

---

# Les aspects pour la réalisation de patrons de conception

## Exemple de réalisation du patron Visiteur

**Ouafa Hachani — Daniel Bardou**

*Equipe SIGMA, LSR-IMAG  
BP 72  
38402 Saint Martin d'Hères Cedex  
{Ouafa.Hachani, Daniel.Bardou}@imag.fr*

---

*RÉSUMÉ. Les patrons de conception par objets offrent de nombreux avantages à être utilisés lors de la conception, ou encore de l'adaptation, d'applications et de composants construits sur la base d'objets. Toutefois la technologie Objet admet certaines limites que tente de repousser une approche plus récente connue sous le nom de programmation par aspects ou plus généralement sous le terme anglais « Advanced Separation of Concerns ». Cet article présente et motive les débuts d'un travail visant à identifier et exploiter les bénéfices issus de l'implémentation de patrons de conception par objets à l'aide d'un langage de programmation par aspects. Nous y présentons en particulier une implémentation du patron Visiteur de [GAM 95] à l'aide du langage AspectJ [KIC 02].*

*ABSTRACT. Object-oriented design patterns are useful for designing, or adapting, software programs or components, based on objects. Object-orientation has however some limitations that a more recent approach known as aspect-orientation, or more generally as "Advanced Separation of Concerns" try to eliminate. This paper presents and motivate the beginning of some work aiming to identify and gain from the benefits of implementing object-oriented design patterns using an aspect-oriented programming language. We more particularly present an implementation of the Visitor design pattern [GAM 95] using the AspectJ programming language [KIC 02].*

*MOTS-CLÉS : patrons de conception par objets, programmation et conception par aspects, évolution, réutilisation.*

*KEYWORDS : object-oriented design patterns, aspect-oriented programming and aspect-oriented software design, software evolution, software reuse.*

---

## 1. Introduction

Les patrons de conception par objets (ou *object-oriented design patterns*) définissent des solutions génériques à des problèmes récurrents en conception par objets [GAM 91, GAM 95, BUS 96, FRO 99]. Etablis à partir de solutions éprouvées, parfois par rétro-conception, on reconnaît que leur utilisation peut accélérer la phase de conception d'une application ou d'un composant, et en faciliter l'adaptation, l'évolution et la réutilisation. Parce qu'ils relèvent de la technologie Objet, qui prône par essence une décomposition des systèmes en terme d'objets (de classes) dénotant les entités (les types d'entités) du domaine d'application plutôt qu'en terme de fonctionnalités, ils sont néanmoins le plus souvent définis comme des collaborations entre plusieurs classes. Leur identification dans l'implémentation d'une application ou d'un composant est de ce fait rendue difficile, et ne repose généralement que sur une documentation rigoureuse, ou au mieux sur une organisation minutieuse des programmes en fichiers, bibliothèques, ou paquetages (suivant les possibilités du langage de programmation utilisé).

La programmation par aspects (ou *AOP* pour *Aspect-Oriented Programming*) [KIC 97] et plus globalement le courant qui lui est associé sous le nom de *AOSD* (*Aspect-Oriented Software Design*), visent à repousser ces limites en proposant des techniques permettant la programmation explicite d'aspects, c'est à dire d'unités de décomposition des programmes transversales aux classes qui constituent les unités de décomposition primaires des langages de programmation par objets<sup>1</sup>. S'il est possible d'implémenter un patron de conception (ou du moins son imitation, c'est à dire son application dans un cadre particulier), il devrait alors également être possible de l'identifier, l'isoler dans la totalité des programmes d'une application ou d'un composant, afin de pouvoir l'adapter, le faire évoluer ou le réutiliser.

Cet article présente et motive le début d'un travail mené sur l'implémentation par aspects de patrons de conception par objets, dans le but d'en améliorer la traçabilité de la phase de conception aux phases ultérieures du cycle de développement d'une application ou d'un composant. La section 2 relève plusieurs problèmes liés à l'utilisation de patrons de conception dans le cadre strict de l'approche Objet, le patron *Visiteur* de [GAM 95] y est considéré pour illustrer nos propos. La section 3 présente une nouvelle implémentation de ce patron à l'aide d'AspectJ, une extension de Java permettant la programmation par aspects. Les avantages de cette solution sont ensuite discutés en section 4, et nous concluons en exposant les perspectives de ce travail en section 5.

