

---

# Une Machine à Objets Extensibles pour la Séparation des Préoccupations

**Frédéric Duclos - Jacky Estublier – Rémy Sanlaville**

Laboratoire Logiciels, Systèmes Réseaux - IMAG  
220 Rue de la Chimie, Domaine Universitaire - B.P. 53  
38041 Grenoble Cedex 9  
{Frederic.Duclos, Jacky.Estublier, Remy.Sanlaville}@imag.fr

---

## **Résumé :**

*La "séparation des préoccupations" est une approche qui vise à regrouper le code correspondant à un même aspect et à rendre les aspects indépendants.*

*Elle est illustrée, entre autres, par la Programmation Orientée Aspect dont il existe plusieurs implémentations telles que AspectJ, HyperJ ou encore JAC.*

*D'autres travaux, tels EJB et CCM, proposent implicitement une séparation des préoccupations dans le domaines spécifique de la technologie des composants, avec un nombre limité de services à travers le concept de conteneur.*

*Nos travaux s'inscrivent dans cette lignée en proposant une machine à objets étendus pour définir des conteneurs extensibles. Cette machine est facilement adaptable pour permettre la séparation des préoccupations dans différents domaines d'application.*

**Mots clés :** *séparation des préoccupations, Programmation Orientée Aspect (AOP), Machine à Objets Étendus (EOM), conteneurs extensibles, EJB, CCM.*

## **Abstract :**

*"Separation of concerns" aims to manage pieces of code that focus on the same aspect and to separate the different aspects.*

*Several tools, like AspectJ, HyperJ or JAC are implementation of Aspect Oriented Programming and have as explicit goal to provide separation of concerns.*

*Other works like EJB and CCM provide implicitly a separation of concerns in the specific area of component technology, through a restricted number of containers.*

*Our work focuses on separation of concerns by providing an extended object machine, in which extensible containers can be described and used. This machine could be adapted for different domains through the definition of a specific controller.*

**Keywords:** *separation of concerns, AOP, Extended Object Machine (EOM), Extensible Containers, Enterprise Java Beans (EJB), Corba Component Model (CCM).*

## 1. Introduction

La « séparation des préoccupations » [19] est considérée comme l'une des approches les plus prometteuses du génie logiciel. Selon cette approche, un programme écrit avec la technologie « objet » contient deux parties principales:

- Une partie fonctionnelle qui implémente les services pour lesquels l'objet a été écrit ; c'est la préoccupation « principale » de l'objet.
- Une partie qui adapte l'objet à un environnement et/ou à une application particulière. Cette partie regroupe des préoccupations « secondaires » parfois complexes et disséminées dans tout le code source de l'application.

La séparation de ces deux parties permettrait d'une part de réduire le travail et l'expertise demandés au programmeur de l'objet, et d'autre part de mieux réutiliser la partie fonctionnelle des objets dans d'autres environnements et d'autres applications. Pour cela, il faudrait que l'objet ne contienne que sa partie fonctionnelle, et que son adaptation soit effectuée à l'extérieur de l'objet.

La « séparation des préoccupations », outre la simplification du travail du programmeur et la réutilisation de code existant, vise à améliorer l'évolution et la lisibilité des logiciels en traitant chacune des préoccupations indépendamment les unes des autres.

La Programmation Orientée Aspect (*Aspect Oriented Programming AOP*) [18] est une façon d'obtenir la séparation des préoccupations, avec différentes implémentations telles que AspectJ [2], JAC [15] ou encore HyperJ [13].

En parallèle, le CBSE (*Component Based Software Engineering*) vise à réduire les dépendances entre les différentes entités logicielles (composants) lors de la compilation et lors de l'exécution. Dans le cadre de la programmation par composant, les industriels ont abordé le problème de la séparation des préoccupations, en proposant des modèles de composant tels que les Enterprise Java Beans (EJB) [21] et le modèle de composant de Corba (CCM) [5]. Ces modèles proposent le concept de « conteneur » gérant des services non-fonctionnels indépendamment de la programmation des composants.

