

# Composants et aspects

Jacques Noyé, Rémi Douence, Mario Südholt  
Projet Obasco EMN - INRIA  
École des Mines de Nantes  
La Chantrerie - 4, rue Alfred Kastler  
BP 20722  
44307 Nantes Cedex 3  
{Jacques.Noye, Remi.Douence, Mario.Sudholt}@emn.fr

23 septembre 2004

## Résumé

La programmation par aspects s'est développée dans le contexte de la programmation par objets comme une réponse à la difficulté de localiser et d'encapsuler certaines préoccupations, dites non fonctionnelles, en utilisant les moyens de structurations classiques (dont les interfaces sont essentiellement définies en terme d'appels de fonctions, dans un sens très général). Les aspects peuvent donc être vus comme tout à fait complémentaires, tout en étant très proches des composants. Ce chapitre se propose de faire le point sur la complémentarité et l'intégration de la programmation par composants et par aspects. Il présente la programmation par aspects, son intérêt dans le cadre des modèles industriels de composants ainsi qu'un certain nombre de langages de composants et d'aspects qui illustrent les rapports entre les concepts de composants et d'aspects.

## 1 Introduction

Comme les composants, les aspects répondent à une problématique de séparation des préoccupations [Dij76, Par72]. Il s'agit de regrouper dans une entité logicielle distincte du code (ou des spécifications, en adoptant une définition élargie du terme « logiciel ») correspondant à une préoccupation donnée, sachant qu'une hypothèse de base est qu'il est possible de décomposer le système ou les systèmes auxquels on s'intéresse en des préoccupations relativement indépendantes, qu'il est possible de traiter (décrire, analyser, implémenter...) sans devoir se référer aux autres préoccupations ou tout au moins en s'y référant de manière marginale et bien définie (typiquement au moyen de notions d'interface ou de contrat). L'existence de ces entités donne de la structure au logiciel et permet ainsi de mieux en maîtriser la complexité [Bro95]. En particulier, dans un contexte de programmation à grande échelle [DK76], ces entités deviennent des unités de configuration et de déploiement du logiciel.

De ce point de vue, composants et aspects ont pour ancêtres communs les modules [PDN86]. Ces notions se sont toutefois développées de manière indépendante dans un contexte où la notion de module s'était retrouvée quelque peu effacée devant la pénétration des langages à objets et l'idée que les classes fournissaient une alternative avantageuse aux modules. L'importance prise par les composants et les aspects a montré que les capacités de structuration du couple classe/objet ne sont pas suffisantes dans le contexte des logiciels actuels. On se trouve aujourd'hui à un moment charnière où ce point semble acquis et où suffisamment de progrès ont été faits à la fois dans le domaine de la programmation par composants et dans le domaine de la programmation par aspects pour que se pose la question de la complémentarité et de l'intégration de ces deux approches.

Ce chapitre se propose de faire le point sur cette question. Pour ce faire, nous présentons d'abord la programmation par aspects. Nous explicitons ensuite la pertinence de cette approche dans le cadre des modèles industriels de composants et analysons six propositions

concrètes d’extension de ces modèles introduisant explicitement des aspects. Nous étudions finalement un certain nombre de langages académiques de composants et d’aspects qui illustrent les rapports entre les concepts de composants et d’aspects et ouvrent la voie vers de nouveaux langages de programmation intégrant les deux approches. Afin de focaliser la discussion nous nous intéresserons principalement à la programmation et n’aborderons que succinctement la production de descriptions intermédiaires partielles et non exécutables<sup>1</sup>.

## 2 Présentation de la programmation par aspects

Dans cette section, nous introduisons la programmation par aspects. Pour ce faire nous commençons par une présentation très informelle (« la programmation par aspects expliquée à ma grand mère » en quelque sorte). Puis pour rendre nos explications plus concrètes nous proposons un survol d’AspectJ, qui est devenu une référence, en comparant les mécanismes de l’héritage à ceux de la programmation par aspects. Enfin nous généralisons cette présentation notamment en prédisant les défis que la programmation devrait relever et donc ce que le programmeur d’aspects peut attendre des outils de demain.

### 2.1 La métaphore de la maison

« Diviser pour régner » est une stratégie récurrente pour appréhender un problème complexe. Le problème complexe est décomposé en sous-problèmes simples et une solution au problème complexe est un assemblage de solutions aux sous-problèmes simples.

Prenons l’exemple d’une maison<sup>2</sup>. Pour un maçon une maison est un assemblage de pièces, une pièce est un assemblage de murs, et un mur est un assemblage de briques. Pour un peintre une maison est un assemblage de pièces, et une pièce est un assemblage de murs. Pour un électricien une maison est un assemblage de réseaux électriques, un réseau électrique est un assemblage de branches, et une branche est un assemblage de prises sur un fil.

La décomposition du maçon et celle du peintre sont compatibles (même si le peintre reste plus abstrait et ne parle pas de brique). On peut même dire que le travail du peintre étend celui du maçon en travaillant la surface que ce dernier a produit. Le peintre pourra donc utiliser le plan du maçon pour calculer les surfaces et acheter sa peinture. Malheureusement la décomposition du maçon et de l’électricien ne sont pas compatibles. Par exemple, les fils électriques traversent les murs, ainsi deux prises sur un même fil peuvent se trouver dans deux pièces différentes et deux prises dans une même pièce ne partagent pas nécessairement un fil. En conséquence, le plan complet de la maison avec murs et fils électriques est difficile à lire. Il est avantageux d’avoir plusieurs plans qui correspondent aux différents corps de métier. Par exemple, lorsqu’il s’agit de choisir un fusible, on préférera le plan de l’électricien qui fait abstraction des murs.

Le défi pour un architecte est de concevoir incrémentalement une maison en se focalisant d’abord sur la maçonnerie seule, puis sur l’électricité seule, avant de réunir ces deux préoccupations.

On retrouve cette problématique dans le domaine du logiciel : en Java un programme complexe est un assemblage de packages, et un package est un assemblage de classes. Par exemple, dans le serveur Apache on trouve une classe qui gère l’analyse syntaxique de XML et deux autres classes qui gèrent la reconnaissance de motif dans les URL [Sit01]. Mais un programme complexe est aussi un assemblage de différentes préoccupations. Par exemple, toujours dans Apache, la gestion des requêtes et la gestion des sessions sont deux préoccupations différentes qui ne respectent pas la même décomposition : le code de gestion des sessions ne peut pas être modularisé dans une classe, une sous-classe ou même un package mais se trouve éparpillé dans plusieurs classes qui gèrent les requêtes. La programmation par aspects propose de programmer de façon modulaire ces préoccupations puis de générer automatiquement le programme complet qui les réunit.

---

<sup>1</sup>En théorie, il n’est pas si facile de clairement séparer ces deux niveaux. En pratique, nous parlerons de Java, ArchJava [ACN02] et AspectJ [KHH<sup>+</sup>01] plutôt que d’UML, des langages de description d’architecture [MT00] et du *Concern Manipulation Environment* [IBM].

<sup>2</sup>Cette section ne devrait pas être prise au sérieux par quiconque ayant l’intention de bâtir une maison ; les auteurs avouent leur ignorance quant aux métiers du bâtiment.

Ces préoccupations sont très fréquentes dès que l'on considère un logiciel suffisamment complexe. On peut donc citer de nombreux exemples d'aspect potentiel : la prélecture dans un système de fichiers [CKFS01], les explications dans les solveurs de contraintes [DJ02], la stratégie d'ordonnancement dans un système d'exploitation [ÅLS<sup>+</sup>03] ou encore les rabais dans une application de commerce électronique [DMS01].

On peut résumer la programmation par aspects à l'aide de deux termes : quantification et non-anticipation [FF00]. En effet, les nouvelles préoccupations peuvent être greffées sur les programmes de base de manière déclarative grâce à la quantification. Par exemple, un aspect de sécurité pourrait dire : « *pour toutes* les communications distantes, encrypter les données ». De plus, ces nouvelles préoccupations ne sont *pas anticipées* par le programme de base. Dans le cas contraire on parle de canevas, ou de logiciel « à trous » dans lesquels viendront s'insérer les nouvelles fonctionnalités ce qui peut compliquer considérablement le programme de base. Dans le cadre de la maison, le maçon devrait truffer les murs de gaines afin d'anticiper les possibles topologies du réseau électrique. La programmation par aspects est donc une alternative à l'utilisation de canevas et évite de compliquer le programme de base.

