abord le paradigme de la programmation par aspects (section 2). Puis, nous présentons les différentes approches permettant de mettre en œuvre ce paradigme (section 3), dont nous dressons un bilan en dernier lieu (section 4).

2. Programmation par aspects

2.1. Motivation : limites de réutilisation avec l'approche objet

2.1.1. Exemple d'aspects

Pour illustrer les limites de l'approche objet, considérons l'exemple d'une librairie électronique¹ représentée sur la figure 1. Les clients utilisent un réseau pour consulter la liste des ouvrages disponibles et éventuellement les commander. Plusieurs clients peuvent être connectés en même temps et plusieurs commandes peuvent être traitées en parallèle.

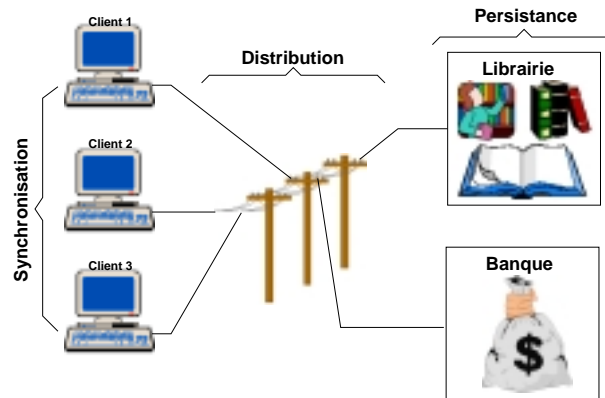


Figure 1. Exemple d'aspects d'une application

La conception objet de cette application nous conduit à définir notamment les classes *Librairie*, *Client*, *Ouvrage* et *Banque*. Par ailleurs, nous pouvons identifier au moins trois aspects différents correspondant à des propriétés non-fonctionnelles de cette application : la distribution, la persistance et la synchronisation. L'aspect distribution apparaît du fait que les instances des différentes classes en jeu se trouvent sur des sites distants. Cet aspect regroupe entre autre la gestion de ces communications distantes et la définition du protocole de communication utilisé. L'aspect persistance apparaît parce que des objets comme, par exemple, les ouvrages mis en vente, doivent survivre à un arrêt de l'application. La sauvegarde de tels objets sur un support de masse et la gestion des mises à jour de cette sauvegarde font partie de la définition de cet aspect persistance. Quant à l'aspect synchronisation, il est mis en évidence, notamment, par le besoin de synchroniser les accès concurrents aux structures des objets. C'est le cas, par exemple, lorsqu'un client demande le prix d'un ouvrage pendant que le libraire modifie ce même prix. L'aspect synchronisation englobe la gestion de tels accès concurrents.

1. Cet exemple est partiellement inspiré des exemples de Kiczales et al. [KIC 97].

L'implémentation de l'application de la librairie électronique dans un langage à objets conventionnel tel que C++ introduit deux problèmes qui compromettent la réutilisation [HUR 95] [KIC 97]. D'une part, l'approche objet conduit au *mélange du code* source correspondant aux services de l'application avec celui correspondant aux trois aspects distribution, synchronisation et persistance. Et d'autre part, elle conduit à la *dispersion du code* correspondant à chacun de ces aspects.

2.1.2. Mélange du code

Nous illustrons le mélange du code par une implémentation possible de la classe **Ouvrage** et de la synchronisation des accès à la variable d'instance **prix**. La figure 2 en donne le code correspondant dans une syntaxe SMALLTALK. Les lignes de code représentant l'aspect synchronisation (en italique) sont intimement enchevêtrées avec les lignes correspondant à des services réalisés par les ouvrages. De tels enchevêtrements nuisent à la lisibilité du code rendant difficiles sa compréhension, sa maintenance et sa réutilisation. Par exemple, il est difficile de réutiliser la classe **Ouvrage** dans un contexte nécessitant une autre politique de synchronisation.

La solution polymorphique, qui consiste à utiliser l'héritage pour introduire cette autre politique de synchronisation n'est pas satisfaisante. En effet, la définition de sous-classes qui modifient les règles de synchronisation pose le problème d'*anomalie d'héritage* [MAT 93]. Ce problème se traduit par l'obligation de redéfinir dans les sous-classes plusieurs, voire la totalité, des méthodes de la superclasse, pour obtenir le comportement désiré. Ce qui réduit d'autant les possibilités de réutilisation offertes habituellement par l'héritage.

```
Ouvrage instanceVariableNames: 'titre auteur prix monitor'
écrirePrix: nouveauPrix
monitor readWriteProtect. "Interdire la lecture et l'écriture"
prix := nouveauPrix.
monitor readWriteRelease. "Autoriser la lecture l'écriture"

lirePrix
|prixActuel|
monitor writeProtect. "Interdire l'écriture"
prixActuel := prix.
monitor writeRelease. "Autoriser l'écriture"
↑prixActuel.
```

Figure 2. Exemple de mélange des aspects avec la programmation par objets

Une solution à ce problème d'anomalie d'héritage pourrait être la construction des applications suivant des schémas de conception (*design patterns*) tel que les schémas "Stratégie" ou "État" [GAM 95]. Cependant, cette solution peut s'avérer très lourde à mettre en œuvre [DUC 97] [BOI 00]. En effet, elle augmente considérablement le nombre des classes et des indirections, provoquant un accroissement des communica-

tions préjudiciable pour les performances. Par ailleurs, la multiplication des classes et des interdépendances complique la maintenance et l'évolution des applications.

Le problème d'anomalie d'héritage et la limitation des possibilités de réutilisation qui en découle ne sont pas spécifiques à la synchronisation. Les mêmes limitations se posent chaque fois qu'un objet est concerné par une ou plusieurs propriétés non-fonctionnelles (par exemple, un ouvrage doit être persistant et synchronisé). La définition des services réalisés par l'application ne peut être facilement réutilisée indépendamment des propriétés non-fonctionnelles. Inversement, il est difficile de réutiliser une propriété non-fonctionnelle dans un autre contexte.

2.1.3. *Dispersion du code*

La réutilisation d'une propriété non-fonctionnelle est d'autant plus difficile que sa définition se trouve dispersée et de ce fait ne peut être isolée pour être réutilisée. En effet, une même propriété peut concerner plusieurs objets. Par exemple, les ouvrages et les clients de notre librairie électronique doivent être tous persistants. Or, avec l'approche objet, le code définissant la persistance se trouve dispersé et partiellement dupliqué dans les classes `Ouvrage` et `Client`. Il est donc difficile de modifier ou de réutiliser ce code bien que la politique de persistance de l'application est conceptuellement considérée comme une unique propriété.

2.2. *Principe de la programmation par aspects*

2.2.1. *Concepts*

Une solution aux problèmes de mélange et de dispersion du code des propriétés non-fonctionnelles rencontrés avec la programmation par objets, consiste à séparer et découpler leurs définitions comme le veut le principe de *separation of concerns* (littéralement séparation des préoccupations) [DIJ 76] [HUR 95]. Partant de ce principe, Gregor Kiczales et al. ont formalisé cette séparation et les différentes étapes de réalisation des applications dans ce cadre pour aboutir au paradigme de programmation appelé *programmation par aspects* (*AOP : Aspect Oriented Programming*) [KIC 97]. Ce paradigme de programmation consiste à structurer les applications en modules indépendants qui représentent les définitions de différents aspects.

La décomposition d'une application fait apparaître :

- **un aspect de base** : il définit l'ensemble des services (i.e. "fonctionnalités") réalisés par l'application. Autrement dit, l'aspect de base correspond au "Quoi" de l'application. Par exemple, l'aspect de base de la librairie électronique présentée précédemment définit les services "rechercher un ouvrage", "passer une commande".