Notre travail sur la séparation des préoccupations s'est découpé en deux phases successives. L'objectif de la première phase était de proposer des conteneurs extensibles pour notre modèle de composant. A la différence des conteneurs de CCM et de EJB, les conteneurs extensibles que nous proposons permettent à l'utilisateur du modèle de composant de définir et d'utiliser ses propres services non fonctionnels. L'objectif de la seconde phase de notre travail est de généraliser l'expérience acquise sur les conteneurs extensibles afin de l'appliquer également aux applications à base d'objets.

Nous qualifions de *Non-Fonctionnel* ce que nous voulons rajouter à une entité logicielle sans avoir à repenser la programmation de cette entité. Nous nous plaçons, de plus, du point de vue de l'utilisateur d'un système de séparation de préoccupation plutôt que du point de vue de l'implémenteur de ce système. En pratique, l'objectif de notre travail est de faciliter la programmation des applications en adaptant le système aux besoins du programmeur, et non l'inverse. Nous considérons que l'utilisation de services non-fonctionnels doit avoir le moins d'impact possible sur le code des entités logicielles existantes.

La section 2 de ce papier présente les technologies pour la séparation des préoccupations dont nous nous sommes inspirées pour nos travaux. La section 3 résume nos recherches sur les conteneurs extensibles appliqués à notre modèle de composants. La section 4 décrit notre technologie d'objets étendus pour la généralisation de nos travaux dans le monde objet. Nous concluons par la section 5.

## 2. Les technologies illustrant la séparation des préoccupations

### *Modèles de composant à base de conteneurs : CCM et EJB*

L'objectif des EJB et de CCM est de répondre à des problèmes importants du monde industriel. L'un de ces problèmes concerne la gestion de différents services non-fonctionnels sur les composants tels que la persistance, les transaction, la distribution, etc., de manière transparente pour le programmeur des composants. Ces services sont gérés par des conteneurs qui encapsulent les instances de composants. Cette approche permet d'isoler différents métiers dans le développement d'applications à base de composants, chaque métier ayant son domaine d'expertise. Le fournisseur de la plate-forme est un expert dans le domaine de la distribution, de la sécurité, des transactions, etc. Il offre les outils nécessaires à la gestion de ces services. Le programmeur de composant, quant à lui, s'intéresse particulièrement à la fonctionnalité du composant. L'assembleur d'application connecte des composants et paramétrise les services offerts par la plate-forme. Enfin, une personne est chargée du déploiement de l'application ainsi créée.

Par rapport à l'objectif de nos travaux, CCM et EJB ne permettent pas à l'utilisateur de définir ses propres services non-fonctionnels, il n'est pas non plus possible de changer la sémantique des services existants. Les conteneurs CCM et EJB ne sont pas extensibles. Leur nombre est fixé et il n'est possible que de les paramétrer.

Dans l'exemple de CCM et EJB, nous pouvons constater que la gestion des services non-fonctionnels n'est pas totalement transparente à la programmation des composants. Au contraire, ils ne définissent qu'un cadre de programmation pour les composants où les développeurs doivent implémenter un certain nombre d'interfaces de rappel (appelées *callback*). Par exemple, dans les EJB, les composants de type

session avec état doivent implémenter l'interface *SessionSynchronization* contenant les méthodes *afterBegin*, *beforeCompletion* et *afterCompletion* pour que le conteneur puisse gérer les transactions sur le composant. Cela implique que le programmeur doit implémenter des méthodes qui ne sont pas liées directement à la fonctionnalité du composant.

Dans un contexte de conteneurs extensibles, les interfaces de rappel représentent un problème important : il n'est pas réaliste ou peu avantageux de demander à un programmeur d'implémenter pour chacun des composants toutes les interfaces de rappel nécessaires aux services non-fonctionnels dont il a besoin.

D'un autre côté, CCM et EJB utilisent des interfaces internes. Ces interfaces permettent aux composants de communiquer certaines informations au conteneur. Par exemple, dans les EJB, la méthode *setRollbackOnly()* peut être utilisée par le composant pour informer le conteneur que la transaction en cours doit échouer.