---

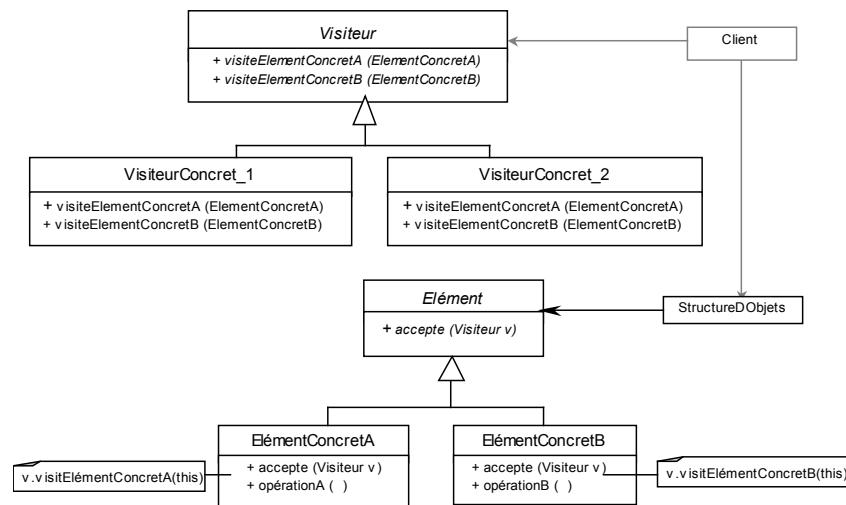
1. [KIC 97] définit un aspect comme toute unité de programme entrecoupant un ou plusieurs « composants », et un composant comme l'unité primaire de décomposition des programmes. Cette définition très générale est destinée à pouvoir être appliquée à tout paradigme de programmation, mais nous nous concentrons ici sur la programmation par objets et n'utilisons pas volontairement le terme de composant, mais plutôt celui d'objet (ou de classe) par souci de clarté. De même, nous ne considérons pas dans cet article des langages à objets sans classes, tels que les langages à prototypes [BAR 98].

## 2. Problèmes liés aux patrons de conception par objets

Si les patrons de conception sont reconnus comme étant utiles, leur utilisation n'en soulève pas moins quelques problèmes, dont certains ont d'ailleurs été relevés par les auteurs même des patrons. Nous en avons principalement relevé quatre que nous détaillons dans cette section sur un exemple d'imitation du patron *Visiteur* de [GAM 95].

### 2.1. Rappel du patron *Visiteur*

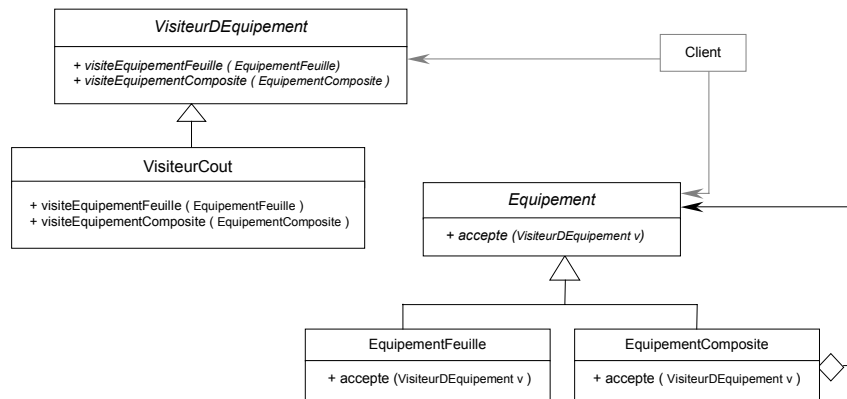
Le patron *Visiteur* permet d'ajouter un ou plusieurs comportements sur une hiérarchie de classes, en les plaçant dans une autre hiérarchie. La figure 1 montre la structure de ce patron. Les classes *VisiteurConcret\_1* et *VisiteurConcret\_2* correspondent chacune à un comportement ajouté à chacune des sous-classes de la classe abstraite *Elément*. Ces comportements sont déclenchés par appel d'une opération *accepte*, et sont déclinés en une version pour chaque sous-classe de *Elément* dans les classes tenant le rôle de visiteur concrets.