- **plusieurs aspects non-fonctionnels** : ils définissent les mécanismes qui régissent l'exécution de l'application. Autrement dit, les aspects non-fonctionnels définissent le "Comment" de l'application. Par exemple, la librairie électronique comporte les aspects non-fonctionnels distribution, persistance et synchronisation. Chaque as-

pect non-fonctionnel est indépendant des autres et définit un mécanisme d'exécution particulier.

La construction d'une application à partir de différents aspects nécessite une étape "d'assemblage". En effet, les aspects étant des modules définis séparément les uns des autres, il est nécessaire de les composer entre eux afin de "construire" l'application. D'où le besoin d'un mécanisme de **composition** pour réaliser cet assemblage (cf. figure 3).

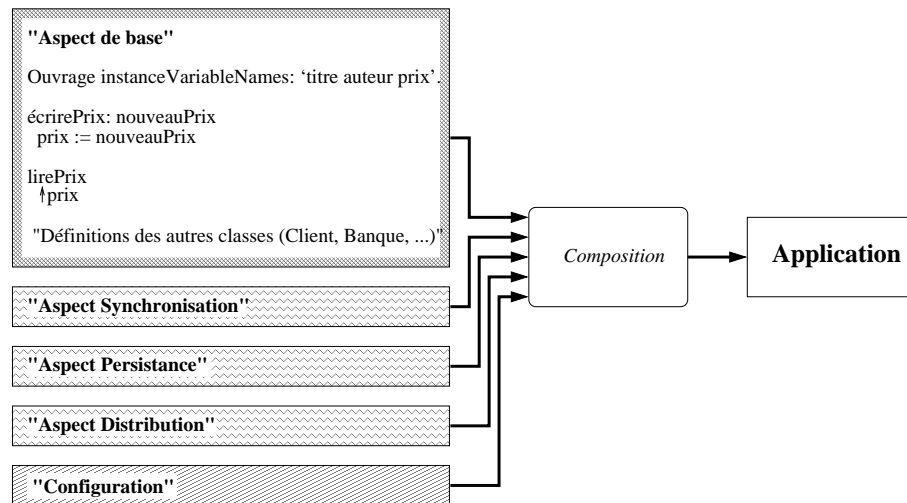


Figure 3. Construction d'une application avec la programmation par aspects

La composition de l'aspect de base avec les aspects non-fonctionnels se traduit par l'établissement d'une jonction entre ces aspects. Cette jonction s'établit au niveau d'un ensemble de points du flot d'exécution de l'aspect de base, appelés **points de jonction** [KIC 97]. Tout point de ce flot d'exécution constitue un point de jonction potentiel, susceptible d'être utilisé pour la composition. Ainsi, une invocation de méthode, une boucle ou une affectation sont autant de points de jonction potentiels. Dans l'exemple de la librairie électronique, les accès à la variable d'instance `prix` de la classe `Ouvrage` sont des points de jonction utilisés pour composer l'aspect de base avec l'aspect synchronisation. Les points de jonction permettent la composition des aspects pour la construction de l'application. Ils représentent donc l'une des abstractions importantes de la programmation par aspects.

Afin de réaliser la composition de l'aspect de base avec les aspects non-fonctionnels, il est nécessaire de connaître les points de jonction à utiliser. Pour ce faire, les développeurs doivent définir la **configuration des aspects**. Cette configuration consiste à indiquer les points de jonction à utiliser pour composer chaque aspect non-fonctionnel avec l'aspect de base. Notons que la variété de points de jonction poten-

tiels qui peuvent être désignés dans la configuration dépend fortement de la mise en œuvre de la programmation par aspects.

La composition d'un aspect non-fonctionnel avec l'aspect de base peut utiliser plusieurs points de jonctions. Inversement, un même point de jonction peut servir pour la composition de différents aspects non-fonctionnels avec l'aspect de base. Ce partage de points de jonction nécessite de disposer de mécanismes de composition qui permettent une "superposition" harmonieuse des différents aspects non-fonctionnels. Il s'agit de modifier le flot d'exécution de l'aspect de base de manière à prendre en compte tous les aspects non-fonctionnels tout en préservant leurs sémantiques respectives.

2.2.2. Exemple de programmation par aspects

En guise d'exemple de programmation par aspects, la figure 3 donne la structure de notre application de la librairie électronique, dans un cadre de programmation par aspects idéal. Quatre modules, correspondant aux quatre aspects identifiés constituent cette application. Le premier module correspond à l'aspect de base défini dans une syntaxe à la SMALLTALK. Il définit les services rendus par l'application indépendamment des aspects non-fonctionnels. En particulier, la classe `Ouvrage` est définie sans aucun élément sur sa synchronisation. La politique de synchronisation est définie dans un autre module correspondant à l'aspect synchronisation. De même pour les autres aspects de l'application. La configuration des aspects désigne les points de jonction correspondant à la lecture et à l'écriture des variables d'instance de la classe `Ouvrage`. Ces points de jonction sont utilisés pour réaliser la composition de l'aspect non-fonctionnel synchronisation avec l'aspect de base.

Comme les définitions des différents aspects sont découplés, il est possible de les réutiliser indépendamment les uns des autres. Par exemple, il est possible de construire une nouvelle application de librairie électronique avec une politique de synchronisation différente. Dans ce cas, à l'exception de l'aspect synchronisation, tous les aspects seraient réutilisés. Il suffirait juste de mettre à jour la configuration des aspects pour construire la nouvelle application.

3. Mise en œuvre de la programmation par aspects

Les principaux problèmes posés par la mise en œuvre de la programmation par aspects sont la représentation et la composition des aspects. Nous exposons dans cette partie les différentes approches qui tentent de répondre à ces questions.

Dans un premier temps, nous introduisons le critère de classification des différentes approches étudiées. Nous présentons ensuite les différentes approches pour programmer avec des aspects. Un même exemple (celui de synchronisation des instances de la classe `Ouvrage` extrait de l'exemple de la librairie électronique) est utilisé pour les illustrer.

3.1. Critère de classification

Pour définir notre critère de classification des différentes techniques de mise en œuvre de la programmation par aspects, nous partons du concept de *programme* informatique. Un programme est une séquence d'instructions destinées à produire un *résultat*. L'exécution de ce programme est effectuée par une plate-forme (système d'exploitation, machine virtuelle, ...) qui *interprète* la séquence d'instructions.

Reconsidérons les définitions de l'aspect de base et des aspects non-fonctionnels dans ce contexte. L'aspect de base décrit les services d'une application, alors que les aspects non-fonctionnels interviennent sur la manière de réaliser ces services. En l'absence d'aspects non-fonctionnels, nous pouvons considérer l'aspect de base comme un programme P_0 écrit pour un interprète I_0 par défaut. L'exécution de P_0 par I_0 produit un résultat R_0 donné. Les aspects non-fonctionnels interviennent dans le traitement de sorte à produire un résultat final R_1 différent du résultat R_0 . Ce nouveau résultat R_1 peut être obtenu aussi bien en transformant le programme P_0 qu'en transformant l'interprète I_0 . En effet, comme tout résultat d'un traitement informatique est induit du couple $\langle \text{Programme}, \text{Interprète} \rangle$, il est possible de modifier ce résultat en intervenant sur le programme ou sur l'interprète ou sur les deux. Chacun de ces types d'intervention définit une catégorie de mise en œuvre de la programmation par aspects.