Les interfaces de rappel d'une part et les interfaces internes d'autre part montrent que certaines préoccupations gérées par les conteneurs (la transaction par exemple) ne sont pas séparées du code fonctionnel du composant. Ceci peut être vue comme une violation du principe de la séparation des préoccupations, mais peut être vue également comme un exemple de la limite de cette approche : certaines préoccupations soit ne sont pas (complètement) indépendantes, soit le programmeur souhaite une meilleure maîtrise de la préoccupation (pour des raisons de performance par exemple). Pour la transaction, une partie de la préoccupation est rendue transparente, une autre est volontairement laissée à la discrétion du développeur. L'histoire montre toutefois que, à terme, la performance est souvent sacrifiée lorsqu'il en résulte une simplification pour le développeur.

### ***La Programmation Orientée Aspect***

La séparation des responsabilités [19] a inspiré un grand nombre de recherches telles que la séparation multidimensionnelle des préoccupations d'IBM [14], les filtres de composition [1] ou encore la Programmation Orientée Aspect (POA) [18][3].

Dans cette dernière approche, chaque aspect représente une des préoccupations d'une application. La philosophie de la POA est de développer une application en programmant chaque aspect séparément puis de les tisser ensembles pour construire l'application.

En pratique, il existe plusieurs implémentations telles que JAC [15] et AspectJ [2]. Elles sont basées sur la notion de point de jonction représentant un point particulier lors de l'exécution d'une application. Des langages permettent d'enrichir l'exécution des points de jonction en associant des méthodes d'aspects dans le cas d'AspectJ et des programmes d'aspect dans le cas de JAC. Ces modèles permettent aussi d'ajouter des membres à des classes existantes. Du point de vue de leur

fonctionnement, les phases de tissage se basent sur la manipulation du code source ou du « byte code » java ; de manière statique pour AspectJ et de manière dynamique pour JAC.

### 3. Composants, architecture, CVM et conteneurs extensibles

Les modèles de composant existants tels que Microsoft .Net [17], les Enterprise Java Beans de Sun [21] ou encore Corba Component Model de l'OMG [5] ne fournissent pas la flexibilité dont nous avons besoin pour les conteneurs extensibles.

Nous avons donc défini notre propre modèle de composant [4], à partir duquel nous avons construit un modèle de conteneur extensible [8][9] permettant de créer et d'utiliser des services non fonctionnels. Lors de ces travaux, nous nous sommes fixé comme objectif d'offrir une gestion de services non fonctionnels sans pour autant disposer et modifier le code source des composants sur lesquels les services sont appliqués. Ces contraintes étaient issues de notre collaboration avec la société Dassault Systèmes [6][20].

Ce modèle de composant est géré par une Machine Virtuelle de Composant (CVM) munie de propriétés d'extension de trois sortes :

- les règles qui, attachées à la CVM, peuvent en modifier le comportement de la CVM. Celle-ci contient un ensemble de méthodes appelées *méta-opérations* dont les règles altèrent l'exécution selon trois politiques : *before*, *after* et *around*, de la même manière que dans AspectJ [2].
- Les attributs ajoutés permettent d'associer des données aux objets. La sémantique de ces attributs est laissée au soin de leurs utilisateurs. Cette sorte d'extension permet par exemple, d'associer une règle à un objet d'architecture afin d'en modifier le comportement pour offrir des services non fonctionnels.
- Les ajouts d'interfaces : notre CVM ne s'intéresse qu'à la phase d'exécution d'une application. Cependant, le cycle de vie de l'application est beaucoup plus large, de l'assemblage de composants à l'exécution de l'application chez un client, via les phases de test ou encore de déploiement. Nous avons ainsi jugé important de permettre l'extension du comportement des objets d'architecture selon la phase de cycle de vie de l'application. Par exemple, les personnes chargées du déploiement peuvent, via le mécanisme d'ajout d'interface, ajouter une méthode *select()* dans certains objets d'architecture pour choisir les versions des composants en vue du déploiement. L'implémentation réelle de cette méthode est déléguée à un objet écrit à cet effet. L'ajout d'interfaces à la CVM est équivalent aux interfaces de rappel non plus pour les composants mais pour la plate-forme. L'intérêt ici est que l'ajout d'interfaces est effectué de manière transparente à la CVM.

La CVM et ses propriétés d'extensions sont l'équivalent d'un Protocole à Méta-Objets (MOP pour *Meta-Object Protocol*) [16]. En effet, les instances de composants peuvent utiliser les propriétés d'extension pour modifier le comportement de la CVM, ce qui est équivalent à modifier des méta-objets.