**Figure 1.** Diagramme de classes UML du patron *Visiteur* de [GAM 95]

Considérons, dans la figure 2, l'exemple d'une imitation du patron *Visiteur* dans une application dans laquelle on retrouve des classes représentant des équipements électroniques organisés dans une hiérarchie d'agrégation (on peut y reconnaître une imitation du patron *Composite* [GAM 95]). Le patron *Visiteur* a été utilisé pour ajouter une opération de calcul du coût d'un équipement, ce qui a donné lieu à la

définition des classes *VisiteurDEquipement* et *VisiteurCoût* d'une part, et l'ajout des opérations *accepte* dans la classe *Equipement* et ses sous-classes. L'opération *accepte* définie en *EquipementFeuille* fait appel à l'opération *visiteEquipementFeuille* de *VisiteurCoût* et l'opération *accepte* définie en *EquipementComposite* fait appel à l'opération *visiteEquipementComposite* de *VisiteurCoût*.



**Figure 2.** Exemple d'imitation du patron *Visiteur*

Le principal avantage de l'utilisation de *Visiteur* est qu'il suffit alors de créer de nouvelles sous-classes de *VisiteurDEquipement* pour ajouter de nouveaux comportements, sans aucune modification dans la hiérarchie des équipements. En revanche, cette conception présente plusieurs problèmes que nous détaillons dans les sous-sections suivantes.

## 2.2. Problème de confusion

Il est difficile lors de l'utilisation d'un patron de distinguer les définitions qui relèvent de l'imitation du patron de ce qui n'en relève pas. Cela tient essentiellement du fait que l'imitation d'un patron nécessite une modification de classes déjà existantes et l'ajout de nouvelles classes. Dans notre exemple, à moins de se reposer sur une documentation rigoureuse ou sur une convention de nommage, rien ne permet de reconnaître le statut particulier qu'ont les opérations *accepte* parmi toutes les opérations définies dans les classes d'équipements. De même, aucun élément particulier dénote le fait que les classes visiteurs détiennent des comportements pour le compte d'autres classes.

### **2.3. Problème d'indirection**

*Visiteur*, comme de nombreux autres patrons de [GAM 95], utilise la délégation explicite (la redirection de messages) pour activer les comportements des éléments déportés dans les classes visiteurs. Dans notre exemple, l'indirection est due au fait qu'un appel à l'opération *accepte* d'un équipement particulier se traduit en l'exécution d'une opération de *VisiteurCoût*. Cela diminue la compréhension du code, augmente les communications entre objets et introduit des dépendances préjudiciables à l'évolution de l'application.

### **2.4. Problème de rupture d'encapsulation**

L'utilisation de la délégation explicite dans le patron *Visiteur*, et dans de nombreux autres patrons, pose de plus un problème relatif à l'encapsulation. Dans notre exemple, il est fort probable que l'opération permettant effectivement de calculer le coût d'un équipement, effectivement définie et exécutée dans le contexte de la classe *VisiteurCoût* doit accéder à des attributs (voire d'autres opérations) définies dans la classe d'équipement délégrant ce comportement. L'utilisation du patron *Visiteur* impose de rendre ces attributs publics (ou du moins de leur définir des accesseurs publics) alors que l'on aurait pu souhaiter les déclarer privés ou protégés. Ce problème de rupture de l'encapsulation, d'ailleurs reconnu dans [GAM 95], est tout à fait similaire aux problèmes d'encapsulation liés au mécanisme de délégation implicite [BAR 96].

### **2.5. Problèmes liés à l'héritage**

La généralisation est extensivement utilisée dans les patrons de conception pour répartir les comportements entre les différentes classes intervenantes. L'application d'un patron sur une classe possédant déjà une super-classe peut en conséquence être rendue difficile dans le cadre d'une implémentation avec un langage de programmation ne permettant pas l'héritage multiple, ou un héritage multiple limité à l'héritage d'interfaces, tel que Java. De plus ceci renforce encore l'interdépendance des classes d'application et des patrons et est un frein à leur réutilisation séparée. Ce problème ne concerne pas directement le patron *Visiteur* qui s'applique sur une hiérarchie de classes déjà constituée, mais il peut néanmoins apparaître dans une variante où l'on désirerait ajouter des comportements à des classes n'apparaissant pas dans la même hiérarchie de généralisation.

## 2.6. Importance des problèmes relevés

Les problèmes détaillés dans les sous-sections précédentes ne se limitent pas à l'utilisation de *Visiteur* et concernent de nombreux patrons de [GAM 95]. Nous avons pu observer dans [HAC 02] que ces problèmes se posent lors de l'utilisation de plusieurs patrons, le tableau 1 fait état de ces observations.