3.2. Approche par transformation de programme

Cette section décrit les techniques de mise en œuvre qui s'attachent à faire varier uniquement le programme dans le couple $\langle \text{Programme}, \text{Interprète} \rangle$. L'idée est donc de construire un nouveau programme P_1 à partir de l'aspect de base (programme P_0) et des aspects non-fonctionnels pour produire un nouveau résultat R_1 , et ce, sans changer l'interprète par défaut I_0 . Il s'agit donc de transformer le programme P_0 de sorte à y introduire les traitements qui représentent les aspects non-fonctionnels.

3.2.1. Description de l'approche par transformation de programme

Cette approche, mise en œuvre dans différents travaux [CHI 95] [KIC 97] [CHI 98] [FRA 98] [LIE 99], consiste à définir chaque aspect non-fonctionnel sous la forme d'un ensemble de règles de transformation à appliquer à l'aspect de base. Par exemple, considérons l'aspect non-fonctionnel qui consiste à produire une trace dans un fichier de journalisation pour certaines opérations de l'aspect de base. Cet aspect de journalisation peut être représenté par des règles de transformation qui consistent à insérer le code de production de trace au début des méthodes qui représentent les opérations à tracer.

Une des difficultés de cette approche consiste à construire des règles de transformation génériques, i.e. indépendantes de l'aspect de base. Le but étant de définir des aspects non-fonctionnels découplés de l'aspect de base, et de ce fait facilement réutilisables. Cet objectif peut être atteint en paramétrant les règles de transformation. Dans

ce contexte, les règles de transformation et donc les aspects non-fonctionnels sont définis en terme de points de jonction “abstrait”. Ces points de jonction permettent de référencer, sans couplage, des éléments de l’aspect de base. La configuration consiste à lier les points de jonction “abstrait” à des éléments concrets de la définition de l’aspect de base (affectations, boucles, ...). Une fois ce lien établi, la composition des aspects peut avoir lieu, puisque les règles de transformation peuvent s’appliquer à l’aspect de base.

Le processus de transformation que représente la composition des aspects produit un nouveau programme où les différents aspects sont fusionnés. Dans ce programme monolithique, les lignes de code représentant les différents aspects (de base ou non-fonctionnels) sont intimement enchevêtrées. Cet enchevêtrement est analogue à celui obtenu lorsqu’une application est développée dans un langage de programmation par objets conventionnel. Aussi, l’approche par transformation de programme s’apparente à automatiser le mélange de code qui aurait été réalisé manuellement par les développeurs.

Lorsqu’une application comporte différents aspects non-fonctionnels, la composition des aspects se traduit par l’application des règles de transformation représentant l’ensemble de ces aspects non-fonctionnels. L’ordre de réalisation des différentes transformations est important. En effet, l’application de différentes règles de transformation suivant des ordres différents ne conduit généralement pas au même résultat. Par exemple, la transformation qui introduit le cryptage des arguments de certains appels de méthodes (aspect cryptographie) et la transformation qui introduit la génération de trace (aspect journalisation) ne peuvent être appliquées dans un ordre quelconque. Pour remédier à cet impératif sémantique, certaines mises en œuvre offrent aux développeurs des constructions permettant l’ordonnement des aspects non-fonctionnels et donc des règles de transformation [LOP 98b] [LIE 99] [XER 00].

3.2.2. Exemple d’un représentant de l’approche par transformation de programme : le langage AspectJ

Issu des travaux menés au Xerox-PARC sur la programmation par aspects [KIC 97], ASPECTJ est l’un des rares langages à expliciter la notion d’aspect [LOP 98b] [XER 00]. Cette extension de JAVA permet de définir l’aspect de base sous forme de classes JAVA. Des constructions syntaxiques spécifiques dans un langage dédié permettent la définition des aspects non-fonctionnels et leur configuration. Les développeurs utilisent ces constructions pour définir les différents aspects non-fonctionnels d’une application sous forme de règles de transformation simples.

La conception initiale de ASPECTJ n’a pas été pensée pour séparer explicitement la configuration des points de jonction du code des aspects non-fonctionnels. En effet, les règles de transformation qui représentent les aspects non-fonctionnels référencent explicitement des noms de classes ou de méthodes définies dans l’aspect de base. Conscients de cette limitation, les concepteurs de ASPECTJ proposent, dans les dernières versions, d’utiliser l’héritage entre aspects et la notion d’*aspect abstrait* pour découpler les définitions des aspects de leur configuration [XER 00]. Cependant, ce

découplage dépend fortement de la discipline des développeurs, puisqu'aucun mécanisme ne l'encourage.

L'utilisation de ASPECTJ pour notre exemple de synchronisation de la classe `Ouvrage` aboutit à la définition de trois modules (voir figure 4). Le premier module correspond à l'aspect de base implémentant la définition de la classe `Ouvrage`. Le second module correspond à la définition de l'aspect synchronisation (`Synchronizer`). La configuration est définie dans le troisième module (`OuvrageSynchronizer`).

Dans ASPECTJ, le mot-clé **aspect** introduit une construction qui représente un aspect non-fonctionnel et éventuellement la configuration associée. Cette construction s'apparente à une définition de classe étendue avec les définitions des points de jonction utilisés par l'aspect, ainsi que les transformations réalisées par l'aspect. Les points de jonctions sont regroupés dans des ensembles identifiés par un nom et introduits par le mot-clé **pointcut**. Ces ensembles permettent de retrouver les lignes de code à transformer. Dans notre exemple, les lignes 19 à 22 définissent l'ensemble `synchronizeCut` qui regroupe les points de jonction correspondant à la réception des invocations des méthodes `lirePrix()` et `écrirePrix()` de la classe `Ouvrage`.

Les règles de transformation sont introduites par l'intermédiaire de mots-clés tels que **before** et **after**. Le mot-clé utilisé désigne le type de transformation à réaliser. Ce mot-clé est suivi par l'identifiant de l'ensemble des points de jonctions qui doivent être affectés par la transformation. Les lignes 4 à 10 de la figure 4 représentent deux règles de transformation qui affectent l'ensemble des points de jonctions `synchronizeCut`. Comme les points de jonction de cet ensemble correspondent à des réceptions d'invocations de méthodes, les transformations introduisent le code de synchronisation au début (mot-clé **before**) et à la fin (mot-clé **after**) des corps de ces méthodes.

Les constructions **aspect** supportent l'héritage grâce au mot-clé **extends** emprunté à JAVA. Cette notion d'héritage a été utilisée dans l'exemple de la figure 4 pour favoriser la réutilisation. Un aspect synchronisation découplé de toute application est défini dans le module `Synchronizer`. En effet, `Synchronizer` ne définit que les règles de transformations à réaliser et les méthodes et variables nécessaires pour assurer la synchronisation. L'ensemble des points de jonction `synchronizeCut` étant qualifié **abstract**, sa définition doit être donnée dans les "aspects" qui héritent de `Synchronizer`. Cette définition est réalisée dans `OuvrageSynchronizer` qui est couplé à la classe `Ouvrage` et joue le rôle de configuration. De plus, le constructeur `OuvrageSynchronizer` (lignes 23-27) utilise les méthodes définies dans `Synchronizer` pour définir les règles de synchronisation (méthodes auto-exclusives ou mutuellement exclusives).