## 2.2 AspectJ

Comme nous le verrons dans la section 2.3, la programmation par aspects ne se résume pas à AspectJ. Néanmoins AspectJ [KHH<sup>+</sup>01] est aujourd'hui une référence pour la programmation par aspects et un bon moyen d'aborder le domaine. AspectJ est un langage qui étend Java et permet de définir des aspects. AspectJ est aussi un tisseur qui mélange le code du programme de base écrit en Java avec celui des aspects afin de générer une application complète.

AspectJ peut être vu comme un macroprocesseur structuré qui crée des variantes des méthodes du programme de base en injectant des morceaux de code définis par les aspects au début, à la fin ou à la place du corps de ces méthodes. Nous tentons de proposer ici une vue moins opérationnelle d'AspectJ en comparant le mécanisme d'héritage à celui de la programmation par aspects.

Java permet la programmation incrémentale grâce à l'héritage : le programmeur va ajouter de nouvelles fonctionnalités sans modifier une ligne du programme existant. AspectJ permet la programmation incrémentale grâce aux aspects. Le programmeur va ajouter de nouvelles fonctionnalités sans modifier une ligne du programme de base. Dans cette sous-section nous illustrons AspectJ en développant différentes variantes d'un aspect de profilage qui comptabilise certains appels de méthode. Cet aspect « jouet » nous permet d'introduire AspectJ en douceur, nous renvoyons le lecteur à la distribution d'AspectJ pour des exemples d'aspects plus réalistes.

Une sous-classe définit de nouveaux champs et de nouvelles méthodes. Un aspect ressemble à une classe : il définit de nouveaux champs, de nouvelles méthodes. Par exemple, l'aspect `Profiling` définit un champ entier `nbCalls` :

```
aspect Profiling {  
  
    int nbCalls = 0;  
}
```

L'héritage définit une nouvelle classe mais il ne modifie pas à lui seul le comportement du programme initial. Cette nouvelle classe doit ensuite être *explicitement* utilisée (c'est-à-dire instanciée) par le programmeur. Un aspect ne définit pas de nouvelle classe, ni de nouvelles méthodes, mais modifie à lui seul le comportement du programme initial (le programme de base). En effet, sans autres intervention du programmeur, les actions de l'aspect sont invoquées *implicitement* par le programme de base. À cette fin le programmeur associe un *point de coupe* à chaque action d'un aspect. Un *point de coupe* est une signature de méthode du programme de base. Lorsqu'une méthode du programme de base est appelée, un *point jonction* est généré. Les actions des aspects dont les points de coupe correspondent à ce point de jonction sont appelées. Par exemple, la nouvelle version de l'aspect `Profiling` ci-dessous compte et affiche le nombre d'appels à la méthode `foo` de la classe `Foo` du programme de base :

```
aspect Profiling {
```

```

int nbCalls = 0;

void around() : call(void Foo.foo()) {
    nbCalls++;
    System.out.println(thisJoinPoint + " " + nbCalls);
    proceed();
}
}

```

A l'exécution, le point de jonction courant est représenté par la variable `thisJoinPoint` qui contient des informations sur l'appel de méthode du programme de base (par exemple, référence sur le receveur, valeur des arguments) qui peuvent être exploitées par l'action.

De la même manière qu'une méthode redéfinie dans une sous-classe utilise `super()` pour invoquer la méthode originale de la super-classe, une action d'un aspect utilise `proceed()` pour invoquer la méthode originale du programme de base. Par exemple, si `proceed()` est supprimé de l'aspect `Profiling` alors l'aspect comptera encore les appels à la méthode `foo` mais cette méthode ne sera jamais exécutée.

L'héritage (simple) associe une sous-classe à une super-classe, et une méthode `foo` redéfinie dans une sous-classe est associée (via `super`) à la méthode `foo` de la super-classe. En AspectJ, un point de coupe peut être une disjonction de plusieurs signatures et une signature peut être générique en utilisant des jokers `*`. Une méthode d'un aspect peut donc être associée via un point de coupe à plusieurs classes et méthodes du programme de base. Par exemple, en remplaçant `call(void Foo.foo())` par `call(void Foo.foo()) || call(void Bar.bar())` on compte aussi les appels à `bar`.

Nous avons vu qu'en AspectJ un point de coupe permet d'associer une même action d'un aspect à différentes classes et méthodes du programme de base. Un aspect permet donc de superposer sa propre structuration à celle du programme de base. En AspectJ, un aspect permet aussi de ne pas respecter la structuration *de l'exécution* du programme de base. Un point de coupe peut en effet aussi exprimer une dépendance dans le flot de contrôle entre deux points de jonction. Par exemple, en remplaçant `call(void Foo.foo())` par `cflow (call(void Foo.foo())) && call(void Bar.bar())` on compte les appels à `bar` qui ont lieu pendant l'exécution de la méthode `foo` et ce quelque soit la longueur de la chaîne d'appels entre ces méthodes.

L'héritage définit une nouvelle classe qui doit ensuite être instanciée explicitement par le programmeur. Un aspect définit des champs et peut donc être instancié. Un aspect n'est pas instancié explicitement mais déclarativement. Par défaut, un aspect a une unique instance. Dans notre exemple, il existe un unique compteur qui compte les appels de méthode du programme de base. AspectJ, grâce au exemple, le mot clé `pertarget` permet aussi d'associer une nouvelle instance d'un aspect à chaque objet du programme de base. La nouvelle version de l'aspect `Profiling` permet ainsi de compter les appels à la méthode `foo` indépendamment pour chaque instance de `Foo` :

```

aspect Profiling pertarget(call(void Foo.foo())) {
    int nbCalls = 0;

    void around() : call(void Foo.foo()) {
        nbCalls++;
        System.out.println(thisJoinPoint + " " + nbCalls);
        proceed();
    }
}

```

Nous arrêtons ici notre survol d'AspectJ qui offre de nombreuses autres possibilités. Par exemple, AspectJ ne s'intéresse pas uniquement aux appels de méthodes du programme de base, mais aussi à la lecture/écriture des champs, ou encore à la capture des exceptions. AspectJ permet aussi la modification structurelle du programme de base (par exemple introduction de nouveaux champs, introduction de nouvelles méthodes, modification du graphe d'héritage). Le lecteur intéressé se référera à la dernière distribution d'AspectJ disponible à [aspectj.org](http://aspectj.org).

## 2.3 La programmation par aspects après-demain

Comme il a été écrit plus haut, la programmation par aspects ne se résume pas à AspectJ, et la programmation par aspects ne se limite pas au monde objets. Il existe de nombreuses propositions. Plutôt que de les passer en revue en détail, nous identifions les principales pistes explorées et les défis à relever.

AspectJ propose un langage de points de coupe assez simple (essentiellement des signatures génériques de méthode composées avec des opérateurs logiques). Des propositions ont été faites pour enrichir ce langage et y intégrer par exemple des notions de séquence [DFS04b], de logique temporelle [ÅLS<sup>+</sup>03], de flot de contrôle [Sd03], de flot de données [MK03].

Un point de coupe spécifie où un aspect doit être tissé dans le programme de base. Lorsque deux aspects doivent être tissés à un même point, il y a conflit. Des propositions ont été faites pour détecter (statiquement au tissage [DFS02, DFS04a] ou dynamiquement à l'exécution [DS03]) ces conflits. Ils peuvent alors être résolus par le programmeur qui spécifie la composition correcte (par exemple en ordonnant les aspects). Par exemple, AspectJ a été intégré à différents environnements de programmation qui annotent les points de conflits dans le programme de base et propose l'instruction `dominate` pour ordonner les aspects. Ces notions de conflit et de composition d'aspects devraient s'affiner et fournir à la fois plus de sûreté et un plus grand pouvoir d'expression au programmeur.

AspectJ propose un langage généraliste, Java, pour définir le corps des méthodes (*advice*) des aspects. Ces méthodes peuvent donc modifier arbitrairement la sémantique du programme de base. Il est préférable de proposer des langages restreints afin d'enrichir le comportement du programme de base sans le casser [CL03]. Des propositions ont été faites dans le cadre de la sécurité (où la seule action est `exit()` pour terminer l'application avant de violer un invariant [CF00]), des ordonnanceurs de processus (où les actions se limitent à élire un processus actif et interdisent par exemple de perdre ou de dupliquer des processus [ÅLS<sup>+</sup>03]).