	Confusion	Indirection due à la délégation	Dépendance due à l'héritage	Rupture d'encapsulation
<i>Visiteur</i>	x	x		x
<i>Stratégie</i>	x	x		x
<i>Médiateur</i>	x	x		x
<i>Observateur</i>	x	x	x	x
<i>Adaptateur</i>	x	x	x	x
<i>Pont</i>	x	x		x
<i>Décorateur</i>	x	x		x

**Tableau 1 :** *Problèmes relevés pour quelques patrons de [GAM 95]*

Ces quatre types de problèmes découlent de deux problèmes récurrents en programmation, ceux de la dispersion et de l'enchevêtrement de code [HUR 95], [KIC 97] :

- la confusion est le résultat de l'enchevêtrement du code relatif au patron dans les différentes classes ;

- l'utilisation de la délégation et du mécanisme d'héritage dans la réalisation de ces patrons est due à la dispersion de l'implantation du patron dans les différentes classes intervenant dans le patron, et de cela découle l'obligation de rompre l'encapsulation.

C'est essentiellement afin de s'affranchir de ces deux problèmes que [KIC 97] ont proposé la programmation par aspects, que nous considérons dans la section suivante.

## 3. Utilisation d'aspects dans l'implémentation de patrons de conception

Nous rappelons brièvement dans cette section les principes de la programmation par aspects, puis nous expliquons comment le langage de programmation par aspects AspectJ peut être utilisé pour implémenter le patron *Visiteur* de [GAM 95]. Nous discutons des avantages de cette implémentation dans la section 4.

### 3.1. *Programmation par aspects*

La programmation par aspects [KIC 97] propose de structurer les programmes en les décomposant en *aspects* et classes. Un aspect est une unité de décomposition implémentant une propriété transversale au découpage de l'application en terme de classes. Les problèmes d'enchevêtrement et de dispersion du code peuvent être résolus grâce à cette double décomposition en aspects et composants : on peut définir autant d'aspects sur une classe que nécessaire pour y placer le code qui s'y trouverait enchevêtré, et un aspect permet de regrouper le code qui se trouverait dispersé dans plusieurs classes.

L'interaction entre aspects et classes est définie à l'aide de *points de jonctions*, des points du flot d'exécution des programmes : il est par exemple possible de considérer comme point de jonction en AspectJ [KIC 01], l'appel à une opération, le retour d'une opération, l'accès en écriture ou en lecture à un attribut [XER 02]. La composition des aspects et des composants, appelée *tissage* (ou *weaving*) [KIC 97] a généralement lieu lors de la composition. Retarder cette composition relâche le couplage entre les aspects et les classes, offrant ainsi de nouvelles perspectives de réutilisation. Nous nous proposons d'en étudier les avantages dans l'implémentation de patrons de conception.

### 3.2. *Patron Visiteur par aspects*

Pour pallier les problèmes liés aux patrons de conception (relevés en section 2), et permettre de retracer leur utilisation dans l'implémentation d'une application ou d'un composant, nous proposons de réaliser ces patrons à l'aide de la programmation par aspects. A cette fin, nous considérons plus le catalogue de patrons de Gamma comme un catalogue de problèmes récurrents en conception par objets, plutôt qu'un catalogue de solutions. En effet, les solutions ne sont décrites dans [GAM 95] qu'en termes de classes, et notre propos est de tirer parti des nouvelles possibilités offertes par la programmation par aspects. Nous retenons donc essentiellement l'intention et les indications d'utilisation des patrons auxquels nous nous intéressons.

L'intention de Visiteur étant d'ajouter des comportements à une hiérarchie de classes existantes, nous proposons d'utiliser les introductions<sup>2</sup> d'AspectJ [XER 02] qui permettent d'ajouter statiquement des opérations à une classe, sans modifier la définition de celle-ci. La hiérarchie des classes visiteurs et l'ajout de méthodes accepte se retrouvent alors inutiles, et il suffit de définir une introduction pour chaque version d'opération devant être ajoutée. La figure 3 montre la structure que nous proposons pour le patron *Visiteur* réalisé à l'aide d'aspects. La notation utilisée dans cette figure est une extension d'UML intégrant les aspects proposée par

---

2. Une introduction définie dans un aspect, permet d'ajouter une définition d'attribut ou d'opération à une classe.

[SUZ 99] ; les relations de réalisation (lignes pointillées avec pointe de flèche triangulaire) y relie chaque aspect à ses classes concernées.