3.3. Approche par transformation d'interprète

Cette section décrit les techniques de mise en œuvre qui s'attachent à faire varier uniquement l'interprète dans le couple $\langle \text{Programme}, \text{Interprète} \rangle$ (cf. section 3.1). Le point de départ est l'aspect de base qui représente un programme P_0 dont l'interprétation par un interprète par défaut I_0 produit un résultat R_0 . L'idée de cette

```

“Aspect de base”
public class Ouvrage {
    String titre, auteur;
    float prix;
    public void écrirePrix(float nouveauPrix) {
        prix = nouveauPrix;}
    public float lirePrix() {
        return prix;}
}

“Aspect Synchronisation”
1: abstract aspect Synchronizer{
2:     // Les points de jonctions utilisés sont à définir dans les sous-aspects
3:     abstract pointcut synchronizeCut();
4:     // Traitements à associer aux points de jonction
5:     before ( ): synchronizeCut() {
6:         // Placer les verrous associés à la méthode exécutée
7:     }
8:     after ( ): synchronizeCut() {
9:         // Ôter les verrous associés à la méthode exécutée
10:    }
11:    public void addSelfex(String methName){
12:        // Ajouter un verrou interdisant 2 exécutions simultanées de “methName”
13:    }
14:    public void addMutex(String [ ] methNames){
15:        // Ajouter un verrou interdisant l’exécution simultanée des méthodes “methNames”
16:    }
17: }

“Configuration”
18: aspect OuvrageSynchronizer extends Synchronizer {
19:     // Points de jonction utilisés
20:     pointcut synchronizeCut():
21:         receptions(void Ouvrage.écrirePrix(float))||
22:         receptions(float Ouvrage.lirePrix( ));
23:     OuvrageSynchronizer() {
24:         // Règles de synchronisation
25:         addSelfex("écrirePrix");
26:         addMutex(new String[ ] {"lirePrix", "écrirePrix"});
27:     }
28: }

```

Figure 4. Synchronisation de la classe Ouvrage dans ASPECTJ version 0.7 Bêta 9

approche est de construire un nouvel interprète I_1 à partir des aspects non-fonctionnels et de l'interprète par défaut I_0 pour produire un nouveau résultat R_1 , et ce, sans changer le programme P_0 . Il s'agit donc de modifier la sémantique d'interprétation du programme P_0 pour prendre en compte les aspects non-fonctionnels.

3.3.1. Description de l'approche par transformation d'interprète

Un interprète est une entité qui est le plus souvent complexe et difficile à comprendre et à modifier. En effet, sélectionner les mécanismes internes pertinents et les transformer afin d'introduire les aspects non-fonctionnels ne sont pas des tâches triviales. D'où le besoin d'un cadre qui guide et facilite les transformations. Les différents représentants de l'approche par transformation d'interprète que nous avons pu étudier [WAT 88] [KIC 94] [STR 96] [DEM 97] [LUN 98] [PRY 99] ont tous adopté un même cadre : la *réflexion*².

D'après Brian Smith [SMI 90], la réflexion est *la capacité d'une entité à s'auto-représenter et plus généralement à se manipuler elle-même, de la même manière qu'elle représente et manipule son domaine d'application premier*. La réflexion caractérise donc la propriété d'un système capable de raisonner et d'agir sur lui-même. Ainsi, un système réflexif est capable de contrôler son propre fonctionnement et de l'adapter en fonction des évolutions des besoins ou du domaine d'application du système.

La réflexion permet de séparer naturellement l'aspect de base des aspects non-fonctionnels. En effet, une application développée à l'aide d'un langage réflexif comporte deux niveaux³ : *un niveau de base* et *un niveau méta* (cf. figure 5). Le niveau de base décrit les services réalisés par l'application (i.e. le "Quoi"), et représente donc l'aspect de base. Quant au niveau méta, il décrit la manière d'interpréter le niveau de base (i.e. le "Comment"), et représente donc l'ensemble des aspects non-fonctionnels. La séparation des aspects non-fonctionnels les uns des autres peut néanmoins être obtenue en structurant le méta-niveau de sorte que chaque classe décrive un unique aspect non-fonctionnel.

Le méta-niveau est peuplé d'objets particuliers appelés *méta-objets*. Ces méta-objets peuvent être liés à des objets du niveau de base par l'intermédiaire d'un lien appelé *lien méta*. L'activité de chaque objet du niveau de base est régie par les méta-objets auxquels il est associé au travers le lien méta. Par exemple, les méthodes invoquées sur un objet O sont interprétées par les méta-objets associés à O .

La configuration des aspects non-fonctionnels consiste à définir pour chaque objet du niveau de base, le(s) méta-objet(s) chargé(s) d'en contrôler l'exécution. Cette définition s'accompagne de la désignation des points de jonction où les méta-objets

2. Certains auteurs préfèrent utiliser le terme *réflexivité*.

3. Une application réflexive peut comporter virtuellement une infinité de niveaux [SMI 90]. Mais, nous limitons la discussion ici aux deux premiers niveaux qui sont les seuls à nous intéresser ici.

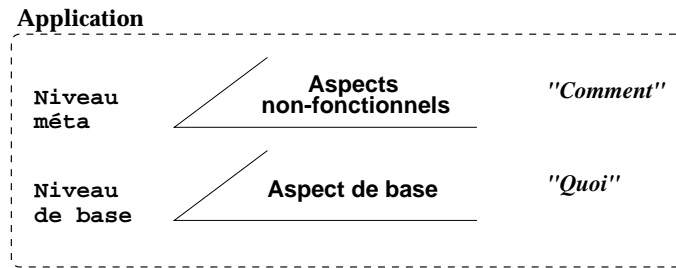


Figure 5. Correspondance entre aspects et niveaux d'une application réflexive

doivent intervenir. En dehors des points de jonction ainsi désignés, l'interprétation du niveau de base est identique à celle réalisée avec l'interprète par défaut.

Partant d'une configuration donnée, la composition des aspects est réalisée en deux temps. D'abord, l'interprète par défaut est étendu avec les définitions des méta-objets et donc avec les définitions des aspects non-fonctionnels. Puis, l'aspect de base est composé avec les aspects non-fonctionnels par l'intermédiaire du lien méta.

La difficulté de cette approche consiste à composer les méta-objets représentant des aspects non-fonctionnels qui interviennent à un même point de jonction. Cette composition se traduit soit par la composition de méta-objets (coopération de méta-objets [MUL 95] [OLI 98]), soit par la composition de classes de méta-objets (notamment par héritage [BOU 99a]).

Notons que les mises en œuvre basées sur la *réflexion à la compilation*⁴ telles que OPENC++ version 2 [CHI 95] ou OPENJAVA [CHI 98] ne font pas partie de l'approche par transformation d'interprète. En effet, elles utilisent la représentation des éléments de programmes (classes, méthodes, ...) sous forme d'objets à la compilation comme cadre pour opérer la transformation de programme. En revanche, des mises en œuvre telles que les *filtres de composition*⁵ [AKS 92] [AKS 98] font partie des approches par transformation d'interprète. Cette dernière s'apparente à une mise en œuvre réflexive, puisque les filtres jouent le rôle de méta-objets qui interprètent les invocations de méthodes émises ou reçues par les objets.

3.3.2. Exemple d'un représentant de l'approche par transformation d'interprète : le langage MetaclassTalk

Le langage METACLASSTALK est une extension réflexive de SMALLTALK où les classes jouent le rôle de méta-objets [BOU 99b]. Pour ce faire, les classes sont instances de *méta-classes* (i.e. classes de classes) qui définissent des méthodes pour contrôler la structure (allocation et accès) et le comportement des instances (envoi,

4. *Compile-time reflection.*

5. *Composition Filters.*

réception de messages). Les définitions des métaclasse décrivent donc certains mécanismes d'exécution des objets du niveau de base et donc les aspects non-fonctionnels. En choisissant de ne définir qu'un seul aspect non-fonctionnel par métaclasse, META-CLASSTALK sépare clairement les définitions des différents aspects non-fonctionnels. La séparation entre les aspects non-fonctionnels et l'aspect de base est quant à elle automatique, puisque l'aspect de base est défini dans les classes du niveau de base alors que les aspects non-fonctionnels sont définis dans les métaclasse.