### ***Conteneurs extensibles***

Nos travaux sur les conteneurs extensibles dans notre modèle de composant se base sur un langage de description des services non fonctionnels et sur un langage d'utilisation de ces services. Le langage de définition de services regroupe l'ensemble des règles et les différentes opérations de préparation pour offrir le service non fonctionnel désiré. Le langage d'utilisation de service permet d'associer le service aux différents composants d'une application. Un compilateur utilise les informations de description et d'utilisation des services pour associer les règles correspondantes aux différentes méta-opérations de la CVM.

Cela correspond à un conteneur extensible dans lequel le nombre de services n'est pas déterminé à priori. Dans le cas idéal, un service est indépendant d'une application particulière et réutilisable dans diverses applications. Nous avons cependant été confronté aux problèmes d'interactions entre les composants et les services. Ces interactions sont illustrées par les interfaces de rappel dans les EJB et CCM. Ce problème est traité dans la section suivante.

### ***Interfaces de rappel dans notre modèle de conteneur extensible***

Les interfaces de rappel représentent un problème important des conteneurs extensibles, elles montrent que les préoccupations ne sont pas toujours orthogonales à la fonctionnalité des composants. Du point de vue du programmeur de composants, il n'est pas raisonnable d'implémenter toutes les interfaces de rappel exigées par tous les services non-fonctionnels existants à un moment donné, cette tâche deviendrait trop lourde et trop coûteuse. Nous avons contourné ce problème en généralisant la notion d'interface de rappel et en extériorisant l'implémentation du code des composants. Cependant, un objet extérieur à un composant ne peut pas atteindre les champs privés de ce composant. Nous proposons de généraliser les interfaces de rappel par une interface particulière permettant à l'objet implémentant des interfaces de rappel de se comporter comme s'il était interne au composant. Cette interface de rappel généralisée contient les méthodes *getAttributeValue(String)*, *setAttributeValue(String, Object)* et enfin *invoke(Method, Object[])*. Les objets implémentant les interfaces de rappel sont fournis lors de la description de l'utilisation des services non fonctionnels sur un composant.

En ce qui concerne l'équivalent des interfaces internes de EJB et de CCM, nous offrons un mécanisme similaire permettant au programmeur de composant d'utiliser

explicitement certains services. Cela ne pose pas de problème dans notre approche car dans ce cas, le service non fonctionnel est considéré comme une prolongation voulue de la fonctionnalité du composant.

### ***Discussion***

Du point de vue de la séparation des préoccupations, ce travail est un mélange entre les approches telles que la Programmation Orientée Aspect et les modèles de composant à conteneurs : nous permettons de gérer des services non fonctionnels de manière transparente à la programmation des composants, et nous ne limitons pas le nombre de services possibles. Nous pouvons, d'une part, regrouper les composants dans une bibliothèque de composants et, d'autre part, regrouper les services non fonctionnels dans une bibliothèque de services. Ces deux bibliothèques sont indépendantes, ce qui permet une bonne réutilisation des composants et des services.

## **4. La machine à objets étendus**

Les mécanismes d'extensions utilisés précédemment ne s'appliquent qu'à notre CVM ; les objets la composant prévoient explicitement les mécanismes d'extension.

Or ces extensions seraient très utiles pour appliquer des services non fonctionnels à des objets et non seulement à des composants. Notre idée est de fabriquer une machine à objets étendus EOM (pour *Extended Object Machine*) dans laquelle des objets seraient « étendus » par des attributs, des interfaces et dont les méthodes pourraient être modifiées. Cette machine a pour objectif de rendre des services équivalents aux approches actuelles basées sur des mandataires (cf. conteneurs de EJB / CCM) et sur de la modification de code (cf. AspectJ), mais de manière plus flexible, c'est-à-dire plus dynamique et ne demandant aucune contrainte de programmation aux applications à instrumenter.