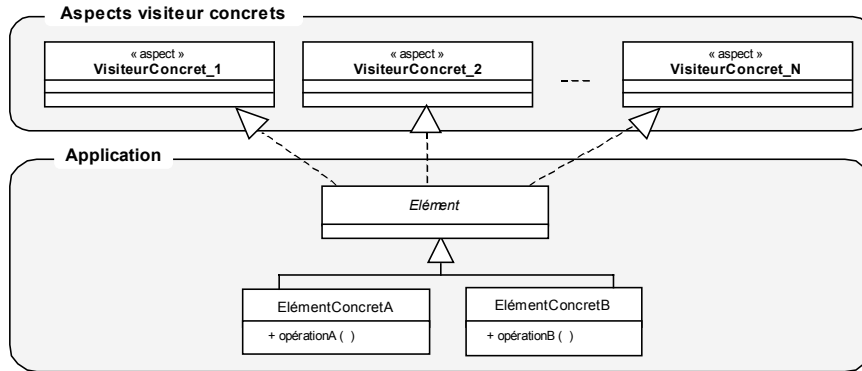


Figure 3. Structure du patron *Visiteur par aspects*

Les aspects *VisiteurConcret<sub>n</sub>* constituent le patron *Visiteur*, chacun d’eux contient les introductions permettant l’ajout d’un comportement aux classes éléments. Ajouter un nouveau comportement à ces classes revient donc à créer un nouvel aspect visiteur concret, de même qu’il suffit de supprimer l’un de ces aspects pour retirer un comportement. Le patron se trouve parfaitement séparé du reste de l’application, ce qui facilite son évolution et sa réutilisation ; il en est de même pour le code propre à l’application.

La figure 4 montre l’application de ce *Visiteur par aspects* à notre exemple de hiérarchie d’équipements. Un seul aspect *VisiteurCoût* a été défini dans cet exemple, il contient une introduction de l’opération *calculCoût* (un nom d’opération plus explicite qu’*accepte*) pour chaque classe de la hiérarchie d’équipements. La figure 5 montre une partie du code de l’aspect *VisiteurCoût* en AspectJ.

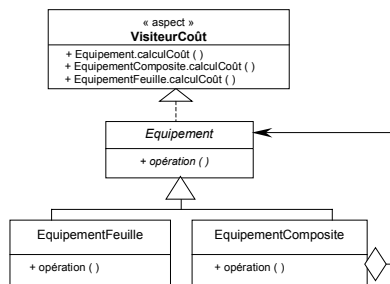


Figure 4. Exemple d’imitation du patron *Visiteur par aspects*

```

privileged aspect VisiteurCoût { // privileged permet l'accès aux propriétés privées des classes
concernées

    // introduction de l'opération calculCoût dans les classes Equipement et EquipementComposite
    public Monnaie (Equipement || EquipementComposite).CalculCout ( ) { ... }

    // introduction de l'opération calculCoût dans la classe EquipementFeuille
    public Monnaie EquipementFeuille.calculCoût ( ) { ... }
    ...
}

```

**Figure 5.** *Squelette du code de l'aspect VisiteurCoût*

#### 4. Discussion

Nous discutons dans cette section la solution d'implémentation par aspects du patron *Visiteur* de [GAM 95] que nous avons présenté dans la section précédente.

##### 4.1. Résolution des problèmes liés à l'utilisation des patrons de conception

La solution de réalisation par aspects du patron *Visiteur* que nous proposons résout les problèmes relevés en section 2, nous l'expliquons ci-dessous.

- Le problème de confusion ne se pose plus, le code relatif à l'application du patron étant clairement séparé du code relatif au reste de l'application. En particulier, il n'est pas nécessaire d'ajouter une méthode accepte dans la hiérarchie sur laquelle s'applique *Visiteur*.

- Le problème d'indirection est également résolu dans la mesure où les opérations ajoutées sont directement et statiquement introduites dans les classes de la hiérarchie sur laquelle s'applique le patron.

- Le problème de rupture d'encapsulation est évité : la délégation n'est plus utilisée et, dans notre exemple, l'opération de calcul du coût est directement exécutée dans le contexte d'une classe d'équipement.