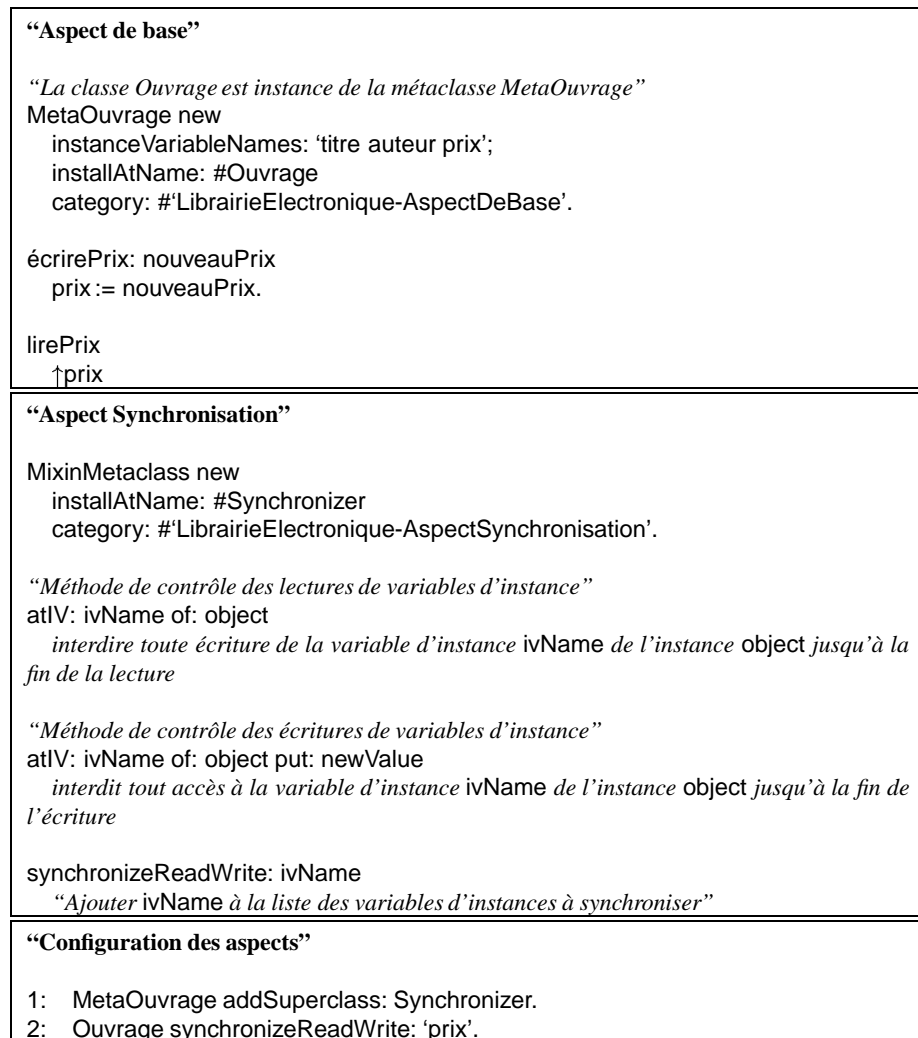


Figure 6. Représentation des aspects à l'aide des métaclasse

Reprenons notre exemple de synchronisation de la classe `Ouvrage`, dans le cadre de la programmation par métaclasse en `METACLASSTALK`. L'aspect de base correspond à la définition des services réalisés par un ouvrage comme par exemple la possibilité de lire ou de modifier le prix. La définition de la classe `Ouvrage` décrit ces services indépendamment de la manière de les réaliser (cf. figure 6). L'aspect synchronisation apparaît sous la forme de la définition de la métaclasse `Synchronizer` qui redéfinit les méthodes de contrôle des lectures/écritures des variables d'instance (respectivement “`atIV: ivName of: object`” et “`atIV: ivName of: object put: newValue`”) pour introduire la synchronisation. Ces méthodes sont implicitement appelées pour réaliser les accès à la structure des objets du niveau de base.

La définition de la métaclasse `Synchronizer` étant indépendante de la définition de la classe `Ouvrage`, nous avons donc une séparation claire entre l'aspect non-fonctionnel synchronisation et l'aspect de base. Grâce à cette séparation, il est possible de réutiliser un aspect indépendamment de l'autre. De manière générale, il est possible de disposer de bibliothèques de métaclasse qui seront réutilisées dans diverses applications.

La configuration de l'aspect synchronisation est réalisée grâce aux expressions qui apparaissent dans le module de configuration. La première d'entre elles (ligne 1 de la figure 6) rend `MetaOuvrage` (la métaclasse de `Ouvrage`) sous-classe de la métaclasse `Synchronizer`. Le comportement de la classe `Ouvrage` s'en trouve étendu, notamment avec la méthode `synchronizeReadWrite`: utilisée dans la ligne 2 de la configuration. Cette méthode prend comme argument le nom de la variable d'instance `prix` dont les accès représentent des points de jonction utilisés par l'aspect synchronisation.

3.4. Approches hybrides

Dans les deux sections précédentes, nous avons présenté deux approches opposées pour la mise en œuvre de la programmation par aspects. Entre ces deux extrêmes, il existe des approches hybrides qui transforment aussi bien le programme que l'interprète, pour introduire les aspects non-fonctionnels. En effet, la transformation du programme ou celle de l'interprète ne sont pas des tâches aisées. Suivant le langage utilisé et les outils disponibles, l'une peut s'avérer plus adaptée que l'autre pour certains aspects particuliers. D'où l'intérêt des approches hybrides.

Pour illustrer une approche hybride, prenons l'exemple de deux aspects non-fonctionnels (optimisation et synchronisation), qui sont plus faciles à réaliser avec des techniques de mise en œuvre différentes. Dans notre exemple, la transformation de programme est plus adaptée pour l'aspect optimisation, alors que la transformation d'interprète est plus adaptée pour l'aspect synchronisation.

Notre aspect optimisation consiste à limiter le nombre de boucles `for` d'un programme `JAVA`. La figure 7(a) montre l'exemple d'une classe de vecteurs qui définit une méthode `plus()` utilisant une boucle `for` pour réaliser l'addition entre vecteurs. Un

| | |
|--|--|
| <pre> Vecteur a, b, c, somme; ... somme = a.plus(b.plus(c)); ... class Vecteur{ ... Vecteur plus(Vecteur other){ Vecteur resultat = new Vecteur(this.size()); for(int i = 0; i < this.size(); i++){ resultat.putElement(i, this.getElement(i) + other.getElement(i)); } return resultat; } } </pre> | <pre> Vecteur a, b, c, somme; ... somme = new Vecteur(a.size()); for(int i = 0; i < this.size(); i++){ somme.putElement(i, a.getElement(i)+ b.getElement(i)+ c.getElement(i)); } ... </pre> |
| (a) | (b) |

Figure 7. Aspect optimisation

programme qui additionne trois vecteurs conduit à deux invocations de cette méthode. Ainsi, deux boucles `for` sont exécutées. L'optimisation de ce programme est plus facile à réaliser par transformation de programme. Elle consiste à fusionner les deux boucles comme le montre la figure 7(b). Une solution utilisant la transformation d'interprète est difficile à mettre en œuvre, car elle nécessite que l'interprète analyse le programme avant de l'exécuter.