AspectJ macroexpande le code des aspects dans le programme de base quand cela est possible. Dans certains cas un certain nombre de tests dynamiques persistent. Des travaux ont été proposés afin d'analyser statiquement le programme de base et de supprimer une partie de ces tests dynamiques [Sd03].

De plus, le tisseur d'AspectJ permet de tisser des aspects sur des aspects (en plus du programme de base). Cette possibilité peut être contrôlée avec la construction `within` qui permet de restreindre un point de coupe à une portion de code (et évite qu'un aspect ne s'applique sans fin à lui-même). Cette notion de portée doit être étudiée afin de permettre un meilleur contrôle du tisseur [DFS02] et par exemple de maîtriser l'expansion de code.

AspectJ concerne la *programmation* par aspects. Différents travaux considèrent les aspects à d'autres étapes du cycle de vie du logiciel, par exemple dès la conception [HPP00]. Un continuum devrait être établi afin de prendre en compte les aspects à toutes les étapes du cycle de vie.

Les aspects existaient avant la programmation par aspects mais devaient être tissés à la main dans le programme de base par le programmeur. Des travaux proposent aujourd'hui d'analyser des applications existantes afin d'extraire des aspects [EV04]. Cette opération inverse du tissage devrait permettre de réusiner ces applications pour améliorer leur modularité et donc faciliter leur maintenance.

Il est naturel de vouloir rendre les aspects réutilisables. Quelques travaux ont proposé d'adapter différentes techniques à cette fin. On peut notamment citer, l'utilisation de types [WZL03] ou de protocoles [DFS04a]. Il est aussi naturel de souhaiter rendre l'opération de tissage dynamique et réversible afin de changer à l'exécution l'ensemble des aspects utilisés par une application.

Enfin, AspectJ est un langage de programmation par aspects généraliste. En effet, le langage de programmation des actions est Java, et le langage de définition des points de coupe repose sur les mécanismes du langage Java (par exemple, appel de méthode, affectation d'un champ). De plus, le programme de base est une boîte blanche dont tout le code source est accessible. Lorsque le programme de base est un assemblage de composants, ces caractéristiques doivent être révisées (par exemple, actions limitées à différents services techniques, composants vus comme des boîtes noires ou grises). La prochaine section, passe en revue les propositions pour la programmation par aspects dans le cadre des modèles industriels de composants. Alors que la suivante étudie les propositions de langages de composants et d'aspects dans le cadre académique.

## 3 Aspects et modèles industriels de composants

Nous nous tournons maintenant vers la relation entre les aspects et les modèles de composants industriels, tels que les *Enterprise JavaBeans* (les EJB) [Sun03], le *Corba Component Model* (CCM) [Obj02] et *.NET* de Microsoft [Mic], lesquels ont été introduits dans le chapitre [réf](#) de ce volume. Les modèles de composants y sont présentés sans détailler la définition et l'utilisation des services techniques associés. Dans cette section, nous présentons les caractéristiques du support pour les aspects dans les modèles industriels et en particulier leurs services techniques principalement à l'exemple des EJB.

### 3.1 Intégration existante d'aspects dans des modèles de composants industriels

L'analyse de l'intégration d'aspects dans les modèles de composants industriels nécessite d'abord de tenir compte de la raison d'être de ces modèles : la réalisation d'architectures à trois niveaux (*3-tiers architectures*). Ce type d'architecture sert essentiellement à la mise en œuvre de systèmes d'information d'entreprises.

Par conséquent, les modèles de composants industriels supportent explicitement un certain nombre de fonctionnalités qui sont fondamentales dans ce domaine, notamment la distribution du calcul, la répartition des données, la persistance, le comportement transactionnel et la sécurité. La distribution du calcul et la répartition de données sont prises en charge par la structuration d'applications en composants, les « beans » du modèle des EJB, qui peuvent être déployés sur différentes machines, permettent l'appel de services via un réseau à travers leurs interfaces distantes et la gestion des communications synchrones aussi bien qu'asynchrones. En revanche, les trois services techniques - la persistance, les transactions et la sécurité - sont des fonctionnalités transverses qui ne peuvent pas être encapsulées dans des beans, c'est-à-dire ces fonctionnalités sont, en fait, des aspects dans le sens introduit précédemment.

Les EJB mettent en œuvre des modèles particuliers de transactions, de sécurité et de persistance. Techniquement, ces modèles sont réalisés sous forme de canevas à objets qui permettent l'implémentation de ces aspects en trois phases. D'abord, lors de la construction des composants, les méthodes des classes constituant ces canevas sont utilisées pour implémenter une politique par composant pour ces trois aspects. Lors de l'assemblage d'une application à partir de différents beans, des contraintes sur ces politiques peuvent être définies à l'aide des descripteurs de déploiement. Finalement, des politiques génériques sont configurées en instanciant des paramètres définis dans les descripteurs lors du déploiement.<sup>3</sup>

#### 3.1.1 L'aspect de sécurité dans les EJB

Concrètement, le canevas constituant l'aspect de sécurité dans les EJB implémente un modèle de sécurité basé sur la notion de rôle [RSC<sup>+</sup>96]. Un rôle est essentiellement un ensemble de composants assujettis aux mêmes règles de contrôle d'accès. L'aspect de sécurité des EJB fournit deux mécanismes différents pour la définition de stratégies de contrôle d'accès : un mécanisme associant des droits d'exécution de méthodes d'interfaces déclarativement, c'est à dire après l'implémentation des composants et sans la modification des implémentations ; un autre mécanisme pour le contrôle d'accès appliqué directement au niveau de l'implémentation.

Au temps de la construction des beans un contrôle d'accès peut être en particulier défini par des méthodes testant l'appartenance d'un composant à un rôle. Plus précisément, lors de la construction d'un bean, le développeur peut explicitement mettre en place une politique de contrôle d'accès en s'appuyant sur les méthodes de l'interface `EJBContext : getCallerPrincipal()` détermine l'identité du composant ayant appelé une méthode et `isCallerInRole(String roleName)` vérifie l'appartenance à un rôle. Cette politique est paramétrée par les rôles utilisés. Ces rôles doivent être déclarés à l'aide d'éléments `security-role-ref` dans le descripteur de déploiement du bean.

Reprenant l'exemple du compte bancaire introduit dans le chapitre [réf](#), voir figure 6, un test de sécurité restreignant le transfert de sommes importants aux managers de la banque

---

<sup>3</sup>Il est à noter que la version 3.0 de la spécification des EJB [EJB04], qui est en préparation, utilise encore la même architecture pour les services techniques, mais fournit un certain nombre de mécanismes destinés à faciliter leur expression syntaxique.

```

public class BanqueImpl implements SessionBean \{
    javax.ejb.SessionContext context;
    ....
    private void effectuer\_virement(String nom\_compte\_credit,
                                    String nom\_compte\_debit,
                                    float somme\_transfert) \{
        java.security.Principal principal = context.getCallerPrincipal();
        String callerId = principal.getName();

        if (somme\_transfert > SEUIL
            && ! context.isCallerInRole("mgr")
            && callerId != "Dupont")

            return ;
        ...
    \}
    ...
\}

```

FIG. 1 – Définition de stratégie de sécurité au temps de construction

ainsi qu'à l'employé « Dupont » peut être exprimé comme il est indiqué figure 1.

Lors de l'assemblage, un modèle de sécurité plus abstrait peut être défini en terme des rôles « logiques » associés aux utilisateurs - tels que le rôle d'administrateur ou le rôle d'employé. Les rôles logiques sont définis à l'aide de l'élément `security-role` du descripteur de déploiement. L'assembleur de l'application a également la possibilité d'associer un ensemble de droits d'exécution de méthodes à des rôles logiques à l'aide de l'élément `method-permission`. Ces droits permettent ou interdisent l'exécution de toutes les méthodes ou des méthodes particulières des interfaces maison, distante ou de services web. De cette manière, des modèles de sécurité basés sur l'accès aux méthodes des composants peuvent être définis sans modification de l'implémentation des composants uniquement par déclaration au niveau du descripteur de déploiement. Pour le cas où des beans font usage de rôles définis au niveau de leur implémentation, les rôles d'implémentation doivent être projetés sur les rôles logiques à l'aide d'éléments `role-link`. Finalement, l'assembleur peut utiliser l'élément `security-identity` afin de définir quelle identité de sécurité doit être utilisée pendant l'exécution d'une méthode : soit l'identité est héritée de l'appelant (en spécifiant l'attribut `use-caller-identity`), soit elle est définie explicitement par l'attribut `run-as`.