- Les problèmes liés à l'héritage, bien que peu présents sur l'exemple de *Visiteur*, peuvent être également diminués. Il est par exemple possible d'ajouter un comportement à des classes issues de hiérarchies de généralisation différentes en utilisant les introductions adéquates. De plus, AspectJ permet d'intercaler un aspect dans la hiérarchie d'héritage entre une classe et sa super-classe. Ce problème a pu être résolu dans le cadre de la réalisation d'autres patrons de Gamma, tel que *Observateur*, par exemple [HAC 02].

#### 4.2. Variantes

La réalisation de *Visiteur* par aspects que nous proposons préconise de prévoir un aspect pour chaque comportement ajouté. Si nous en avons ainsi décidé, c'est dans un souci de conserver l'un des avantages de la structure initiale de *Visiteur*, le fait que chaque classe visiteur concret localise toutes les définitions relatives à un comportement ajouté. Ce n'est qu'une réalisation possible parmi d'autres, il est également possible de définir les introductions ajoutant tous les comportements voulus au sein d'un aspect visiteur unique, ou encore de prévoir plusieurs aspects visiteurs regroupant des introductions correspondant à des services liés les uns aux autres ou interdépendants. On peut enfin noter qu'un langage tel qu'AspectJ permet l'héritage entre aspects et la définition d'aspects abstraits [XER 02], ce qui laisse de nombreuses possibilités quant à l'organisation des aspects jouant le rôle de visiteurs.

#### 5. Travaux connexes

Il existe à notre connaissance peu de travaux menés sur les patrons de conception et la programmation par aspects, nous en citons quelques-uns ci-dessous.

– [LOR 98] affirme que le patron *Visiteur* implémenté d'une certaine manière dans une version modifiée de Java est une forme de programmation par aspects et n'a de fait que peu de rapports avec notre travail (le cheminement de sa pensée est inverse à notre démarche).

– [NOD 01] se sont eux intéressés à l'implémentation des patrons *Observateur*, *Composite* et *Adaptateur* de [GAM 95] en AspectJ et Hyper/J [OSS 01] : leur démarche est assez proche de la nôtre mais néanmoins différente. Ils s'attachent à définir très fidèlement un aspect correspondant à chaque classe introduite dans l'implémentation d'un patron de conception par objets, puis à assurer une liaison entre les aspects ainsi obtenus et les classes d'application en introduisant des aspects supplémentaires, alors que nous nous attachons à tirer parti au maximum des possibilités de la programmation par aspects en ne retenant que l'intention des patrons. Notre proposition d'implémentation par aspects d'*Observateur* [HAC 02] est par exemple différente de la leur.

– [HAN 02] se propose aussi de fournir des implémentations de patrons de conception en AspectJ. Nous partageons avec ce travail très récent, plus exhaustif que le nôtre puisque la totalité des 23 patrons de [GAM 95] est traitée, un objectif et un domaine d'expérimentation communs. Nous avons rapidement comparé quelques-uns de leurs résultats aux nôtres et nous avons certaines différences. Par exemple, leur réalisation de *Visiteur* par aspects s'attache à reproduire une partie de la structure initialement proposée dans [GAM 95], alors qu'en nous focalisant sur l'intention de *Visiteur*, nous aboutissons à une solution différente.

## 6. Conclusion

Nous avons présenté et motivé dans cet article une réalisation du patron *Visiteur* de [GAM 95] à l'aide d'AspectJ. Nous avons relevé plusieurs problèmes liés à l'utilisation de *Visiteur* dans sa réalisation initiale et expliqué comment notre réalisation permet d'éviter ces problèmes. Le mécanisme d'introduction n'est pas unique en AspectJ, et ce langage propose également d'autres mécanismes et concepts absents des langages strictement objet (points de jonction, points de recouvrement, perfectionnement, ...). Si les introductions sont suffisantes pour réaliser *Visiteur*, ces mécanismes peuvent s'avérer utiles dans la réalisation d'autres patrons, les perfectionnements sont en particulier une bonne alternative à la délégation (voir, par exemple, les patrons *Observateur* et *Stratégie* par aspects dans [HAC 02]).

Une idée importante de notre approche est que, pour tous les patrons sur lesquels nous nous penchons, nous nous efforçons d'aboutir à une solution dans laquelle une imitation de patron peut être identifiée dans le code par un aspect. Au delà du fait que cela présente l'avantage de résoudre *de facto* le problème de confusion, cela devrait avoir pour effet de pouvoir considérer les patrons non seulement comme éléments de conception réutilisables, mais aussi comme éléments d'implémentation réutilisables. Enfin, passer de l'utilisation d'un patron à celle d'un patron alternatif dans une conception, correspondrait au niveau du code à passer d'un aspect à un autre.