Notre aspect synchronisation consiste à synchroniser les accès à des champs `public` d'une classe `JAVA`. Étant `public`, ces champs peuvent être accédés directement dans différentes classes. De ce fait, la synchronisation par transformation de programme est difficile à réaliser, car elle nécessite une analyse lourde de type pour trouver l'ensemble des accès. En comparaison, il est aisé de mettre en œuvre cette synchronisation par transformation de l'interprète, puisque tous les accès aux champs d'un objet sont réalisés par l'interprète.

`JAVASSIST` [CHI 00] est l'un des rares représentants de l'approche hybride. En effet, il offre à la fois une bibliothèque d'outils de transformation de programme et des capacités réflexives qui permettent de transformer l'interprète. Il est ainsi possible de choisir d'opérer les transformations du programme ou de l'interprète suivant que l'on souhaite privilégier l'efficacité ou la flexibilité. Les aspects non-fonctionnels pour lesquels l'efficacité est primordiale seraient introduits par transformation de programme, alors que les aspects où la flexibilité est la plus importante seraient introduits par transformation de l'interprète.

4. Programmation par aspects : acquis et défis

À ce jour, la programmation par aspects a été expérimentée pour réaliser une application “réelle” [KER 99]. Il s’agit d’un système d’enseignement à distance via le web. La programmation par aspects a permis de faciliter l’évolution de cette application et de supporter sa montée en charge progressive. Ainsi, les interventions ont été limitées aux seuls aspects non-fonctionnels qui reflétaient l’évolution. Les autres aspects ont pu être réutilisés.

Si cette première et unique expérimentation “grandeur nature” a été un succès, il n’en reste pas moins que les mises en œuvre existantes n’ont pas résolu tous les problèmes que soulève ce paradigme. Nous proposons dans cette section de faire un point sur ces différents problèmes. Il ne s’agit pas d’évaluer les différentes approches identifiées, mais de faire un bilan des problèmes résolus et de ceux qui restent ouverts. Nous n’abordons pas ici les problèmes de conception, comme par exemple l’identification des aspects des applications, qui sortent du cadre de cet article. Mais, nous nous intéressons exclusivement à quelques problèmes qui doivent être traités dans les mises en œuvre de la programmation par aspects, à savoir :

- la *composition* des aspects,
- leur *réutilisabilité*,
- l’*expressivité* de la mise en œuvre,
- la *facilité de programmation*,
- et les possibilités d’*adaptabilité dynamique* qu’elle offre.

4.1. Composition

La composition des aspects entre eux est une condition cruciale pour l’exploitation d’une mise en œuvre de la programmation par aspects. Néanmoins, si des solutions existent, aucune n’est totalement satisfaisante.

La difficulté de composition apparaît lorsque des aspects non-fonctionnels différents sont composés avec l’aspect de base à travers un même point de jonction. Dans de nombreux cas, les transformations (de programme ou d’interprète) correspondant aux aspects non-fonctionnels “en conflit” conduisent à des résultats différents suivant qu’un aspect ou un autre est privilégié.

Pour illustrer ce problème, considérons une application qui réalise des calculs coûteux dont les résultats sont confidentiels. Cette application dispose de deux aspects non-fonctionnels que sont l’optimisation et la sécurité. L’optimisation consiste à utiliser une anté-mémoire (*cache*) où sont stockés les résultats de calculs. La sécurité consiste à authentifier les demandeurs de calculs, avant de réaliser les traitements. La composition de ces deux aspects, optimisation et sécurité, est critique. En effet, il est impératif que l’authentification du demandeur d’un calcul ait lieu avant d’extraire le résultat du calcul de l’anté-mémoire. Autrement dit, la composition des deux aspects

sécurité et optimisation n'est pas commutative, puisqu'il est impératif d'ordonner les aspects de manière à privilégier l'aspect sécurité. Cependant, les aspects étant définis indépendamment les uns des autres, aucune information sur leur composition n'est disponible. Dès lors, l'outil de composition ne dispose pas d'assez d'informations pour déterminer le résultat attendu par les développeurs.

Pour tenter de résoudre ce problème, quelques mises en œuvre de la programmation par aspects permettent aux développeurs d'étendre les définitions des aspects non-fonctionnels avec une information qui permet à l'outil de composition de décider de l'aspect à privilégier [LOP 98b] [LIE 99] [BOU 99a]. Cependant, ces solutions ne permettent que de restreindre le champ d'apparition du problème de composition des aspects, sans l'éliminer complètement. En effet, elles ne permettent que de construire une relation d'ordre partiel entre les aspects non-fonctionnels. Certains cas restent indécidables et nécessitent l'intervention des développeurs pour guider la composition.

4.2. Réutilisabilité

La réutilisabilité des aspects dépend du "niveau" de séparation offert par la mise en œuvre. Ce niveau est maximal lorsque l'aspect de base, les aspects non-fonctionnels et la configuration sont tous découplés et donc définis séparément les uns des autres. Dans ce cas, il est possible de réutiliser n'importe quel aspect indépendamment des autres.

Avec les mises en œuvre qui s'appuient sur la transformation de programme, le problème de réutilisation peut se poser pour les aspects non-fonctionnels. En effet, la réutilisation des aspects non-fonctionnels peut être compromise lorsque leurs définitions comportent des règles de transformations qui référencent directement l'aspect de base. Ce problème rencontré notamment avec les premières versions du langage ASPECTJ [LOP 98b], est résolu dans d'autres mises en œuvre [BEU 99] [LIE 99] qui séparent les définitions des aspects non-fonctionnels de la configuration des aspects.

Dans les mises en œuvre qui opèrent une transformation de l'interprète, l'utilisation de la réflexion constitue un cadre qui sépare naturellement l'aspect de base des aspects non-fonctionnels. En revanche, ce cadre ne garantit pas l'isolation de la configuration et la généralité des aspects. Ce problème admet cependant une solution [ROB 99] [WEL 99] [BOU 00] qui doit être prise en compte dans la mise en œuvre pour prévoir une infrastructure qui permet d'isoler la configuration.

4.3. Expressivité

L'expressivité d'une mise en œuvre désigne les possibilités qu'elle offre pour représenter différents aspects non-fonctionnels. Le nombre et la variété de ces aspects sont directement corrélés à la variété des points de jonction accessibles dans la mise en œuvre. Une approche donnant accès à peu de points de jonction, présente une expressivité limitée car le développeur éprouve des difficultés à programmer certains aspects.

Plusieurs points de jonction ont été identifiés et sont utilisés dans différentes mises en œuvre. Ainsi, le début et la terminaison d'une invocation de méthode constituent deux types de points de jonction parmi les plus utilisés. Il semble acquis que de tels points de jonction doivent être accessibles dans toute mise en œuvre de la programmation par aspects.

4.4. Facilité de programmation

La facilité de programmation des aspects non-fonctionnels et de leur configuration sont des éléments déterminants pour le choix d'une mise en œuvre particulière. Dans de nombreuses mises en œuvre, cette programmation est une tâche difficile puisqu'elle s'appuie sur des langages impératifs généralistes (SMALLTALK, JAVA, ...). Pour remédier à ce problème, Gregor Kiczales et al. [KIC 97] ont suggéré d'utiliser des langages déclaratifs, éventuellement dédiés à des aspects non-fonctionnels particuliers. Si cette idée est séduisante, elle pré-suppose de connaître l'ensemble des aspects non-fonctionnels (ensemble a priori infini) pour construire l'ensemble des langages dédiés.