Lors du déploiement, des (groupes d')identités des composants faisant partie de l'application à exécuter et l'environnement logiciel cible sont associés aux rôles logiques à l'aide de l'élément `security-identity`. Finalement, notons que le déployeur a un contrôle complet sur le descripteur de déploiement dans le modèle des EJB : il peut, en particulier, invalider et modifier les spécifications données lors des phases précédentes.

Reconsidérant l'exemple du compte bancaire, le descripteur de la figure 2 spécifie les contraintes suivantes relevant de la sécurité :

- le composant « MaBanque » fait partie des banques privées ;
- lors de la construction du composant, un rôle de sécurité « mgr » a été créé ;
- en ce qui concerne l'assemblage, trois rôles de sécurité sont créés, pour les managers, les employés et les stagiaires de la banques ;
- lors de l'assemblage, le rôle « mgr » est lié au rôle « Manager » ;
- tous les employés ont le droit d'effectuer des virements.

### 3.1.2 Insuffisances de la réalisation d'aspects à base de canevas

Les aspects standard des modèles industriels sont appropriés pour une classe importante d'applications d'entreprise comme l'ont démontré par Kim et Clarke [KC02]. Néanmoins, l'intégration d'aspects sous forme de canevas dans les modèles de composants industriels souffre de différentes déficiences.

```

<ejb-jar>

<session>
  <ejb-class>MaBanque</ejb-class>
  <security-identity>
    <run-as>
      <role-name>BanquePrivee</role-name>
    </run-as>
  </security-identity>
</session>

<security-role-ref>
  <role-name>mgr</role-name>
</security-role-ref>

<assembly-descriptor>
  <security-role>
    <role-name>Manager</role-name>
  </security-role>
  <security-role>
    <role-name>Employe</role-name>
  </security-role>
  <security-role>
    <role-name>Stagiaire</role-name>
  </security-role>

  <security-role-ref>
    <role-name>Mgr</role-name>
    <role-link>Manager</role-link>
  </security-role-ref>

  <method-permission>
    <role-name>Employe</role-name>
    <method>
      <ejb-name>MaBanque</ejb-name>
      <method-name>effectuer\_virement</method-name>
    </method>
  </method-permission>
  ...
</assembly-descriptor>

</ejb-jar>

```

FIG. 2 – Ex.: descripteur de déploiement

D'abord, les canevas intégrés dans ces modèles ne fournissent que des modèles très simples pour les fonctionnalités correspondantes. Les modèles de contrôle d'accès à l'aide de rôles [RSC<sup>+</sup>96] supportent, par exemple, l'expression de certaines relations entre rôles, telles que l'héritage de droits d'accès d'un rôle donné par un sous-rôle. De telles relations ne peuvent pas être exprimées directement par le modèle de sécurité des EJB.

En outre, le mécanisme standard d'extension de canevas par héritage de classes du canevas, n'est pas suffisant pour enrichir les modèles de base : en particulier, ce mécanisme d'extension est limité à l'interface utilisée lors de la phase de construction des beans. Il ne permet pas, par exemple, d'étendre les moyens rudimentaires de spécification d'ensembles de méthodes pour lesquelles des permissions peuvent être définies lors de l'assemblage.

À part les restrictions inhérentes à chacun des modèles d'aspects, les canevas ne fournissent aucun support pour la combinaison des aspects entre eux. Une stratégie de sécurité spécifique aux beans persistants, par exemple, ne peut être définie qu'en utilisant les interfaces de bas niveau de l'aspect de sécurité et des conventions sur des méthodes exportées. En revanche, il n'est pas possible de spécifier déclarativement une stratégie de sécurité en combinant un aspect de sécurité et un aspect de persistance.

Finalement, les modèles de composants industriels n'incluent pas de support pour d'autres aspects, services techniques ou fonctionnalités « métier ».

### 3.2 Vers un support pour des aspects flexibles et extensibles

Afin de s'affranchir des restrictions détaillées précédemment, différentes approches ont été proposées qui explorent l'extension de l'architecture des modèles de composants industriels.

Dans ce qui suit nous classons ces approches selon deux dimensions : d'abord, si les aspects sont introduits par une modification du concept de conteneur ou si les aspects s'appliquent à d'autres éléments de l'architecture ; ensuite, si des mécanismes linguistiques spécifiques sont introduits ou s'il s'agit d'une extension sous forme de canevas à objets.

Notons d'abord que les spécificités des modèles industriels contraignent fortement les approches de la programmation par aspects qui peuvent y être raisonnablement appliquées. D'abord, du fait de l'importance de la notion de conteneur pour l'aiguillage de requêtes entre clients et services, le conteneur représente le point d'ancrage naturel pour la réalisation d'aspects. D'autres parties de ces architectures, par exemple les clients (via leur représentation côté serveur) et les bibliothèques fournissant les services techniques standard, ne sont pas utilisées comme base pour la réalisation d'aspects. Ensuite, les modèles industriels étant des modèles de composants à boîtes noires, c'est-à-dire qui ne permettent pas l'accès à l'implémentation des composants après la phase de construction d'un composant, toute technique se basant sur le code source - généralement très populaire dans le domaine de la programmation par aspects - n'est que d'un intérêt limité pour la réalisation d'aspects dans le contexte de ces modèles.

Par conséquent, l'extension du conteneur est utilisée par presque tous les travaux pour la réalisation d'un modèle d'aspects plus expressif. Nous présentons par la suite six de ces extensions : AspectJ2EE de Cohen et Gil [CG04], Caesar de Pichler et al. [POM03], JBoss AOP de JBoss Inc. [BCF<sup>+</sup>04], AES de Choi [Cho00], CVM de Duclos et al. [DEM02] et CWeP de Fariás [Far03]. JBoss AOP est l'extension pour la programmation par aspects du serveur d'applications libre JBoss, lequel inclut une implémentation de la spécification EJB, tandis qu'AspectJ2EE, Caesar, AES, CVM et CWeP sont des propositions académiques. Ces extensions couvrent les différentes techniques proposées pour l'extension de modèles industriels (et subsument, en particulier, les approches [BV98, Gru00]).

La table 1 résume les principales caractéristiques de ces six approches en les groupant en trois parties<sup>4</sup> :

- les objectifs (caractéristiques n<sup>os</sup> 1-4) de ces extensions ;
- la définition d'aspects (n<sup>os</sup> 5-10), en particulier, les moyens d'expressions pour les coupes et les actions, avec ou sans support linguistique dédié ;
- le mécanisme de tissage d'aspects (n<sup>os</sup> 11, 12), en particulier, sa compatibilité avec le modèle standard des EJB.

---

<sup>4</sup>Les auteurs ont conscience que cette classification n'est en rien la seule raisonnable, même pour l'ensemble des caractéristiques données ; ils espèrent pourtant que le lecteur sera convaincu de sa logique inhérente.

	AspectJ2EE [CG04]	Caesar [POM03]	JBoss AOP [BCF+04]	AES [Cho00]	CVM [DEM02]	CWeP [Far03]
1/ Support pour aspects nouveaux	oui					
2/ Ré-ingénierie d'aspects standard	oui	non considéré	non	oui	non considéré	non considéré
3/ Préservation « boîte noire »	oui					
4/ impl. d'aspects - liaison aux comp.	descripteur de liaison	interfaces de collaboration	descripteur de liaison	architecture méta	architecture méta	protocoles explicites
5/ Modèle de coupes	ess. AspectJ, pas d'introductions, <code>remotecall</code>	ess. AspectJ, pas d'introductions	ess. AspectJ, introduction de mixins	appels de méthodes	appels de méthodes	expressions régulières d'appels
6/ Modèle d'actions	Java	Java	Java	Java	Java	manipulation de protocoles
7/ Composition d'aspects, résolution de conflits	chaînage par sous-classage	programmable	chaînage par pile d'interception	chaînage par métaprogrammation	chaînage par règles d'application	analyse d'interactions, opérateurs de composition
8/ Généricité	paramétrage par coupes abstraites et champs	polymorphisme de type	pas de support	pas de support	pas de support	pas de support
9/ Instanciation d'aspects	composants	composants	VM, classes, composants	objets	composants	composants
10/ Support linguistique	déf. d'aspects : descripteur	déf. d'aspects, activation ( <code>deploy</code> )	déf. d'aspects : descripteur	déf. d'aspects : descripteur	déf. d'aspects, règles d'activation	protocoles à états finis
11/ Mécanisme de tissage	aspects étendant les composants	génération de proxies	génération de proxies	appels au métaniveau	appels au métaniveau	monitorage de protocoles
12/ Temps de tissage	assemblage, déploiement	activation dynamique d'aspects	déploiement, activation dynamique	exécution : instantiation de métaobjets	déploiement, activation dynamique	exécution

TAB. 1 – Caractéristiques principales de quelques extensions aspects pour des modèles industriels

### 3.2.1 Objectifs

Le tout premier objectif est évidemment de fournir un support pour la définition d'aspects. Les six approches autorisent la définition d'aspects autres que les services techniques prédéfinis habituels. La situation est légèrement différente en ce qui concerne ces aspects prédéfinis, tels que la sécurité : JBoss AOP ne les autorise pas mais requiert le recours aux aspects standard fournis par le serveur d'application JBoss. Parmi les approches académiques, AspectJ2EE et AES fournissent des nouvelles versions des aspects standard : les autres ne les considèrent pas mais fournissent des moyens en principe appropriés pour leur mise en place.