Si nous considérons notre CVM comme une application, alors notre machine est capable d'étendre les objets de cette application, ce qui permet d'ajouter la gestion de services non fonctionnels. Cette approche est applicable à tous les modèles de composants basés sur une machine d'exécution. Même les modèles ne disposant pas de CVM peuvent tirer parti de notre EOM. Par exemple certains services non fonctionnels peuvent être proposés pour des Java Beans, par extensions des beans eux-mêmes.

### ***Fonctionnement de notre machine d'exécution***

Suite aux besoins exprimés par Dassault Systèmes, nous avons cherché une solution qui soit potentiellement applicable aussi bien à du byte code, mais aussi aux

binaires exécutables classiques. Cette section présente le fonctionnement de notre EOM pour le langage Java, nous n'avons pas encore vérifié pour d'autres langages tel que C++.

Une classe est rendue extensible en générant une sous-classe contenant toute la mécanique nécessaire à la gestion des extensions :

- La sous-classe redéfinit toutes les méthodes déclarées comme étendues de la classe et invoque un contrôleur de règles chargé de déclencher et de gérer l'ordonnancement des règles concernées,
- Une table permettant de gérer les attributs additionnels est insérée.

En ce qui concerne les interfaces ajoutées, la sous-classe implémente chacune des méthodes de l'interface, ces méthodes redirigent les appels vers un objet qui implémente effectivement cette interface. Il s'agit du concept de délégation.

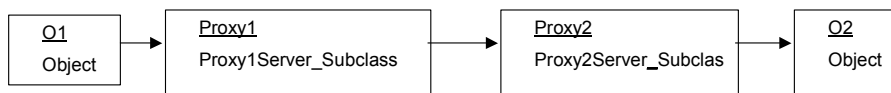
Une fois la sous-classe générée, nous changeons le nom de la classe originale (avec BCEL *Byte Code Engineering Library* [7]), la sous-classe prend alors le nom de la classe originale. Les clients de la classe originale vont alors utiliser, de façon transparente notre sous-classe.

Pour des questions de flexibilité et de performance, nous avons trouvé un compromis entre extensions statiques et extensions dynamiques. Les règles et les attributs sont dynamiques, c'est-à-dire qu'il est possible d'en ajouter pendant l'exécution de l'application. Par contre, les ajouts d'interfaces sont traités statiquement car il faut générer les méthodes correspondantes.

Au lieu de générer une sous-classe de l'objet, nous aurions pu modifier directement le code binaire de cet objet. Cependant nous n'avons pas opté pour ce choix pour des questions de simplicité. En effet, il est beaucoup moins difficile de générer du code source que de générer et d'insérer du code binaire dans un objet existant.

### ***Règles vs mandataires***

Les mandataires sont souvent utilisés pour rendre des services non-fonctionnels. Il est alors intéressant de comparer les règles avec les mandataires. La Figure 2 montre une solution classique pour ajouter plusieurs services non fonctionnels à un objet O2.



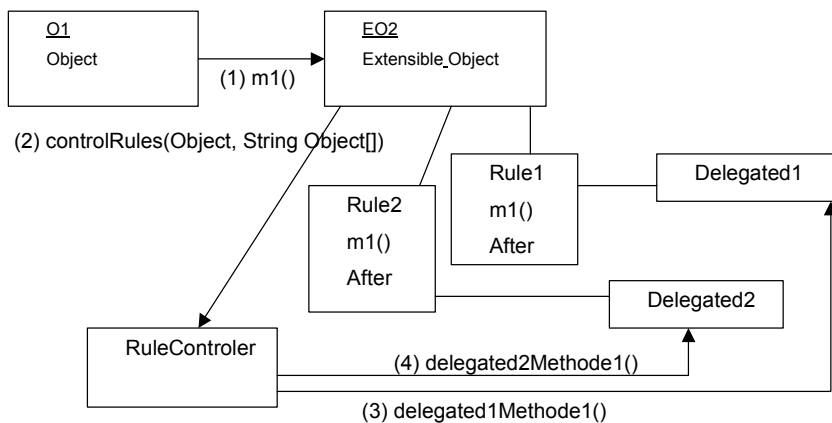
**Figure 2 Utilisation de mandataires**

Un objet O1 croit communiquer directement avec un objet O2, mais des mandataires (*Proxys* [12]) sont insérés entre O1 et O2 pour fournir différents services. Du point de vue des appels de méthodes, cette manipulation est transparente car O1 référence O2 par un attribut de type O2 (ou de l'une de ses interfaces), type qui est respecté car les mandataires sont des sous-classes de O2.