Une voie prometteuse consiste à utiliser des outils de développement qui prennent en compte la notion d'aspect et de points de jonction. C'est le cas par exemple des extensions de Emacs, ForteJava et plus récemment JBuilder développées pour AspectJ [XER 00]. Ces outils simplifient la navigation entre aspect de base et aspects non-fonctionnels.

4.5. Adaptabilité dynamique

L'adaptabilité dynamique correspond à la possibilité de changer, d'ajouter ou de supprimer, à l'exécution, les aspects non-fonctionnels d'une application, de manière à faire évoluer, dynamiquement, les mécanismes d'exécution de l'application. Cette possibilité permet de répondre à la problématique délicate des systèmes devant évoluer sans interrompre leur exécution (administration de systèmes, chaîne de montages, ...). Ces évolutions peuvent se faire à l'initiative soit des développeurs, soit des systèmes qui s'auto-modifiraient pour prendre en compte les évolutions de leurs environnements d'exécution.

Pendant longtemps, l'adaptabilité dynamique des aspects non-fonctionnels n'avait pas été abordée pour les mises en œuvre de la programmation par aspects fondées sur la transformation de programme. Cette question a finalement été traitée dans des travaux récents [BOI 00], qui ouvrent ainsi la voie en la matière en définissant un cadre permettant de changer dynamiquement les aspects non-fonctionnels des applications.

Avec les mises en œuvre basées sur la transformation de l'interprète, de tels changements ainsi que l'ajout-suppression des aspects non-fonctionnels sont possibles, notamment par l'utilisation de la réflexion. En effet, les entités du méta-niveau représentant les aspects non-fonctionnels dans les systèmes réflexifs peuvent être dynamiquement modifiées ou remplacées. À titre d'exemple, ces possibilités ont été exploitées

pour adapter dynamiquement les stratégies de représentation et d'exécution d'un bus logiciel CORBA [LED 99].

5. Conclusion

La programmation par aspects semble être une voie prometteuse du génie logiciel pour réduire les coûts de développement, d'évolution et de maintenance des logiciels. En effet, elle augmente les opportunités de réutilisation en définissant une nouvelle façon de structurer les applications. Pour une application donnée, cette structuration consiste à décrire séparément l'aspect de base, les différents aspects non-fonctionnels, et la configuration des aspects. L'aspect de base correspond aux définitions des services que doit réaliser l'application, alors que les aspects non-fonctionnels décrivent la manière de réaliser ces services. La configuration des aspects définit le détail des liaisons entre aspect de base et aspects non-fonctionnels. L'application finale est produite par la composition des différents aspects avec comme liant (*glue*) la configuration des aspects.

Dans cet article, nous avons identifié et étudié trois approches pour la mise en œuvre de la programmation par aspects. Elles partagent le fait de considérer les aspects non-fonctionnels comme des transformations qui sont opérées sur le couple $\langle \text{Programme}, \text{Interprète} \rangle$. L'aspect de base est en effet considéré comme un programme écrit pour un interprète par défaut. Les aspects non-fonctionnels visent à modifier le résultat produit par ce couple. Pour réaliser cette modification, les mises en œuvre de la programmation par aspects peuvent s'appuyer soit sur la transformation du programme, soit sur la transformation de l'interprète. Une troisième approche hybride est bien entendu possible et consiste à transformer aussi bien le programme que l'interprète.

La mise en œuvre de la programmation par aspects se trouve confrontée à différents problèmes et notamment à l'automatisation de la composition des aspects et à l'expressivité nécessaire dans les langages utilisés pour la définition des aspects non-fonctionnels. Ainsi, le développeur doit intervenir dans le processus de composition pour gérer les conflits entre les aspects non-fonctionnels qui s'appuient sur les mêmes points de jonction. Par ailleurs, l'identification de l'ensemble des points de jonction nécessaires pour définir facilement tous les aspects non-fonctionnels est encore une question ouverte. La réponse à cette question permettrait de définir des langages plus expressifs pour la définition des aspects non-fonctionnels.

Une nouvelle tendance consiste à généraliser le concept d'aspect pour l'étendre à toutes les propriétés *transversales* des applications [BOU 99b] [IBM 00] [XER 00]. Ainsi, toute propriété transversale – qu'elle représente un mécanisme d'exécution ou une "fonctionnalité" de l'application – peut être isolée sous forme d'aspect. Cette évolution introduit alors la notion d'**aspects fonctionnels**. Un aspect fonctionnel est la définition des interactions entre un ensemble d'objets en vue de réaliser un service donné ("fonctionnalité"). La fonction de recherche dans un document, la mise

à jour de l’affichage d’une application représentent des exemples d’aspects fonctionnels. La notion d’aspect de base persiste, mais son rôle est restreint à la définition du squelette de l’application. La généralisation du concept d’aspect élargit considérablement le champ d’applications de la programmation par aspects et soulève ainsi de nouvelles questions. Comment représenter un aspect fonctionnel ? Quels sont les nouveaux points de jonctions à utiliser ? Comment composer les aspects fonctionnels avec les autres aspects ? Quelles sont les techniques possibles de mise en œuvre ? Ces questions rejoignent les préoccupations de plusieurs travaux, comme la programmation par contextes [VAN 97] [VAN 98] ou la programmation adaptative [LIE 96] [MEZ 98].

En résumé, la programmation par aspects est un paradigme en pleine effervescence, qui laisse entrevoir de nouvelles perspectives de réutilisation. Même si de nombreuses questions restent ouvertes, nous sommes enclins à croire que la programmation par aspects connaîtra dans les années à venir un essor semblable à celui qu’a connu la programmation par objets !

Remerciements

Les auteurs tiennent à remercier sincèrement les différentes personnes qui ont bien voulu relire et commenter les différentes versions de ce papier.

6. Bibliographie

- [AKS 92] AKSIT M., BERGMANS L., « An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach », *Proceedings of ECOOP’92*, 1992.
- [AKS 98] AKSIT M., TEKNERDOGAN B., « Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters », Lopes et al. [LOP 98a].
- [BEU 99] BEUGNARD A., « How to make aspects re-usable, a proposition », Black et al. [BLA 99].
- [BLA 99] BLACK A., KENDALL L., AKSIT M., BERGMANS L., Eds., *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP’99*, Lisbon, Portugal, Juin 1999.
- [BOI 00] BOINOT P., MARLET R., MULLER G., NOYÉ J., CONSEL C., « A Declarative Approach for Designing and Developing Adaptive Components », *Proceedings of the 15th IEEE International Conference on Automated Software Engineering, ASE’00*, Grenoble, Septembre 2000.
- [BOU 99a] BOURAQADI-SAÂDANI M. N., « Un cadre réflexif pour la programmation par aspects », *Proceedings of LMO’99*, Villefranche sur Mer, Janvier 1999, Hermès.
- [BOU 99b] BOURAQADI-SAÂDANI N. M. N., « UN MOP Smalltalk pour l’étude de la composition et de la compatibilité des métaclases. Application à la programmation par aspects. », Thèse de Doctorat, Université de Nantes, École des Mines de Nantes, Juillet 1999.
- [BOU 00] BOURAQADI N. M. N., « Concern Oriented Programming using Reflection », OOPSLA 2000 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Octobre 2000.