Ensuite, deux autres objectifs sont primordiaux. D'abord, afin de pouvoir utiliser des composants développés par des tiers, la préservation d'un modèle de composition à « boîte noire », c'est-à-dire ne permettant pas à un aspect d'accéder directement à l'implémentation des composants, est impérative. Ce critère est satisfait par les six approches. Ensuite, une séparation stricte entre l'implémentation des aspects et leur liaison, c'est-à-dire application, aux composants est cruciale pour le développement de bibliothèques d'aspects indépendants des composants. Toutes les approches supportent une telle séparation explicitement même si différents moyens sont utilisés pour y arriver : AspectJ2EE et JBoss AOP principalement utilisent le mécanisme standard de déploiement de composants pour la liaison d'aspects qui sont préalablement traduits dans des composants particuliers ; AES et CVM sont bâtis sur des architectures méta permettant de réaliser la séparation recherchée en terme de la séparation entre niveau de base et métaniveau ; finalement, Caesar et CWeP réalisent cette séparation sur la base de leurs interfaces qui sont enrichies par rapport au standard EJB, Caesar utilise des interfaces de collaboration pour lier aspects et composants, CWeP lie les aspects aux protocoles explicites faisant partie des interfaces des composants EJB étendus.

### 3.2.2 Définition d'aspects

Les six extensions considérées fournissent un large éventail de techniques pour la définition d'aspects. En ce qui concerne le modèle de coupes, l'interception de l'appel de méthodes par le conteneur est la coupe de base fournie par toutes les extensions (et constitue l'unique coupe primitive des approches « méta » AES et CVM). AspectJ2EE, Caesar et JBoss AOP transposent essentiellement le modèle des coupes d'AspectJ aux EJB incluant, par exemple, des coupes pour l'accès aux valeurs des champs et des coupes sur le flot de contrôle. AspectJ2EE y ajoute une coupe `remotecall` afin de pouvoir définir le traitement d'un appel à la fois du côté du client et du côté du serveur, c'est-à-dire à différents niveaux d'une application multi-niveaux (*tier-cutting*). Pourtant, les « introductions » d'AspectJ, qui permettent de modifier structurellement les classes et qui impliqueraient une violation du principe de « boîte noire », en sont exclues. Seul JBoss AOP fournit une forme d'introduction qui est compatible avec ce principe et qui réalisent une forme de *mixins*, permettant ainsi le paramétrage de composants par un mécanisme en connu dans le monde des objets. Finalement, CWeP fournit des coupes d'expressions régulières sur des séquences d'appels de méthodes, coupes plus abstraites qui ne peuvent être émulées par des aspects complexes dans les modèles à la AspectJ.

Les modèles d'actions des extensions, à part celui de CWeP, sont assez similaires. Ils permettent essentiellement de définir des méthodes Java et de les appeler avant, après ou à la place d'un point d'exécution correspondant à une coupe. CWeP se différencie en fournissant des opérateurs pour la manipulation des protocoles faisant partie de l'interface des composants. Ces opérateurs permettent de modifier la structure d'un protocole aussi bien que son état dynamique.

À part les modèles généraux pour la définition d'aspects, trois caractéristiques avancées sont d'un intérêt particulier en ce qui concerne l'expressivité des aspects : la composition d'aspects et la résolution de conflits éventuels, le paramétrage ainsi que l'instanciation d'aspects (cf. table 1, n<sup>os</sup> 7-9). D'abord, la composition d'aspects et la résolution de conflits sont typiquement traitées à l'aide de séquences d'aspects à exécuter à un point donné. Des mécanismes particuliers peuvent être fournis à cet effet (les règles d'application de CVM, les piles de chaînage de JBoss AOP) ou des mécanismes standard y être appliqués (ordonner les aspects par héritage dans le cas d'AspectJ2EE, des métaprogrammes pour AES). Caesar et CWeP fournissent des mécanismes plus élaborés pour ce problème : le premier permet l'ordonnancement arbitraire entre aspects à l'aide de ses interfaces de collaborations, le second fournit des analyses statiques d'interactions entre aspects et un jeu correspondant d'opérateurs

de compositions pour la résolution de conflits identifiés. En ce qui concerne des éléments de généralité, il est possible de faire dépendre l'implémentation d'un aspect par des coupes abstraites et des valeurs de champs (AspectJ2EE) et d'utiliser la liaison tardive d'action d'un aspect en exploitant le polymorphisme de type entre interfaces de collaboration de Caesar. Finalement, différentes possibilités d'instanciation d'aspects sont fournies. Elles régissent en particulier la création de différents états des aspects en fonction de différents entités de l'application de base. Ces possibilités incluent typiquement l'association d'une instance d'aspects à des composants (en fait des instances de composants). Certaines approches permettent aussi d'explicitement l'instanciation d'aspects par VM ou classes de composants uniquement (JBoss AOP) ou, dans le cas d'AES, pour un objet arbitraire de l'application sous-jacente.

Finalement, une discussion des moyens de définition des aspects serait incomplète sans évoquer la question du support linguistique correspondant. Typiquement, une syntaxe particulière est introduite pour la liaison d'aspects à l'application de base via des coupes. Les coupes sont définies dans des descripteurs (par exemple, de déploiement), c'est-à-dire en dehors de la définition des composants, et ne nécessitent donc pas d'accès à l'implémentation des composants. En outre, plusieurs extensions définissent des constructions syntaxiques supplémentaires pour la spécification de l'activation des aspects (`deploy` de Caesar, règles d'activation de CVM). CWeP offre une syntaxe spécifique pour la construction et manipulation de protocoles à l'aide d'aspects.

### 3.2.3 Tissage d'aspects

Une fois les aspects définis selon les modèles étendus discutés précédemment, se pose la question de leur tissage avec un ensemble de composants existants (cf. table 1, n<sup>os</sup> 11, 12). Les six extensions ont pour but de préserver la compatibilité avec le support d'exécution standard du modèle industriel de composants sous-jacent. Elles utilisent toutes des variantes d'appels à l'aide de *proxies*<sup>5</sup>. Ainsi, Caesar et JBoss AOP génèrent des classes de proxies lors du déploiement. AES et CVM repose sur une architecture à (un) métaniveau et l'appel proxy est de fait réalisé par un appel au métaniveau. Comme dans AspectJ2EE les aspects doivent étendre les classes de l'application sous-jacente, les proxies sont en fait remplacés par des versions spécialisées d'une méthode englobant la fonctionnalité aspect aussi bien que l'appel à la méthode métier de la superclasse appropriée. CWeP réalise le tissage d'aspects à l'aide d'un moniteur d'exécution des protocoles de composants. L'utilisation des schémas de proxy permet d'effectuer le tissage à différents moments du cycle de vie d'un composant : typiquement un aspect peut être tissé lors de l'assemblage et déploiement. En outre, il est possible d'activer (ou de désactiver) un aspect lors de l'exécution.

### 3.2.4 Discussion

Afin d'évaluer les extensions, considérons les insuffisances identifiées dans la section 3.1.2 :

- aspects standard limités ;
- impossibilités de définition de nouveaux aspects (techniques ou métier) ;
- moyens inappropriés de définition d'aspects, en particulier pour spécifier la liaison avec l'application sous-jacente, l'instanciation d'aspects et la composition d'aspects.