Le travail que nous présentons n'en est encore qu'à son début, nous espérons élargir rapidement le champ des patrons considérés à l'ensemble des patrons de [GAM 95]. De la même manière, nous n'avons pour l'instant considéré que le langage AspectJ, mais il nous reste à étudier dans quelles mesures nos propositions sont applicables dans d'autres langages de programmation par aspects tels que HyperJ [OSS 00] ou AspectS [HIR 01]. En améliorant la traçabilité des patrons de conception dans l'implémentation des applications et des composants dans lesquels ils sont utilisés, les résultats de ce travail devrait permettre d'améliorer la réutilisation et l'adaptation de ces derniers.

## 6. Bibliographie

- [BAR 96] Bardou D., Dony C., « Split Objects : a Disciplined Use of Delegation within Objects », *Proceedings of OOPSLA'96, ACM SIGPLAN Notices*, Vol. 31, N°10, 1996.
- [BAR 98] Bardou D., « Etude des langages à prototypes, du mécanisme de délégation et de son rapport à la notion de point de vue », *Thèse de doctorat de l'Université de Montpellier II*, 1998.
- [BUS 96] Buschman F., « What is a pattern ? », *Object Expert*, vol. 1, n°3, 1996, p.17-18.
- [FRO 99] Front-Conte A., Giraudin J.-P., Rieu D., Saint-Marcel C., « Réutilisation et patrons d'ingénierie », *Génie Objet : Analyse et Conception de l'évolution*, 1999, p. 91-136.

- [GAM 91] Gamma E., « Object-Oriented Software Developments based on ET++ : Design Patterns, Class Library, Tools », *PhD thesis at University of Zürich*, 1991.
- [GAM 95] Gamma E., Helm R., Johnson R., Vlissides J., « *Design Patterns : Elements of Reusable Object-Oriented Software* », Addison-Wesley, 1995.
- [HAC 02] Hachani O., « Apport de la programmation par aspects dans l'implémentation des patrons de conception par objets », *Mémoire de DEA à l'Université Grenoble 1*, 2002.
- [HAN 02] Hannemann J., Kiczales G., « Design Pattern Implementation in Java and AspectJ », *to appear in Proceedings of OOPSLA 2002*, 2002.
- [HIR 01] Hirschfeld R., « AspectS – AOP with Squeak », *OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.
- [HUR 95] Hursh W., Lopes C., « Separation of Concerns », *Rapport NU-CCS-95-03, College of Computer Science, Northeastern University*, 1995.
- [KIC 97] Kiczales G., Lamping J., Menhdhekar A., Maeda C., Lopes C., Loingtier J.-M., Irwin J. « Aspect-Oriented Programming », *In proceedings of ECOOP'97, Lecture Notes in computer Science*, Vol. 1241, Springer-Verlag, 1997, p. 220-242.
- [KIC 01] Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J., Griswold W.G., « An Overview of AspectJ », *In proceedings of ECOOP 2001, Lecture Notes in computer Science*, Vol. 2072, Springer-Verlag, 2001, p. 327-353.
- [LOR 98] Lorenz D.H., « Visitor Beans : An Aspect-Oriented Pattern », *ECOOP'98 Workshop on Aspect-Oriented Programming*, 1998.
- [NOD 01] Noda N., Kishi T., « Implementing Design Patterns Using Advanced Separation of Concerns », *Submitted to OOPSLA 2001 Workshop on ASOC in OOS*, 2001.
- [OSS 00] Ossher H., Tarr P., « Hyper/J : Multi-dimensional separation of concerns for Java™ ». *In proceedings of the 22<sup>nd</sup> ICSE june 2000, 4-11 Limerick Irland*, p. 734-737, *ACM 2000*.
- [OSS 01] Ossher H., Tarr P., « Hyper/J : Multi-dimensional separation of concerns for Java ». *In proceedings of ICSE 2001*, 2001, p. 275-284.
- [SUZ 99] Suzuki J., Yamamoto Y., « Extending UML with Aspects : Aspect Support in the Design Phase », *ECOOP'99 Workshop on Aspect-Oriented Programming*, 1999.
- [XER 02] Xerox Corporation, The AspectJ Programming Guide, <http://aspectj.org/doc/dist/progguide/index.html>, 2002.