- [CHI 95] CHIBA S., « A Metaobject Protocol for C++ », *Proceeding of OOPSLA'95*, 1995, p. 285-299.
- [CHI 98] CHIBA S., TATSUBORI M., « Yet Another java.lang.Class », *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, Brussels, Belgium, Juillet 1998.
- [CHI 00] CHIBA S., « Load-time Structural Reflection in Java », BERTINO E., Ed., *Proceedings of ECOOP 2000*, Sophia Antipolis and Cannes, Juin 2000.
- [DEM 97] DEMPSEY J., CAHILL V., « Aspects of System Support for Distributed Computing », LOPES C., KICZALES G., TEKINERDOGAN B., MENS K., Eds., *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'97*, Jyväskylä, Finland, Juin 1997.
- [DIJ 76] DIJKSTRA E. W., *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J., 1976.
- [DUC 97] DUCASSE S., « Réification des schémas de conception: une expérience », *Proceedings of LMO'97*, Roscoff, October 1997, Hermès, p. 95-110.
- [FRA 98] FRADET P., SÜDHOLT M., « AOP: towards a generic framework using program transformation and analysis », Lopes et al. [LOP 98a].
- [GAM 95] GAMMA E., HELEM R., JOHNSON R., VLISSIDES J., *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [HUR 95] HURSCH W. L., LOPES C. V., « Separation of Concerns », rapport n° NU-CCS-95-03, Février 1995, College of Computer Science, Northeastern University, Boston, MA.
- [IBM 00] IBM, « Multi-Dimensional Separation of Concerns: Software Engineering using HyperSpaces », <http://www.research.ibm.com/hyperspace/>, 2000.
- [KER 99] KERSTEN M. A., MURPHY G. C., « Atlas: A Case Study in Building a Web-based Learning Environment using Aspect-oriented Programming », NORTHROP L., Ed., *Proceedings of OOPSLA'99*, Denver, Colorado, USA, Novembre 1999.
- [KIC 94] KICZALES G., PÆPCKE A., « Open Implementations and Metaobject Protocols », Expanded tutorial notes, 1994.
- [KIC 96] KICZALES G., « Beyond The Black Box: Open Implementation », *IEEE Software*, vol. 13, n° 1, 1996.
- [KIC 97] KICZALES G., LAMPING J., MENDHEKAR A., MAEDA C., LOPES C., LOINGTIER J.-M., IRWIN J., « Aspect-Oriented Programming », AKŞIT M., MATSUOKA S., Eds., *Proceedings of ECOOP'97*, n° 1241 LNCS, Springer-Verlag, Juin 1997, p. 220-242.
- [LED 99] LEDOUX T., « OpenCorba: A Reflective Open Broker », COINTE P., Ed., *Meta-Level Architectures and Reflection*, n° 1616 LNCS, Saint-Malo, Juillet 1999, Springer, p. 197-214, *Proceedings of Reflection'99*.
- [LIE 96] LIEBERHERR K. J., *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, Boston, 1996, ISBN 0-534-94602-X.
- [LIE 99] LIEBERHERR K., LORENZ D., MEZINI M., « Programming with Aspectual Components », rapport n° NU-CCS-99-01, Mars 1999, College of Computer Science, Northeastern University, Boston, MA.
- [LOP 98a] LOPES C., KICZALES G., TEKINERDOGAN B., DE MEUTER W., Eds., *Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'98*, Brussels, Belgium, Juillet 1998.

- [LOP 98b] LOPES C. V., KICZALES G., « *ECOOP'98 Workshop Reader* », Chapitre Recent Developments in AspectJ, N° 1543 LNCS, Springer-Verlag, 1998.
- [LUN 98] LUNAU C. P., « Is Composition of Metaobjets = Aspect Oriented Programming », Lopes et al. [LOP 98a].
- [MAT 93] MATSUOKA S., YONEZAWA A., « *Research Directions in Concurrent Object-Oriented Programming* », Chapitre Analysis of inheritance anomaly in object-oriented concurrent programming languages, MIT Press, 1993.
- [MEZ 98] MEZINNI M., LIEBERHERR K., « Adaptive Plug-and-Play Components for Evolutionary Software Development », *Proceedings of OOPSLA'98*, Vancouver, Canada, October 1998, ACM, p. 97-116.
- [MUL 95] MULET P., MALENFANT J., COINTE P., « Towards a Methodology for Explicit Composition of MetaObjects », *Proceedings of OOPSLA'95*, Austin, Texas, Octobre 1995, p. 316-330.
- [OLI 98] OLIVA A., BUZATO L. E., « Composition of Meta-Objects in Guarana », *Proceedings of the OOPSLA'99 Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada, Octobre 1998.
- [PRY 99] PRYOR J., BASTÁN N., « A Reflective Architecture for the Support of Aspect Oriented Programming in Smalltalk », Black et al. [BLA 99].
- [ROB 99] ROBBEN B., VANHAUTE B., JOOSEN W., VARBAETEN P., « Non-fonctionnal Policies », COINTE P., Ed., *Meta-Level Architectures and Reflection*, n° 1616 LNCS, Saint-Malo, Juillet 1999, Springer, p. 74-94, *Proceedings of Reflection'99*.
- [SMI 90] SMITH B. C., « What Do You Mean, Meta? », *Workshop on Reflection and Metalevel Architectures in OO Programming, ECOOP/OOPSLA'90*, Ottawa, Ontario, Canda, Octobre 1990.
- [STE 96] STEYAERT P., LUCAS C., MENS K., D'HONDT T., « Reuse Contracts: Managing the Evolution of Reusable Assets », *Proceedings of OOPSLA'96*, 1996, p. 268-285.
- [STR 96] STROUD R. J., WU Z., « *Advances in Object-Oriented Metalevel Architectures and Reflection* », Chapitre Using Metaobject Protocols to Satisfy Non-Functional Requirements, p. 31-52, CRC Press, 1996.
- [VAN 97] VANWORMHOUDT G., CARRÉ B., DEBRAUWER L., « Programmation par objets et contextes fonctionnels. Application de CROME à Smalltalk », *Proceedings of LMO'97*, Roscoff, October 1997, Hermès, p. 223-239.
- [VAN 98] VANWORMHOUDT G., « Programmation par contextes en Smalltalk », *L'Objet*, vol. 3, n° 4, 1998, p. 429-444.
- [WAT 88] WATANABE T., YONEZAWA A., « Reflection in an Object-Oriented Concurrent Language », *Proceedings of OOPSLA'88*, ACM, 1988.
- [WEL 99] WELCH I., STROUD R., « From Dalang to Kava - the Evolution of a Reflective Java Extension », COINTE P., Ed., *Proceedings of Reflection'99*, n° 1616 Lecture Notes in Computer Science, Saint-Malo, Juillet 1999, Springer, p. 2-21.
- [XER 00] XEROX, « aspectj.org Site », <http://www.aspectj.org>, 2000.

Article reçu le 15 novembre 1999.

Version révisée le 15 juin 2000.

Rédacteur responsable : CHRISTOPHE DONY

Noury M. N. Bouraqadi-Saâdani est actuellement ingénieur de recherche au département informatique de l'École des Mines de Nantes. Il occupe ce poste au sein de l'équipe objets, composants et modèles depuis 1999, année d'obtention de son Doctorat en informatique. Ses travaux de recherche portent sur l'utilisation de la réflexion pour la construction des systèmes distribués et à agents mobiles. Il étudie également les problèmes de composition et de programmation par aspects.

Thomas Ledoux est enseignant-chercheur au département informatique de l'École des Mines de Nantes (EMN). Il a obtenu son Doctorat en informatique sur le thème de la réflexion dans les systèmes répartis en 1998. Ses principaux domaines d'intérêt sont les architectures réflexives, la méta-programmation, la programmation par aspects, le middleware et les agents mobiles. Il est actuellement responsable technique pour la partie EMN du projet RNTL ARCAD qui a pour objectif de définir une infrastructure réflexive extensible pour l'exécution de composants répartis adaptables.