Notons d'abord que les extensions présentées offrent des éléments de solutions pour chacun de ces problèmes. La définition de nouveaux aspects, en particulier, constitue leur point fort majeur, ce qui est validé par un grand nombre d'aspects spécifiques réalisés. JBoss AOP a été utilisé, par exemple, pour définir des aspects pour la mise en cache et l'agrégation de composants ainsi que l'introduction d'un mécanisme non standard de communication asynchrone.

Pourtant, la majorité des extensions actuelles sont caractérisées par l'intégration d'un modèle d'aspects limité. En particulier, des mécanismes tels que les interfaces de collaborations de Caesar (qui généralisent les possibilités de liaison entre aspects et composants) ainsi que les coupes non atomiques de CWeP (définies sur une notion d'interface de composants plus expressive) sont encore confinées au domaine de la recherche même s'ils ne demandent pas l'accès à l'implémentation des composants et sont compatibles avec les modèles industriels de composants standard.

---

<sup>5</sup>Pas d'accord. À discuter avec Mario. (Jacques).

Plus généralement, la recherche dans le domaine de la programmation par aspects s'occupe actuellement pour une large partie de la définition d'aspects plus expressifs, voir la section suivante. L'intégration de ces mécanismes en respectant les contraintes s'imposant dans le cadre d'une industrialisation de composants sur étagère reste d'actualité.

## 4 Langages de composants et aspects

### 4.1 Des langages hybrides basés sur la notion de collaboration

Un certain nombre de propositions, les *Mixin Layers* [SB02] et sa variante Java, les *Java Layers* [CL01, Car02], Jiazzi [MFH01], Caesar [MO03] et les *Aspectual Components* (ou AC) [LLO03], bien que développés dans des contextes et avec des perspectives différentes, définissent des langages, le plus souvent des extensions de Java, basés sur la notion de *collaboration*, qui s'avère posséder à la fois certaines propriétés des composants et certaines propriétés des aspects. La notion de collaboration est née dans le contexte de la conception par objets et a été formalisée dans [RAB<sup>+</sup>92]. Il s'agit de considérer une application comme la composition de *collaborations*, chaque collaboration étant définie par un ensemble d'objets et un *protocole* déterminant comment ces objets interagissent. Ce protocole définit pour chaque objet le rôle que celui-ci joue dans la collaboration. Dans son expression la plus simple, un rôle est lui-même défini par l'ensemble des méthodes prenant part à la collaboration. Cet ensemble inclut à la fois les méthodes fournies par l'objet et les méthodes requises pour que l'objet joue son rôle. Le point clef est qu'un objet est susceptible d'intervenir dans plusieurs collaborations. Une collaboration peut donc être considérée comme un composant en tant qu'unité de composition (et de compilation) avec ses interfaces définissant ses constituants, les rôles, avec leurs méthodes fournies et requises. Elle peut aussi être considérée comme un aspect traversant la structure des classes.

Voici, par exemple, une définition du protocole correspondant au schéma de conception *Observateur* [GHJV94], inspiré de Caesar [MO03]. Caesar utiliserait le même mot-clef `interface` pour `collaboration` et `role`. AC utilise lui des mots-clefs plus parlants (`collaboration` et `role`) mais ne permet pas de définir séparément interface et implémentation. Noter que les définitions des deux rôles sont mutuellement récursives.

```
collaboration ObserverProtocol {
  role Subject {
    provided void addObserver(Observer o);
    provided void removeObserver(Observer o);
    provided void changed();
    expected String getState();
  }
  role Observer {
    expected void notify(Subject s);
  }
}
```

Les propositions mentionnées ci-dessus diffèrent de nombreuses façons, en particulier en terme de langage de coupe, de langage de liaison et de composition des collaborations. Pour ce qui est du premier point, le langage de coupe, les différences sont essentiellement syntaxiques. Il s'agit avant tout d'altérer le comportement de méthodes appartenant à d'autres collaborations et donc, en terme d'aspects, d'avoir accès à des points de coupe correspondant aux réceptions des appels de méthode. Le deuxième point, le langage de liaison, est pour une part dépendant du troisième, la manière de composer les collaborations et nous allons donc continuer notre discussion à partir de ce critère.

#### 4.1.1 Composition par héritage avec les *Mixin Layers*

Une première idée des *Mixin Layers* ou *couches de mixins* est simplement d'utiliser l'héritage pour composer des collaborations (rappelez-vous de l'analogie précédemment évoquée entre ajout d'un aspect et extension d'une classe par héritage). Une deuxième idée, afin de découpler la définition d'une collaboration de la collaboration qu'elle va étendre est de définir chaque rôle

à l'aide d'un *mixin*. Un mixin est défini par G. Bracha et W. Cook [BC90] comme une sous-classe abstraite, c'est-à-dire la définition d'une sous-classe qui peut être appliquée à différentes superclasses. D'une certaine manière, un mixin est paramétré par sa superclasse. En guise d'exemple, voici comment on pourrait définir un aspect `Profiling` s'appliquant à `Foo.foo()` à l'aide de Jam, une extension de Java avec des mixins [ALZ03], à comparer avec la définition de la section 2.2:

```

mixin Profiling {
    inherited void foo();

    int nbCalls = 0;

    public void foo() {
        nbCalls++;
        System.out.println("foo " + nbCalls);
        super.foo();
    }
}

class FooWithProfiling = Profiling extends Foo {}

```

Les mixins ne permettent de définir qu'un rôle unique. Une couche de mixins combine différents mixins, chacun correspondant à un rôle de la collaboration que la couche de mixins implémente. **À faire : préciser ici comment sont mis en relation les éléments de deux couches successives. Donner un exemple en Java Layers.**

Bien que l'introduction de mixins facilite l'utilisation de l'héritage, elle n'en gomme pas certains défauts, en particulier son caractère statique et ses problèmes de robustesse. Le caractère statique de l'héritage fait qu'une fois définie, une couche de mixins ne pourra pas être remplacée dynamiquement par une autre couche de mixins. Les problèmes de robustesse de l'héritage ont été bien répertoriés sous le nom de problème de la classe de base (ou super-classe) fragile (voir, par exemple, [MS98]). De plus, l'héritage ne fournit qu'un langage de liaison très limité. Ainsi, le mixin `Profiling` dans l'exemple ci-dessus est indépendant de la classe à profiler mais fixe le nom de la méthode à considérer. L'application de ce mixin au profilage d'une méthode `bar()` requiert la création d'un nouveau mixin adaptateur faisant la liaison entre la méthode `Profiling.foo()` et la méthode `bar()`. Finalement, l'opération de composition outre le fait qu'elle est essentiellement asymétrique (c'est en fait le cas de toutes les propositions considérées ici), demande à effectuer une linéarisation des compositions en ne considérant qu'une couche de mixins à la fois.

Un mot sur Jiazzi (diffère essentiellement par son langage de liaison ?).

#### 4.1.2 Composition par agrégation : Caesar

Caesar [MO03] se présente comme un langage d'aspects visant à régler certains problèmes observés dans AspectJ, notamment en ce qui concerne la structuration des aspects et l'absence de déploiement dynamique de ceux-ci. Ces critiques sont essentiellement résolues en introduisant la notion de collaboration sous la forme d'*interface de collaboration d'aspect* (*aspect collaboration interface*) et en effectuant la composition des aspects par l'intermédiaire d'encapsulateurs. Par rapport aux couches de mixins, le dernier point correspond à remplacer la composition par héritage par une composition par agrégation, une alternative bien connue!

Pour chaque aspect (on pourrait aussi dire collaboration), Caesar distingue son *interface* (ou protocole), son *implémentation* et, pour chaque programme de base ciblé, un code de *liaison* adapté.

L'exemple de l'observateur...

```

interface ObserverProtocol {
    interface Subject {
        provided void addObserver(Observer o);
        provided void removeObserver(Observer o);
        provided void changed();
        expected String getState();
    }
}

```

```

interface Observer {
    expected void notify(Subject s);
}
}
...

```

On voit que Caesar minimise le nombre de concepts : la notion d'interface couvre à la fois la définition des rôles et des collaborations et la notion de classe leur implémentation. On voit aussi que l'interface et l'implémentation d'un aspect n'inclut aucun élément extérieur à Java et ressemble fort à ce qu'on pourrait considérer comme un composant Java. Le point un peu curieux toutefois est que l'implémentation ne fait pas apparaître d'appel à la méthode requise `getState()`, celle-ci est en fait nécessaire à la définition de la méthode `notify()` dans le code de liaison.