Cette approche pose néanmoins plusieurs problèmes de mise en œuvre :

- La mise en place la chaîne de mandataires demande de remplacer l'utilisation du *new* de Java par l'appel à une usine pour l'objet O2 (*factory* [12]), ce qui impose des contraintes de programmation à O1. Cette approche ne peut pas être utilisée pour du code existant.
- Étant des sous-types de l'objet serveur, les mandataires sont spécifiques à ce serveur et ne sont pas réutilisables pour d'autres objets.

La Figure 3 montre un exemple de fonctionnement des règles : L'objet O1 référence directement l'objet extensible EO2. Deux règles sont attachées à l'objet EO2, elles interceptent la même méthode m1() de EO2. Lorsque O1 invoque la méthode m1() sur EO2 (1), le contrôle est passé au contrôleur de règles (2).



**Figure 3 Fonctionnement des règles**

Le contrôleur appelle successivement les objets délégués indiqués par les règles (3 et 4), puis exécute m1(). Dans cette situation, l'exécution de m1() est complétée par l'exécution des méthodes déléguées de chacune des règles.

Cette approche offre les avantages suivants:

- Il n'est nul besoin d'imposer des contraintes de programmation aux clients de l'objet étendu. Tout objet peut bénéficier du service.

-Le contrôleur est un interpréteur, il est possible d'ajouter et de supprimer des règles dynamiquement.

-Les objets délégués effectuent les mêmes opérations que les mandataires de la situation précédente, mais ne dépendent plus de l'objet étendu. Les objets délégués peuvent être utilisés tel quel pour tout objet qui souhaite bénéficier de ce service.

Le contrôleur, indépendant de la machine à objet étendu, peut être spécialisé pour des classes d'application ayant des besoins spécifique (performance, dynamique, coordination...). Dans notre cas, nous avons optimisé notre gestionnaire de règle pour une gestion des conteneurs extensibles. Les règles ne sont interprétées que lors du premier accès à l'objet, les objets délégués sont ensuite directement exécutés, et les mandataires, s'il y en a, sont directement chaînés ; c'est une approche JIT (*just in time*) appliquée à l'interprétation des règles.

## 5. Conclusion

Au cours des dernières années, nous avons défini différents modèles de composant et implémenté leur plate-forme d'exécution que nous appelons CVM. Nous avons utilisé une approche inspiré de l'AOP et des MOP pour définir et implanter un système de conteneur extensibles.

Les différentes implémentations que nous avons faites nous ont amené à généraliser nos travaux, et à proposer notre machine à objets étendus. Notre OEM peut être vue comme une implémentation de l'AOP similaire en bien des points à JAC. Nous estimons toutefois que les implémentations de l'AOP que nous connaissons sont spécialisées pour un langage (Java le plus souvent) ; elles imposent d'avoir soit le source, soit le « byte code » des programmes à étendre. Elle sont monolithiques, assez lourdes, et imposent un langage de description d'aspect qu'il n'est pas possible d'étendre ou de spécialiser.

Nous pensons que notre EOM contribue au domaine de la séparation des préoccupations car notre système est architecturé en trois parties bien distinctes sur trois niveaux d'abstraction : la machine d'extension, l'interpréteur de règle, et le système à conteneur extensibles.

La machine d'extension est légère et performante ; elle peut être implantée pour divers langages, y compris, à priori, pour du binaire exécutable ; elle peut être utilisée pour de nombreux usages.

Le système de règle et sont interpréteur/compilateur, sont spécifiques au type d'application envisagé. De fait, nous avons déjà développé deux systèmes de règles, l'un spécialisé pour les fédérations [10][11], l'autre pour les conteneurs extensibles [8]. Bien d'autres peuvent être définis en fonctions des contraintes que l'on se donne. En particulier, pour aborder l'épineux problème de la compatibilité/cohérence de la composition d'aspect, on envisage des règles et des

langages différents, permettant de calculer statiquement certaines propriétés. De ce point de vue AspectJ ou JAC ne sont qu'une implémentation particulière, pour un usage particulier.