Grosso modo : le cœur des aspects est propre. Il y a juste une petite couche de glu (le code de liaison) qui gère l'interception et l'adaptation des noms.

### 4.1.3 Composition par superposition : Aspectual Components

[LLM99, LLO03]. Encore une autre histoire de composition : la superposition (*superimposition* en Anglais).

### 4.1.4 Discussion

- Ne serait-il pas possible de combiner les bons points de ces approches.
- Open Modules [Ald04b, Ald04a].
- Paramétrisme.
- `cflow` : facile ?
- références à intégrer : JasCo [SVJ03], [Far03, GPRZ04], [PFFT02, PFT03] ?

## 5 Conclusion

Convergence en cours. Importance de la non-anticipation et problèmes associés. Mettre de l'intelligence dans les interactions.

## Références

- [ACN02] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In Magnusson [Mag02], pages 187–197.
- [Aks03] Mehmet Aksit, editor. *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003)*, Boston, Massachusetts, USA, March 2003. ACM Press.
- [Ald04a] Jonathan Aldrich. Open modules: Modular reasoning in aspect-oriented programming. In Clifton et al. [CLL04], pages 7–18.
- [Ald04b] Jonathan Aldrich. Open modules: Reconciling extensibility and information hiding. In Ernst [Ern04].
- [ÅLS<sup>+</sup>03] Rickard A. Åberg, Julia L. Lawall, Mario Südholt, Gilles Muller, and Anne-Françoise Le Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 196–204, Montréal, Canada, March 2003. IEEE Computer Society Press.
- [ALZ03] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam - designing a Java extension with mixins. *ACM Transactions on Programming Languages and Systems*, 25(5):641–712, September 2003.
- [BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In Meyrowitz [Mey90], pages 303–311.

- [BCF<sup>+</sup>04] Bill Burke, Austin Chau, Marc Fleury, Adrian Brock, Andy Godwin, and Harald Gliebe. JBoss aspect oriented programming. <http://www.jboss.org/developers/projects/jboss/aop>, February 2004.
- [Bro95] F.P. Brooks. *The Mythical Man-Month, Essays on Software Architecture*. Addison-Wesley, anniversary edition, 1995.
- [BV98] Gregory Blank and Gene Vayngrib. Aspects of enterprise java beans. In Christina Lopes, Bedir Tekinerdogan, Wolfgang De Meuter, and Gregor Kiczales, editors, *Aspect-Oriented Programming Workshop at ECOOP'98*, pages 37–40, Enschede, The Netherlands, July 1998.
- [Car02] Richard Cardone. *Language and Compiler Support for Mixin Programming*. PhD thesis, The University of Texas at Austin, May 2002.
- [CF00] Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *Conference Record of POPL'00: The 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, Boston, MA, USA, January 2000. ACM Press.
- [CG04] Tal Cohen and Joseph (Yossi) Gil. AspectJ2EE = AOP + J2EE: Towards an aspect based, programmable and extensible middleware framework. In Martin Odersky, editor, *ECOOP 2004 - Object-Oriented Programming, 18th European Conference*, number 3086 in Lecture Notes in Computer Science, Oslo, Norway, June 2004. Springer-Verlag.
- [Cho00] Jung Pil Choi. Aspect-oriented programming with enterprise javabeans. In *Proceedings of the Fourth International Enterprise Distributed Object Computing Conference (EDOC 2000)*, pages 252–261, Makuhari, Japan, September 2000. IEEE Computer Society Press.
- [CKFS01] Yvonne Coady, Gregor Kiczales, Mike Feeley, and Greg Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *Proceedings of the Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9)*, pages 88–98, Vienna, Austria, September 2001.
- [CL01] R. Cardone and C. Lin. Comparing frameworks and layered refinement. In Harrold and Schäfer [HS01], pages 285–294.
- [CL03] Curtis Clifton and Gary T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. TR 03-15, Department of Computer Science, Iowa State University, December 2003.
- [CLL04] Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors. *FOAL 2004 Proceedings - Foundations of Aspect-Oriented Languages - Workshop at AOSD 2004*, volume TR 04-04. Department of Computer Science, Iowa State University, March 2004.
- [DEM02] Frédéric Duclos, Jacky Estublier, and Philippe Morat. Describing and using non functional aspects in component based applications. In Kiczales [Kic02], pages 65 – 75.
- [DFS02] Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT Conference, GPCE 2002 - Proceedings*, volume 2487 of *Lecture Notes in Computer Science*, pages 173–188, Pittsburgh, PA, USA, October 2002. Springer-Verlag.
- [DFS04a] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In Lieberherr [Lie04], pages 141–150.
- [DFS04b] Rémi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In Mehmet Akşit, Siobhán Clarke, Tzilla Elrad, and Robert E. Filman, editors, *Aspect-Oriented Software Development*. Addison-Wesley, 2004. to appear.
- [Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.
- [DJ02] Rémi Douence and Narendra Jussien. Non-intrusive constraint solver enhancements. In *First AOSD workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)* [Uni02].

- [DK76] F. DeRemer and H. Kron. Programming-in-the-large versus programming-in-the-small. *IEEE Transactions on Software Engineering*, SE-2(2):80–86, 1976.
- [DMS01] Rémi Douence, Olivier Motelet, and Mario Südholt. Sophisticated crosscuts for e-commerce. In *ECOOP 2001 Workshop on Advanced Separation of Concerns*, Budapest, Hungary, jun 2001.
- [DS03] Rémi Douence and Mario Südholt. Un modèle et un outil pour la programmation par aspects événementiels. In Briot J.-P. and J. Malenfant, editors, *Langages et modèles à objets - LMO'03*, pages 105–117, Vannes, France, January 2003. Hermès. RSTI série L'objet, 9(1-2).
- [EJB04] EJB 3.0 Expert Group. Enterprise JavaBeans specification, version 3.0, early draft, 2004. <http://java.sun.com/products/ejb/docs.html>.
- [Ern04] Erik Ernst, editor. *SPLAT: Software engineering Properties of Languages for Aspect Technologies - Workshop at AOSD 2004*, March 2004.
- [EV04] Ran Ettinger and Mathieu Verbaere. Untangling: a slice extraction refactoring. In Lieberherr [Lie04], pages 93–101.
- [Far03] Andrés Farías. *Un modèle de composants avec des protocoles explicites*. PhD thesis, École des Mines de Nantes, December 2003.
- [FF00] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA 2000*, Minneapolis, USA, October 2000.
- [GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [GPRZ04] Steffen Göbel, Christoph Pohl, Simone Röttger, and Steffen Zschaler. The COM-QUAD component model — enabling dynamic selection of implementations by weaving non-functional aspects. In Lieberherr [Lie04], pages 74–82.
- [Gru00] John Grundy. Multi-perspective specification, design and implementation of software components using aspects. *International Journal of Software Engineering and Knowledge Engineering*, 10(6), 2000. World Scientific Publishing Co.
- [HPP00] Wai-Ming Ho, Francois Pennaneac'h, and Noel Plouzeau. Umlaut: A framework for weaving uml-based aspect-oriented designs. In *Technology of Object-Oriented Languages and Systems (TOOLS 33)*, pages 324–334, Saint Malo, France, June 2000. IEEE Computer Society Press.
- [HS01] Mary Jean Harrold and Wilhelm Schäfer, editors. *Proceedings of the 23rd International Conference on Software Engineering*, Toronto, Canada, May 2001. IEEE Computer Society Press.
- [IBM] The Concern Manipulation Environment website. <http://www.research.ibm.com/cme>.
- [Jul98] E. Jul, editor. *ECOOP'98 - Object-Oriented Programming - 12th European Conference*, volume 1445 of *Lecture Notes in Computer Science*, Brussels, Belgium, July 1998.
- [KC02] Howard Kim and Siobhán Clarke. The relevance of AOP to an applications programmer in an EJB environment. In *First AOSD workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)* [Uni02].
- [KHH<sup>+</sup>01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Lindskov Knudsen [LK01], pages 327–353.
- [Kic02] Gregor Kiczales, editor. *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, Enschede, The Netherlands, April 2002. ACM Press.
- [Lie04] Karl Lieberherr, editor. *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004)*, Lancaster, UK, March 2004. ACM Press.
- [LK01] J. Lindskov Knudsen, editor. *ECOOP 2001 - Object-Oriented Programming, 15th European Conference*, number 2072 in *Lecture Notes in Computer Science*, Budapest, Hungary, June 2001. Springer-Verlag.