Concernant les conteneurs extensibles, nous avons montré qu'une approche inspirée des protocoles à méta objets (MOP) permet de généraliser le concept de conteneur, et que l'implémentation de cette approche peut elle même être généralisée pour tout système à composant et à tout objet, grâce à notre machine à objets étendus.

Le fait que notre OEM soit très générale et qu'elle peut être « branchée » sur tout système à composant et tout objet, offre une approche qui devrait permettre l'utilisation des mêmes conteneurs pour des objets et des composants hétérogènes. En appliquant une approche MOP, on devrait aussi pouvoir faire interopérer ces composant hétérogènes.

Nous pensons qu'une telle approche devrait amener à repenser profondément la conception des modèles de composant, et la construction des infrastructures qui les supportent. C'est le travail que nous avons commencé à réaliser dans notre équipe.

## 7. Bibliographie

- [1] M. Aksit, L. Bergmans and S. Vural. "An object-oriented language-database integration model: The composition-filters approach". Proceedings of ECOOP 1992.
- [2] AspectJ. <http://aspectj.org>
- [3] N.M.N. Bouraqadi-Saâdani, T. Ledoux. "Le point sur la programmation par aspects". Techniques et sciences informatiques Volume 20 – n°4/2001, pages 505-528
- [4] H. Cervantes, J.M. Favre, F. Duclos. "Describing Hierarchical Compositions of Java Beans with the Beanome Language", European joint Conferences on Theory and Practice of Software (ETAPS) 2002
- [5] Corba Component Model. <http://www.omg.org>
- [6] Dassault Systèmes. <http://www.3ds.com/>
- [7] M. Dahm. "Byte Code Engineering with the BCEL API". Technical Report B-17-98, Freie Universität Berlin, Institut für Informatik, April 2001.
- [8] F.Duclos. "Environnement de Gestion de Services Non-Fonctionnels dans les Applications à Composants". Thèse de doctorat de l'Université Joseph Fourier, le 8 octobre 2002.
- [9] F.Duclos, J.Estublier, P.Morat. "Describing and Using Non Functional Aspects in Component Based Applications". In Proceedings of the 1st Aspect-Oriented

12 Journées « Systèmes à composants adaptables et extensibles », 17 et 18 octobre 2002.

Software Development (AOSD'02), ACM Press, Enshede, Pays-Bas, 22-26 Avril 2002.

- [10] J. Estublier, H. Verjus and P.Y. Cunin. "Designing and Building Software Federations", 1st Conference on Component Based Software Engineering (CBSE). Varsovie (poland). 4-6 September 2001.
- [11] J. Estublier, H. Verjus and P.Y. Cunin. "Modelling and Managing Software Federations". European Conference on Software Engineering (ESEC). Wien (Austria). September 10-14, 2001.
- [12] E. Gamma, R. Helm, R. Johnson, J. Vlissides. "Design Patterns". Addison-Wesley, 1995
- [13] IBM. HyperJ. <http://www.alphaworks.ibm.com/tech/hyperj>.
- [14] IBM, Multi-Dimentional Separation Of Concerns, [www.research.ibm.com/hyperspace](http://www.research.ibm.com/hyperspace)
- [15] Java Aspect Components. <http://jac.aopsys.com/>
- [16] G. Kiczales, J. des Rivieres, D. G. Bobrow. "The Art of the Metaobject Protocol". The MIT Press, Cambridge, Massachusetts, 1991. ISBN 0-262-11158-6 (hc.), second printing, 1992.
- [17] Microsoft .Net. <http://www.microsoft.com/net/default.asp>
- [18] Kiczales, G., et al. "Aspect-Oriented Programming". Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Vol. LNCS 1241. Springer-Verlag. June 1997.
- [19] Lopes C. V. and Hirsch W. L., "Separation of Concerns", College of Computer Science, Northeastern University, Boston, February 1995
- [20] Y. Ledru, R. Sanlaville and J. Estublier. "Defining an Architecture Description Language for Dassault Systèmes". ISAW4. Limerick (Ireland). June 2000. Pages 115-120.
- [21] Sun Microsystems Enterprise JavaBeans. <http://java.sun.com/products/ejb/>