- [LLM99] Karl Lieberherr, David Lorenz, and Mira Mezini. programming with aspectual components. Technical Report NU-CCS-99-01, Northeastern University, March 1999.
- [LLO03] David H. Lorenz, Karl Lieberherr, and Johan Ovlinger. Aspectual collaborations: Combining modules and aspects. *The Computer Journal*, 46(5):542–565, September 2003.
- [Mag02] Boris Magnusson, editor. *ECOOP 2002 - Object-Oriented Programming, 16th European Conference*, number 2374 in Lecture Notes in Computer Science, Málaga, Spain, June 2002. Springer-Verlag.
- [Mey90] Norman Meyrowitz, editor. *Proceedings of ECOOP-OOPSLA*, Ottawa, Canada, October 1990. ACM Press.
- [MFH01] S. McDirmid, M. Flatt, and W.C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In Vlissides [Vli01].
- [Mic] Microsoft Corp. Microsoft .NET Framework. <http://msdn.microsoft.com/netframework/>.
- [MK03] Hidehiko Masuhara and Kazunori Kawauchi. Dataflow pointcut in aspect-oriented programming. In *First Asian Symposium on Programming Languages and Systems (APLAS'03)*, Beijing, China, November 2003.
- [MO03] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In Aksit [Aks03].
- [MS98] L. Mikhajlov and E. Sekerinski. A study of the fragile base class. In Jul [Jul98], pages 355–382.
- [MT00] N. Medvidovic and R.N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.
- [Obj02] Object Management Group. CORBA components, June 2002. Version 3.0.
- [Par72] D. Parnas. On the criteria for decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.
- [PDN86] R. Prieto-Diaz and J.M. Neighbors. Module interconnection languages. *The Journal of Systems and Software*, 6(4):307–334, November 1986.
- [PFFT02] Mónica Pinto, Lidia Fuentes, Mohamed E. Fayad, and Jose María Troya. Separation of coordination in a dynamic aspect oriented framework. In Kiczales [Kic02], pages 134–140. Short paper.
- [PFT03] Mónica Pinto, Lidia Fuentes, and Jose María Troya. DAOP-ADL: an architecture description language for dynamic component and aspect-based development. In Pfenning and Smaragdakis [PS03], pages 118–137.
- [POM03] R. Pichler, K. Ostermann, and M. Mezini. On aspectualizing component models. *Software Practice and Experience*, 33(10):957–974, 2003.
- [PS03] Frank Pfenning and Yannis Smaragdakis, editors. *Proceedings of the 1st ACM/SIGPLAN Conference on Generators and Components*, volume 2830 of *Lecture Notes in Computer Science*, Erfurt, Germany, September 2003. Springer-Verlag.
- [RAB<sup>+</sup>92] T. Reenskaug, E. Anderson, A. Berre, A. Hurlen, A. Landmark, O. Lehne, E. Nord-Hagen, E. Ness-Ulseth, G. Oftedal, A. Skaar, and P. Stenslet. OORASS: Seamless support for the creation and maintenance of object-oriented systems. *Journal of Object-Oriented Programming*, 5(6):27–41, October 1992.
- [RSC<sup>+</sup>96] S. Ravi, S. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-based access control models. *Computer*, 29(2):38–47, February 1996.
- [SB02] Yannis Smaragdakis and Don Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255, April 2002.
- [Sd03] Damien Sereni and Oege de Moor. Static analysis of aspects. In Aksit [Aks03], pages 30–39.

- [Sit01] AspectJ Site. Aspect-oriented programming in Java with AspectJ, 2001. <http://www.parc.com/research/csl/projects/aspectj/downloads/OReilly2001.pdf>.
- [Sun03] Sun Microsystems. Enterprise JavaBeans specification, version 2.1, 2003. <http://java.sun.com/products/ejb/docs.html>.
- [SVJ03] Davy Suvéé, Wim Vanderperren, and Viviane Jonckers. JasCo; an aspect-oriented approach tailored for component-based software development. In *Proc. of 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, pages 21–29. ACM Press, March 2003.
- [Szy02] C. Szyperski. *Component Software*. Addison-Wesley, 2002. 2nd edition.
- [Uni02] University of Twente. *First AOSD workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, Enschede, The Netherlands, April 2002.
- [Vli01] John Vlissides, editor. *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications (OOPSLA'01)*, Tampa Bay, FL, USA, October 2001. ACM Press.
- [WZL03] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 127–139. ACM Press, 2003.

## A Glossaire

- Action** (*Advice*) Un morceau de code qui met en œuvre une partie d’une préoccupation.
- Aspect** (*Aspect*) Éléments d’un programme / fonctionnalités associés à un point de vue, *A well modularized implementation of a crosscutting concern*. Gregor Kiczales
- AspectJ** À la fois un langage (surensemble de Java dédié à la programmation par aspects) et sa mise en œuvre (i.e., un tisseur).
- Association** (*Association*) Une spécification (souvent déclarative) des classes d’équivalence entre points de jonction et instances d’aspects. Déclare quand un aspect doit être instancié et à quels points de jonction il est associé.
- Base (programme de)** (*Base Program*) un programme fonctionnel/opérationnel qui met en œuvre un point de vue.
- Cycle de vie d’un composant** Ensemble de phases pendant lesquelles un composant est développé, sujet à exécution ou autrement traité. Dans les modèles de composants industriels quatre phases sont typiquement distinguées : la phase de *construction* d’un composant, l’*assemblage* d’une application par la composition de composants, le *déploiement* d’un assemblage sur un environnement matériel et logiciel ainsi que l’*exécution* de l’application. Les approches pour l’application de la programmation par aspects aux composants peuvent être caractérisées par les moyens d’expression et de tissage d’aspects spécifiques aux différentes phases du cycle de vie qu’elles utilisent.
- Composant** (*Component*) *Components are for composition. Composition enables prefabricated “things” to be reused by rearranging them in ever-new composites.*  
*Software components are executable units of independent production, acquisition, and deployment that can be composed into a functioning system.*  
*To enable composition a software component adheres to a particular component model and targets a particular component platform.* C. Szyperski [Szy02]
- Conteneur** (*Container*) La couche de logiciel qui vient entourer un composant et permet ses interactions avec le système d’exécution.
- Collaboration** (*Collaboration*) Un ensemble de classe et un *protocole* déterminant comment les instances de ces classe interagissent. Le protocole définit le *rôle* de l’objet au sein de la collaboration. En général, un rôle ne fait appel qu’à un sous-ensemble des champs d’un objet et un objet peut jouer plusieurs rôles (c’est-à-dire prendre part à différentes collaborations).
- Mixin** (*Mixin*) Une définition de sous-classe paramétrée par sa superclasse, c’est-à-dire qui va pouvoir étendre différentes superclasses.

- Module** (*Module*) Un ancêtre commun aux composants et aux aspects.
- Point de jonction** (*Joinpoint*) Un point du programme de base auquel un aspect peut être lié.
- Point de coupe** (*Pointcut*) Une spécification (souvent déclarative) des points de jonction du programme de base qui doivent implicitement appeler le code d'un aspect.
- Phase du cycle de vie** Voir « Cycle de vie ».
- Programmation incrémentale** Processus de développement qui consiste à obtenir une succession de programmes de plus en plus complets.
- Séparation des préoccupations** (*Separation of concerns*) *Let me try to explain to you, what to my taste is characteristic for all intelligent thinking. It is, that one is willing to study in depth an aspect of one's subject matter in isolation for the sake of its own consistency, all the time knowing that one is occupying oneself only with one of the aspects.*  
*We know that a program must be correct and we can study it from that viewpoint on ly; we also know that is should be efficient and we can study its efficiency on another day [...]*  
*But nothing is gained on the contrary by tackling these various aspects simultaneously. It is what I sometimes have called "the separation of concerns" [...]*  
*A scientific discipline separates a fraction of human knowledge from the rest: we have to do so, because, compared with what could be known, we have very, very small heads.*  
 Edsger W. Dijkstra
- Service technique** Fonctionnalité transversale dans les modèles de composants industriels ; en particulier, terme groupant trois fonctionnalités : la persistance, le comportement transactionnel et la sécurité.
- Tisseur** (*Weaver*) Un compilateur qui mélange un programme de base et des aspects pour produire un programme qui met en œuvre de multiples points de vue.